GNU C Kütüphanesi Başvuru Kılavuzu

Çeviren:
Nilgün Belma Bugüner
Belgeler Grubu Yöneticisi

Çeviren:
Yaşar Dereli
Dokuz Eylül Üniversitesi

17 Ocak 2007

Özet

Bu kitap GNU C kütüphanesini belgelendirir.

Çeviri, 23 Nisan 2006'da kılavuzun libc–2.4 paketiyle dağıtılan sürümünden güncellenmiştir.

[The GNU C Library Reference Manual]'in 0.10 baskısının çevirisidir.

İçindekiler

Özgün Belge için Yasal Uyarı

Türkçe Çeviri için Yasal Uyarı

Teşekkür

Kılavuzun çevirisi sırasında zorlandığım ya da çelişkiye düştüğüm yerlerde, yardımlarını esirgemeyen,

- kardeşim Yücel Haluk Bugüner[B4]'e ve
- yerelleştirme ile ilgili çalışmalarda yoldaşım olan Deniz Akkuş[B5]'a

teşekkür ederim.
Nilgün Belma Bugüner[B6] — 18 Kasım 2004

Çeviri Hakkında

GNU C Kütüphanesi Kılavuzu gibi kapsamlı bir C işlevleri kılavuzu şimdiye dek Türkçe'ye çevrilmemişti (bildiğim kadarıyla). Her ne kadar C dili ile geliştirilen bir yazılım teamülen tamamen İngilizce sözcüklerle yazılıyorsa da, ülkemizde hiç İngilizce bilmeyenlerin de C ile yazılım geliştirebildiklerini biliyoruz.

Belgeler.org ya da az bilinen ismimizle "Linux Belgelendirme Çalışma Grubu" olarak özgür kaynak kodlu yazılımların özgür belgelerini Türkçeye çevirme görevini *gönüllü* olarak üstlendiğimizden, ilk olarak çok gereksinim duyulan bu kılavuzu çevirmeye karar verdik ve kılavuzun İngilizce tam metnini tefrika bölümüne yerleştirip çeviriye gönüllü katılım daveti yaptık (17 Ağustos 2003).

İlk bölümleri ben çevirdim, bir süre sonra Yaşar Dereli bazı bölümlerin çevirisini üstlendi ve *Şifrelemeyle İlgili İşlevler* (sayfa: 803), *Soketler* (sayfa: 398), *Süreçler* (sayfa: 685) ve *Borular ve FIFOlar* (sayfa: 393) oylumlarını çevirdi. Evlendikten sonra sanırım pek fırsat bulamadı ve bunu sürdüremedi. Başka üstleniciler de oldu ama onlar bu kararlarının gereğini yerine getirmediler. Sonuçta, Yaşar Dereli'nin çevirdiği oylumlar hariç kılavuzun kalanı benim tarafımdan çevrildi.

Böyle bir kılavuzun ilk defa çevrilmeye çalışılmasının bazı zorlukları vardı. Bazı terimlerin türkçe karşılıkları hiç yoktu.

Kılavuzda C diline özgü işlev, makro, veri türü, HTML dosya isimleri gibi sözcükler dışında İngilizce terim yoktur. Dosya isimlerine dokunulmamasının sebebi özgün metinle çeviri arasında kolay bağ kurulabilmesini sağlamaktır.

C dilinde yazılım geliştirenlerin yadırgayacakları ama okuma ve anlama kolaylığı sağladığı için yapılan bir uygulamaya burada dikkat çekmek isterim. Kılavuzda işlevlerin argümanları da türkçeye çevrilmiş ve bu yapılırken türkçeye özgü karakterler de kullanılmıştır. Halbuki C dilinin sözdizimsel yapısında ascii 127 karakter dışında karakter kullanılamaz. Bu kod parçalarında ascii 127 dışında karakter kullanırsanız kodu derlerken hata oluşur. Bu bakımdan bu işlevleri kopyala/yapıştır yöntemiyle alıp kullanmayın (Ek–B'deki listelerde bu işlem uygulanmamıştır, onları kopyalayıp yapıştırabilirsiniz.)

Kılavuzda kullanılan dile özel bir dikkat gösterilmiştir. Kavramsal karmaşaya yol açmamak için terimler özenle seçilmiştir. Kimi yerde bir terim size şaşırtıcı gelebilir. Bunun sebebi, orada kullanılması gerektiğini düşündüğünüz terimin içerdiği kavramla daha iyi örtüşen bir yerde o terimin kullanılmış olması, orada ise yine o terim kullanıldığında bir kavram kargaşasına yol açacak olmasıdır. Bu bakımdan kılavuz içinde karşılaştığınız yeni terimleri bulunduğu bağlam içinde değerlendirmelisiniz. Örneğin, "thread" için şimdiye kadar çeşitli kaynaklarda "kanal" karşılığı kullanılmış (bence hiç de kavramla örtüşmüyor). Kılavuz içinde ise dosya akımları ile dosya tanıtıcılarından birlikte bahsedilirken kullanılan bir terim bu "kanal" hazretleri. Dolayısıyla kılavuz kapsamında "thread" için "evre" karşılığı kullanılmıştır. Ayrıca, aynı bağlamda olmasa da, "reentrant" ve "thread–safe" kılavuz boyunca eşanlamlı kullanılmış olduğundan her ikisine de "evresel" denmiştir. Burada bahsedemeyeceğim daha çook yeni terim var ama metni okurken, metin anlamlarınıda içerdiğinden onları kolay kolay farketmeyecek ve doğal olarak sizler de onları kullanmaya başlayacaksınız.

Ayrıca, özellikle bu kılavuz için, işlevlerin HTML sayfalar üzerindeki şimdiki görüntüsünü elde etmek için ve kılavuzun sonundaki dizinleri elde etmek için Docbook–xsl kodlarında değişiklik yapılmıştır. Eğer kılavuzun

PDF çıktısını almak için XML taslaklarını standart DocBook–xsl kodları ile derlerseniz bu çıktıyı elde edemezsiniz (hatta hiç derleyemezsiniz). Bunun yerine XML taslakları aldığınız yerde bulunan belgeler–xsl kodunu (`cvsroot/sitesrc/docbook/`) ve tabii belgeler–dtd kodunu alıp kullanmalısınız. (Bu durum belgeler.org'daki bütün belgelerin XML taslakları için geçerlidir. belgeler.org herşeyiyle özel ve bu özellikleriyle bir bütünlük içindedir.)

Kılavuzun, C diliyle yazılım geliştirenler ve C diliyle yazılım geliştirmeseler bile tüm Linux kullananlar tarafından en azından bir kere okunmasını öneririm. Kafalarındaki pek çok sorunun yanıtını bu kılavuzun içinde bulacaklarına ve bu kılavuzu başucu kılavuzu haline getireceklerine eminim.

Eğer kılavuzunun metni içinde çevrilmemiş, çevrildiği halde orada kalmış metinlere rastlarsanız ya da bir imla hatası bulursanız lütfen belgeler.org üzerinden bildirin. Çevrilmeden bırakmış metinleri çevirip yollarsanız özellikle memnun olurum `:-)` (Var, biliyorum, bazan birbiriyle ilişkili olduğundan konudan konuya atlayarak çevirdim, geri dönüp onları da çevirdim, bazıları gözümden kaçmış olabilir, ama şimdi yerlerini bulmak o kadar zor ki. Salt metin 2,5 MB dile kolay...)

Nilgün Belma Bugüner — 18 Kasım 2004.

I. Giriş

İçindekiler

C dili, girdi/çıktı işlemleri, bellek yönetimi, metin düzenleme ve benzeri işlemlerin uygulanabilmesi için hiçbir yerleşik çözüm sağlamaz. Bu çözümler yazılımlarınızla ilintileyip derleyebileceğiniz bir standart ***kütüphane*** içinde tanımlanır. Bu belgenin konusu olan GNU C kütüphanesi, ISO C standardında belirtilmiş olan tüm kütüphane işlevlerine ek olarak Unix işletim sisteminin diğer türevleri ile POSIX'e özgü olan kütüphane işlevlerini ve GNU sistemine özel oluşumlarını tanımlar.

Bu kılavuz, GNU kütüphanesinin özelliklerini nasıl kullanacağınızı size anlatmayı amaçlar. Diğer sistemlere taşınamayacak şeylerin neler olduğuna, hangi özelliklerin hangi standartlara uyduğuna ayrıca yeri geldikçe değinilmiştir. Ancak bu kılavuzdaki bilgiler tam bir taşınabilirlik sağlamak amacında değildir.

1. Başlarken

Bu kılavuz, sizin en azından, biraz da olsa C yazılım geliştirme diline ve temel yazılım geliştirme kavramlarına aşina olduğunuz varsayımıyla hazırlanmıştır. Özellikle, "geleneksel" ISO C öncesi dilden ziyade ISO standardı olan C ile bir aşinalık olduğu varsayılmıştır.

GNU C kütüphanesi, herbiri birbirleriyle yakın ilişkili özelliklere göre gruplanmış bildirimleri ve tanımları içeren, yazılımınızı işlerken C derleyicisi tarafından kullanılan çeşitli ***başlık dosyaları*** içerir. Örneğin `stdio.h` başlık dosyasında girdi ve çıktı işlemleri ile ilgili özellikler, `string.h` dosyasında ise dizge işleme araçları bildirilmiştir. Bu kılavuzdaki konular genelde başlık dosyalarının içeriğine bağlı olarak bölümlenmiştir.

Bu kılavuzu ilk kez okuyorsanız, giriş bölümünü tamamen okumalı, diğer bölümlere de bir göz gezdirmelisiniz. GNU C kütüphanesi *epeyce* çok işlev barındırır ve onların nasıl kullanıldığını hatırlamanızı beklemek pek gerçekçi olmaz. Yazılımınızı geliştirirken kütüphane işlevlerini ne zaman kullanacağınızı ve onların akılda kalmayan özelliklerini bu kılavuzun neresinde bulacağınız bilmek için bu kılavuza bir göz gezdirmeniz gerekir.

2. Standartlar ve Taşınabilirlik

Bu bölüm GNU C kütüphanesinin üzerine inşa edildiği çeşitli standartlar ve kaynaklar hakkındadır. Bu standartlar ve kaynaklar, ISO C ve POSIX standartları ile Sistem V ve Berkeley Unix gerçeklemelerinden oluşur.

Bu kılavuzun birincil hedefi GNU kütüphanesinin içerdiklerini daha verimli nasıl kullanacağınızı anlatmaktır. Ancak, yazılımlarınızı bu standartlarla uyumlu yapmak ya da GNU dışındaki işletim sistemlerine de taşınabilir

kılmakla ilgileniyorsanız, bu, kütüphaneyi nasıl kullanacağınızı etkiler. Bu bölümde bu standartlara kısaca değinilecek, böylece kılavuzun içinde bunlara rastladıkça onların ne olduğunu bileceksiniz.

Kütüphane Oluşumlarının Özeti (sayfa: 827) bölümünde kütüphanenin içerdiği tüm işlevlerin ve sembollerin bir alfabetik listesini bulabilirsiniz. Bu liste ayrıca her işlev ve sembolün hangi standartlara uyduğu bilgisini de içerir.

2.1. ISO C

GNU C kütüphanesi Amerikan Ulusal Standartları Enstitüsü (ANSI) tarafından yayınlanmış [American National Standard X3.159–1989—"ANSI C"] ve daha sonra Uluslararası Standartlar Teşkilatı (ISO) tarafından kabul edilmiş [ISO/IEC 9899:1990, "Programming languages—C"] C standardı ile uyumludur. Daha genel bir kabul gördüğünden dolayı biz ISO C standardını referans alıyoruz. GNU kütüphanesini oluşturan içerik ve başlık dosyaları, ISO C standardı tarafından belirtilenlerin bir üst kümesidir.

ISO C standardına tamamen bağlı kalmak istiyorsanız, yazılımınızı GNU C derleyicisi ile derlerken `-ansi` seçeneğini kullanmalısınız. Bu seçenekderleyiciye ek özellikler açıkça istenmediği sürece kütüphane başlık dosyalarından *sadece* ISO C standardı olan şeyleri tanımlamasını söyler. Bu konuda daha fazla bilgi *Özellik Sınama Makroları* (sayfa: 25) bölümünde bulunabilir.

Kütüphaneye sadece ISO C özelliklerini içerecek şekilde sınırlama getirilebilmesi önemlidir, çünkü ISO C, kütüphane gerçeklemelerinde tanımlanan isimlere sınırlama getirir ve GNU genişletmesi bu sınırların dışına çıkar. Bu sınırlamalar hakkında daha fazla bilgi edinmek için *Anahtar Sözcükler* (sayfa: 24) bölümüne bakınız.

Bu kılavuzda ISO C ile diğer kabuller arasındaki farkların tüm ayrıntıları verilmeye çalışılmamıştır. Sadece farklı platformlarda çalışan yazılımlar geliştirebilmeniz için bir fikir verir.

2.2. POSIX (Taşınabilir İşletim Sistemi Arayüzü)

POSIX (The Portable Operating System Interface)

GNU kütüphanesi ayrıca, ***Bilgisayar Ortamları için Taşınabilir İşletim Sistemi Arayüzü*** olarak da bilinen ISO *POSIX* ailesi standartlarla da (ISO/IEC 9945) uyumludur. Bunlar ayrıca ANSI/IEEE Std 1003 olarak da yayınlanmıştır. POSIX genellikle Unix işletim sisteminin çeşitli sürümlerinden türetilmişitir.

POSIX standartları ile belirtilen kütüphane oluşumları ISO C ile belirlenenlerin bir üst kümesidir. POSIX, yeni ek işlevler ya da ISO C işlevlerine eklenen bazı özellikleri belirtir. Genellikle POSIX standartları tarafından tanımlanan ek gereksinimler ve işlevsellik, farklı işletim sistemi ortamlarında çalışabilen genel yazılım geliştirme dilinden ziyade işletim sistemlerinin belli çeşitlerine düşük seviyede destek sağlamak amacındadır.

GNU C kütüphanesi [ISO/IEC 9945–1:1996, the POSIX System Application Program Interface] tarafından belirlenen işlevlerin tümünü gerçekler. Kılavuzda bu standart bahis konusu olduğunda POSIX.1 nitelemesi kullanılacaktır. Bu standart tarafından ISO C oluşumlarına ek birincil genişletmeler dosya sistemi arayüzü ilkelleri (*Dosya Sistemi Arayüzü* (sayfa: 351)), aygıta özel uçbirim denetim işlevleri (*Düşük Seviyeli Uçbirim Arayüzü* (sayfa: 442)) ile süreç denetim işlevlerini (*Süreçler* (sayfa: 685)) içerir.

[ISO/IEC 9945–2:1993, the POSIX Shell and Utilities standard] (POSIX.2) içindeki bazı oluşumlara da GNU kütüphanesinde yer verilmiştir. Bunlar kalıp eşleştirme uygulamaları ile düzenli ifadeleri kullanan uygulamalardır (Bkz. *Şablon Eşleme* (sayfa: 212)).

2.3. Berkeley Unix

GNU C kütüphanesi özellikle 4.2 BSD, 4.3 BSD, 4.4 BSD Unix sistemleri (*Berkeley Unix* olarak da bilinir) ile *SunOS* (biraz Unix Sistem V işlevselliği de içeren bir 4.2 BSD türevi) gibi bazı Unix sürümlerindeki, yazılı standart haline gelmemiş oluşumları da tanımlar. Bu sistemler ISO C ve POSIX oluşumlarının çoğunu destekler. 4.4 BSD ve SunOS'un yeni dağıtımları hepsini destekler.

BSD oluşumları, *sembolik bağları* (sayfa: 365), **select** *işlevini* (sayfa: 323), *BSD sinyal işlevlerini* (sayfa: 641) ve *soketleri* (sayfa: 398) içerir.

2.4. SVID (Sistem V Arayüzü Tanımlaması)

Sistem V Arayüzü Tanımlaması (SVID – The System V Interface Description) AT&T Unix System V işletim sistemini tanımlayan bir belgedir. *POSIX standardına* (sayfa: 21) ek bazı özellikler içeren bir üst küme tanımlar.

GNU C kütüphanesi, ISO C ve POSIX standartları tarafından gerekli görülmeyen ancak SVID tarafından gerekli görülen özellikleri içeren System V Unix ve diğer Unix sistemleri (SunOS gibi) ile uyumluluk için gerekli oluşumları tanımlar. Diğer yandan, SVID tarafından gerekli görülen ancak genelde az kullanışlı ve daha zor anlaşılır olan oluşumları içermez. (Aslında Unix System V'in kendisi de onların hepsini sağlamaz.)

Sistem V desteği sağlayan oluşumlar, paylaşımlı bellek ve süreçler arası iletişim için yöntemler, **hsearch** ve **drand48** ailesi işlevler, **fmtmsg** ile çeşitli matematik işlevleri içerir.

2.5. XPG (X/Open Taşınabilirlik Kılavuzu)

X/Open Taşınabilirlik Kılavuzu, X/Open Company, Ltd. şirketi tarafından yayınlanmış POSIX'den daha bir genel standarttır. X/Open, Unix telif hakkına sahiptir ve XPG de bir Unix sistemi olarak kabul edilen sistemler için gereksinimleri belirtir.

GNU C kütüphanesi X/Open Taşınabilirlik Kılavuzunun 4.2 sürümü ile tüm X/Open Unix genişletmelerine ve XSI (X/Open Sistem Arayüzü) uyumlu sistemlerde ortak olan tüm genişletmelere uyar.

POSIX'in üstüne yapılan eklemeler esas olarak Sistem V ve BSD sistemlerinde bulunan işlevsellikten türetilmiştir. Sistem V sistemlerindeki bazı berbat yanlışlıklar da düzeltilmiştir.

3. Kütüphanenin Kullanımı

Bu bölümde GNU C kütüphanesinin kullanımıyla ilgili bazı pratik çözümlere yer verilmiştir.

3.1. Başlık Dosyaları

C yazılımları tarafından kullanılan kütüphaneler gerçekte iki ana parçadan oluşur: veri türlerinin ve makroların tanımlandığı, değişkenlerin ve işlevlerin bildirildiği ***başlık dosyaları*** ile değişken ve işlev tanımlarının bulunduğu ***arşiv***, yani asıl kitaplık.

(C ile ilgili bilgileri hatırlayalım: Bir ***bildirim*** bir işlev ya da bir değişkenin varlığı ve türü hakkında bilgi verir. İşlev bildirimi ayrıca argümanlarının türleri hakkında da bilgi verir. Bildirim işlemi yoluyla, derleyiciler nesnelerin hangi özelliklere sahip olduklarını anlarlar. ***Tanımlama*** ise derleyiciye değişken için bellekte yer ayırmasını ya da işlevin ne yaptığını belirtmek içindir. Yani tanımlama ile derleyici bellekte bir yer açarken, bildirim de bunu yapmaz.) GNU C kütüphanesindeki oluşumları kullanırken, yazılımınızın kaynak koduna ilgili başlık dosyalarını dahil etmeniz gerekir. Böylece derleyici bu oluşumların bildirimlerini edinir ve onları doğru olarak işler. Yazılımınız önişlemci tarafından işlendikten sonra ilintileyici bunların sağladığı bilgilerle arşiv dosyasındaki tanımlamalara ulaşır.

Başlık dosyaları bir yazılımın kaynak koduna **#include** önişlemci deyimi ile dahil edilir. C dili bu deyimi iki şekilde kabul eder; Birinci kullanım,

```
#include "başlık"
```

şeklindedir ve bununla kendi yazdığınız, yazılımınızın farklı parçaları arasındaki arayüzleri açıklayan bildirim ve tanımları içeren *başlık* isimli bir başlık dosyasını kaynak kodunuza dahil edersiniz. İkinci kullanımı ise,

```
#include <dosya.h>
```

şeklindedir ve bununla bir standart kütüphanenin bildirim ve tanımlarının bulunduğu `dosya.h` başlık dosyasını kaynak kodunuza dahil edersiniz. Bu dosya normalde sistem yöneticiniz tarafından standart bir yere konmuştur. C kütüphanesinin başlık dosyalarını kaynak kodunuza dahil etmek istediğinizde bu ikincisini kullanmalısınız.

Genellikle, **`#include`** önişlemci deyimleri C kaynak kodunun ilk satırlarında bulunur. Kaynak dosyalarınızın başına bu kodun ne yaptığına ilişkin bazı açıklamalar koyuyorsanız, *özellik sınama makrolarının* (sayfa: 25) tanımlarını hemen altındaki satırlara koyun ve ardından da **`#include`** önişlemci deyimlerini yerleştirin.

GNU C kütüphanesindeki başlık dosyaları birbiriyle ilgili oluşumların tür ve makro tanımları ile değişken ve işlev bildirimleri şeklinde gruplanarak oluşturulmuştur. Bu nedenle kullanmak istediğiniz özellikler ile örtüşen çok sayıda başlık dosyasını kaynak kodunuza dahil etmeniz gerekebilir.

Bazı kütüphane başlık dosyaları başka bir kütüphanenin başlık dosyalarını da içerebilir. Bu bir yazılım geliştirme tarzı ile ilgilidir ve siz buna pek bel bağlamayın; en iyisi kullandığınız kütüphane oluşumları için gerekli başlık dosyalarını kendiniz yine de kodunuz dahil edin. Kaynak koduna aynı başlık dosyasının defalarca dahil edilmiş olmasının bir önemi yoktur. İlk dahil edilenden sonrakiler etkisizdir.

Uyumluluk bilgisi

ISO C gerçeklemelerinin hepsi başlık dosyalarının sıralaması ve defalarca içerilmesi durumlarında bu şekilde davranır. Ancak, çok eski C gerçeklemelerinde bu geleneksel bir durum haline gelmemişti.

Belirtmek gerekir ki, bir işlevin bildirildiği bir başlık dosyasını kodunuza dahil etmek zorunda değilsiniz. Bu kılavuzdaki belirtimlere uyarak, o işlevi kendiniz de bildirebilirsiniz. Ancak bu önerilmez, çünkü başlık dosyası o işlevi bir makro tanımı olarak içeriyor olabilir.

3.2. Makro Olarak Tanımlanmış İşlevler

Bu kılavuzda bir işlev olarak bahsettiğimiz bazı şeyler aslında bir makro tanımı olabilir. Bu durum makro tanımı da işlevin yaptığını yaptığından yazılımın çalışması açısından bir bir sorun çıkmaz. Kütüphane işlevlerinin makro eşdeğerlerinin argümanları işlev çağrılarındaki gibi değerlendirilir. İşlev tanımları yerine bu makro tanımlarının yapılmasının sebebi, makrolar satıriçi sembolik dile özgü yorumlar şeklinde üretebildiğinden, bir işlevden daha hızlı çalışabilmeleridir.

Bir kitaplık işlevinin adresini almak, bu bir makro olarak tanımlanmış olsa bile çalışır. Bu bağlam içerisinde çalışmasının sebebi, işlevin adından sonra, bir makro çağrısını tanımak için gereken sol parantezin olmamasıdır.

Bazan bir makro tanımının bir işlev olarak kullanılmasını istemeyebilirsiniz. Bu, yazılımınızda hata ayıklamayı kolaylaştırır. Bunu yapmanın iki yolu vardır:

- Makro çağrılarına özel olarak, işlev ismini parantez içine alarak bir makro tanımından kurtulabilirsiniz. Bunun çalışmasının sebebi işlev isminin sözdizimsel olarak artık bir makro çağrısı olarak algılanmamasıdır.
- Makro tanımını kaynak kodunuzun içinde **`#undef`** önişlemci deyimi ile (oluşumun açıklamasında aksi belirtilmediği sürece) devredışı bırakabilirsiniz.

Örneğin, `stdlib.h` başlık dosyasında **`abs`** isimli bir işlevin bildirimi olduğunu kabul edelim:

```
extern int abs (int);
```

Ve, **`abs`** için bir makro tanımı olsun:

```
#include stdlib.h
int f (int *i) { return abs (++*i); }
```

Burada **abs**, hem bir makro hem de bir işlevdir. Aşağıdaki örneklerde ise **abs** sadece bir işlevdir, bir makro değildir.

```
#include stdlib.h
int g (int *i) { return (abs) (++*i); }
```

```
#undef abs
int h (int *i) { return abs (++*i); }
```

Makro tanımları, asıl işlevin yaptığını yapmaktan başka bir makro da tanımladığından bu yöntemlere gerçekte hiç ihtiyaç yoktur. Ayrıca, bir makro tanımının kaldırılması yazılımınızın daha yavaş çalışmasına sebep olacaktır.

3.3. Anahtar Sözcükler

Kütüphanedeki veri türü, makro, değişken ve işlevlerin isimlerinden ISO C standardında belirtilenler koşulsuz olarak rezerve edilmiştir. Yazılımınızda bu isimleri yeniden tanımlayamazsınız. Kütüphanedeki diğer isimler ise, onların bildirildiği veya tanımlandığı başlık dosyalarını içerdiği taktirde onlarda bu gruba dahil olur. Bu sınırlamaların çeşitli nedenleri vardır:

- Örneğin, standart `exit` işlevinin yaptığından farklı işler yapan bir `exit` işleviniz varsa, kodunuzu okuyan başkaları, kodunuzu anlamakta çok zorluk çekecektir. Bu durumlardan kaçınırsanız, yazılımınız hem daha kolay anlaşılır olur, hem de modülerliği ve yeniden düzenlenebilirliği artar.
- Yeniden tanımlanan bir işlev onu kullanan başka kütüphane işlevlerinin hatalı çalışmasına sebep olabilir. Yani, yeniden tanımlama mümkün olsaydı diğer işlevler düzgün çalışamayabilecekti.
- Derleyiciden eniyileme yapması istendiğinde, bir işlevin kullanıcı tarafından yeniden tanımlanması mümkün olmadığında, derleyici bu işlevlere eniyileme yapıp yapmayacağına daha iyi karar verebilecektir. Bazı kütüphane oluşumlarında örneğin *argüman sayısı değişken işlevler* (sayfa: 814) ile çalışmada ve *yerel olmayan çıkışlarda* (sayfa: 593) C derleyicisinin bir bölümünün bu işlevlerle bir bütünlük içinde çalışması gerekir. Ayrıca gerçeklenme açısından, derleyici dilin yerleşik parçaları olarak bunlarla daha kolay çalışabilir.

Bu kılavuzda belgelenmiş olan isimlere ek olarak, tek altçizgi (_) ile başlayan tüm harici isimler (genel işlevler ve değişkenler) ile nerede ve nasıl kullanılmış olurlarsa olsunlar iki altçizgi ile veya bir altçizgiden sonra bir büyük harfle başlayan tüm isimler rezerve sözcüklerdir. Kütüphane ve başlık dosyalarında tanımlanan işlevler, değişkenler ve makroların dahili kullanım amaçlı olanları, yazılımcının isim kullanım alanını daraltmamak ve yazılımcının kullanacağı olası isimlerle çakışma olmaması için bu yöntemle seçilmektedir.

Bazı isim sınıfları, C dilinin ve POSIX.1 ortamının gelecekteki geliştirmelerinde kullanılmak üzere ayrılmıştır. Bu isimleri şimdi kullandığınızda bir sorun çıkmayacak olsa da gelecekte C ve POSIX standartları ile çelişebilecektir. Bu nedenle onları şimdiden kullanmamaya başlamanız önerilir.

- Bir büyük **E** harfi ile başlayan, bir sayı veya büyük harf ile devam eden tüm isimler hata kodlarının isimleri olarak ayrılmıştır. Bkz. *Hata Bildirme* (sayfa: 31).
- **is** veya **to** ile başlayan ve küçük harf ile devam eden isimler karakter sınama ve dönüşüm işlevleri için kullanılmak üzere ayrılmıştır. Bkz. *Karakterle Çalışma* (sayfa: 82).
- **LC_** ile başlayan ve büyük harflerden oluşan isimler yerel nitelikleri belirleyen makroların isimleri olarak kullanılabilir diye ayrılmıştır. Bkz. *Yereller ve Uluslararasılaştırma* (sayfa: 164).
- Tüm bilinen matematik işlevlerinin isimleri, **f** veya **l** harfi ile sonlandırılmış olarak **float** ve **long double** argümanlarla kullanılmak üzere işlev isimleri olarak ayrılmıştır. Bkz. *Matematik* (sayfa: 475).

- **SIG** ile başlayan ve büyük harflerden oluşan tüm isimler sinyal isimleri olarak ayrılmıştır. Bkz. *Standart Sinyaller* (sayfa: 604).
- **SIG_** ile başlayan ve büyük harflerden oluşan tüm isimler sinyal eylemlerinin isimleri olarak ayrılmıştır. Bkz. *Basit Sinyal İşleme* (sayfa: 611).
- **str**, **mem** veya **wcs** ile başlayan ve küçük harflerden oluşan isimler dizi ve dizge işlevlerinin isimleri olarak ayrılmıştır. Bkz. *Diziler ve Dizgeler* (sayfa: 90).
- **_t** ile biten isimler veri türlerinin isimleri için ayrılmıştır.

Bunlara ek olarak bazı başlık dosyalarının kullanımına bağlı olarak o başlık dosyasına özel bazı isimler rezerve edilmştir. Bu başlık dosyalarını kullandığınızda bu isimleri kullanmamaya çalışmalısınız.

- **d_** ile başlayan isimler **dirent.h** başlık dosyası ile rezervedir.
- **l_**, **F_**, **O_** ve **S_** ile başlayan isimler **fcntl.h** başlık dosyası ile rezervedir.
- **gr_** ile başlayan isimler **grp.h** başlık dosyası ile rezervedir.
- **_MAX** ile biten isimler **limits.h** başlık dosyası ile rezervedir.
- **pw_** ile başlayan isimler **pwd.h** başlık dosyası ile rezervedir.
- **sa_** ve **SA_** ile başlayan isimler **signal.h** başlık dosyası ile rezervedir.
- **st_** ve **S_** ile başlayan isimler **sys/stat.h** başlık dosyası ile rezervedir.
- **tms_** ile başlayan isimler **sys/times.h** başlık dosyası ile rezervedir.
- **c_**, **V**, **I**, **O** ve **TC** ile başlayan isimler ile **B** ile başlayıp bir rakam ile devam eden isimler **termios.h** başlık dosyası ile rezervedir.

3.4. Özellik Sınama Makroları

Bir kaynak dosyasını derlerken kullanabileceğiniz özelliklerden istediklerinizi ***özellik sınama makroları***nı tanımlayarak kontrol edebilirsiniz.

Yazılımınızı **gcc -ansi**[1] kullanarak derlediğinizde, bir veya birkaç özellik makrosu tanımlamamışsanız sadece ISO C kütüphanesinin özelliklerini elde edebilirsiniz.

Bu makroları kaynak kod dosyalarınızın en tepesinde **#define** önişlemci deyimini kullanarak tanımlayabilirsiniz. Bu deyimler, **#include** deyimlerinden de *önce olmalıdır*. Daha iyi açıklamak için bu deyimlerden önce sadece açıklamalar bulunabilir diyebiliriz. Bundan başka GCC'nin `-D` seçeneği ile de bu makrolar kullanılabilirse de makroların kaynak dosyaların kendisinde tanımlanması daha iyidir.

Bu sistem çok sayıda standardı destekleyen kütüphane oluşturmayı mümkün kılar. Farklı standartlar çoğunlukla bir başka standardın üzerine birşeyler ekler ve birbirileriyle de uyumsuzlardır. Geniş kapsamlı standartların gerektirdiği işlev isimleri kullanıcıya kalan isim alanını da küçültür. Bu ukalalıktan öte, pratikte sorunlar oluşturur. Örneğin, bazı GNU dışı yazılımlar, bu kütüphanede bulunan `getline` işlevinin yaptığından tamamen farklı işlemler yürüten bir `getline` işlevi içerebilir ve bunlar eğer tüm özellikleriyle ayrım yapmaksızın etkin kılınırsa uyumluluk sağlanamaz.

Bu durumdan, bir yazılım ancak sınırlı sayıda standarda uygun olabilir sonucu çıkarılmamalıdır. Üzerinde ayrım yapılamayan geniş standartlar uyumluluk için yetersizdir. Standart dışı başlık dosyalarının içerilmesinden ya da standart içinde tanımlanmamış özelliklerin katıştırılmasından sizi koruyamaz.

_POSIX_SOURCE makro

Bu makroyu **#define** ile belirtirseniz, ISO C özelliklerine ek olarak POSIX.1 standardının (IEEE Standard 1003.1) işlevselliğine de sahip olunur.

Bir pozitif tamsayıyı **_POSIX_C_SOURCE** makrosuna atarsanız **_POSIX_SOURCE** etkisiz olacaktır.

_POSIX_C_SOURCE makro

Bu makroya atayacağınız pozitif tamsayılarla hangi POSIX özelliklerinin etkin olacağını belirleyebilirsiniz. Daha büyük değerler daha büyük işlevsellik sağlar.

Bu makroya **1** ya da daha büyük bir tamsayı atarsanız, POSIX.1 standardının 1990 sürümündeki (IEEE Standard 1003.1–1990) işlevselliği elde edersiniz.

Bu makroya **2** ya da daha büyük bir tamsayı atarsanız, POSIX.2 standardının 1992 sürümündeki (IEEE Standard 1003.2–1992) işlevselliği elde edersiniz.

Bu makroya **199309L** ya da daha büyük bir tamsayı atarsanız, POSIX.1b standardının 1993 sürümündeki (IEEE Standard 1003.1b–1993) işlevselliği elde edersiniz.

Daha büyük değerler ise geleceğe yönelik genişletmeleri etkin kılacaktır. POSIX standardının gelişim sürecinde bu değerler tanımlandıkça, GNU C kütüphanesi onlar standart haline geldikçe destekleyecektir. **_POSIX_C_SOURCE**'a **199506L** ya da daha büyük bir tamsayı atarsanız, POSIX.1 standardının 1996 sürümündeki (ISO/IEC 9945–1: 1996) işlevselliği elde edersiniz.

_BSD_SOURCE makro

Bu makro belirtilirse, 4.3 BSD Unix'den türetilmiş işlevselliğe ek olarak ISO C, POSIX.1 ve POSIX.2 işlevselliği elde edilir.

4.3 BSD Unix'den türetilmiş bazı oluşumlar POSIX.1 standardındaki bazı özelliklerle çelişir. Bu makro kullanıldığında 4.3 BSD tanımlamaları POSIX.1 tanımlamalarına göre öncelikli olur.

4.3 BSD ile POSIX.1 arasındaki bazı çakışmalardan dolayı, BSD uyumluluğu ile derlediğiniz yazılımlarınızı ilintilerken özel ***BSD uyumluluk kütüphanesi*** kullanmanız gerekebilir. Bu farkları giderebilmek için bazı işlevlerin iki farklı yolla tanımlanmaları gerekir. Biri normal C kütüphanesi, diğeri uyumluluk kütüphanesidir. Yazılımınızda **_BSD_SOURCE** makrosunu kullanıyorsanız, `-lbsd-compat` seçeneği ile derlemelisiniz. Bu derleyici veya ilintileyiciye, işlevleri normal C kütüphanesinde değil, uyumluluk kütüphanesinde araması gerektiğini söyler.

_SVID_SOURCE makro

Bu makro belirtilirse, SVID'den türetilmiş işlevselliğe ek olarak ISO C, POSIX.1, POSIX.2 ve X/Open işlevselliği de elde edilir.

_XOPEN_SOURCE makro
_XOPEN_SOURCE_EXTENDED makro

Bu makro belirtilirse, X/Open Taşınabilirlik Kılavuzunda açıklanan işlevsellik elde edilir. Bu POSIX.1 ve POSIX.2 işlevselliğinin üstünde bir genişletme içerdiğinden **_POSIX_SOURCE** ve **_POSIX_C_SOURCE** kendiliğinden tanımlanmış olur.

Tüm Unixlerin aynı olması ilkesinden hareketle, sadece BSD ve SVID'de geçerli işlevselliği de içerir.

Eğer **_XOPEN_SOURCE_EXTENDED** makrosu da ayrıca belirtilmişse, fazladan bir işlevsellik eklenir. Bu fazladan işlevler, X/Open Unix ticari sürümünün gerektirdiği tüm işlevleri sağlayacaktır.

_XOPEN_SOURCE makrosuna 500 değeri atanırsa, mevcut işlevselliğe ek olarak Single Unix Specification, version 2 içindeki yeni tanımlar da içerilecektir.

_LARGEFILE_SOURCE makro

Bu makro belirtilirse tüm önceki standarlardaki bazı yetersizlikleri düzelten ek işlevler tanımlanmış olur. Özellikle, **fseeko** ve **ftello** işlevlerini kullanmak içindir. Bu işlevler olmaksızın ISO C arayüzü (**fseek**, **ftell**) ile düşük seviyeli POSIX arayüzü (**lseek**) arasında bazı sorunlar çıkacaktır.

Bu makro, Büyük Dosya Desteğinin (LFS – Large File Support) bir parçası olarak kütüphaneye dahil edilmiştir.

_LARGEFILE64_SOURCE makro

Bu makroyu belirtirseniz, içerdiği ek işlevlerle 32 bitlik sistemlerdeki 2GB'lık dosya büyüklüğü sınırı aşılabilir. Bu arayüz, büyük dosyaları desteklemeyen sistemlerde kullanılamaz. Doğal dosya büyüklüğü 2GB'dan büyük olan sistemlerde (örn, 64 bitlik sistemler) ise yeni işlevler, mevcutlarla eşdeğerdedir.

Bu yeni işlevsellik mevcut olan türler ve işlevlerle yer değiştirilerek kullanılır. Bu yeni nesnelerin isimlerinde maksadını ifade edecek şekilde **64** bulunur. Örneğin, `off_t` yerine `off64_t`, `fseeko` yerine `fseeko64` gibi.

Bu makro, Büyük Dosya Desteğinin (LFS – Large File Support) bir parçası olarak kütüphaneye dahil edilmiştir. 64 bitlik kullanım halen genele yansımadığından bu bir geçiş dönemi arayüzüdür. (Aşağıdaki **_FILE_OFFSET_BITS** makrosunun açıklamalarına bakınız).

_FILE_OFFSET_BITS makro

Bu makro sistemde hangi dosya sistemi arayüzünün kullanılacağını belirlemekte kullanılır. **_LARGEFILE64_SOURCE**, bir ek arayüz olarak 64 bitlik arayüzü etkin kılar. **_FILE_OFFSET_BITS** ise 64 bitlik arayüzün eski arayüzle değiştirilmesini mümkün kılar.

_FILE_OFFSET_BITS belirtilmemişse veya **32** değeri ile belirtilmişse hiçbir etkisi olmaz. 32 bitlik arayüz, 32 bitlik sistemlerde 32 bitlik `off_t` türü ile kullanılmaya devam eder.

Makro, **64** değeri ile belirtilmişse, büyük dosya arayüzü eski arayüzle değiştirilir. Yani işlevler farklı isimler altında olmaz, eski işlev isimleri yenileri için geçerli olur. Siz `fseeko` işlevini çağırdığınızda aslında `fseeko64` işlevini kullanmış olursunuz.

Bu makro sadece, sistem büyük dosyalarla çalışabileceğiniz bir mekanizma sağlıyorsa belirtilebilir. 64 bitlik sistemlerde bu makro etkisizdir. ***64** isimli işlevler normal işlevlerle eşdeğerdir.

Bu makro, Büyük Dosya Desteğinin (LFS – Large File Support) bir parçası olarak kütüphaneye dahil edilmiştir.

_ISOC99_SOURCE makro

ISO C standardı yeniden gözden geçirilip düzeltilinceye kadar yeni özellikler geniş uygulama alanı bulsalar bile özdevinimli olarak etkinleştirilmez. Buna rağmen GNU libc yeni standardın tam bir gerçeklemesine sahiptir ve bu yeni özellikler **_ISOC99_SOURCE** makrosu ile etkinleştirilebilir.

_GNU_SOURCE makro

Bu makro belirtildiğinde herşey etkin olur: SO C89, ISO C99, POSIX.1, POSIX.2, BSD, SVID, X/Open, LFS ve GNU oluşumları. Bu durumda POSIX.1 ile BSD arasındaki çakışmalar için öncelik POSIX.1'dedir.

_GNU_SOURCE makrosunun tam etkili ancak BSD'nin öncelik almasını isterseniz, bunu makroların belirtilme sırasını aşağıdaki gibi düzenleyerek yapmalısınız:

```
#define _GNU_SOURCE
#define _BSD_SOURCE
#define _SVID_SOURCE
```

Bunu yaparsanız, derleyici ya da ilintileyiciye **-lbsd-compat** seçeneğini vererek yazılımınızın BSD uyumluluk kütüphanesi ile ilintilenmesini sağlamalısınız.

Bilgi

Bunu yapmayı unutursanız, yazılımın çalışması sırasında çok tuhaf hatalarla karşılaşabilirsiniz.

_REENTRANT	makro
_THREAD_SAFE	makro

Bu makroları belirttiğinizde bazı işlevlerin ***evresel*** (`reentrant`) sürümleri bildirilmiş olur. Bu işlevlerin bazıları POSIX.1c içinde belirtilmişse de bir çoğu GNU libc'ye özeldir, bir kısmı da diğer bazı sistemlerde kullanılmaktadır. Sorun standartlaştırmadaki gecikmeden kaynaklanmaktadır.

Birtakım başka sistemlede ise C kütüphanesinin buna özel bir sürümü yoktur. Sadece tek bir sürüm vardır ancak derleme sırasında bu durumun belirtilmesi gerekir.

Yeni yazacağınız yazılımlarda **_GNU_SOURCE** kullanmanızı öneririz. GCC'ye `-ansi` seçeneğini belirtmez ve bu makrolardan hiçbirini kaynak kodunuzda belirtmezseniz bunun etkisi **_POSIX_C_SOURCE** makrosunun 2 ile, **_POSIX_SOURCE**, **_SVID_SOURCE** ve **_BSD_SOURCE** makrolarını 1 ile atamakla eşdeğerdedir.

Daha geniş özellikler içeren bir özellik sınama makrosu ile birlikte bu makronun kapsamında olan makrolardan birini ayrıca belirtmenin bir etkisi yoktur. Örneğin, **_POSIX_C_SOURCE** makrosundan sonra **_POSIX_SOURCE** makrosunun belirtilmesinin bir etkisi olmayacaktır. Benzer şekilde **_GNU_SOURCE** makrosundan sonra belirtilen **_POSIX_SOURCE** veya **_POSIX_C_SOURCE** ya da **_SVID_SOURCE** makrolarının bir etkisi olmaz.

Yalnız, **_BSD_SOURCE** makrosu hiçbir makronun kapsamında değildir. Bu, BSD ile POSIX arasındaki bazı çakışmalardan dolayı hangisinin öncelik alacağını belirleyebilmek bakımından böyledir. Diğer özellik sınama makrolarına ek olarak belirtilen **_BSD_SOURCE** bir etki oluşturacaktır: POSIX ile çakışan özellikler için BSD özellikleri öncelikli olacaktır.

4. Kılavuzun Yol Haritası

Bu bölümde bu kılavuzun devamındaki kısımların içeriği hakkında kısaca bilgi verilmiştir.

- *Hata Bildirme* (sayfa: 31) kısmında kütüphane tarafından raporlanan hataların nasıl saptandığı açıklanmıştır.
- *Kütüphanedeki C Dili Oluşumları* (sayfa: 813) kısmında C dilinin standart parçaları için kütüphanedeki destek hakkında bilgiler bulunmaktadır. Bunlar, **sizeof** işleci, **NULL** sembolik sabiti gibi şeyler ile argüman sayısı değişken işlevlerin nasıl yazıldığı, sayısal türlerin aralıkları ve çeşitli özellikleri gibi konulardır. Ayrıca kodunuza basit tuzaklar koyarak, sınama sonucuna göre uyarı iletileri alabileceğiniz basit bir hata ayıklama mekanizmasından da bahsedilmiştir.
- *Sanal Bellek Ayırma ve Sayfalama* (sayfa: 47) kısmında sanal belleğin özdevimli ayrılması de dahil sanal ve gerçek belleğin kullanımı ve yönetimi anlatılmıştır. Yazılımınızın baştan ne kadar belleğe ihtiyacı olacağını bilemiyorsanız belleği gerektikçe özdevimli ayırabilir ve göstericiler üzerinden yönetebilirsiniz.
- *Karakterle Çalışma* (sayfa: 82) kısmı karakter sınıflandırma işlevleri (`isspace` gibi) ile harf büyüklüğü dönüşümü yapan işlevler hakkında bilgi içerir.
- *Diziler ve Dizgeler* (sayfa: 90) kısmında, dizgeler (boş sonlandırmalı karakter dizileri), genel amaçlı bayt dizileri işlevleri ile kopyalama ve karşılaştıma gibi amaçlara uygun işlevler hakkında bilgi verilmiştir.
- *Girdi/Çıktı İşlemlerine Giriş* (sayfa: 231) kısmı kütüphanedeki girdi ve çıktı oluşumlarına kapsamlı bir bakış ile dosya isimleri gibi genel kavramlar hakkında bilgiler içermektedir.
- *Akımlar Üzerinde Giriş/Çıkış* (sayfa: 236) kısmında akımlarla (veya **FILE *** nesneleri) ilgili giriş/çıkış işlemleri anlatılmıştır. Bunlar `stdio.h` dosyasındaki normal C kütüphanesi işlevleridir.

- *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305) kısmı dosya tanımlayıcılar üzerindeki giriş/çıkış işlemleri hakkında bilgi içerir. Dosya tanıtıcılar Unix ailesi işletim sistemlerine özel bir düşük seviyeli mekanizmadır.
- *Dosya Sistemi Arayüzü* (sayfa: 351) kısmında dosyaların silinmesi, isimlerinin değiştirilmesi, yeni dizin oluturulması gibi dosyalar üzerindeki işlemler anlatılmıştır. Bu kısım ayrıca dosyaların sahipleri, dosya koruma kipleri gibi dosya özelliklerine nasıl erişildiği hakkında bilgileri de içerir.
- *Borular ve FIFOlar* (sayfa: 393) kısmı süreçler arası basit iletişim mekanizması hakkında bilgiler içerir. Boruhatları, biri diğerinin çocuğu olan iki süreç arasındaki iletişim için kullanılırken FIFO'lar aynı makina üzerindeki ortak bir dosya sistemi paylaşan süreçler arasındaki iletişim için kullanılır.
- *Soketler* (sayfa: 398) kısmı bir ağ üzerinden haberleşen farklı makinalar üzerinde çalışan süreçler arasındaki daha karmaşık bir süreçler arası iletişim mekanizması hakkında bilgiler içerir. Bu kısımda ayrıca internet konak adreslemesi ile sistemin ağ veritabanının nasıl kullanıldığı da anlatılmıştır.
- *Düşük Seviyeli Uçbirim Arayüzü* (sayfa: 442) kısmında bir uçbirim aygıtının özniteliklerini nasıl değiştirebileceğiniz anlatılmıştır. Örneğin bir kullanıcının yazdığı karakterlerin görüntülenmesini engellemek isterseniz bu kısmı okuyun.
- *Matematik* (sayfa: 475) kısmı matematik işlevleri kütüphanesi hakkındadır. Gerçek sayılarla trigonometrik veya üstel işlevler, tamsayılarda bölümden kalan işlevleri ve rasgele sayı üreteçleri gibi konulara değinilmiştir.
- *Aritmetik İşlevleri* (sayfa: 506) kısmında basit aritmetik işlemler, gerçek sayılarla analiz ve dizgelerdeki sayıların okunması için işlevler anlatılmıştır.
- *Arama ve Sıralama* (sayfa: 203) kısmı, dizilerdeki değerlerin sıralanması ve aranması için karşılaştırma işlevleri hakkında bilgi içerir.
- *Şablon Eşleme* (sayfa: 212) kısmında düzenli ifadeler ve kabuk dosya isim kalıpları ile kabukta olduğu gibi sözcüklerin yorumlanması için işlevlere yer verilmiştir.
- *Tarih ve Zaman* (sayfa: 538) kısmında, takvim ve işlemci zamanının ölçümleri ile zamanlayıcıların ayarlanması ile ilgili işlevler anlatılmıştır.
- *Karakter Kümeleriyle Çalışma* (sayfa: 126) kısmında, **`char`** veri türü ile ifade edilemeyen karakter kümelerini kullanarak karakterler ve dizgeler üzerinde nasıl işlem yapılacağı açıklanmıştır.
- *Yereller ve Uluslararasılaştırma* (sayfa: 164) kısmında, kütüphanenin davranışını etkileyen ülke ve dil seçimi ile ilgili bilgiler bulunmaktadır. Örneğin harflerin sırası hangi karakter hangisinin büyüğü ya da küçüğüdür, parasal değerler ve tarih dizgeleri nasıl biçimlenir gibi bilgiler bu kısımdadır.
- *Yerel Olmayan Çıkışlar* (sayfa: 593) kısmında, `setjmp` ve `longjmp` işlevleri anlatılmıştır. Bu işlevlerle **`goto`** deyiminin yaptığı gibi bir işlevden diğerine atlamak için kullanılan atlama işlevleridir.
- *Sinyal İşleme* (sayfa: 601) kısmı sinyallar hakkındadır. Sinyal nedir, bir sinyal alındığında neler yapılabilir ve yazılımınızın önemli bir bölümü çalışırken alınan sinyallerin engellenmesi gibi bilgiler içerir.
- *Temel Yazılım ve Sistem Arayüzü* (sayfa: 644) kısmında yazılımınızın komut satırı seçeneklerine ve ortam değişkenlerine nasıl erişebileceğiniz anlatılmıştır.
- *Süreçler* (sayfa: 685) kısmı, yeni süreçlerin başlatılması ve uygulamaların çalıştırılması hakkında bilgi içerir.
- *İş Denetimi* (sayfa: 716) kısmında, uçbirimin denetimi ve süreç grupları üzerindeki işlemler için işlevler açıklanmıştır. Bu işlevler, kabuk yazmakla ilgileniyorsanız özellikle iş denetimi konusundadır.
- *Sistem Veritabanları ve İsim Hizmetleri Seçimi* (sayfa: 733) kısmı, sistem veritabanındaki isimlerde arama, hangi veritabanı için hangi hizmetin kullanıldığının saptanması ve bu hizmetlerin nasıl gerçekleştirildiği gibi bilgiler içerir. Böylece kendi hizmet yordamlarınızı oluşturabilirsiniz.
- *Kullanıcı Veritabanı* (sayfa: 760) ve *Grup Veritabanı* (sayfa: 762), kısımlarında kullanıcı ve grup veritabanlarına nasıl erişebileceğiniz anlatılmıştır.

- *Sistem Yönetimi* (sayfa: 769) kısmında, yazılımınızın altında çalıştığı donanım ve yazılımlar hakkındaki bilgilerin alınması ve denetimini sağlayan işlevler anlatılmıştır.
- *Sistem Yapılandırma Parametreleri* (sayfa: 784) kısmında, çeşitli işletim sistemlerinin sınırları hakkında nasıl bilgi edinileceği anlatılmıştır. Bu parametrelerin çoğu POSIX ile uyumluluk içindir.
- *Kütüphane Oluşumlarının Özeti* (sayfa: 827) kısmında kütüphanenin içerdiği tüm işlevler, değişkenler ve makrolar ile veri türleri ve işlev prototipleri hakkında bunların hangi standarda uyumlu olduğu ve hangi sistemden türetildiği gibi bilgiler özet halinde verilmiştir.
- *Kütüphanenin Sürdürülmesi* (sayfa: 956) kısmında, GNU C kütüphanesinin sisteminize kurulması açıklanmıştır. Ayrıca bulduğunuz hataların nasıl raporlanacağı, yeni bir sisteme kütüphanenin nasıl taşınacağı, kütüphaneye yeni işlevlerin nasıl ekleneceği gibi bilgilere de yer verilmiştir.

Kullanmak istediğiniz kütüphane oluşumunun ismini biliyorsanız doğrudan *Kütüphane Oluşumlarının Özeti* (sayfa: 827) kısmına bakın. Bu kısım oluşum hakkında özet bir bilgi verdiği gibi oluşum hakkında daha ayrıntılı bilgiyi nerede bulabileceğiniz bilgisini de içerir. Bu ek bölüm ayrıca, örneğin, bir işlevin argümanlarının türleri ve sırasını bilmek istediğinizde faydalıdır. Ayrıca her işlev, değişken veya makronun türetildiği sistem veya standardın ne olduğunu da bu kısımdan öğrenebilirsiniz.

II. Hata Bildirme

İçindekiler

GNU C kütüphanesindeki çoğu işlev hata durumlarını saptar ve bildirir, siz de bu hata durumlarına göre yazılımınızın davranışlarını düzenleyebilirsiniz. Örneğin, bir girdi dosyasını açarken, dosyanın gerçekte açılıp açılmadığına bakmalı ve eğer kütüphane işlevinin çağrısı başarısız olmuşsa bir hata iletisi basmalı başarılı ise sıradaki eylemi gerçekleştirmelisiniz.

Bu kısımda hata bildirme mekanizmasının nasıl çalıştığı anlatılmıştır. Bu mekanizmayı kullanabilmek için yazılımınız `errno.h`(B65) dosyasını içermelidir.

1. Hata Denetimi

Kütüphane işlevlerinin çoğu başarısız olduğu durumlarda özel bir değer döndürür. Bu özel değer genellikle ya **-1** veya bir boş gösterici ya da bu amaç için tanımlanmış **EOF** gibi bir sabitttir. Bu geridönüş değeri size sadece bir hatanın oluştuğu bilgisini verir. Hatanın çeşidini bulmak için **errno** değişkeninde saklanan hata koduna bakmanız gerekir. Bu değişken `errno.h` başlık dosyasında bildirilmiştir.

`volatile int` **errno** — değişken

errno değişkeni sistem hata numarasını içerir. Siz **errno** değişkeninin değerini değiştirebilirsiniz.

errno değişkeni **volatile** olarak bildirildiğinden bir *sinyal tuzağı* (sayfa: 617) tarafından herhangi bir anda değiştirilebilir. Doğru yazılmış bir sinyal tuzağı **errno** değişkeninin değerini saklayabildiğinden ya da eski değerine döndürebildiğinden, sinyal tuzaklarını yazmak dışında bu konuda kaygılanmanız gerekmez.

errno değişkeni yazılım ilklendirildiğinde sıfır değerine sahiptir. Birçok kütüphane işlevi bir hata oluştuğunda sıfırdan farklı bir değerle dönmeyi garanti eder. Bu hata durumları her işlev için listelenmiştir. Bu işlevler başarılı olduklarında **errno** değişkeninin değerini değiştirmezler. Bu nedenle, başarılı bir işlev çağrısı sonucunda **errno** değişkeninin değeri sıfır olmayabileceğinden, bir işlev çağrısının başarılı olup olmadığını **errno** değişkeninin değerine bakarak saptamaya çalışmamalısınız. Bunu yapmanın doğru yolu her işlev için belgelendirilmiştir. *Eğer* çağrı başarısızsa **errno** değişkeninin değerine bakabilirsiniz.

Birçok kütüphane işlevi, çağırdığı bir başka kütüphane işlevinin başarısızlığı durumunda **errno** değişkeninin değerini sıfırdan farklı bir değere ayarlayabilir. Kütüphane işlevlerinin, işlev bir hata geri döndürdüğünde **errno** değişkeninin değerini değiştirdiği kabulune göre hareket etmelisiniz.

Uyumluluk bilgisi

ISO C, bir makro olarak gerçekleştirilebileceğinden **errno**'nun bir değişken değil bir "değiştirilebilir sol taraf değeri" olduğunu belirtir. Örneğin, GNU sisteminde olduğu gibi ***__errno ()** benzeri bir işlev çağrısı ile ilişkilendirilmiş olabilir. Bu nedenle GNU kütüphanesi, GNU dışı sistemlerde de doğru sonuçlar verir.

sqrt ve **atan** gibi bazı kütüphane işlevleri bir hata durumunda **errno** değerini de ayarladıklarından kusursuz doğrulukta bir değer döndürürler. Bu gibi işlevler için, işlev çağrısından önce **errno** değişkenine sıfır değerini atamalı ve sonrasında **errno** değişkeninin değerine bakmalısınız.

Tüm hata kodları için `errno.h`[B67] başlık dosyasında birer makro olarak tanımlanmış birer sembolik isim vardır. Bu isimler daima bir **E** harfi ile başlar ve bir rakam ya da büyük harf ile devam eder. Bu isimleri birer *anahtar sözcük* (sayfa: 24) gibi ele almalısınız.

Hata kodu değerlerinin hepsi pozitif tamsayılardır ve tamamı farklı değerlere sahiptir; bu ikisi dışında: **EWOULDBLOCK** ve **EAGAIN** aynı değere sahiptir. Değerleri farklı olduğundan hata kodlarını **switch** deyiminde etiket olarak kullanabilirsiniz; ama **EWOULDBLOCK** ve **EAGAIN** isimlerinden sadece birini kullanmalısınız. Yazılımınız bu sembolik sabitlerin özel değerleri dışında bir kabulde bulunmamalıdır.

errno değeri herzaman bu makroların değerlerine karşılık değildir; bazı kütüphane işlevleri kendi özel durumlarına özgü hata kodları döndürebilir. Belli bir kütüphane işlevi için anlamlı olabilen bu değerleri içeren işlevler kılavuzda belirtilmiştir.

GNU dışı sistemlerin hemen hepsinde bir sistem çağrısı, argüman olarak bir geçersiz gösterici ile çağrılmışsa **EFAULT** ile dönebilir. Bu sadece yazılımınızdaki bir yazılım hatasının sonucu olarak görülebileceğinden ve "GNU sistemlerinde asla bu hatayı göremeyeceğinizden" işlevlerin açıklamalarında **EFAULT** hatasına hiç yer verilmemiş ama bu değer kütüphaneye konmuştur.

Bazı Unix sistemlerinde birçok sistem çağrısı, bir argüman olarak verilmiş yığıt ya da çekirdek içindeki bir göstericinin, yığıtı genişletmeye çalışmak gibi anlaşılmaz bir sebeple başarısız olması durumunda **EFAULT** ile dönebilir. Bu gibi durumların oluşmaması için sistemdeki yığıt belleği yerine durağan ya da özdevimli bellek ayırmaya çalışmalısınız.

2. Hata Kodları

Hata kodu makroları `errno.h` başlık dosyasında tanımlıdır. Hepsi tamsayı sabitlerle sonuçlanır. Bu hata kodlarının GNU sisteminde oluşmayanları için yani GNU sistemi dışındaki sistemler için olanları GNU kütüphanesinde bulunmaktadır.

`int` **EPERM**
: İşleme izin verilmedi; sadece dosyanın (ya da diğer özkaynağın) ya da süreçlerin sahibinin yapabileceği işlemler.

`int` **ENOENT**
: Böyle bir dosya ya da dizin yok. Sistemde bulunacağı varsayılmış bir dosya ya da dizinin bulunamadığı durum.

`int` **ESRCH**
: Belirtilen süreç kimliği ile eşleşen bir süreç yok.

`int` **EINTR**
: İşlev çağrısı engellendi; herhangi bir anda çağrının tamamlanmasını engelleyen bir sinyalin oluştuğu durum. Bu durum oluştuğunda çağrıyı tekrarlamalısınız.

  EINTR ile başarısızlık dışında yakalanan tüm sinyaller için işlevlerin tekrar devreye girmesini sağlayabilirsiniz. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

`int` **EIO**
: Giriş/Çıkış hatası; genellikle fiziksel okuma ve yazma hataları için kullanılır.

`int` **ENXIO**
: Böyle bir adres ya da aygıt yok. Sistem bir dosya olarak belirttiğiniz bir aygıtı kullanmaya çalışır ve aygıtı bulamazsa bu hata oluşur. Bu hatayı alıyorsanız ya aygıt dosyası doğru olarak oluşturulmamış veya fiziksel aygıt bulunamamış ya da makinaya doğru bağlanmamıştır.

`int` **E2BIG**

Argüman listesi çok uzun; *exec* (sayfa: 688) işlevlerinden biri ile çalıştırılan yeni bir uygulama argümanlarla çağrıldığında çok fazla bellek alanı ayrılmaya çalışılırsa oluşur. GNU sistemlerinde bu durum asla oluşmaz.

int **ENOEXEC**
Çalıştırılabilir dosya biçimi geçersiz. Bu durum *exec* (sayfa: 688) işlevleriyle saptanır.

int **EBADF**
Dosya tanımlayıcı hatalı; Örneğin dosya tanımlayıcının açılmamış olduğu ya da sadece yazmak için açılmış bir tanımlayıcıdan okuma yapıldığı gibi durumlarda oluşur.

int **ECHILD**
Hiç alt süreç yok. Alt süreçlerle çalışılması durumunda, üzerinde çalışılacak bir alt süreç yoksa oluşur.

int **EDEADLK**
Kısırdöngü önlendi. Bir sistem özkaynağının ayrılması sırasında bu özkaynağın başka bir özkaynağın varlığına bağlı olması ve o özkaynağında ayrılacak özkaynağa bağlı olması gibi bir durumda her iki özkaynağın bir diğerini beklemesi gibi bir durumdur. Sistem bu gibi durumlarda durumu bildirmeyi garanti etmez. Eğer bu hatayı almışsanız, sistem tam çökeceği sırada sizi uyarabilmiş ve size bir şans verebilmiş demektir... Örnekler için *Dosya Kilitleri* (sayfa: 346) kısmına bakınız.

int **ENOMEM**
Yeterli bellek yok. Sistem kapasitesini doldurduğu için sanal bellek ayıramıyordur.

int **EACCES**
İzinler yetersiz; dosya izinlerinin yapılmaya çalışılan işlem için yetersiz olduğu durum.

int **EFAULT**
Hatalı adres; geçersiz bir göstericinin saptandığı durum. GNU sistemlerinde bu hatayı asla almazsınız, onun yerine bir sinyal alırsınız.

int **ENOTBLK**
Bir blok özellikli dosya gereken yerde verilen dosya uygun değilse oluşur. Örneğin, Unix sistemlerinde normal bir dosyayı bir dosya sistemi olarak sisteme bağlamaya çalışırsanız bu hatayı alırsınız.

int **EBUSY**
Aygıt ya da özkaynak meşgul; bir sistem kaynağının kullanımda olduğu için paylaşılamadığında oluşur. Örneğin, bir bağlı dosya sisteminin kökünü bir dosya olarak silmeye çalışırsanız bu hatayı alırsınız.

int **EEXIST**
Dosya var. Mevcut bir dosyanın yeni bir dosya olarak belirtildiği durum.

int **EXDEV**
Dosya sistemlerine uygunsuz bir bağ oluşturulmaya çalışıldığı saptandı. Bu durum sadece *link* (sayfa: 364) kullanıldığında bir yandan da *rename* (sayfa: 369) ile dosyanın ismi değiştirilmeye çalışıldığında oluşur.

int **ENODEV**
Belli bir sıraya uygun bir aygıtın verilmesi umulan bir işleve yanlış aygıt türü verildi.

int **ENOTDIR**
Bir dizin gerekliyken belirtilen dosya bir dizin değil.

int **EISDIR**
Dosya bir dizin; bir dizini yazmak için açamaz veya bir dizin belirten bir sabit bağı silemezsiniz.

int **EINVAL**
Geçersiz argüman. Bir kütüphane işlevi yanlış argümanla çağrıldığında oluşan çeşitli sorunlara dikkat çekmek için kullanılır.

int **EMFILE**
Mevcut süreç çok fazla dosya açmış ve daha fazlasını açamaz. Tanımlayıcıların tekrarlanması bu sınıra yaklaşılmasına sebep olur.

BSD ve GNU sistemlerinde açık dosyaların sayısı gerektiğinde arttırılabilmesini sağlayan bir özkaynak sınırıyla denetlenir. Bu hatayı alırsanız ya **RLIMIT_NOFILE** sınırını arttırın ya da sınırsız yapın. Bkz. *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

int **ENFILE**
Sistemin bütününde çok fazla farklı dosya açılışı var. Bilgi: Birbirlerine bağlı kanallar tek bir dosya açılışı sayılır; bkz. *İlintili Kanallar* (sayfa: 316). Bu hata GNU sistemlerinde hiçbir zaman oluşmaz.

int **ENOTTY**
İlgisiz G/Ç denetimi işlemi. Örneğin, uçbirim kiplerinin normal bir dosya için kullanılmaya çalışılması.

int **ETXTBSY**
Çalışmakta olan bir dosyaya yazma denemesi ya da yazmak için açılmış bir dosyanın çalıştırılmasının denenmesi. Çoğunlukla bir hata ayıklama uygulaması ile çalışmakta olan bir yazılımın dosyasının yazmak için açılmaya çalışılması bu hatanın ortaya çıkmasına sebep olur. Sembolik sabitin ismi "text file busy" (metin dosyası meşgul) tümcesinden kısaltılmıştır. Bu hata GNU sistemlerinde oluşmaz çünkü gerekirse metin kopyalanır.

int **EFBIG**
Dosya çok büyük; dosyanın uzunluğunun sistemde mümkün olan dosya uzunluğundan büyük olduğu durum.

int **ENOSPC**
Aygıt üzerinde yer yok; disk doluyken üzerine bir dosyanın yazılmaya çalışılması durumu.

int **ESPIPE**
Konumlama işlemi geçersiz (örneğin bir boruhattında).

int **EROFS**
Salt okunur dosya sisteminde birşeyler değiştirilmeye çalışıldı.

int **EMLINK**
Çok fazla bağ var; tek bir dosyaya çok sayıda bağ varsa, dosyanın ismi bağlarla zaten değiştirilmişse ve `rename` ile dosyanın ismi değiştirilmeye çalışılırsa bu durum ortaya çıkabilir. Bkz. *Dosya İsimlerinin Değiştirilmesi* (sayfa: 369).

int **EPIPE**
Kırık boruhattı; bir boruhattının diğer ucunda okuyacak bir sürecin olmaması durumu. Her kütüphane işlevi bu hatayla dönebilir ve ayrıca bir **SIGPIPE** sinyali üretir. Bu sinyal yazılım tarafından yakalanmıyor ya da engellenmiyorsa yazılımı sonlandıracaktır. Bu yüzden eğer yazılımınız **SIGPIPE** sinyalini yakalamıyor ya da engellemiyorsa yazılımınız bu hatayı asla almayacak ama sonlanacaktır.

int **EDOM**
Sayısal argüman alan dışı; matematik işlevlerinde bir argüman işlev tarafından tanımlanmış alandan taşarsa bu hata oluşur.

int **ERANGE**

Kapsama hatası; matematik işlevleri tarafından kullanılır. Sonuç, olması gereken değer aralığının altında ya da üstünde ise bu hata oluşur.

`int` **EAGAIN**
Özkaynak geçici olarak kullanımdışı; daha sonra tekrar denerseniz çağrı çalışacaktır. **EWOULDBLOCK** makrosu ile aynıdır ve GNU C kütüphanesinde bu ikisi aynıdır.

Bu hata bir kaç farklı durumda ortaya çıkabilir:

- Engellenmemesi öngörülmüş bir nesne üzerinde nesneyi engelleyecek bir işlem yürütülmeye çalışılırsa bu hata alınabilir. Bir işlemin tekrarlanması, bazı dış koşullar nesnenin okuma, yazma veya bağlanma için hazır hale gelmesini sağlayana dek engellenecektir. İşlemin mümkün olacağı zamanı *select* (sayfa: 323) işlevini kullanarak bulabilirsiniz.

  

  Taşınabilirlik Bilgisi:

  Eski Unix sistemlerinin çoğunda, bu durum **EWOULDBLOCK** ile **EAGAIN** hatakodundan farklı bir durum olarak ele alınmıştır. Yazılımınızın taşınabilirliği açısından her iki hata koduna da bakmalı ve onları aynı olarak değerlendirmelisiniz.

- Bir geçici özkaynak yokluğu bir işlemi imkansız yapabilir. `fork` işlevi bu hatayı döndürebilir. Varlığı umulanın geçici olarak yokluğunu ifade eder. Bu durumda yazılımınız işlemi tekrarlayabilir ve işlemi gerçekleştirebilir. Şüphesiz tekrar denemeden önce bir kaç saniye beklemek ve az bulunan özkaynakları diğer süreçlerin serbest bırakmasına izin vermek daha iyidir. Bu tür kıtlık durumlarına sıkça rastlanır ve tüm sistemi etkiler. Böyle bir durumda etkileşimli bir yazılım kullanıcıya bilgi verip kendi komut döngüsüne geri dönmelidir.

`int` **EWOULDBLOCK**
GNU C kütüphanesinde bu hata**EAGAIN** (yukarıda) ile aynıdır. Değerler bütün işletim sistemlerinde daima aynıdır.

Eski Unix sistemlerinin çoğunda, C kütüphanelerinde **EWOULDBLOCK**, **EAGAIN** hata kodundan farklıdır.

`int` **EINPROGRESS**
Engellenmemesi öngörülmüş bir nesne üzerinde başlatılmış bir işlem tamamlanmadı. Daima bloklanması gereken bazı işlevler (örn, *connect* (sayfa: 422)) hiçbir zaman **EAGAIN** hatası vermez. Onun yerine, bir işlemin başlamış olduğunu ve tamamlanmasının biraz vakit alacağını belirten **EINPROGRESS** hatası döner. Çağrı tamamlanmadan önce nesneyi değiştirmeye çalışmak ise **EALREADY** hatasına yolaçar. Kuyruktaki işlemin ne zaman tamamlanacağını bulmak için *select* (sayfa: 323) işlevini kullanabilirsiniz.

`int` **EALREADY**
Engellenmemesi öngörülmüş bir nesne üzerindeki işlem hala sürüyor.

`int` **ENOTSOCK**
Bir soket gerektiği halde belirtilen dosya bir soket değil.

`int` **EMSGSIZE**
Bir sokete gönderilen iletinin uzunluğu desteklenenden fazla.

`int` **EPROTOTYPE**
İstenen protokol bu soket türünü desteklemiyor.

`int` **ENOPROTOOPT**

Belirttiğiniz *soket seçeneği* (sayfa: 438) soket tarafından kullanılmakta olan protokol için uygun değil.

`int` **EPROTONOSUPPORT**
Soket istenen iletişim protokolünü desteklemiyor (istenen protokolün tamamiyle geçersiz olduğu durumda). Bkz. *Bir Soketin Oluşturulması* (sayfa: 420).

`int` **ESOCKTNOSUPPORT**
Soket türü desteklenmiyor.

`int` **EOPNOTSUPP**
İstediğiniz işlem desteklenmiyor. Bazı soket işlevleri tüm soket türleri için uygun değildir ve bunlar tüm iletişim protokolleri için gerçekleştirilmemiş olabilirler. GNU sisteminde bir nesne istenen işlemi desteklemiyorsa birçok çağrı bu hatayı verebilir. Sunucunun bu çağrı için hiçbir şey yapamayacağı genel bir durumu belirtir.

`int` **EPFNOSUPPORT**
İstediğiniz soket iletişim protokolü ailesi desteklenmiyor.

`int` **EAFNOSUPPORT**
Bir soket için belirtilmiş adres ailesi desteklenmiyor; adres ailesinin soket üzerinde kullanılan protokolle bağdaşmadığını belirtir. Bkz. *Soketler* (sayfa: 398).

`int` **EADDRINUSE**
İstenen *soket adresi* (sayfa: 401) kullanımda.

`int` **EADDRNOTAVAIL**
İstenen soket adresi kullanıma uygun değil; örneğin, yerel konak ismi ile eşleşmeyen bir soket ismi verilmeye çalışılması. Bkz. *Soket Adresleri* (sayfa: 401).

`int` **ENETDOWN**
Ağ çökük olduğundan soket işlemi başarısız oldu.

`int` **ENETUNREACH**
Uzak konağı içeren alt ağ erişilemez olduğundan soket işlemi başarısız oldu.

`int` **ENETRESET**
Uzak konak çöktüğünden ağ bağlantısı sıfırlandı.

`int` **ECONNABORTED**
Ağ bağlantısı yerel olarak sonlandırıldı.

`int` **ECONNRESET**
Ağ bağlantısı yerel konağın denetimi dışında kapandı. Örneğin, uzak makinanın yeniden başlatılıyor olması veya önlenemeyen bir protokol saldırısı bu hataya sebep olabilir.

`int` **ENOBUFS**
Çekirdeğin G/Ç tamponlarının hepsi kullanımda. GNU'da bu hata daima **ENOMEM** ile eşanlamlıdır. Ağ işlemlerinin birinden ya da diğerinden alabilirsiniz.

`int` **EISCONN**
Zaten bağlı olan bir sokete bağlanmayı denediniz. Bkz. *Bir Bağlantının Oluşturulması* (sayfa: 422).

`int` **ENOTCONN**
Soket hiçbir şeye bağlı değil. Veriyi hedefini belirtmeksizin bir soket üzerinden aktarmaya çalışırsanız bu hatayı alırsınız. Bağlantısız soketlerde (örneğin UDP) bu hata yerine **EDESTADDRREQ** hatası alınır.

`int` **EDESTADDRREQ**
Sokete öntanımlı hedef adresi belirtilmemiş. Veriyi hedefini belirtmeksizin `connect` ile bir bağlantısız soket üzerinden aktarmaya çalışırsanız bu hatayı alırsınız.

`int` **ESHUTDOWN**
Soket zaten kapatılmış.

`int` **ETOOMANYREFS**
Çok fazla başvuru: uçlar birbirine bağlanamıyor

`int` **ETIMEDOUT**
Bir zamanaşımı belirtilmiş bir soket işlemine zamanaşımı süresinde bir yanıt gelmedi.

`int` **ECONNREFUSED**
Bir uzak konak ağ bağlantısına izin vermedi (genelde uzak konağın istenen hizmeti sunmadığı durumlarda oluşur).

`int` **ELOOP**
Bir dosya ismine bakılırken çok seviyeli sembolik bağlar saptandı. Bu durum çoğunlukla kapalı bir çevrim oluşturan sembolik bağları belirtir.

`int` **ENAMETOOLONG**
Dosya ismi çok uzun (**PATH_MAX**'dan daha uzun; bkz. *Dosya Sistemi Kapasite Sınırları* (sayfa: 795)) veya konak ismi çok uzun (`gethostname` veya `sethostname` işlevlerinden döner; bkz. *Konak İsimlendirmesi* (sayfa: 769)).

`int` **EHOSTDOWN**
İstenen ağ bağlantısındaki uzak konak çökük.

`int` **EHOSTUNREACH**
İstenen ağ bağlantısındaki uzak konak erişilebilir değil.

`int` **ENOTEMPTY**
Dizin boş değil, işlem için dizin boş olmalı. Bu hata genelde boş olmayan bir dizini silmeye kalkınca oluşur.

`int` **EPROCLIM**
Bu hata, yeni bir süreç üzerinde kullanıcı başına sınır `fork` ile aşıldığında oluşur. **RLIMIT_NPROC** sınırı hakkında daha ayrıntılı bilgiyi *Özkaynak Kullanımın Sınırlanması* (sayfa: 575) bölümünde bulabilirsiniz.

`int` **EUSERS**
Çok fazla kullanıcı olduğundan dosya kotası sistemi bozuldu.

`int` **EDQUOT**
Kullanıcının disk kotası aşıldı.

`int` **ESTALE**
Eskimiş NFS dosya kaydı. Sunucu konak üzerindeki dosya sisteminin yeniden düzenlenmesi nedeniyle NFS sisteminin iç bozulmasını belirtir. Bu bozulmanın giderilmesi için yerel konak üzerinde NFS dosya sisteminin önce ayrılıp sonra tekrar bağlanması gerekir.

`int` **EREMOTE**
Bir uzak dosya sistemi zaten kullanımda olan NFS bağlama dosyası ismi ile sisteme bağlanmaya çalışıldı. (Bu bazı işletim sistemlerinde bir hatadır, ancak GNU sistemlerinde bu hata kodunu imkansız yaparak onun çalışacağını umuyoruz.)

`int` **EBADRPC**

RPC yapısı hatalı

`int` **ERPCMISMATCH**
RPC sürümü yanlış

`int` **EPROGUNAVAIL**
RPC uygulaması kullanılabilir değil

`int` **EPROGMISMATCH**
RPC uygulaması sürümü yanlış

`int` **EPROCUNAVAIL**
RPC, uygulama için hatalı yordam yürüttü

`int` **ENOLCK**
Kullanılabilir bir kilit yok. Bu hata dosya kilitleme oluşumları tarafından kullanılır. Bilgi için *Dosya Kilitleri* (sayfa: 346) bölümüne bakınız. Bu hata GNU sistemlerinde hiçbir zaman üretilmez, ancak NFS sunucusunun çalıştığı başka bir işletim işletim sistemindeki bir işlemden kaynaklanabilir.

`int` **EFTYPE**
İlgisiz dosya türü ya da biçimi. Dosya işlem için yanlış türdendir ya da bir veri dosyası yanlış biçimlidir.

Bazı sistemlerde, **chmod** ile bir dizin olmayan dosyaya yapışkan bit belirtilmeye çalışılırsa bu hata oluşur. Bkz. *Dosya İzinlerinin Atanması* (sayfa: 380).

`int` **EAUTH**
Kimlik kanıtlama hatası.

`int` **ENEEDAUTH**
Kimlik kanıtlayıcı gerekli

`int` **ENOSYS**
İşlev henüz gerçeklenmedi. Bu hata çağrılan işlevin C kütüphanesinde ya da işletim sisteminde henüz mevcut olmadığını belirtir. Bu hatayı alıyorsanız, C kütüphanesinin ya da işletim sisteminin yeni bir sürümünü sisteminize kurmadığınız sürece aynı işlev için bu hatayı almaya devam edeceksiniz demektir.

`int` **ENOTSUP**
Desteklenmiyor. Bir işlev bu hatayı döndürüyorsa, ona verilen bir parametrenin geçerli olduğunu ancak istenen işlevselliğin bulunmadığı anlaşılır. İşlevin henüz bir komutu, bir seçeneği ya da bir bayrak bitini gerçekleştirmediği anlamına gelebilir. Bir işleve bir parametre içinde verilen bazı nesneler, örneğin bir dosya tanıtıcısı ya da port, verilen diğer parametrelerle desteği sağlayamadığı anlamına da gelebilir. Farklı bir dosya tanıtıcısı ya da farklı bir parametre değerleri aralığı ile destek sağlanmış olabilir.

İşlevin tamamı gerçeklenmemişse bu hata değil **ENOSYS** döner.

`int` **EILSEQ**
Bir çokbaytlı karakter çözümlemesinde, işlev bir geçersizlik saptadı ya da bayt sırası tamamlanmamış veya verilen geniş karakter geçersiz.

`int` **EBACKGROUND**
GNU sisteminde, **term** protokolünü destekleyen sunucular, çağrıcı uçbrimin önalan işlem grubunda değilse bazı işlemler için bu hatayı döndürebilir. Aslında kullanıcılar bu hatayı görmezler. Çünkü örneğin `read` ve `write` gibi işlevler hatayı **SIGTTIN** eya **SIGTTOU** sinyaline dönüştürürler. İşlem grupları ve bu sinyaller hakkında daha fazla bilgi edinmek için *İş Denetimi* (sayfa: 716) bölümüne bakınız.

`int` **EDIED**

GNU sisteminde, açılan bir dosya bir uygulama tarafından dönüştürüldüğünde veya çevirici uygulama dosyaya bağlanmadan önce başlatılırken ölürse bu hata oluşur.

`int` **`ED`**
Deneyimli kullanıcı neyi yanlış yaptığını bilir.

`int` **`EGREGIOUS`**
Ne yaptınız?

`int` **`EIEIO`**
Evine dön ve sütünü iç.

`int` **`EGRATUITOUS`**
Bu hata kodunun bir kullanım amacı yok.

`int` **`EBADMSG`**
Hatalı ileti

`int` **`EIDRM`**
Belirteç kaldırıldı

`int` **`EMULTIHOP`**
Çoklu sıçrama denendi

`int` **`ENODATA`**
Kullanılabilir veri yok

`int` **`ENOLINK`**
Bağ kopmuştu

`int` **`ENOMSG`**
İstenen türde ileti yok

`int` **`ENOSR`**
Akımdışı özkaynaklar

`int` **`ENOSTR`**
Aygıt bir akım değil

`int` **`EOVERFLOW`**
Tanımlı veri türü için değer çok büyük

`int` **`EPROTO`**
Protokol hatası

`int` **`ETIME`**
Zamanlayıcı zamanaşımına uğradı

`int` **`ECANCELED`**
İşlem iptal edildi. Bir işlem daha tamamlanmadan herhangi bir anda iptal edilmiş. Bkz. *Eşzamansız G/Ç* (sayfa: 327). `aio_cancel` işlevini çağırdığınızda normal sonuç olarak işlemin tamamlanması bu hata ile etkilenir. Bkz. *Eşzamansız G/Ç İşlemlerinin İptal Edilmesi* (sayfa: 336).

Aşağıdaki hata kodları Linux/i386 çekirdeğinde tanımlıdır ve henüz belgelendirilmemiştir. (Ç.N: Hata iletileri özgün belgede yoktur, onları ben ekledim.)

```
int ERESTART  Engellenen sistem çağrısı yeniden başlatılmalı

int ECHRNG    Kanal numarası aralık dışında

int EL2NSYNC  2. seviye eşzamanlı değil

int EL3HLT    3. seviye kapandı

int EL3RST    3. seviye sıfırlandı

int ELNRNG    Bağ numarası aralık dışında

int EUNATCH   Protokol sürücüsü bağlı değil

int ENOCSI    Kullanılabilir bir CSI yapısı yok

int EL2HLT    2. seviye kapandı

int EBADE     Geçersiz değişim

int EBADR     İstek tanımlayıcı hatalı

int EXFULL    Değişim kotası doldu

int ENOANO    Anot yok

int EBADRQC   İstek kodu geçersiz

int EBADSLT   Yuva geçersiz

int EDEADLOCK Dosya kilitlemede kısırdöngü hatası

int EBFONT    Yazıtipi dosyasının biçimi hatalı

int ENONET    Makina ağ üzerinde değil

int ENOPKG    Paket kurulu değil

int EADV      Dikkat çekme hatası

int ESRMNT    Srmount hatası

int ECOMM     Gönderme sırasında iletişim hatası

int EDOTDOT   RFS'e özgü hata

int ENOTUNIQ  İsim ağ üzerinde eşsiz değil
```

```
int EBADFD    Dosya tanımlayıcı hatalı durumda
int EREMCHG   Uzak adres değişti
int ELIBACC   Gerekli bir paylaşımlı kütüphaneye erişilemiyor
int ELIBBAD   Bozulmuş bir paylaşımlı kütüphaneye erişim
int ELIBSCN   a.out içindeki .lib bölümü bozulmuş
int ELIBMAX   Çok fazla paylaşımlı kütüphane ilintilenmeye çalışılıyor
int ELIBEXEC  Bir paylaşımlı kütüphane doğrudan çalıştırılamaz
int ESTRPIPE  Akımlarda boruhattı hatası
int EUCLEAN   Yapı temizlik gerektiriyor
int ENOTNAM   İsimli türde bir XENIX dosyası değil
int ENAVAIL   Kullanılabilir bir XENIX semaforu yok
int EISNAM    Bir isimli dosya mı
int EREMOTEIO Uzak G/Ç hatası
int ENOMEDIUM Ortam bulunamadı
int EMEDIUMTYPE Ortam türü yanlış
int ENOKEY
int EKEYEXPIRED
int EKEYREVOKED
int EKEYREJECTED
int EOWNERDEAD
int ENOTRECOVERABLE
```

3. Hata İletileri

Kütüphane, bir kütüphane çağrısının başarısız olması durumunda yazılımınızda özelleştirilebilir biçimde bilgilendirici hata iletileri oluşturabilmenizi kolaylaştıran işlevler ve değişkenler içerir. `strerror` ve `perror` işlevleri belirtilen bir hata kodu için standart hata iletisini verirler. `program_invocation_short_name`

değişkeni ise hatanın saptandığı yazılımın ismine erişebilmenizi sağlar.

```
char *strerror(int hatanum)                                    işlev
```

strerror işlevi *hatanum* ile belirtilen *hata kodunu* (sayfa: 31) bir açıklayıcı hata iletisi dizgesi ile eşleştirir ve bu dizgeye bir gösterici ile döner.

hatanum normalde **errno** değişkeninden alınır.

strerror ile döndürülen dizgeyi değiştirmemelisiniz. Ayrıca, **strerror** işlevine daha sonra yapacağınız çağrılar, dizgenin üzerine yazılması ile de sonuçlanabilir. (**strerror** bir kütüphane işlevi tarafından da çağrılabileceğinden döndürdüğü hata iletisinin sizin istediğiniz hata iletisi olacağının hiçbir garantisi yoktur.)

strerror işlevi `string.h`[B96] başlık dosyasında bildirilmiştir.

```
char *strerror_r(int    hatanum,                               işlev
                 char   *tampon,
                 size_t n)
```

strerror_r işlevi süreçteki tüm evreler (threads) tarafından paylaşılan süreç için ayrılmış durağan bellek içindeki evreye özel hata iletisini döndürmek dışında **strerror** işlevi gibi çalışır. Hata iletisini tutacak bellek bölgesi kullanıcı tarafından belirlenen ve *tampon* ile başlayan *n* bayt uzunlukta bir tampondur.

Bu tampona NULL karakteri de dahil olmak üzere en fazla *n* karakter yazılabileceğinden yazılımcı bu alanın büyüklüğünü yeterince büyük seçmelidir.

strerror tarafından döndürülen dizgenin gerçekten o anki evrenin son çağrısına karşılık olacağının hiçbir garantisi olmadığından çok evreli (multi–threaded) yazılımlarda daima **strerror_r** işlevi kullanılmalıdır.

strerror_r işlevi bir GNU oluşumu olup `string.h`[B97] başlık dosyasında bildirilmiştir.

```
void perror(const char *ileti)                                 işlev
```

Bu işlev standart hataya (***stderr* (sayfa: 237)**) bir hata iletisi basar. Standart hatanın yönünü değiştirmez.

perror işlevini *ileti* argümanına değer olarak bir boş gösterici ya da bir boş dizge belirterek çağırırsanız o andaki **errno** değerine karşılık gelen hata iletisini sonuna bir satırsonu karakteri ekleyerek basar.

ileti argümanı boş değilse, **perror** bu dizgeyi çıktısına önek olarak alır ve arkasına bir ikinokta üstüste ile boşluk ekledikten sonra **errno** değerine karşılık gelen hata iletisini sonuna bir satırsonu karakteri ekleyerek basar.

perror işlevi `stdio.h`[B99] başlık dosyasında bildirilmiştir.

strerror ve **perror** belirtilen hata kodu için tamamen aynı hata iletisini üretir, ancak üretilen ileti sistemden sisteme farklı olabilir. GNU sisteminde iletiler oldukça kısadır; çok satırlı hata iletileri yoktur (Ç.N: çeviriler bu kurala uymayabilir). Her hata iletisi büyük harfle başlar ve sonunda herhangi bir noktalama işareti bulunmaz.

Uyumluluk Bilgisi:

strerror işlevi ISO C89 ile gelmiştir. Eski C sistemlerinin çoğu hala bu işlevi desteklememektedir.

Uçbirimden girdi okumayan yazılımların çoğu bir sistem çağrısı başarısız olduğunda kendini sonlandıracak şekilde tasarlanır. Teamül olarak, böyle bir yazılımın verdiği hata iletileri dizin ismi içermeden yazılımın ismi ile başlamalıdır. Bu isim **program_invocation_short_name** değişkeninde bulunur. Dizini de içeren tam dosya ismi ise **program_invocation_name** değişkeninde saklanır.

`char *`**program_invocation_name** değişken

Bu değişkenin değeri o anda çalışmakta olan sürecin çağrıldığı yazılım ismidir. **argv[0]** (sayfa: 645) ile aynıdır. Bu bir dosya ismi olmak zorunda da değildir. Çoğunlukla da dizin bilgisi içermez.

`char *`**program_invocation_short_name** değişken

Bu değişkenin değeri, dizin isimleri kaldırıldığında o anda çalışmakta olan süreci çağırmakta kullanılan isimdir. Yani **program_invocation_name** değişkenin değeri olan dizgedeki son bölü çizgisi ve onun solundaki herşey kaldırıldığında kalan isim bu değişkenin değeridir diyebiliriz.

Kütüphane ilklendirme kodu `main` işlevi çağrılmadan önce bu değişkenlerin her ikisine de değerlerini atar.

Taşınabilirlik Bilgisi:

Bu iki değişken GNU oluşumudur. Yazılımınızın GNU dışı sistemlerde çalışmasını istiyorsanız yazılımın ismini `main` işlevinin **argv[0]** argümanının değerinden dizin isimlerini ayıklayarak elde etmelisiniz. `main` işlevi ile hiç etkileşime girmeden hata bildirme yordamlarında yazılım ismi içeren iletileri kolayca üretmek için bu değişkenleri kütüphaneye koyduk.

Aşağıdaki bir dosya açılamadığında hatayı gösteren bir örnek yeralmaktadır. `acil_susam_acil` işlevi *isim* isimli dosyayı okumak için açmayı dener, dosya sorunsuz olarak açılırsa bir akım ile döner. `fopen` kütüphane işlevi bir sebeple dosyayı açamazsa bir boş gösterici ile döner. Bu durumda `acil_susam_acil` işlevi `strerror` işlevini kullanarak bir hata iletisi oluşturur ve süreci sonlandırır. Hata kodunu `strerror` işlevine aktarmadan önce başka kütüphane çağrıları yapacaksak, onu bir değişkene kaydetmeliyiz, çünkü arada çağırdığımız diğer işlevler **errno** değerini değiştirebilir.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FILE *acil_susam_acil (char *isim)
{
  FILE *stream;

  errno = 0;
  stream = fopen (isim, "r");
  if (stream == NULL)
    {
      fprintf (stderr, "%s: %s dosyası açılamadı; %s\n";,
               program_invocation_short_name, isim, strerror (errno));
      exit (EXIT_FAILURE);
    }
  else
    return stream;
}
```

perror işlevi ISO C standardına uyumlu tüm sistemlerde geçerli olduğundan daha taşınabilirdir. Ancak **perror** işlevinin ürettiği hata iletisi tam isteneni sağlamaz ve yaptığı işlemi değiştirmek ya da geliştirmek için bir yol yoktur. Örneğin, GNU kodlama standardı, hata iletilerinin yazılımın ismi ile başlamasını ve bazı girdi

dosyalarını okuyan yazılımların girdi dosyasının ismi ile dosya okunurken hata oluşturan satırın numarası gibi bilgilerinde hata iletisinde içerilmesini gerektirir. Bu gibi durumlar için GNU projelerinde geniş çapta kullanılan iki işlev vardır. Bu işlevler `error.h` başlık dosyasında bildirilmiştir.

```
void error(int          durum,                                          işlev
           int          hatanum,
           const char *biçim,
           …)
```

error işlevi yazılımın çalışması sırasında oluşan genel sorunları bildirmek amacıyla kullanılır. *bicim* argümanı, `printf` ailesi işlevlerde olduğu gibi dizgeyi biçimlemekte kullanılır. **perror** gibi **error** işlevi de bir hata kodunu metin şeklinde basmak için kullanılabilir. **perror** işlevinden farklı olarak *hatanum* argümanıyla işleve hata numarası açıkça belirtilir. İşlevin bu özelliği, yukarıda bahsedilen, arada çağrılan başka işlevlerin **errno** değerini değiştirebilmesi sorununu ortadan kaldırır.

error işlevi önce yazılımın ismini basar. Eğer yazılım, **error_print_progname** isimli bir genel değişken tanımlamış ve onu işleve göstermişse, bu işlev yazılımın ismini basmak için çağrılacaktır. Aksi takdirde, **program_name** kütüphane değişkenindeki dizge kullanılır. Yazılımın isminden sonra iki nokta üstüste ve bir boşluk gelir. Onun ardına da *bicim* argümanı ile hazırlanan dizge basılır. *hatanum* argümanı sıfırdan farklı bir değere sahipse, biçim dizgesinden sonra ikinokta üstüste ve bir boşluk konduktan sonra bu değere karşılık olan hata iletisi basılır. Her durumda çıktı bir satırsonu karakteri ile sonlandırılır.

Çıktı standart hataya (**stderr**) yönlendirilir. Eğer standart çıktı çağrı öncesi yönlenimli değilse çağrı sonrası dar yönlenimli (narrow–oriented) olacaktır.

İşlev *durum* parametresi sıfırdan farklı bir değere sahip olmadıkça bir geridönüş değeri ile dönecektir. Bu durumda işlev **exit** işlevini parametre olarak *durum* değerini vererek çağıracak ve bu durumda asla dönmeyecektir. **error** işlevi bir geridönüş değerine sahipse, bu **error_message_count** kütüphane değişkeninin değerinin bir arttırılması ile elde edilen o ana kadar bildirilmiş hata iletilerinin sayısına karşılık gelen bir sayı olacaktır.

```
void error_at_line(int          durum,                                  işlev
                   int          hatanum,
                   const char   *dosyaismi,
                   unsigned int satırno,
                   const char   *biçim,
                   …)
```

error_at_line işlevi **error** işlevine çok benzer. Ek olarak *dosyaismi* ve *satırno* parametrelerini içerir. Yazılımın ismi ile *bicim* ile üretilen dizge arasına giren ek bilgiler dışında diğer parametrelerin yorumlanması **error** işlevindeki gibidir.

Yazılımın isminden sonraki iki nokta üstüsteden sonra *dosyaismi* ile verilen dosya ismi tekrar bir iki nokta üstüste ve ardından *satırno* ile verilen satır numarası ve bir iki nokta üstüste basılır.

Bu ek çıktı, bir girdi dosyasındaki hatanın yerini belirtmekte kullanılır.

error_one_per_line genel değişkenine sıfırdan farklı bir değer verilirse **error_at_line**, aynı dosya ismi ve aynı satır için başka hatalar varsa onları basmayacaktır. Hata bir ardardalık arzadiyorsa ardarda olmayan tekrarlar yakalanmaz.

error işlevi gibi bu işlevde *durum* sıfır olduğunda döner. Sıfırdan faklı ise **exit** çağrılır. İşlev bir geridönüş değerine sahipse, bu **error_message_count** kütüphane değişkeninin değerinin bir arttırılması ile elde edilen o ana kadar bildirilmiş hata iletilerinin sayısına karşılık gelen bir sayı olacaktır.

Yukarıda değinildiği gibi **error** ve **error_at_line** işlevleri **error_print_progname** isimli bir değişken tanımlanarak özelleştirilebilir.

`void (*error_print_progname)(void)` değişken

error_print_progname değişkenine sıfırdan farklı bir değer atanmışsa, bu değerin gösterdiği yazılım veya işlev **error** veya **error_at_line** işlevleri tarafından çağrılır.

İşlevin hatayı standart çıktıya basacağı ve akımın yönünü saptayabileceği umulur.

Değişken geneldir ve tüm evrelerce paylaşılır.

`unsigned int error_message_count` değişken

error_message_count değişkeninin değeri **error** veya **error_at_line** işlevlerinin her geri dönüşünde bir arttırılır. Değişken geneldir ve tüm evrelerce paylaşılır.

`int error_one_per_line` değişken

error_one_per_line değişkeni sadece **error_at_line** işlevi ile etkilidir. Normalde **error_at_line** işlevi her çağrı için bir çıktı oluşturur. **error_one_per_line** değişkenine sıfırdan farklı bir değer atanırsa, **error_at_line** işlevi aynı dosyanın aynı satırındaki diğer hataların iletilerini basmaz. Değişken geneldir ve tüm evrelerce paylaşılır.

Bir girdi dosyasını okuyan ve hataları bildiren bir yazılım aşağıdakine benzer:

```
{
  char *satir = NULL;
  size_t boyu = 0;
  unsigned int satirno = 0;

  error_message_count = 0;
  while (! feof_unlocked (fp))
    {
      ssize_t n = getline (&satir, &boyu, fp);
      if (n <= 0)
        /* Dosya sonu ya da hata.  */
        break;
      ++satirno;

      /* Satır işlemleri.  */
      …

      if (Satırdaki hata saptandı)
        error_at_line (0, hatanum, dosyaismi, satirno,
                       "hata iletisi metni %s", bir_degisken);
    }

  if (error_message_count != 0)
    error (EXIT_FAILURE, 0, "%u hata bulundu", error_message_count);
}
```

error ve **error_at_line** işlevleri GNU kodlama standardına göre uygulama geliştirmek isteyen yazılımcıların seçimi olmalıdır. GNU libc aynı amaçlar için BSD'de kullanılan işlevleri de içerir. Bu işlevler **err.h** başlık dosyasında bildirilmiştir. Uyumluluk amacıyla kütüphaneye dahil edildiklerinden, genelde kullanılmamaları önerilir.

`void warn(const char *biçim, …)` işlev

warn işlevi, **error** işlevinin birlikte çalıştığı genel değişkenleri kullanmaması dışında

```
error (0, hatanum, biçim, parametreler)
```

çağrısına eşdeğerdir.

```
void vwarn(const char *biçim,                                    işlev
           va_list)
```

vwarn işlevi, **warn** gibidir, farklı olarak *biçim* ile belirtilen biçim dizgesinin *va_list* (sayfa: 814) türünde verilen parametrelerle düzenlenmesini sağlar.

```
void warnx(const char *biçim, …)                                 işlev
```

warnx işlevi, **error** işlevinin birlikte çalıştığı genel değişkenleri kullanmaması dışında

```
error (0, 0, biçim, parametreler)
```

çağrısına eşdeğerdir. **warn** işlevinden farkı ise, hata numarası dizgesinin basılmamasıdır.

```
void vwarnx(const char *biçim,                                   işlev
            va_list)
```

vwarnx işlevi, **warnx** gibidir, farklı olarak *biçim* ile belirtilen biçim dizgesinin *va_list* (sayfa: 814) türünde verilen parametrelerle düzenlenmesini sağlar.

```
void err(int          durum,                                     işlev
         const char *biçim,
         …)
```

err işlevi,

```
error (durum, hatanum, biçim, parametreler)
```

çağrısı ile **error** işlevinin birlikte çalıştığı genel değişkenleri kullanmaması ve *durum* değişkenine sıfır verildiğinde bile yazılımın çıkması dışında aynıdır.

```
void verr(int          durum,                                    işlev
          const char *biçim,
                      va_list)
```

verr işlevi, **err** gibidir, farklı olarak *biçim* ile belirtilen biçim dizgesinin *va_list* (sayfa: 814) türünde verilen parametrelerle düzenlenmesini sağlar.

```
void errx(int          durum,                                    işlev
          const char *biçim,
          …)
```

errx işlevi,

```
error (durum, 0, biçim, parametreler)
```

çağrısı ile **error** işlevinin birlikte çalıştığı genel değişkenleri kullanmaması ve *durum* değişkenine sıfır verildiğinde bile yazılımın çıkması dışında aynıdır. **err** işlevinden farkı ise, hata numarası dizgesinin basılmamasıdır.

```
void verrx(int          durum,                                   işlev
           const char *biçim,
                       va_list)
```

verrx işlevi, **errx** gibidir, farklı olarak *biçim* ile belirtilen biçim dizgesinin *va_list* (sayfa: 814) türünde verilen parametrelerle düzenlenmesini sağlar.

III. Sanal Bellek Ayırma ve Sayfalama

İçindekiler

Bu kısımda GNU C kütüphanesi kullanılan bir sistemde belleği kullanımı ve süreçlerin yönetiminden bahsedilmiştir.

GNU C kütüphanesi sanal belleği farklı yollarla özdevimli ayırmak için çeşitli işlevler içerir. Bu çeşitlilik genellik ve tutarlılık anlamındadır. Kütüphane ayrıca gerçek belleğin ayrılması ve sayfalamanın denetlenmesi için işlevler de sağlar.

Bellek Eşlemli G/Ç (sayfa: 319) bu kısımda ele alınmamıştır.

1. Süreç Belleği Kavramları

Bir sürecin kullanabildiği çok temel özkaynaklardan biri bellektir. Sistemlerin belleği düzenlemesi için çok çeşitli yollar olmakla birlikte, genellikle tercih edilen, sıfırdan çok büyük değerlere kadar büyüyebilen adreslerle her sürecin bir doğrusal sanal adres alanına sahip olmasıdır. Onun bir süreklilik içermesi gerekmez; örneğin, bu adreslerin hepsi gerçekte veri saklamak için değildir.

Sanal bellek sayfalara bölünmüştür (genellikle 4 kilobaytlık). Sanal bellekteki her sayfa, bir gerçek bellek sayfası (***çerçeve*** de denir) veya ikincil bir saklama alanı ya da çoğunlukla disk alanı olarak gerçeklenir. Disk alanı, takas alanı olabileceği gibi sıradan bir disk dosyası da olabilir. Gerçekte, bir sayfa tamamen sıfırlardan oluşabilir ve içinde hiçbir şey bulunmayabilir (bir sayfanın sadece sıfırlardan oluştuğunu belirten bir bayrak vardır). Gerçek belleğin veya yedekleme alanının bir çerçevesi çok sayıda süreç için çok sayıda sanal sayfayı destekleyebilir. Bu normal bir durumdur. Örneğin, sanal bellek GNU C kütüphanesi tarafından işgal edilebilir. Bu durumda **`printf`** işlevini içeren gerçek bellek sayfası, işlevi kullanan çok sayıda sürecin herbiri için bir sanal bellek sayfası olur.

Bir yazılımın bir sanal sayfanın herhangi bir parçasına erişebilmesi için sayfa ("ilintili") bir gerçek çerçeve tarafından yedeklenir. Ancak gerçekte sanal bellek gerçek bellekten daha büyük olduğundan, sayfalar düzenli olarak, gerçek bellek ile yedekleme deposu arasında ileri ve geri taşınması, bir sürecin onlara erişmesi gerektiğinde gerçek belleğe alınması ve gerek kalmadığında da yedekleme deposuna geri taşınması gerekir. Bu devinime ***sayfalama*** denir.

Bir süreç bir sayfaya erişmeye çalışır ve sayfa gerçek belleğe kopyalanamazsa, bu olaya ***sayfalama hatası*** ("page fault") denir. Bir sayfalama hatası oluştuğunda çekirdek süreci askıya alır ve sayfayı bir gerçek sayfa çerçevesine kopyalar. Bu işleme ***gerçek belleğe sayfalama*** ("paging in" or "faulting in") denir. Bu işlemin ardından artık sayfa gerçek bellekte erişilebilir olduğundan çekirdek tarafından askıya alınan süreç işlemine kaldığı yerden devam ettirilir. Aslında, tüm sayfalar sürece daima gerçek bellekteymiş gibi görünür. Bir şey dışında: bir makina dili komutun bekletilme süresi, olsun olsun ancak bir kaç nanosaniye sürer, çünkü çekirdek gerçek belleğe sayfalama işlemini bu kadar kısa sürede tamamlayabilir. Bu işleme duyarlı yazılımlar için kullanılan denetim işlevleri *Sayfaların Kilitlenmesi* (sayfa: 77) bölümünde anlatılmıştır.

Her sanal adres alanı içinde, bir süreç hangi adreslerin ne olduğu bilgisine sahiptir ve bu sürece ***bellek ayırma*** denir. Ayırma süreci aslında kıt özkaynakları toplayarak kendine ayırır, ancak sanal bellek söz konusu olduğunda ana amaç bu değildir, çünkü orada genellikle herhangi bir ihtiyaca yetecekten fazlası vardır. Bir süreç içindeki bellek ayırma işleminin ana konusu iki farklı şeyin aynı bellek baytlarında olmadığından emin olmaktır.

Süreçler belleği iki farklı yolla ayırır: çalıştırma sırasında ve yazılımsal olarak. Aslında, *çatallama* (sayfa: 687) üçüncü bir yol olmakla beraber çok enteresan değildir.

Çalıştırma (exec) bir işlemdir; bir süreç için bir sanal bellek ayırmak, ana yazılımı bu alana yüklemek ve yazılımı çalıştırmaktır. Bu, "exec" ailesi işlevlerle (örn, **`execl`**) yapılır. İşlem bir yazılım dosyası (bir çalıştırılabilir dosya) alır, içindeki tüm verinin yükleneceği alanı ayırır ve yazılımı da yükledikten sonra denetimi yazılıma aktarır. Bu veri çoğunlukla yazılımın komutlarından (***metin***) oluşmakla birlikte yazılımdaki dizgeler, sabitler ve hatta bazı değişkenleri de içerir: *durağan saklama sınıfı C değişkenleri* (sayfa: 49).

Bir yazılım çalışmaya başladıktan sonra ek bellek kazanmak için yazılımsal ayırma yapar. GNU C kütüphanesi

kullanan bir C yazılımında, iki çeşit yazılımsal ayırma kullanılabilir: özdevinimli (automatic) ve özdevimli (dynamic). Bkz. *C Yazılımlarında Bellek Ayırma* (sayfa: 49).

Bellek eşlemli G/Ç diğer bir özdevimli sanal bellek ayırma şeklidir. Belleğin bir dosyayla eşlenmesi, bir sürecin belli bir adres aralığındaki içeriğinin belirtilen normal bir dosyanın içeriği ile eşdeğerde olacağının bildirilmesidir. Sistem sanal belleğin dosyanın içeriğini içermesini sağlar ve eğer siz bellekte değişiklik yaparsanız sistem aynı değişiklikleri dosyaya yazar. Sanal belleğin çalışması ve sayfalama hatalarının sonuçları itibariyle, yazılım sanal belleğe erişinceye kadar, sistemin dosyayı okumak için G/Ç işlemleri yapması ve ona gerçek bellek ayırması için bir sebep yoktur. Bkz. *Bellek Eşlemli G/Ç* (sayfa: 319).

Belleğin yazılımsal ayrılması gibi yazılım, yazılımsal olarak onu serbest bırakabilir. Çalıştırma işlemi ile ayrılan belleği siz serbest bırakamazsınız. Ancak yazılımın çalışmasını sonlandırırsanız (exit) ona ayrılan tüm belleğin serbest kaldığından söz edilebilir, ancak adres alanı kesintiye uğramadan kalır ki bu nokta gerçekten tartışma götürür. Bkz. *Yazılımın Sonlandırılması* (sayfa: 681).

Bir sürecin adres alanı bölütlere bölünür. Bir ***bölüt*** kesintisiz bir sanal adres aralığıdır. Üç önemli bölüt vardır:

metin bölütü
: Bir yazılımın komutlarını, dizgelerini ve durağan sabitlerini içerir. Çalıştırma işlemi tarafından ayrılır ve yaşamı boyunca aynı boyutta sanal adres alanında kalır.

veri bölütü
: Yazılımın çalışan deposudur. İlk ayırma ve ilk yükleme işlemi çalıştırma işlemi tarafından yapılabilir ve süreç tarafından *Veri Bölütünün Boyunun Değiştirilmesi* (sayfa: 77) bölümünde açıklandığı gibi genişletilebilir veya daraltılabilir. Alt ucu sabittir.

yığıt bölütü
: Bir yazılım yığıtı içerir. Bir yığıtın büyüdüğü gibi büyür ama küçüldüğü gibi küçülmez.

2. Yazılım Verisine Saklama Alanı Ayrılması

Bu bölümde sıradan yazılımların kendi verilerini nasıl yönettiği, meşhur `malloc` işlevi ile GNU C kütüphanesi ve GNU derleyicisine özel oluşumlardan bahsedilmiştir.

2.1. C Yazılımlarında Bellek Ayırma

C dili, C yazılımlarındaki değişkenler için iki çeşit bellek ayırma işlemini destekler:

Durağan bellek ayırma
: Bir durağan (static) veya genel (global) değişken tanımında durağan ayırma yapılır. Her durağan veya genel değişken sabit uzunlukta bir bellek bloğu tanımlar. Bu blok, yazılım ilk çalıştırıldığında bir kere ayrılır ve asla serbest bırakılmaz (durağan özelliği).

Özdevinimli bellek ayırma
: Bir işlev argümanı ya da bir yerel değişken gibi bir özdevinimli bir değişken için özdevinimli bellek ayırması uygulanır. Bir özdevinimli (automatic) değişken için bellek alanı bir birleşik deyim bildirimine girildiğinde ayrılır ve birleşik deyimin çalışması sona erdiğinde serbest bırakılır. GNU C'de bir özdevinimli saklama alanının uzunluğu bir ifade ile belirtilebilir ve değiştirilebilir. Diğer C gerçeklemelerinde bu uzunluk bir sabit olmalıdır.

Bir üçüncü bellek ayırma türü ise C değişkenleri tarafından desteklenmeyen ama GNU C kütüphanesi üzerinden kullanılabilen ***özdevimli bellek ayırma***dır.

2.1.1. Özdevimli Bellek Ayırma

Özdevimli (dynamic) bellek ayırma, yazılımların çalışması sırasında bazı bilgilerin saklanmasını sağlayan bir tekniktir. Yazılımınızın bir bellek ihtiyacı olduğunda ancak ne kadar bellek ihtiyacı olacağı baştan belli olmayan durumlarda bu teknik faydalıdır.

Örneğin, bir girdi dosyasından okunan bir satırı saklamanız gerekebilir; bir satırın uzunluğunun belli bir sınırı olmadığından belleği özdevimli olarak ayırmanız ve satırdan daha fazlasını okumanız gerekirse özdevimli olarak bu alanı büyütmeniz gerekir.

Veya, bir veri girdisi olarak her kayıt ya da her tanım için bir blok gerekebilir; bunlardan kaç tane olacağını baştan bilemeyeceğinizden, her yeni kayıt için yeni bir blok veya her yeni tanım okunduğunda bir blok ayırmanız gerekir.

Özdevimli bellek ayırması kullandığınızda, bir bellek bloğunun ayrılması yazılımın doğrudan doğruya istediği bir eylemdir. Alanı ayırmak istediğinizde ayırmak istediğiniz uzunluğu bir argüman olarak belirterek bir işlev ya da bir makro çağırırsınız. Bu alanı serbest bırakmak istediğinizde ise başka bir işlev ya da makroyu çağırırsınız. Tüm bu işlemleri ne zaman ve ne sıklıkta gerekiyorsa yapabilirsiniz.

Özdevimli ayırma C değikenlerince "dynamic" isminde bir saklama sınıfı olmadığından desteklenmez ve özdevimli ayrılmış alanda saklanan değerin atandığı bir C değişkeni asla olamaz. Özdevimli ayrılmış bellek almanın tek yolu bir sistem çağrısı üzerinden almak (bu sistem çağrısı genellikle GNU C kütüphanesinin bir işlevi üzerinden yapılır) ve ona erişmenin tek yolu da bir gösterici kullanmaktır. Daha az kullanışlı olduğundan ve özdevimli bellek ayırma işlemi daha fazla hesaplama gerektirdiğinden özdevimli ayırma sadece durağan ya da özdevinimli bellek ayırma uygulanamadığında kullanılır.

Örneğin, bir **`struct foobar`** için özdevimli olarak bellek ayırması yapmak isterseniz, bu alandaki içerik için **`struct foobar`** türünde bir değişken bildiremezsiniz. Ama **`struct foobar *`** gösterici türünden bir değişken bildirebilir ve bu alanın adresini ona değer olarak atayabilirsiniz. Sonra da **`*`** ve **`->`** işleçlerini kullanarak bu alandaki içeriğe erişirsiniz:

```
{
  struct foobar *ptr
    = (struct foobar *) malloc (sizeof (struct foobar));
  ptr->name = x;
  ptr->next = current_foobar;
  current_foobar = ptr;
}
```

2.2. Özgür Bellek Ayırma

En çok kullanılan özdevimli bellek ayırma oluşumu **`malloc`** işlevidir. Bu işlevi kullanarak istediğiniz zaman istediğiniz boyutta bellek bloğu ayırabilir, istediğinizde büyütebilir, küçültebilir ve ihtiyacınız kalmadığında serbest bırakabilir ya da hiç serbest bırakmayabilirsiniz.

2.2.1. Özdevimli Olarak Basit Bellek Ayırma

Bir bellek bloğunu ayırmak için **`malloc`** çağrılır. Bu işlev `stdlib.h` başlık dosyasında bildirilmiştir.

`void *`**`malloc`**`(size_t` *boyut*`)` işlev

> Bu işlev *boyut* bayt uzunlukta yeni bir bellek bloğu ayırabilirse ve bu bloğa bir gösterici ile döner. Bellek ayrılamazsa bir boş gösterici ile döner.

Bloğun içi tanımsızdır; içeriğini kendiniz ilklendirmelisiniz (ya da yerine **`calloc`** kullanın). Bkz. *Temizlenmiş Bellek Ayırma* (sayfa: 53).

Normalde blok içinde saklamak istediğiniz nesnenin türünde bir gösterici olan değere tür dönüşümü uygulamalısınız. Aşağıda bu bu yönteme ilişkin bir örnek vardır. Burada blok, **memset** kütüphane işlevi kullanılarak sıfırlarla ilklendirilmektedir (Bkz. *Kopyalama ve Birleştirme* (sayfa: 94)):

```
struct foo *ptr;
…
ptr = (struct foo *) malloc (sizeof (struct foo));
if (ptr == 0) abort ();
memset (ptr, 0, sizeof (struct foo));
```

ISO C **void *** türünü gerektiğinde özdevinimli olarak başka bir gösterici türüne dönüştürebildiğinden, **malloc** işlevinin sonucunu bir tür dönüşümü yapmaksızın bir gösterici değişkeninde saklayabilirsiniz. Ancak kodu geleneksel C ile çalıştıracaksanız ya da atama işleci kullanmıyorsanız tür dönüşümü yapmanız gerekir.

Bir dizge için yer ayırırken **malloc** işlevine belirteceğiniz argümanın değeri, dizgenin uzunluğu artı bir olmalıdır. Bu, dizgenin bir NULL karakteri ile sonlandırılması ve NULL karakterinin dizgenin uzunluğundan sayılmamasından dolayıdır. Örnek:

```
char *ptr;
…
ptr = (char *) malloc (length + 1);
```

Bu konu hakkında daha ayrıntılı bilgi edinmek için *Dizgelerle İlgili Kavramlar* (sayfa: 90) bölümüne bakınız.

2.2.2. malloc Örnekleri

Ayrılabilecek bellek yoksa **malloc** bir boş gösterici ile döner. **malloc** işlevini her çağırışınızda dönen değere bakmalısınız. **malloc**'u çağıran, bir boş gösterge döndüğünde hatayı bildiren ve sadece sıfırdan farklı bir değerle dönen bir yardımcı işlev yazabilirsiniz. Bu yöntem teamülen **xmalloc** olarak bilinir ve aşağıdaki gibi gerçeklenir:

```
void * xmalloc (size_t size)
{
  register void *value = malloc (size);
  if (value == 0)
    fatal ("sanal bellek tükendi");
  return value;
}
```

Aşağıda ise **malloc** kullanılan (**xmalloc** yoluyla) gerçek bir örnek vardır. `savestring` işlevi bir karakter dizisini yeni ayrılan null sonlandırmalı dizgeye kopyalamaktadır:

```
char * savestring (const char *ptr, size_t len)
{
  register char *value = (char *) xmalloc (len + 1);
  value[len] = '\0';
  return (char *) memcpy (value, ptr, len);
}
```

malloc'un size verdiği blok her türden veriyi tutabilir. GNU sisteminin bellek yönetimine göre, bir çok sistemde adres sekizin katlarıdır, 64 bitlik sistemlerde ise onaltının katlarıdır. Daha yüksek sınırlar (sayfa seviyesi) gerekliyse **memalign**, **posix_memalign** veya **valloc** kullanılır (Bkz. *Bellek Bloklarının Hizalanarak Ayrılması* (sayfa: 54)).

Bilgi

Bir **malloc** çağrısı ile ayrılan bloğun sonuna başka birşeyler için kullanılmak üzere bellek eklenmesi gerekebilir. Daha önce ayrılmış bir bloğu büyütmek için **malloc** kullanmaya çalışırsanız ya önceki ayrılan bloğu ya da yenisini bozmanız mümkündür. Ayrılmış belleği sonradan büyütmek isterseniz **realloc** işlevini kullanın (Bkz. *Bir Bellek Bloğunun Boyutunun Değiştirilmesi* (sayfa: 52)).

2.2.3. `malloc` ile Ayrılan Belleğin Serbest Bırakılması

malloc ile ayırdığınız bir bloğa ihtiyacınız kalmadığında, ayrılan bloğu tekrar kullanılabilir hale getirmek için **free** ile serbest bırakmalısınız. **free** işlevi `stdlib.h` başlık dosyasında bildirilmiştir.

`void` **free**`(void *`*gösterici*`)` işlev

free işlevi *gösterici* ile erişilen bellek bloğunu serbest bırakır.

`void` **cfree**`(void *`*gösterici*`)` işlev

Bu işlev **free** ile aynı işi yapar. SunOS ile geriye uyumluluk için vardır. Bu işlevi değil **free** işlevini kullanmalısınız.

Bir bloğun serbest bırakılması bloğun içeriğini değiştirir. *Bir bloğu serbest bıraktıktan sonra bloğun içindeki bir veriye ulaşabileceğinizi sanmayın (örneğin bir blok zincirindeki sonraki bloğa bir gösterici belirterek.)* Bir bloğu serbest bırakmadan önce içinde kullanacağınız bir bilgi varsa kopyalamayı unutmayın. Aşağıdaki örnekte bir zincirdeki blokların tümü sırayla serbest bırakılmaktadır:

```
struct chain
  {
    struct chain *next;
    char *name;
  }

void
free_chain (struct chain *chain)
{
  while (chain != 0)
    {
      struct chain *next = chain->next;
      free (chain->name);
      free (chain);
      chain = next;
    }
}
```

free işlevi bellek alanını uzun dönemde işletim sistemine döndürür ve süreci küçültür. Sistem genelinde ise, bellek alanını yeniden kullanılmak üzere serbest bırakır ve daha sonraki **malloc** çağrılarını mümkün kılar. Kısa dönemde ise, serbest bırakılan alan **malloc** tarafından süreç dahilinde kullanılmak üzere bir serbest bırakılanlar listesinin parçası olarak bekletilir.

Yazılımın çalışması sona erdiğinde blokların bırakılması gerekmez. Çünkü süreç sonlandığında sürece ayrılan tüm alan sisteme geri verilir.

2.2.4. Bir Bellek Bloğunun Boyutunun Değiştirilmesi

Çoğunlukla bir bellek bloğunu kullanmaya başlarken ihtiyacınız olacak bellek miktarı bilemez ve yaklaşık bir boyut ile bloku ayırırsınız. Örneğin, blok bir dosyadan okunan satırı tutan bir tampon olabilir ve bir satır için yeterli olan tamponunuz başka bir satır için yetersiz kalabilir.

realloc işlevini çağırarak bloğu uzatabilirsiniz. Bu işlev `stdlib.h` başlık dosyasında bildirilmiştir.

```
void *realloc(void  *gösterici,                                    işlev
              size_t yeniboyut)
```

realloc işlevi *gösterici* ile erişilen bloğun boyunu *yeniboyut* ile belirtilen değere ayarlar.

Bloktan sora gelen alan kullanılabilir olsa da **realloc** işlevi bloğu daha fazla serbest alanın bulunduğu yeni bir adrese kopyalamayı gerekli bulabilir. **realloc** işlevi bloğun yeni adresi ile döner. Blokun taşınması gerekirse, **realloc** eski içeriği kopyalar.

realloc işlevine *gösterici* için bir boş gösterici belirtirseniz, işlev, **malloc (***newsize***)** çağrılmış gibi davranır. Bu kullanışlı olabilirse de, ISO C öncesi C gerçeklemeleri bu davranışı desteklemeyebilir ve büyük ihtimalle **realloc** boş bir gösterici ile çökebilir.

realloc işlevi daha büyük blok oluşturamazsa tıpkı **malloc** işlevi gibi bir boş gösterici ile döner. Bu durum oluştuğunda eski bloka dokunulmaz; değiştirilmez ve başka bir konuma taşınmaz.

Çoğu durumda **realloc** başarısız olduğunda eski bloğa ne olduğunun fazla bir önemi kalmaz. Çünkü uygulama yazılımı bellek yetmediğinde çalışmasını sürdüremez ve yapacak tek şey kalır: bir ölümcül hata iletisi ile kendini sonlandırmak. Çoğunlukla bu durum için bir yordam yazılır. **malloc** için benzer yordama **xmalloc** dendiği gibi **realloc** için olanına da **xrealloc** denir:

```
void * xrealloc (void *ptr, size_t size)
{
  register void *value = realloc (ptr, size);
  if (value == 0)
    fatal ("Sanal bellek tükendi");
  return value;
}
```

realloc işevini bloğu küçültmek için de kullanabilirsiniz. Bu genelde daha küçük belleğin yeterli olduğu durumlarda bir bellek tıkanıklığı varsa ayrılan fazla belleği serbest bırakmak için yapılır. Bazı ayırma gerçeklemelerinde bir bloğun küçültülmesi bazan onun kopyalanmasını zorunlu kılabilir ve kullanılabilir bir bellek yoksa işlev başarısız olabilir.

realloc işlevine belirtilen yeni boyut eskisi ile aynıysa işlev hiçbir değişiklik yapmaz ve verdiğiniz adresi geri döndürür.

2.2.5. Temizlenmiş Bellek Ayırma

calloc işlevi belleği ayırdıktan sonra içeriğini sıfırlarla doldurarak temizler. Bu işlev `stdlib.h` başlık dosyasında bildirilmiştir.

```
void *calloc(size_t elmsayısı,                                     işlev
             size_t elmboyu)
```

Bu işlev boyu *elmboyu* olan *elmtsayısı* sayıda eleman içeren bir blok ayırır. Bu içerik **calloc** dönmeden önce sıfırlarla doldurulur.

calloc işlevini aşağıdaki gibi tanımlayabilirsiniz:

```
void * calloc (size_t count, size_t eltsize)
{
  size_t size = count * eltsize;
  void *value = malloc (size);
  if (value != 0)
```

```
    memset (value, 0, size);
  return value;
}
```

Ancak genelde, **calloc** işlevinin dahili olarak **malloc** işlevini çağıracağının garantisi yoktur. Bu nedenle, bir uygulama C kütüphanesi dışında kendi **malloc**, **realloc** ve **free** işlevlerini sağlıyorsa, **calloc** işlevini de mutlaka tanımlamalıdır.

2.2.6. **malloc** için Yeterlik Kaygıları

Diğer sürümlerin aksine GNU C kütüphanesindeki **malloc** işlevi ayırdığı belleğin boyunu ikinin kuvvetlerine ayarlamaz, dolayısıyla istenenden daha küçük ya da daha büyük bir bellek ayırmaz. Komşu bellek parçaları boyutlarına bakılmaksızın bir **free** üzerinden tek parça yapılabilir. Bu, yüksek belleğin küçük kesintili bloklar haline gelmesine sebep olmadan, gerçeklemeyi ayırma kalıplarının her türü için elverişli yapar.

Çok büyük bloklar (bir sayfadan büyük) bu gerçekleme tarafından **mmap** işlevi ile (anonim veya `/dev/zero` üzerinden) ayrılır. Serbest bırakıldıkları anda parçaların sisteme döndürülmesi büyük yarar sağlar. Bu nedenle, daha küçük biri ve her **free** çağrısından sonraki arasında kilitlenen bir büyük blok belleği boşaltır. **mmap** için kullanılacak boyut eşiği **mallopt** ile ayarlanabilir. Ayrıca **mmap** kullanımı tamamen iptal de edilebilir.

2.2.7. Bellek Bloklarının Hizalanarak Ayrılması

Bir blok için **malloc** tarafından döndürülen adres GNU sistemlerinde daima sekizin katlarıdır (64 bitlik sistemlerde onaltının katlarıdır). Bir bloğun daha büyük adres adımları (adım boyu ikinin kuvveti olmalı) ile adreslenmesini isterseniz **memalign**, **posix_memalign** veya **valloc** işlevlerini kullanmalısınız. **memalign** işlevi `malloc.h`, **posix_memalign** işlevi ise `stdlib.h` başlık dosyasında bildirilmiştir.

GNU kütüphanesi ile, **memalign**, **posix_memalign** veya **valloc** ile dönen blokları serbest bırakmak için **free** kullanılır. Bu BSD ile çalışmaz, çünkü BSD bu tür blokları serbest bırakacak bir yol sağlamaz.

```
void *memalign(size_t adımboyu,                                    işlev
               size_t boyut)
```

memalign işlevi *boyut* uzunluktaki bir bloğu *adımboyu*nun katları olan adreslerden birinde ayırır ve bloğun adresi ile döner. *adımboyu* ikinin kuvveti olan bir değer olmalıdır. İşlev büyük miktarda (sayfa kadar veya fazlası) bellek ayırmak için kullanılır.

```
int posix_memalign(void **gösterici,                               işlev
                   size_t adımboyu,
                   size_t boyut)
```

posix_memalign işlevi **memalign** işlevine benzer şekilde *adımboyu*nun katlarına hizalayarak *boyut* uzunluktaki bir tampon ayırır. Ancak işlev *adımboyu* parametresinde bir özelliğin varlığına bağımlıdır: değer, **sizeof (void *)** değerinin ikinin kuvveti katı olmalıdır.

İşlev, belleği ayırmada başarılı olursa, ayrılan belleğin göstericisi ***gösterici**'ye atanır ve işlev sıfır değeriyle döner. Aksi takdirde sorunu belirten bir hata değeri ile döner.

Bu işlev POSIX 1003.1d içinde geçer.

```
void *valloc(size_t boyut)                                         işlev
```

valloc işlevi sayfa boyunda adresleme ile *boyut* uzunlukta blok ayırır. **memalign** kullanılarak aşağıdaki gibi gerçeklenebilir:

```
void * valloc (size_t size)
{
  return memalign (getpagesize (), size);
}
```

Bellek altsistemi hakkında ayrıntılı bilgi *Bellek Parametrelerinin Sorgulanması* (sayfa: 590) bölümünde bulunabilir.

2.2.8. Ayarlanabilir Malloc Parametreleri

Özdevimli bellek ayırma ile ilgili bazı parametreleri **mallopt** ile ayarlayabilirsiniz. Bu işlev bir genel SVID/XPG arayüzüdür ve `malloc.h` başlık dosyasında bildirilmiştir.

```
int mallopt(int param,                                                    işlev
            int değer)
```

mallopt işlevi çağrılırken ayarlanacak parametre *param* ile, değeri ise *değer* ile belirtilir. `malloc.h` başlık dosyasında tanımlı *param* seçenekleri şunlardır:

`M_TRIM_THRESHOLD`
: Belleğin sisteme döndürülmesi sırasında bir negatif argüman ile **sbrk**'in çağrılmasına sağlayacak en tepedeki serbest bırakılabilir parçanın mümkün en küçük boyudur (bayt cinsinden).

`M_TOP_PAD`
: Bu parametre bir **sbrk** çağrısı gerektiğinde sistemden sağlanacak fazladan belleğin miktarını belirler. Ayrıca, bir negatif argümanla **sbrk** çağırarak yığın belleğin (heap) küçültülmesi sırasında tutulan baytların sayısını belirtir. Bu, yığın bellekte gerekli histerezisi oluşturur ve örneğin aşırı miktarlarda sistem çağrılarını engelleyebilir.

`M_MMAP_THRESHOLD`
: Bu değerden daha büyük parçalar **mmap** sistem çağrısı kullanılarak normal yığın belleğin dışında ayrılır. Bu yol, bu parçaların **free** üzerinden sisteme döndürülebilirliğini garanti eder. Bu eşik değerden daha küçük istekler hala **mmap** üzerinden ayrılabilir.

`M_MMAP_MAX`
: **mmap** ile ayrılabilecek parçaların mümkün en büyük sayısıdır. Bu parametreye sıfır değeri atanarak **mmap** kullanımı iptal edilebilir.

2.2.9. Yığın Bellek Tutarlılık Denetimi

mcheck işlevini kullanarak özdevimli ayrılan belleğin tutarlılığının **malloc** tarafından denetlenmesini isteyebilirsiniz. Bu işlev bir GNU oluşumudur ve `mcheck.h` başlık dosyasında bildirilmiştir.

```
int mcheck(void (*çıkış_işlevi) (enum mcheck_status durum))                işlev
```

mcheck işlevinin çağrılması **malloc**'a arasıra tutarlılık denetimi yapmasını söyler. Bu sayede **malloc** ile ayrılan belleği aşan yazma işlemleri yakalanır.

çıkış_işlevi argümanı ile bir tutarsızlık bulunduğunda çağrılacak işlev belirtilir. Burada bir boş gösterici belirtirseniz, **mcheck** işlevi bir ileti basarak öntanımlı olan **abort** (sayfa: 683) işlevini çağırır. Belirttiğiniz işlev ne tutarsızlığı saptandığını söyler ve tutarsızlık türünün belirtildiği bir argümanla çağrılır. Tutarsızlık türleri *aşağıda* (sayfa: 56) açıklanmıştır.

malloc ile bellek ayırma yaptıktan sonra tutarlılık denetimini başlatmak için bazan çok geç kalınabilir. Bu durumda **mcheck** hiçbir şey yapmaz. Eğer çağrı için çok geç kalınmışsa **mcheck -1** ile aksi takdirde **0** ile döner.

mcheck işlevini yeterince erken çağırmanın en kolay yolu, yazılımınızı ilintilerken `-lmcheck` seçeneğini kullanmaktır. Böylece yazılımınızın kaynak kodunu değiştirmeniz gerekmez. Başka bir yol da, yazılımınızı her başlattığınızda bir **mcheck** çağrısı girecek bir hata ayıklayıcı kullanmaktır. Örneğin, aşağıdaki gibi yazılımınızı her başlattığınızda gdb komutları özdevinimli olarak **mcheck**'i çağıracaktır.

```
(gdb) break main
Breakpoint 1, main (argc=2, argv=0xbffff964) at whatever.c:10
(gdb) command 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>call mcheck(0)
>continue
>end
(gdb) …
```

mcheck işlevinin **malloc** işlevinden önce çağrılması gerektiğinden bu kod sadece **malloc** işlevini çağıran nesnelerde böyle bir ilklendirme işlevi yoksa çalışır.

`enum mcheck_status` **mprobe**`(void *`*göstərici*`)` işlev

mprobe işlevi ile bir ayrılmış blok üzerinde tutarlılık denetimi yapılmasını doğrudan doğruya isteyebilirsiniz. Bu denetimin arasıra yapılması için zaten yazılımın hemen başlarında **mcheck** çağrısı yapmış olmalısınız; **mprobe** çağrısı ile çağrı sırasında bir ek tutarlılık denetimi yapılmasını isteyebilirsiniz.

göstərici, bir **malloc** ya da **realloc** işlevinden dönmüş bir gösterici olmalıdır. **mprobe** işlevi ne tür bir tutarsızlık olduğunu belirten bir değer ile döner. Bu değerler aşağıda açıklanmıştır.

`enum` **mcheck_status** veri türü

Bu numaralı tür, bir ayrılmış blokta saptanan tutarsızlığın çeşidini açıklar. Olası değerler:

`MCHECK_DISABLED`
: **mcheck** ilk ayırma işleminden önce çağrılmamış. Hiçbir tutarlılık denetimi yapılamaz.

`MCHECK_OK`
: Bir tutarsızlık yok.

`MCHECK_HEAD`
: Bloğun öncesi değiştirildi. Bu durum genellikle, bir dizi indisi ya da bir göstericinin bloğun başlangıcındaki değerinin altına indirildiğinde ortaya çıkar.

`MCHECK_TAIL`
: Bloktan sonrası değiştirildi. Bu durum genellikle, bir dizi indisi ya da bir göstericinin değeri blok aşılacak kadar arttırıldığında ortaya çıkar.

`MCHECK_FREE`
: Blok zaten serbest bırakılmış.

Diğer bir denetim başlatma yöntemi, **malloc**, **realloc** ve **free** kullanımında yazılım hatalarına karşı bir önlem olarak **MALLOC_CHECK_** ortam değişkeninin belirtilmesidir. **MALLOC_CHECK_** bir denetim yapılması için belirtilmişse, çifte **free** çağrısı veya bir tek baytlık taşma gibi basit hatalara tolerans gösteren (daha az

tutarlı) özel bir gerçekleme kullanılır. Bu tür hatalara karşı bir koruma olmadığından bellek artıkları olabileceği dikkate alınmalıdır.

MALLOC_CHECK_ için **0** değeri kullanılırsa, bir yığın bellek bozulması gözardı edilecektir. **1** kullanılırsa, standart hataya tanımlayıcı bir ileti basılır. **2** olması durumunda ise hemen **abort** çağrılır. Bu son durum genellikle hatanın hemen ardından bir çökmenin kaçınılmaz olduğu dolayısıyla hatanın geriye doğru izlenebilirliğinin çok zor olduğu durumlarda kullanışlıdır.

MALLOC_CHECK_ ile ilgili bir sorun vardır:

SUID ve SGID'li çalıştırılabilirlerinin istismarı sözkonusu olabilir; bunlar, normal yazılım davranışından ayrılarak standart hata tanımlayıcıya bazı şeyler yazar. Bu nedenle, **MALLOC_CHECK_** kullanımı SUID ve SGID'li çalıştırılabilirler için öntanımlı olarak iptal edilmiştir. Ancak sistem yöneticisi tarafından `/etc/suid-debug` dosyası eklenerek etkinleştirilebilir (bu dosyanın içeriği önemli değildir, dosya boş olabilir).

Öyleyse **MALLOC_CHECK_** kullanımı ve `-lmcheck` seçeneğiyle ilintileme arasında ne fark vardır? `-lmcheck` geriye uyumluluk adına vardır. Her ikisi de aynı yazılım hataları için kullanılırsa da **MALLOC_CHECK_** kullandığınızda yazılımı yeniden derlemek gerekmez.

2.2.10. Bellek Ayırma Kancaları

GNU C kütüphanesi, **malloc**, **realloc** ve **free** işlevlerinin davranışını, işlevle ilgili bir kanca işlev belirterek değiştirebilmenizi sağlar. Örneğin, özdevimli bellek ayırma kullanılan yazılımlarda hata ayıklamaya yardımcı olarak bu kancaları kullanabilirsiniz.

Kanca değikenler `malloc.h` başlık dosyasında tanımlanmıştır.

__malloc_hook değişken

Bu değer, her **malloc** çağrısında kullanılan işleve bir göstericidir. **malloc**'a benzeyecek bu işlevi siz tanımlayacaksınız; şöyle:

```
void *function (size_t boyut, const void *çağıran)
```

çağıran'ın değeri, **malloc** çağrıldığında yığıtta bulunan dönüş adresidir. Bu değer yazılımın bellek tüketimini izlemenizi mümkün kılar.

__realloc_hook değişken

Bu değer, her **realloc** çağrısında kullanılan işleve bir göstericidir. **realloc**'a benzeyecek bu işlevi siz tanımlayacaksınız; şöyle:

```
void *function (void *gösterici, size_t boyut, const void *çağıran)
```

çağıran'ın değeri, **realloc** çağrıldığında yığıtta bulunan dönüş adresidir. Bu değer yazılımın bellek tüketimini izlemenizi mümkün kılar.

__free_hook değişken

Bu değer, her **free** çağrısında kullanılan işleve bir göstericidir. **free**'ye benzeyecek bu işlevi siz tanımlayacaksınız; şöyle:

```
void function (void *gösterici, const void *çağıran)
```

çağıran'ın değeri, **free** çağrıldığında yığıtta bulunan dönüş adresidir. Bu değer yazılımın bellek tüketimini izlemenizi mümkün kılar.

__memalign_hook değişken

Bu değer, her **memalign** çağrısında kullanılan işleve bir göstericidir. **memalign**'a benzeyecek bu işlevi siz tanımlayacaksınız; şöyle:

```
void *function (size_t adımboyu, size_t boyut, const void *çağıran)
```

çağıran'ın değeri, **free** çağrıldığında yığıtta bulunan dönüş adresidir. Bu değer yazılımın bellek tüketimini izlemenizi mümkün kılar.

Dikkat

Bu işlevlerden birine bir kanca olarak tanımladığınız işlevin, öncelikle kancayı eski değerine döndürmeden bu işlevi ardışık olarak çağırmadığından emin olmalısınız. Aksi takdirde, yazılımınız sonsuz döngüye takılacaktır. İşlevi ardışık olarak çağırmadan önce tüm kancaları önceki değerlerine döndürdüğünüzden emin olmalısınız. Bir ardışık çağrıdan geri dönüldüğünde, bir kanca kendisini değiştirebileceğinden tüm kancalar yeniden kaydedilmelidir.

__malloc_initialize_hook değişken

Bu değişkenin değeri, malloc gerçekleştirmesi ilklendirildiğinde bir kere çağrılan bir işleve göstericidir. Bu zayıf bir değişkendir ve aşağıdaki gibi bir tanımlama ile yazılım tarafından değiştirilebilir:

```
void (*__malloc_initialize_hook) (void) = ilkendirme_kancam;
```

Bakmak için en iyi zaman, malloc kanca işlevlerinin güvenle kurulduğu zamandır. Kanca işlevler, malloc'la ilgili işlevleri ardışık çağırırsa, malloc'un **__malloc_hook**'a atandığı sırada kendisini düzgün olarak ilklendirmesi gerekir. Diğer yandan, kanca işlevler kendi malloc gerçeklemelerini sağlarlarsa, kancaların yazılımdaki ilk **malloc** çağrısı tamamlanmadan önce atanması yaşamsal önemdedir. Çünkü aksi takdirde, örneğin bir normal kancasız malloc'la sağlanan bir blok daha sonra bir **__free_hook** üzerinden elde edilmelidir.

Her iki durumda da sorun `__malloc_initialize_hook` tarafından gösterilen bir yazılımcı tanımlı işlev içinden ayarlanması ile çözümlenebilir ve böylece kancalar doğru zamanda güvenle ayarlanmış olur.

Aşağıdaki örnekte **__malloc_hook** ve **__free_hook** değişkenlerinin nasıl kullanıldığı gösterilmiştir. **malloc** veya **free** her çağrıldığında bir bilgi basan bir işlev kurulmaktadır. Bu örnekte, yazılımda **realloc** ve **memalign** işlevlerinin bulunmadığı varsayılmıştır.

```
/* __malloc_hook ve __free_hook için prototipler */
#include <malloc.h>

/* Kancalarımız için prototipler  */
static void my_init_hook (void);
static void *my_malloc_hook (size_t, const void *);
static void my_free_hook (void*, const void *);

/* C kütüphanesindeli ilklendirme kancasının üstüne yazalım. */
void (*__malloc_initialize_hook) (void) = my_init_hook;

static void my_init_hook (void)
{
  old_malloc_hook = __malloc_hook;
  old_free_hook = __free_hook;
  __malloc_hook = my_malloc_hook;
  __free_hook = my_free_hook;
}

static void * my_malloc_hook (size_t size, const void *caller)
{
  void *result;
```

```
  /* tüm eski kancaları önceki değerlerine getirelim */
  __malloc_hook = old_malloc_hook;
  __free_hook = old_free_hook;
  /* Ardışık çağıralım */
  result = malloc (size);
  /* belirlenen kancaları kaydedelim */
  old_malloc_hook = __malloc_hook;
  old_free_hook = __free_hook;
  /* printf, malloc'u  tekrar çağırmamalı. */
  printf ("malloc (%u), %p döndürdü\n", (unsigned int) size, result);
  /* Kendi kancalarımızı önceki değerlerine getirelim */
  __malloc_hook = my_malloc_hook;
  __free_hook = my_free_hook;
  return result;
}

static void
my_free_hook (void *ptr, const void *caller)
{
  /* tüm eski kancaları önceki değerlerine getirelim */
  __malloc_hook = old_malloc_hook;
  __free_hook = old_free_hook;
  /* Ardışık çağıralım */
  free (ptr);
  /* belirlenen kancaları kaydedelim */
  old_malloc_hook = __malloc_hook;
  old_free_hook = __free_hook;
  /* printf, free'yi  tekrar çağırmamalı. */
  printf ("gösterici %p serbest bırakıldı\n", ptr);
  /* Kendi kancalarımızı önceki değerlerine getirelim */
  __malloc_hook = my_malloc_hook;
  __free_hook = my_free_hook;
}

main ()
{
  …
}
```

mcheck (sayfa: 55) işlevi böyle kancalar kurularak çalışır.

2.2.11. `malloc` ile Bellek Ayırma İstatistikleri

Özdevimli bellek ayırma hakkında **mallinfo** işlevini kullanarak bilgi edinebilirsiniz. Bu işlev ve onunla ilişkili veri türü `malloc.h` başlık dosyasında bildirilmiştir. Standart SVID/XPG sürümünün oluşumlarıdır.

`struct` **mallinfo** — veri türü

Bu yapı özdevimli bellek ayırıcı hakkında bilgi döndürmek için kullanılır. Aşağıdaki üyelere sahiptir:

`int` **arena**
: **malloc** tarafından **sbrk** ile ayrılan belleğin bayt cinsinden toplam boyudur.

`int` **ordblks**
: Kullanılmayan parçaların sayısıdır. (Bellek ayırıcı dahili olarak işletim sistemindeki bellek parçalarını alır ve tek tek **malloc** çağrılarına dilimleyerek dağıtır. Bkz. *malloc için Yeterlik Kaygıları* (sayfa: 54).)

`int` **smblks**

Bu alan kullanılmamıştır.

`int` **hblks**
mmap ile ayrılan parçaların toplam sayısıdır.

`int` **hblkhd**
mmap ile ayrılan belleğin bayt cinsinden toplam boyudur.

`int` **usmblks**
Bu alan kullanılmamıştır.

`int` **fsmblks**
Bu alan kullanılmamıştır.

`int` **uordblks**
malloc tarafından alınmayan parçaların işgal ettiği toplam boyuttur.

`int` **fordblks**
Serbest bırakılan (kullanımda olmayan) parçaların işgal ettiği belleğin toplam boyudur.

`int` **keepcost**
Normalde yığın belleğin son sınırındaki (sanal adres alanının veri bölütünün yüksek ucu) en tepe dağıtılabilir parçanın boyudur.

`struct mallinfo` **mallinfo**`(void)` işlev

Bu işlev o anki özdevimli bellek kullanımı hakkında `struct mallinfo` yapı türünde bilgi döndürür.

2.2.12. `malloc` ile İlgili İşlevlerin Özeti

Burada **malloc** ile çalışan işlevlerin bir özeti bulunmaktadır:

`void *`**malloc**`(size_t` *boyut*`)`

boyut baytlık bir blok ayırır. Bkz. *Özdevimli Olarak Basit Bellek Ayırma* (sayfa: 50).

`void` **free**`(void *`*göstərici*`)`

malloc ile ayrılan belleği serbest bırakır. Bkz. *malloc ile Ayrılan Belleğin Serbest Bırakılması* (sayfa: 52).

```
void *realloc(void  *göstərici,
              size_t yeniboyut)
```

malloc ile ayrılan belleği büyütür ya da küçültür. Bunu bazan veriyi başka bir yere kopyalayarak yapar. Bkz. *Bir Bellek Bloğunun Boyutunun Değiştirilmesi* (sayfa: 52).

```
void *calloc(size_t elmsayısı,
             size_t elmboyu)
```

malloc kullanarak *elmsayısı* * *elmboyu* baytlık bir blok ayırır ve içini sıfırlarla doldurarak temizler. Bkz. *Temizlenmiş Bellek Ayırma* (sayfa: 53).

`void *`**valloc**`(size_t` *boyut*`)`

Sayfa boyunda adresleme ile *boyut* uzunlukta blok ayırır. Bkz. *Bellek Bloklarının Hizalanarak Ayrılması* (sayfa: 54).

```
void *memalign(size_t adımboyu,
               size_t boyut)
```

*adımboyu*nun katlarında adreslenen *boyut* baytlık bir blok ayırır. Bkz. *Bellek Bloklarının Hizalanarak Ayrılması* (sayfa: 54).

```
int mallopt(int param,
            int değer)
```

Ayarlanabilir parametreleri ayarlar. Bkz. *Ayarlanabilir Malloc Parametreleri* (sayfa: 55).

```
int mcheck(void (*çıkış_işlevi) (void))
```

malloc'a özdevimli ayrıan bellek üzerinde arasıra tutarlılık denetimi yapmasını ve bir tutarsızlık bulunursa *çıkış_işlevi*ni çağırmasını söyler. Bkz. *Yığın Bellek Tutarlılık Denetimi* (sayfa: 55).

```
void *(*__malloc_hook)(size_t       boyut,
                       const void *çağıran)
```

Çağrıldığında **malloc** kullanan bir işleve gösterici.

```
void *(*__realloc_hook)(void       *gösterici,
                        size_t      boyut,
                        const void *çağıran)
```

Çağrıldığında **realloc** kullanan bir işleve gösterici.

```
void (*__free_hook)(void       *ptr,
                    const void *çağıran)
```

Çağrıldığında **free** kullanan bir işleve gösterici.

```
void (*__memalign_hook)(size_t      boyut,
                        size_t      adımboyu,
                        const void *çağıran)
```

Çağrıldığında **memalign** kullanan bir işleve gösterici.

```
struct mallinfo mallinfo(void)
```

Özdevimli bellek kullanımı hakkında bilgi döndürür. Bkz. *malloc ile Bellek Ayırma İstatistikleri* (sayfa: 59).

2.3. Bellek Ayırmada Hata Ayıklama

Özdevimli bellek ayırmasında bellek artıklarının bulunması çöp toplayıcı kullanılmayan dillerle yazılım geliştirilirken oldukça karmaşık bir iştir. Uzun süre çalışan yazılımlar özdevimli olarak ayırdıkları nesnelerin işleri bittiğinde serbest bırakılmalarını sağlamalıdır. Aksi takdirde sistem belleğinin tükenmesi sözkonusu olabilir.

GNU C kütüphanesindeki **malloc** gerçeklemesi bu tür artıkların saptanması ve konumlarının bulunmasıyla ilgili bazı bilgiler sağlar. Bunu yapacak uygulama bir ortam değişkeni ile etkinleştirilen bir özel kipte başlatılmalıdır. Hata ayıklama kipi etkinleştirilirse yazılım için hiçbir hız cezası kesilmez.

2.3.1. İzleme işlevselliğinin kurulması

```
void mtrace(void)                                                    işlev
```

mtrace işlevi çağrıldığında **MALLOC_TRACE** isimli ortam değişkenine bakar. Bu değişkenin bir dosya ismi içereceği kabul edilir. Kullanıcının yazma izni olmalıdır. Dosya zaten mevcutsa içeriği silinir. Ortam değişkeni yoksa ya da değeri yazmak için açılabilir bir geçerli dosya ismi değilse hiçbir şey yapılmaz. **malloc** ve benzerlerinin davranışı değiştirilmez. Belli sebeplerle SUID veya SGID'li uygulamalar içinde bu durum geçerlidir.

İsimli dosya başarıyla açılırsa, **mtrace** işlevi **malloc**, **realloc** ve **free** işlevleri için *kancalar* (sayfa: 57) oluşturur. Bundan sonra bu işlevlerin tüm kullanımları izlenir ve dosyaya listelenir. İzlenen işlevlere yapılan tüm çağrılar için artık bir hız cezası vardır. Yani normal kullanım için izleme etkinleştirilmemelidir.

Bu işlev bir GNU oluşumudur ve genelde diğer sistemler için kullanılabilir değildir. İşlev `mcheck.h` başlık dosyasında bildirilmiştir.

`void` **muntrace**`(void)` işlev

muntrace işlevi **malloc** çağrılarının izlenmesini etkinleştiren **mtrace** çağrısından sonra çağrılabilir. Başarılı bir **mtrace** çağrısı yoksa **muntrace** hiçbir şey yapmaz.

Aksi takdirde **malloc**, **realloc** ve **free** işlevleri için oluşturulan *kancaları* (sayfa: 57) iptal eder ve izleme dosyasını kapatır. İzlenen bir çağrı kalmadığından yazılım tam gaz çalışır.

Bu işlev bir GNU oluşumudur ve genelde diğer sistemler için kullanılabilir değildir. İşlev `mcheck.h` başlık dosyasında bildirilmiştir.

2.3.2. Örnek Yazılım Parçaları

İzleme işlevselliği yazılımın çalışma anı davranışını açıklamasa bile tüm yazılımlarda **mtrace** çağrısı yapmak iyi fikir değildir. Olaya bir bütün olarak bakarsak, siz yazılımınızda **mtrace** kullanarak hata ayıklamak istediğinizde, sistemde çalışan tüm diğer yazılımların da **malloc** çağrılarını izleniyorsa, çıktı dosyası tüm yazılımlar için tek olacağından hata ayıklaması kullanışsız olacaktır. Bu nedenle, sadece hata ayıklama için derlenmişse tek **mtrace** çağrısı olmalı ve bir yazılım şöyle başlamalıdır:

```
#include <mcheck.h>

int main (int argc, char *argv[])
{
#ifdef DEBUGGING
  mtrace ();
#endif
  …
}
```

Yazılımın çalışması sırasında çağrıları izlemek isterseniz gereken tek şey budur. Ayrıca, izlemeyi istediğiniz bir anda **muntrace** ile durdurabilir ve yeni bir **mtrace** çağrısı ile izlemeyi yeniden başlatabilirsiniz. Ancak, çağrılmamış işlev çağrıları olabileceğinden bu istenmeyen sonuçlara yol açabilir.

Bilgi

İzleme işlevselliği sadece uygulama yazılımları için değildir, kütüphaneler de (hatta C kütüphanesinin kendisi de) bu işlevleri kullanabilir.

Bu durum, yazılım sonlanmadan önce **muntrace** çağrısı yapmanın neden iyi bir fikir olmadığı hakkında da bir fikir verir. Kütüphaneler sadece yazılımın **main** işlevinin dönüşü veya bir **exit** çağrısından sonra yazılımın sonlandırıldığı hakkında bilgilendirilir. Dolayısıyla bu olmadan kütüphaneler kullandıkları belleği serbest bırakamazlar.

En iyisi, yazılımın hemen başlarında bir kere **mtrace** çağrısı yapmak ve hiç **muntrace** çağrısı yapmamaktır. Bu durumda yazılımın hemen hemen tüm **malloc** işlevlerinin (yazılımın yapılandırıcıları tarafından çalıştırılanlar ile kütüphanelerin kullandıkları hariç) kullanımları izlenir.

2.3.3. Bellek Hata Ayıklaması için İpuçları

Durumu biliyorsunuz. Yazılım hata ayıklama için hazırlandı ve tüm hata ayıklama oturumları iyi çalışmakta. Ancak hata ayıklaması olmaksızın başlattığınızda hata gördünüz. Bir örnek olarak, hata ayıklamayı kapattığınızda bir bellek artığı görmüş olun. Böyle durumları sezebilirseniz kazanma şansınız olabilir. Aşağıdaki küçük yazılımda bazı şeylerin eşdeğerlerini basitçe kullanalım:

```
#include <mcheck.h>
#include <signal.h>

static void
enable (int sig)
{
  mtrace ();
  signal (SIGUSR1, enable);
}

static void
disable (int sig)
{
  muntrace ();
  signal (SIGUSR2, disable);
}

int
main (int argc, char *argv[])
{
  …

  signal (SIGUSR1, enable);
  signal (SIGUSR2, disable);

  …
}
```

Vs., yazılımcı ortamda **MALLOC_TRACE**'i ayarlayarak yazılımda bellek hata ayıklamasını başlatabilir. Çıktı şüphesiz ilk sinyalden önce oluşan ayırmaları göstermez ama bir bellek artığı varsa yine de gösterilecektir.

2.3.4. İzlerin Yorumlanması

Çıktıya bakarsanız buna benzeyecektir:

```
= Start
 [0x8048209] - 0x8064cc8
 [0x8048209] - 0x8064ce0
 [0x8048209] - 0x8064cf8
 [0x80481eb] + 0x8064c48 0x14
 [0x80481eb] + 0x8064c60 0x14
 [0x80481eb] + 0x8064c78 0x14
 [0x80481eb] + 0x8064c90 0x14
= End
```

İzleme dosyası bir insan tarafından okunup anlamlandırılmayacağı için bu satırların ne anlama geldiğinin bir önemi yoktur. Bu nedenle, okunabilirliğe dikkat edilmemiştir. Bu iş için GNU C kütüphanesi ile birlikte bu izleri yorumlayan ve kullanıcı dostu bir yolla özetleyen bir uygulama gelir. Bu uygulamanın ismi **mtrace**'dir (aslında bir Perl betiğidir) ve bir ya da iki argüman alır. İzleme çıktısının dosya ismi mutlaka belirtilmelidir. İsteğe bağlı olarak verilebilecek ikinci argüman izleme dosyasının isminden önce verilmesi gereken ve bu izi üreten yazılımın ismidir.

```
drepper$ mtrace tst-mtrace log
No memory leaks.
```

Burada `tst-mtrace` isimli yazılım çalıştırılmış ve `log` isimli bir izleme dosyası üretilmiştir. **mtrace** uygulaması tarafından basılan ileti kodla ilgili bir sorun olmadığını göstermiş ve ardından ayrılan tüm bellek serbest bırakılmıştır.

Biz **mtrace** uygulamasını yukardaki örnek izleme dosyası için çalıştırırsak farklı bir çıktı alırız:

```
drepper$ mtrace errlog
- 0x08064cc8 Free 2 was never alloc'd 0x8048209
- 0x08064ce0 Free 3 was never alloc'd 0x8048209
- 0x08064cf8 Free 4 was never alloc'd 0x8048209

Memory not freed:
-----------------
   Address     Size     Caller
0x08064c48     0x14  at 0x80481eb
0x08064c60     0x14  at 0x80481eb
0x08064c78     0x14  at 0x80481eb
0x08064c90     0x14  at 0x80481eb
```

mtrace uygulamasını tek argümanla çağırdık ve bu durumda betik izlerde verilen adresleri anlamlandıramadı. Daha anlamlı bir çıktı için şöyle bir çağrı yapabiliriz:

```
drepper$ mtrace tst errlog
- 0x08064cc8 Free 2 was never alloc'd /home/drepper/tst.c:39
- 0x08064ce0 Free 3 was never alloc'd /home/drepper/tst.c:39
- 0x08064cf8 Free 4 was never alloc'd /home/drepper/tst.c:39

Memory not freed:
-----------------
   Address     Size     Caller
0x08064c48     0x14  at /home/drepper/tst.c:33
0x08064c60     0x14  at /home/drepper/tst.c:33
0x08064c78     0x14  at /home/drepper/tst.c:33
0x08064c90     0x14  at /home/drepper/tst.c:33
```

Ansızın, çıktı daha ayrıntılı oluverdi. Yazılımcı artık hangi işlev çağrısının bozukluğa sebep olduğunu hemen görebilir.

Bu çıktının yorumlanması karmaşık değildir. En fazla iki farklı durum saptanmıştır. İlkinde, ayırma işlevlerinden döndürülmemiş göstericiler için **free** çağrılmış. Bu genelde çok kötü bir sorundur ve çıktının ilk üç satırında benzer bir durum gösterilmiştir. Bu gibi durumlar az görülür, çünkü sorun kendini çok dramatik olarak gösterir: yazılım normal olarak çöker.

Diğer durum saptanması daha zor olan bellek artıklarıdır. Çıktıda gördüğünüz gibi **mtrace** işlevi tüm bu bilgiyi toplamıştır. Bu çıktıya bakarak şunu söyleyebiliriz: yazılım **/home/drepper/tst-mtrace.c** kaynak dosyasının 33. satırındaki bir ayırma işlevini dört kere çağırmış ve yazılım sonlandırılmadan önce bu bellek serbest bırakılmamış. Acaba bu araştırılmayı bekleyen gerçek bir sorun mudur.

2.4. Yığınaklar (Obstacks)

Bir ***yığınak*** bir nesne yığını içeren bir bellek havuzudur. Çok sayıda ayrı yığınak oluşturabilir ve nesneleri bunların içinde ayırabilirsiniz. Her yığınağın içinde ayrılan son nesneye karşılık daima ilki serbest bırakılır ancak her yığınak diğerinden bağımsızdır.

Serbest bırakma ile ilgili bu kısıtlama dışında yığınaklar tamamen geneldir: bir yığınak farklı boyutlarda olabilen çok sayıda nesne içerebilir. Yığınaklar makrolarla gerçeklenmiştir. Böylece, nesneler küçük oldukları sürece ayırma işlemi çok hızlı yapılır. Her nesneyi belli bir adım aralığı ile oluşturmak gerektiğinden her nesne için sabit yer ayrılır.

2.4.1. Yığınak Oluşturma

Yığınaklar üzerinde işlem yapmak için gereken araçlar `obstack.h` başlık dosyasında bildirilmiştir.

`struct` **`obstack`** veri türü

> Bir yığınak **`struct obstack`** türünde bir veri yapısı ile ifade edilir. Bu veri yapısı çok küçük ve sabit boyutludur. Nesnelerin içinde ayrıldıkları alanın nasıl bulunacağı ve yığınağın durumu hakkında bilgileri kaydeder. Nesnelerin kendileri hakkında bir bilgi içermez. Bu yapının içeriğine doğrudan erişmeye çalışmayın; sadece bu bölümde açıklanmış olan işlevleri kullanın.

Değişkenleri **`struct obstack`** türünde bildirebilir ve onları yığınak olarak kullanabilirsiniz ya da bir yığınağı diğer nesne çeşitlerinde olduğu gibi özdevimli olarak ayırabilirsiniz. Yığınakların özdevimli olarak ayrılması yazılımınızın çok sayıda farklı yığınağa sahip olmasını mümkün kılar. (Hatta bir yığınak yapısını bir başka yığınağın içinde ayırabilirsiniz ancak bu pek kullanışlı olmaz.)

Yığınaklar ile çalışan işlevlerin tümü hangi yığınağın kullanılacağının belirtilmesini gerektirir. Bunu **`struct obstack *`** türünden bir gösterici ile yapabilirsiniz. Bundan sonra bir yığınaktan bahsettiğimizde ona bir gösterici üzerinden eriştiğiniz kabul edilecektir.

Yığınaktaki nesneler ***tomar*** olarak adlandırılan büyük bloklar halindedir. **`struct obstack`** yapısı ise bu tomarlardan oluşan bir zincirin kaydını tutar.

Yığınak kütüphanesi, önceki tomara sığmayan bir nesne ayırdığınızda, yeni bir tomar oluşturur. Yığınak kütüphanesi tomarları özdevinimli olarak yönettiğinden bu işlemlere fazla dikkat etmeniz gerekmez. Ancak, yığınak kütüphanesinin bir tomarı ayırırken kullanacağı bir işlevi sağlamanız gerekir. Bu işlev **`malloc`** işlevini doğrudan ya da dolaylı kullanmalıdır. Ayrıca bir tomarı serbest bırakacak bir işlev daha sağlamanız gerekir. Bu konular aşağıdaki bölümde açıklanmıştır.

2.4.2. Yığınakları Kullanıma Hazırlama

Yığınak işlevlerini kullanmayı tasarladığınız her kaynak dosyası `obstack.h`[B134] başlık dosyasını aşağıdaki gibi içermelidir:

```
#include <obstack.h>
```

Ayrıca, kaynak dosyası **`obstack_init`** makrosunu da kullanıyorsa, yığınak kütüphanesi tarafından kullanılmak üzere iki işlev veya makro bildirmeli ya da tanımlamalıdır. Biri, içine nesnelerin paketleneceği bellek tomarlarını ayırmada kullanılacak olan **`obstack_chunk_alloc`** işlevi diğeri ise, içindeki nesneler serbest bırakıldığında bellek tomarlarını sisteme döndürecek olan **`obstack_chunk_free`** işlevidir. Bu makrolar kaynak dosyaları içinde yığınaklar kullanılmadan önce tanımlanmış olmalıdır.

Bunlar **`xmalloc`** *Özgür Bellek Ayırma* (sayfa: 50)) araortamı üzerinden **`malloc`** kullanacak şekilde tanımlanır. Bu tanımlama aşağıdaki makro tanım çifti ile yapılır:

```
#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free
```

Yığınak kullanarak ayırdığınız bellek gerçekte **malloc**'dan gelmesine karşın, yığınakların kullanılması daha hızlıdır çünkü daha büyük bellek blokları halinde olduğundan **malloc** daha az sıklıkta çağrılır. Daha ayrıntılı bilgi için *Yığınak Tomarları* (sayfa: 72) bölümüne bakınız.

Çalışma anında yazılım, yığınak olarak bir **struct obstack** yapısını kullanmaya başlamadan önce bir **obstack_init** çağrısı ile yığını ilklendirmelidir.

```
int obstack_init(struct obstack *yığınak)                                    işlev
```

Nesnelerin ayrılacağı *yığınak* nesnesini ilklendirir. Bu işlev yığınağın **obstack_chunk_alloc** işlevini çağırır. Eğer bellek ayırma işlemi başarısız olursa **obstack_alloc_failed_handler** tarafından gösterilen işlev çağrılır. **obstack_init** işlevi daima 1 ile döner.

Uyumluluk bilgisi

Yığınak kütüphanelerinin daha şekilci sürümleri bu işlev başarısız olduğunda 0 ile döner.

Aşağıda bir yığınak için alan ayrılması ve onun ilklendirilmesi üzerine iki örnek bulunmaktadır. İlkinde yığınak bir durağan değişkendir:

```
static struct obstack myobstack;
…
obstack_init (&myobstack);
```

İkincisinde ise yığınak kendisini özdevimli ayırmaktadır:

```
struct obstack *myobstack_ptr
  = (struct obstack *) xmalloc (sizeof (struct obstack));

obstack_init (myobstack_ptr);
```

```
obstack_alloc_failed_handler                                                 değişken
```

Bu değişkenin değeri, **obstack_chunk_alloc** işlevi bellek ayırma sırasında başarısız olduğunda çağıracağı işleve bir göstericidir. Bu göstericiye belirteceğiniz işlev ya **exit** (*Yazılımın Sonlandırılması* (sayfa: 681)) ya da **longjmp** (*Yerel Olmayan Çıkışlar* (sayfa: 593)) çağırmalı, yani dönmemelidir.

```
void my_obstack_alloc_failed (void)
…
obstack_alloc_failed_handler = &my_obstack_alloc_failed;
```

2.4.3. Bir Yığınağa Nesne Eklenmesi

Bir nesne için bir yığınakta yer ayırmanın en kestirme yolu, **malloc** çağırır gibi **obstack_alloc** çağrısı yapmaktır.

```
void *obstack_alloc(struct obstack *yığınak,                                 işlev
                    int             boyut)
```

İşlev, bir yığınakta *boyut* baytlık bir ilkendirilmemiş blok ayırır ve onun adresi ile döner. Burada *yığınak*, içinde blok ayrılacak olan yığınağı belirten **struct obstack** yapısının adresidir. Her yığınak işlevi ya da makrosu ilk argüman olarak daima bir ** yığınak* belirtmeyi gerektirir.

Bellekte yeni bir tomar ayrılması gerekirse bu işlev yığınağın **obstack_chunk_alloc** işlevini çağırır. **obstack_chunk_alloc** işlevi bellek ayırmada başarısız olursa **obstack_alloc_failed_handler** işlevi çağrılır.

Aşağıdaki örnekte, **string_obstack** yığınağında *string* dizgesinin bir kopyası için yer ayrılmaktadır:

```
struct obstack string_obstack;

char * copystring (char *string)
{
  size_t len = strlen (string) + 1;
  char *s = (char *) obstack_alloc (&string_obstack, len);
  memcpy (s, string, len);
  return s;
}
```

İçeriğini belirterek bir blok ayırmak için **obstack_copy** işlevi kullanılır:

```
void *obstack_copy(struct obstack *yığınak,                    işlev
                   void           *adres,
                   int             boyut)
```

Bu işlev bir blok ayırır ve *adres* adresinden başlayan *boyut* baytlık veriyi ayrılan yere kopyalayarak bloku ilklendirir. **obstack_chunk_alloc** tarafından bellek ayırma işlemi başarısız olursa **obstack_alloc_failed_handler** işlevini çağırır.

```
void *obstack_copy0(struct obstack *yığınak,                   işlev
                    void           *adres,
                    int             boyut)
```

obstack_copy işlevi gibidir, farklı olarak, bir boş karakter ekler. Bu fazladan bayt *boyut*'a dahil değildir.

Bu işlev, bir yığınağa bir dizgeyi bir boş sonlandırmalı dizge olarak kopyalamanın en uygun yoludur. Aşağıdaki bir kullanım örneğine bakınız:

```
char * obstack_savestring (char *addr, int size)
{
  return obstack_copy0 (&myobstack, addr, size);
}
```

Önceki örnekte aynı işi yapan **copystring** işlevi için **malloc** (*Özdevimli Olarak Basit Bellek Ayırma* (sayfa: 50)) kullanılmıştı.

2.4.4. Bir Yığınaktan Nesne Çıkarılması

Yığınaktan bir nesneyi çıkarmak için **obstack_free** işlevi kullanılır. Bir yığınak aslında bir nesne yığıtı olduğundan, serbest bırakılan nesne kendinden sonra ayrılan diğer nesnelerinde serbest bırakılmasına sebep olur.

```
void obstack_free(struct obstack *yığınak,                     işlev
                  void           *nesne)
```

nesne hiçbir şey göstermiyorsa yığınaktan tüm nesneler çıkarılır. Aksi takdirde, gösterdiği nesne ve o nesneden sonra yığınağa konulmuş tüm nesneler serbest bırakılır.

Bilgi

nesne hiçbir şey göstermiyorsa işlevin sonucu ilklendirilmemiş bir yığınaktır. Bir yığınağı boşaltmak ve onu kullanılabilir yapmak isterseniz **obstack_free** işlevini yığınaktaki ilk nesnenin göstericisi ile çağırmalısınız:

```
obstack_free (obstack_ptr, first_object_allocated_ptr);
```

Bir yığınaktaki nesneler tomarlar halinde gruplanır. Bir tomardan tüm nesneler çıkarıldığında yığınak kütüphanesi tomarı özdevinimli olarak serbest bırakır (Bkz. *Yığınakları Kullanıma Hazırlama* (sayfa: 65)). Böylece diğer yığınaklar ve bellek ayırma işlemleri için tomar alanı yeniden kullanılabilir.

2.4.5. Yığınak İşlevleri ve Makroları

Derleyiciye bağlı olarak, yığınakların kullanılmasını sağlayan arayüzler bir işlev ya da makro olarak tanımlanabilir. Yığınak oluşumu ISO C ve geleneksel C içeren tüm C derleyicileri ile çalışır, ancak GNU C derleyicisi dışındaki derleyicileri kullanacaksanız almanız gereken bazı önlemler vardır.

Eski moda, ISO C olmayan bir derleyici kullanıyorsanız, tüm yığınak işlevleri aslında birer makro olarak tanımlanmalıdır. Bu makroları birer işlev gibi çağırabilirsiniz ama bir işlevin sağladığı diğer kolaylıklardan (işlev adresini almak gibi) yararlanamazsınız.

Makroların çağrılması bir özel önlem almayı gerektirir: isim olarak, ilk terim (yığınak göstericisi) herhangi bir yan etki barındırmamalıdır. Örneğin,

```
obstack_alloc (get_obstack (), 4);
```

yazarsanız, **get_obstack** defalarca çağrılacaktır. Ya da yığınak göstericisi argümanı olarak ***obstack_list_ptr++** kullanırsanız, defalarca arttırım uygulanacağından çok tuhaf sonuçlar elde edebilirsiniz.

ISO C'de her işlevin bir makro bir de işlev olarak tanımı vardır. Bir işlevi çağırmadan sadece adresini almak isterseniz işlev tanımı kullanılır. Normal bir işlev çağrısında ise öntanımlı olarak makro tanımı kullanılır, ancak isterseniz işlev ismini parantez içinde kullanarak işlev tanımını kullanabilirsiniz. Örnek:

```
char *x;
void *(*funcp) ();

/* Makro kullanımı.  */
x = (char *) obstack_alloc (ynk_gstr, boyut);

/* İşlev çağrısı.  */
x = (char *) (obstack_alloc) (ynk_gstr, boyut);

/* İşlev adresinin alınması.  */
funcp = obstack_alloc;
```

Aynı durum ISO C'deki standart kütüphane işlevleri için de geçerlidir. Bkz. *Makro Olarak Tanımlanmış İşlevler* (sayfa: 23).

Uyarı

ISO C'de bile, makroları kullanırken ilk terimin yan etkiler oluşturmasına karşı önlem almanız gereklidir.

GNU C derleyicisini kullanıyorsanız, bu önlem gereksizdir, çünkü GNU C'deki çeşitli dil oluşumları, makroların, her argümanının bir kere hesaplanmasını sağlayacak şekilde tanımlanmasına izin verir.

2.4.6. Büyüyen Nesneler

Yığınak tomarları içindeki bellek sıralı erişimle kullanıldığından, bir nesneyi, nesne için ayrılan alanın sonuna bayt ya da baytlar ekleyerek adım adım oluşturmak mümkündür. Bu teknikle, bir nesneye veri aktarırken acaba yer kaldı mı diye düşünmeniz gerekmez. Bu tekniğe ***Nesnelerin Büyütülmesi*** diyoruz. Bu bölümde nesneleri sonuna veri ekleyerek büyütmek için kullanılan özel işlevlere yer verilmiştir.

Bu işlevlerden birini kullanarak nesneye veri eklemek istediğinizde nesnenin büyütülmesi özdevinimli olarak gerçekleştiğinden, bir nesneyi büyütmeye başladığınızda özel olarak hiçbir şey yapmanız gerekmez. Ancak büyütme işlemini bittiğinde, bittiğinin belirtilmesi gerekir. Bunun için **obstack_finish** işlevi kullanılır.

Büyütme işleminin bittiği belirtilene kadar her veri eklemesinde nesne yeni bir tomara kopyalandığı için nesnenin büyütülmesi bitene kadar nesnenin gerçek adresi bilinmez.

Bir yığınak bir nesneyi büyütmek için kullanıldığında, bir başka nesneyi o yığına yerleştiremezsiniz. Bunu yapmayı denerseniz, eklediğiniz nesneye ayrı bir nesne olarak erişilemez; büyütülen nesneye bir alan eklenmiş olur ve eklediğiniz nesne büyütülen nesnenin bir parçası haline gelir.

```
void obstack_blank(struct obstack *yığınak,                    işlev
                   int              boyut)
```

Bir büyüyen nesneye alan oluşturmak için kullanılır. **obstack_blank** ilklendirilmemiş bir alan oluşturur.

```
void obstack_grow(struct obstack *yığınak,                     işlev
                  void           *veri,
                  int             boyut)
```

İlklendirilmiş bir blok eklemekte kullanılır. **obstack_grow** işlevi nesne büyüten bir **obstack_copy** işlevi gibi düşünülebilir. Büyüyen nesneye *boyut* uzunlukta *veri* ekler.

```
void obstack_grow0(struct obstack *yığınak,                    işlev
                   void           *veri,
                   int             boyut)
```

Nesne büyüten bir **obstack_copy0** işlevi gibi düşünülenilir. Büyüyen nesneye *boyut* uzunlukta *veri* sonuna bir boş karakter getirilerek eklenir.

```
void obstack_1grow(struct obstack *yığınak,                    işlev
                   char            c)
```

obstack_1grow işlevi büyüyen nesneye bir defada sadece bir karakter eklemek için kullanılır. Büyüyen nesneye *c* karakterini içeren bir bayt ekler.

```
void obstack_ptr_grow(struct obstack *yığınak,                 işlev
                      void           *veri)
```

Bir oluşumun gösterici değerini büyüyen nesneye eklemek için kullanılır. Büyüyen nesneye *veri* nin değerini içeren **sizeof (void *)** bayt ekler.

```
void obstack_int_grow(struct obstack *yığınak,                 işlev
                      int             veri)
```

Büyüyen nesneye **int** türünde tek bir değer eklemek için kullanılır. Büyüyen nesneye **sizeof (int)** bayt ekler ve *veri*'nin değeri ile ilklendirir.

```
void *obstack_finish(struct obstack *yığınak)                  işlev
```

Bir büyüyen nesne ile işiniz bittiğinde onu kapatmak ve son adresini almak için bu işlevi kullanacaksınız.

Büyüyen bir nesneyi bitirdiğinizde, yığınak, normal nesne ayırma ya da yeni bir büyüyen nesne oluşturmak için kullanılabilir.

Bu işlev **obstack_alloc** işlevi gibi aynı koşullar altında bir boş gösterici ile de dönebilir. Bkz. *Bir Yığınağa Nesne Eklenmesi* (sayfa: 66).

Bir büyüyen nesne ile çalışırken ne büyüklüğe ulaştığını bilmek isteyebilirsiniz. Bir büyüyen nesneyi bitirmeden **obstack_object_size** işlevini kullanarak yığınağın eriştiği boyutu öğrenebilirsiniz:

```
int obstack_object_size(struct obstack *yığınak)                    işlev
```

Bu işlev, büyüyen nesnenin o andaki bayt cinsinden uzunluğu ile döner. Bu işlevin nesneyi bitirmeden çağrılabileceğini unutmayın. Nesneyi bitirdikten sonra **obstack_object_size** işlevi sıfır değeri ile döner.

Bir büyüyen nesne oluşturduktan sonra onu iptal etmek isterseniz, önce onu bitirin sonra da serbest bırakın. Örnek:

```
obstack_free (obstack_ptr, obstack_finish (obstack_ptr));
```

Nesne bir büyüyen nesne değilse bu etkisizdir.

Bir nesneyi küçültmek için **obstack_blank** işlevini bir negatif boyutla çağırabilirsiniz. Ancak nesneyi küçültürken boyunun sıfırın altına inmemesine dikkat edin, aksi takdirde neler olacağı bilinemez.

2.4.7. Çok Hızlı Büyüyen Nesneler

Büyüyen nesneleri büyütürken o anki tomarın yeterli yere sahip olup olmadığının denetlenmesi için işlevler içinde yapılan işlemler bir sürü masraf kalemi oluşturur. Üstelik siz bu nesneleri küçük küçük adımlarla ve sıklıkla büyütürseniz bu israf kaydadeğer olmaya başlar.

Bu masrafı, bu denetimlerin yapılmadığı özel "hızlı büyütme" işlevlerini kullanarak düşürebilirsiniz. Sağlam bir yazılım istiyorsanız bu denetimleri kendiniz yapmalısınız. Bu denetimi en basit yolla, nesneye veri eklemeye hazırlanırken yaparsanız hiçbir şeyi saklamak zorunda kalmazsınız, çünkü normal büyütme işlevleri de bunu zaten böyle yapar. Bu düzenlemeyi ne kadar az sıklıkta yaparsanız yazılımınız o kadar hızlı olur.

obstack_room işlevi kullanılan tomar içindeki kullanılabilir alanın miktarı ile döner:

```
int obstack_room(struct obstack *yığınak)                           işlev
```

Bu işlev, kullanılmakta olan (ya da yeni oluşturulan) *yığınak* büyüyen nesnesine yeni bir tomar oluşturmadan hızlı büyütme işlevleri kullanılarak eklenebilecek baytların sayısı ile döner.

Ne kadar yeriniz olduğunu bildiğinize göre aşağıdaki hızlı büyütme işlevlerini kullanarak büyüyen nesnenize güvenle veri ekleyebilirsiniz:

```
void obstack_1grow_fast(struct obstack *yığınak,                    işlev
                        char           c)
```

yığınak'daki büyüyen nesneye *c* karakterini içeren tek bir bayt eklemekte kullanılır.

```
void obstack_ptr_grow_fast(struct obstack *yığınak,                 işlev
                           void           *veri)
```

yığınak'daki büyüyen nesneye *veri* değerini içeren **sizeof (void *)** bayt ekler.

```
void obstack_int_grow_fast(struct obstack *yığınak,                 işlev
                           int            veri)
```

yığınak'daki büyüyen nesneye *veri* değerini içeren **sizeof (int)** bayt ekler.

```
void obstack_blank_fast(struct obstack *yığınak,                    işlev
                        int             boyut)
```

yığınak'daki büyüyen nesneye ilklendirmeksizin *boyut* bayt ekler.

Kalan yeri **obstack_room** ile tespit ettiğinizde yeterli yer kalmadığını öğrenirseniz hızlı büyütme işlevlerini kullanmak artık güvenilir olmaz. Bu durmda, basitçe normal büyütme işlevlerine geçmek gerekir. Sonuç olarak, az yer kalmışsa normal büyütme işlevleri yeni bir tomar açar ve böylece bol bol yeriniz olur.

Demek ki, **obstack_room** ile yeterli yer kalmadığını her anladığınızda normal büyütme işlevlerini kullanabilecek ve nesneniz yeni bir tomara kopyalandığında tekrar hızlı büyütme işlevlerini kullanabileceğiniz güvenilir bir alana sahip olabileceksiniz.

Burada bir örnek var:

```
void dizge_ekle (struct obstack *yiginak, const char *dizge, int uzunluk)
{
  while (uzunluk > 0)
  {
    int bosyer = obstack_room (yiginak);
    if (bosyer == 0)
    {
      /* Yeterli boş yer yok. Bir karakter eklersek nesnemiz
         yeni bir tomara kopyalacak ve boş yer olacak. */
      obstack_1grow (yiginak, *dizge++);
      uzunluk--;
    }
    else
    {
      if (bosyer > uzunluk)
        bosyer = uzunluk;

      /* Boş yer olduğuna göre hızlı olarak ekleme yapabiliriz. */
      uzunluk -= bosyer;
      while (bosyer-- > 0)
        obstack_1grow_fast (yiginak, *dizge++);
    }
  }
}
```

2.4.8. Bir Yığınağın Durumu

Burada açıklanan işlevler kullanımdaki bir yığınağın durumu hakkında bilgi verirler. Buradaki işlevleri büyümekte olan nesneler hakkında bilgi edinmek için de kullanabilirsiniz.

```
void *obstack_base(struct obstack *yığınak)                          işlev
```

Bu işlev *yığınak*'da büyüyen nesnenin başlangıçta geçici olarak kullanılan adresi ile döner. Bu işlevin hemen ardından büyüyen nesne bitirilirse bu adres dönecektir. Ama bitirme işlemini nesneyi fazlaca büyüttükten sonra yaparsanız dönen adres farklı olabilecektir.

Büyüyen bir nesne yoksa, sonraki nesnenin başlangıç adresi ile döner (tomarın yeni bir nesne için yeterli olduğu varsayımıyla).

```
void *obstack_next_free(struct obstack *yığınak)                     işlev
```

yığınak'ın bulunduğu tomardaki ilk serbest baytın adresi ile döner. Bu adres, büyümekte olan nesnenin sonundadır. Eğer nesne bir büyüyen nesne değilse, **obstack_base** ile dönen adresin aynısı döner.

`int` **obstack_object_size**`(struct obstack *`*yığınak*`)` işlev

Büyümekte olan nesnenin uzunluğu ile döner. Bu değer aşağıdaki ifade ile elde edilen değere eşittir:

```
obstack_next_free (yiginak) - obstack_base (yiginak)
```

2.4.9. Yığınaktaki Verinin Adreslenmesi

Her yığınak bir ***adres boyu***na sahiptir. Yığınağa konulan ilk nesne özdevimli olarak bir adrese konur ve her nesne bu adresin belirtilen adres boyunun katlarındaki adreslere yerleştirilir. Öntanımlı olarak bu adres boyu her veri türünü tutabilecek bir nesneye gereken kadardır.

Adres boyunu öğrenmek için **obstack_alignment_mask** işlevi kullanılır:

`int` **obstack_alignment_mask**`(struct obstack *`*yığınak*`)` makro

Dönen değer bir bit maskesidir; değeri 1 olan bir bit, nesnenin adresinin ilgili bitinin 0 olması gerektiğini belirtir. Maske değeri, nesneler ikinin kuvvetlerindeki adres boylarında adreslendiğinden, ikinin kuvveti eksi birdir. Öntanımlı maske değeri hizalı nesnelerin her veri türünü tutmasını mümkün kılacak bir değerdir: örneğin, maske değeri 3 ise adresleri 4'ün katları olan her veri türü saklanabilir. Maske değeri sıfırsa adresleme birer baytlıktır ve bu durumda adresleme yoktur denir.

obstack_alignment_mask makrosu bir sol taraf değeri olarak yorumlanır, yani maskı siz değiştirebilirsiniz. Örnek:

```
obstack_alignment_mask (yiginak) = 0;
```

Burada, belirtilen yığınakta adresleme işlemi kapatılmıştır.

Bilgi

Adresleme boyu değişikliği bir nesne ayrıldıktan sonra ya da bitirildiğinde etkisizdir. Bir nesneyi hiç büyütmeden yeni bir adres boyu belirtip nesneyi **obstack_finish** ile sıfır uzunlukta bitirip sonraki nesneler için yeni adres boyunu geçerli yapabilirsiniz.

2.4.10. Yığınak Tomarları

Yığınaklar kendilerini büyük tomarlar halinde yer ayırırlar ve bu alanı sizin isteklerinizi karşılamak üzere parsellerler. Başka bir tomar uzunluğu belirtilmediği sürece öntanımlı tomar uzunluğu 4096 bayttır. Tomarın 8 baytlık bölümü nesneleri saklamakta kullanılmaz. Bir tomar uzunluğu belirtilmiş bile olsa, büyük nesneler için gerekirse daha büyük uzunlukta tomarlar ayrılabilir.

Yığınak kütüphanesi tomarları sizin tanımlayacağınız **obstack_chunk_alloc** işlevini çağırarak ayırır. İçindeki tüm nesnelerin çıkarılmasıyla boşalan bir tomarı yine sizin tanımlayacağınız **obstack_chunk_free** işlevini çağırarak serbest bırakır.

Bu iki işlev, **obstack_init** (sayfa: 65) kullanılan her kaynak dosyasında ya makro olarak tanımlanmalı ya da işlev olarak bildirilmelidir. Bunlar makro olarak çoğunlukla aşağıdaki gibi tanımlanır:

```
#define obstack_chunk_alloc malloc
#define obstack_chunk_free free
```

Bunlar basit makrolardır (argümansız). Argümanlı makro tanımları çalışmayacaktır. Bu, **obstack_chunk_alloc** veya **obstack_chunk_free** nin, tek başına kendisi bir işlev değilse bir işlev ismi olarak kullanılacağından böyledir.

Tomarları **malloc** ile ayırırsanızi tomar boyu ikinin kuvvetlerinden biri olmalıdır. Öntanımlı tomar boyu 4096 bayttır. Bu boyut, son tomarda kullanılmadan çok fazla bellek kalmasına yol açmayacak kadar küçük, sıradan isteklere yanıt verecek kadar da büyük bir bellek boyudur.

```
int obstack_chunk_size(struct obstack *yığınak)                    makro
```

Belirtilen yığınağın tomar boyu ile döner.

Bu makro bir sol taraf değeri olarak yorumlandığından ona yeni bir değer atayarak farklı bir tomar boyu belirtebilirsiniz. Bunu yapılması, mevcut ayrılmış tomarları etkilemez, sadece bu atamadan sonra ayrılan tomarlar etkilenir. Bir tomarı bu yöntemle daha küçük boylarda ayırmak mümkünse de, çok fazla nesne kullanacaksanız daha büyük tomar boyları daha verimli olacaktır. Örnek:

```
if (obstack_chunk_size (yiginak-gstr) < yeni–tomar–boyu)
  obstack_chunk_size (yiginak-gstr) = yeni–tomar–boyu;
```

2.4.11. Yığınak İşlevlerinin Listesi

Burada yığınaklarla ilgili işlevlerin bir özet listesine yer verilmiştir. Her biri ilk argüman olarak **struct obstack *** türünde bir yığınak adresi alır.

```
void obstack_init(struct obstack *yığınak)                         işlev
```

Bir yığınağı ilklendirir. Bkz. *Yığınak Oluşturma* (sayfa: 65).

```
void *obstack_alloc(struct obstack *yığınak,                       işlev
                    int             boyut)
```

İlklendirmeden *boyut* baytlık bir nesneye yer ayrılır. Bkz. *Bir Yığınağa Nesne Eklenmesi* (sayfa: 66).

```
void *obstack_copy(struct obstack *yığınak,                        işlev
                   void          *adres,
                   int            boyut)
```

İçeriğini *adres*ten kopyalayarak *boyut* baytlık bir nesneye yer ayırır. Bkz. *Bir Yığınağa Nesne Eklenmesi* (sayfa: 66).

```
void *obstack_copy0(struct obstack *yığınak,                       işlev
                    void          *adres,
                    int            boyut)
```

İçeriğini *adres*ten kopyalayarak *boyut* baytlık bir nesneye sonuna bir boş karakter ekleyerek yer ayırır. Bkz. *Bir Yığınağa Nesne Eklenmesi* (sayfa: 66).

```
void obstack_free(struct obstack *yığınak,                         işlev
                  void          *nesne)
```

nesne ve *nesne*den sonra ayrılan tüm nesneleri yığınaktan çıkarır (serbest bırakır). Bkz. *Bir Yığınaktan Nesne Çıkarılması* (sayfa: 67).

```
void obstack_blank(struct obstack *yığınak,                        işlev
                   int            boyut)
```

Büyüyen nesneye ilklendirilmemiş *boyut* bayt ekler. Bkz. *Büyüyen Nesneler* (sayfa: 69).

```
void obstack_grow(struct obstack *yığınak,                         işlev
                  void          *adres,
                  int            boyut)
```

Büyüyen nesneye içeriğini *adres*ten kopyalarak *boyut* bayt ekler. Bkz. *Büyüyen Nesneler* (sayfa: 69).

```
void obstack_grow0(struct obstack *yığınak,                    işlev
                   void           *adres,
                   int             boyut)
```

Büyüyen nesneye içeriğini *adres*ten kopyalarak *boyut* bayt artı bir boş karakter ekler. Bkz. *Büyüyen Nesneler* (sayfa: 69).

```
void obstack_1grow(struct obstack *yığınak,                    işlev
                   char            karakter)
```

Büyüyen nesneye *karakter* içeren bir bayt ekler. Bkz. *Büyüyen Nesneler* (sayfa: 69).

```
void *obstack_finish(struct obstack *yığınak)                  işlev
```

Büyüyen nesneyi kapatarak (bitirmek) son adresi ile döner. Bkz. *Büyüyen Nesneler* (sayfa: 69).

```
int obstack_object_size(struct obstack *yığınak)               işlev
```

Büyüyen nesnenin o andaki boyu ile döner. Bkz. *Büyüyen Nesneler* (sayfa: 69).

```
void obstack_blank_fast(struct obstack *yığınak,               işlev
                        int             boyut)
```

Yeterli yer olup olmadığını denetlemeksizin büyüyen nesneye ilklendirmeden *boyut* bayt ekler. Bkz. *Çok Hızlı Büyüyen Nesneler* (sayfa: 70).

```
void obstack_1grow_fast(struct obstack *yığınak,               işlev
                        char            karakter)
```

Yeterli yer olup olmadığını denetlemeksizin büyüyen nesneye *karakter* içeren bir bayt ekler. Bkz. *Çok Hızlı Büyüyen Nesneler* (sayfa: 70).

```
int obstack_room(struct obstack *yığınak)                      işlev
```

Büyüyen nesnede o andaki kullanılabilir alanın miktarı ile döner. Bkz. *Çok Hızlı Büyüyen Nesneler* (sayfa: 70).

```
int obstack_alignment_mask(struct obstack *yığınak)            işlev
```

Bir nesnenin başlangıcını hizalamakta kullanılan maske ile döner. Bu bir sol taraf değeridir. Bkz. *Yığınaktaki Verinin Adreslenmesi* (sayfa: 72).

```
int obstack_chunk_size(struct obstack *yığınak)                işlev
```

Ayrılan tomarların boyunu verir, ayrılacak tomarlar için boy belirtilir. Bir sol taraf değeridir. Bkz. *Yığınak Tomarları* (sayfa: 72).

```
void *obstack_base(struct obstack *yığınak)                    işlev
```

Büyüyen nesnenin geçici başlangıç adresi ile döner. Bkz. *Bir Yığınağın Durumu* (sayfa: 71).

```
void *obstack_next_free(struct obstack *yığınak)               işlev
```

Belirtilen büyüyen nesnenin sonundan sonraki adres ile döner. Bkz. *Bir Yığınağın Durumu* (sayfa: 71).

2.5. Değişken Boyutlu Özdevinimli Saklama

alloca işlevi, özdevimli ayrılan ama özdevinimli serbest bırakılan bloklar halinde yarı özdevimli bir bellek ayırma çeşidini destekler.

alloca ile bir bloğun ayrılması doğrudan doğruya yapılan bir eylemdir; istediğiniz kadar blok ayırabilir ve çalışma anında boyunu hesaplayabilirsiniz. Ancak serbest bırakma işlemi **alloca** işlevinin çağrıldığı işlevden çıkıldığında özdevinimli olarak gerçekleşir. Ayrılan alanı doğrudan doğruya serbest bırakmak için bir yol yoktur.

alloca işlevi `stdlib.h` başlık dosyasında bildirilmiştir ve bir BSD oluşumudur.

`void *alloca(size_t` *boyut*`)` işlev

Çağrıldığı işlevin yığıt çerçevesinde ayrılan *boyut* baytlık bir bloğun adresi ile döner.

alloca işlevini bir işlev çağrısının argümanı olarak kullanmayın, yoksa istenmeyen sonuçlar ortaya çıkabilir. Çünkü **alloca** için ayrılan yığıt alanı işlev argümanları için ayrılan alanın ortasındaki yığıtın üzerinde görünecektir. Örneğin, **foo (x, alloca (4), y)** gibi bir çağrı yapmayın.

2.5.1. alloca Örneği

alloca kullanılan bir örnek olarak, iki dizge argüman alıp bunları birleştirip bir dosya ismi elde eden ve ve bu dosyayı açtığında bu dosyaya bir tanıtıcı ile dönen, dosyayı açamazsa açamadığını belirtmek üzere eksi bir ile dönen bir işlev aşağıdadır:

```
int
open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
  stpcpy (stpcpy (name, str1), str2);
  return open (name, flags, mode);
}
```

Aşağıdaki örnekte ise aynı sonuç **malloc** ve **free** işlevleri ile elde edilmektedir:

```
int
open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) malloc (strlen (str1) + strlen (str2) + 1);
  int desc;
  if (name == 0)
    fatal ("sanal bellek tükendi");
  stpcpy (stpcpy (name, str1), str2);
  desc = open (name, flags, mode);
  free (name);
  return desc;
}
```

Gördüğünüz gibi **alloca** ile işlem daha basittir. Ancak **alloca** işlevinin getirileri kadar götürüleri de vardır.

2.5.2. alloca İşlevinin Getirileri

alloca işlevinin **malloc** işlevine tercih edilmesindeki sebepler:

- **alloca** kullanılarak çok küçük bir alan işe yaramaz hale gelir ve işlev çok hızlıdır. (GNU C derleyicisi içinde açık kodludur.)

- **alloca** farklı blok boyları için ayrı havuzlar oluşturmadığından, farklı boylardaki bloklar başka boylarda bloklar oluşturmak için yeniden kullanılabilir. **alloca** bellek parçalanmasına sebep olmaz.

- Yerel olmayan çıkışlar **longjmp** (sayfa: 593) **alloca** işlevini çağıran işlevden de çıkışa sebep olacağından **alloca** ile ayrılan alan özdevinimli olarak serbest bırakılır. Bu, **alloca** işlevinin kullanılması için en önemli sebeptir.

  Bunu örnekleyecek olursak, **open** gibi başarılı olduğunda bir dosya tanıtıcı döndüren ama başarısız olduğunda dönmeyen `open_or_report_error` isimli bir işleviniz olsun. Dosya açılamazsa bir hata iletisi bassın ve **longjmp** kullanarak yazılımınızın komut seviyesine geçsin. *Önceki örnekteki* (sayfa: 75) `open2` işlevini bu işlevi kullanacak şekilde değiştirelim:

```
int
open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
  stpcpy (stpcpy (name, str1), str2);
  return open_or_report_error (name, flags, mode);
}
```

  alloca işlevinin çalışma şeklinden dolayı, ayırdığı bellek bir hata oluşsa bile serbest bırakılır ve bunun için ek bir çaba gerekmez.

  Karşılaştıma için, `open2` işlevinin önceki tanımı (**malloc** ve **free** kullanılan) bu amaç için değiştirildiğinde bir bellek artığı oluşacaktı. Hatta, siz bunu düzeltmek için her değişikliği yapmaya razı olsanız bile, bunu yapmanın daha kolay bir yolu yoktur.

2.5.3. alloca İşlevinin Götürüleri

alloca işlevinin **malloc** ile karşılaştırıldığında bazı götürüleri vardır:

- Makinanın sağladığından daha fazla bellek ayırmaya kalkarsanız temiz bir hata iletisi almazsınız. Bunun yerine bir sonsuz döngünün oluşturduğu gibi, olası bir *bölütleme karışıklığından dolayı* (sayfa: 604) bir ölümcül sinyal alırsınız.

- Bazı GNU dışı sistemlerde daha az taşınabilir olduğundan **alloca** desteği bulunmayabilir. Yine de bazı sistemlerdeki bu eksikliği gidermek için yazılmış daha yavaş bir taklidi C'de vardır.

2.5.4. GNU C Değişken Boyutlu Dizileri

GNU C'de **alloca** kullanımı yerine birçok durumda bir değişken boyutlu dizi kullanılabilir. Önceki örneklerdeki `open2` işlevini bu şekilde değiştirelim:

```
int open2 (char *str1, char *str2, int flags, int mode)
{
  char name[strlen (str1) + strlen (str2) + 1];
  stpcpy (stpcpy (name, str1), str2);
  return open (name, flags, mode);
}
```

Ancak çeşitli sebeplerle, **alloca** daima bir değişken boyutlu diziye eşdeğer değildir:

- Bir değişken boyutlu dizinin bellek alanı, dizi isminin etki alanının sonunda serbest bırakılır. **alloca** ile ayrılan alan ise işlev sonuna kadar kalır.

- **alloca** işlevi her yinelemede ek bir blok ayırmak üzere bir döngü içinde kullanılabilir. Bu değişken boyutlu dizilerle mümkün değildir.

Bilgi

alloca ile değişken boyutlu dizileri aynı işlevde karışık kullanırsanız, bir değişken boyutlu dizinin bildirildiği etki alanının çıkması, bu etki alanının çalışması sırasında **alloca** ile ayrılmış tüm blokların serbest bırakılmasına sebep olur.

3. Veri Bölütünün Boyunun Değiştirilmesi

Bu kısımdaki semboller `unistd.h` başlık dosyasında bildirilmiştir.

Yazılım Verisine Saklama Alanı Ayrılması (sayfa: 49) bölümünde açıklanan işlevlere göre kullanımı daha kolay olduğundan normalde, bu kısımda anlatılan işlevleri kullanacaksınız. Bunlar, kendi kullandıları GNU C Kütüphanesi bellek ayırıcısı için birer arayüz oldukları gibi sistem çağrılarının da basit birer arayüzüdür.

`int` **brk**`(void *`*adres*`)` işlev

brk işlevi çağrı sürecinin veri bölütünün yüksek sonunu *adres* olarak belirler.

Bölüt sonunun adresi bölütteki son baytın adresi artı 1 olarak tanımlıdır.

İşlev, eğer *adres* veri bölütünün düşük sonundan daha düşükse etkisizdir. (Bu durumda işlevin başarılı olduğu varsayılmıştır.)

İşlev, veri bölütü diğer bir bölütün içine girerse ya da *sürecin veri saklama sınırını* (sayfa: 575) aşarsa başarısız olur.

Veri saklama alanı ve yığıtın aynı bölütte olduğu bir ortak geçmişsel bir olgu sebebiyle işlev bu ismi almıştır. Yığıt bölütünün üstünden altına doğru büyürken, veri saklama alanı bölütün altından üstüne doğru büyür ve bunlar arasındaki perdeye perde (*break*) denir.

Başarı durumunda sıfır değeri döner. Başarısızlık durumunda **-1** döner ve **errno** hata değerine ayarlanır. Aşağıdaki **errno** değeri bu işleve özeldir:

ENOMEM
: İstek veri bölütünün başka bir bölütün içine girmesine ya da sürecin veri saklama sınırının aşılmasına sebep oldu.

`void` ***sbrk**`(ptrdiff_t` *delta*`)` işlev

Bu işlev **brk** ile aynı sonucu verir, farklı olarak, veri bölütünün sonundan itibaren bir konum olarak *delta* ile belirtirsiniz ve işlev başarılı olduğunda sıfır yerine veri bölütünün sonunun adres değeri ile döner.

Yani, veri bölütünün o andaki sonunu öğrenmek için **sbrk(0)** kullanabilirsiniz.

4. Sayfaların Kilitlenmesi

Sisteme belirli bir sanal bellek sayfasını bir gerçek bellek sayfa çerçevesi ile ilişkilendirmesini ve bu bağlantıyı tutmasını isteyebilirsiniz. Böylece bir sayfalama hatası oluşmaz. Bu işleme ***bir sayfanın kilitlenmesi*** denir.

Bu kısımdaki işlevler çağrılan sürecin sayfalarını kilitlemek ve bırakmak için kullanılır.

4.1. Sayfalar Neden Kilitlenir?

Sayfalama hataları şeffaf olarak gerçek belleğe alınacak sayfaların sanal bellekte kalmasına sebep olduğundan, bir sürecin nadir olarak sayfaların kilitlenmesi ile ilgilenmesi gerekir. Bununla birlikte iki sebep daha vardır:

- Hız. Bir sayfalama hatası sadece, bir süreç basit bir bellek erişiminin ne kadar süreceğine duyarlı değilse şeffaftır. Zamana bağlı süreçler, özellikle gerçek zamanlı süreçler bekleyemez ya da çalışma hızındaki bir değişikliği gideremeyebilirler. Bir süreç bu sebeple sayfaların kilitlenmesine ihtiyaç duyabildiği gibi ayrıca işlemci kullanımı bakımından diğer süreçlere göre öncelik alması gerekebilr. Bkz. *Sürecin İşlemci Önceliği ve Zamanlama* (sayfa: 578)

  Bazı durumlarda, hangi sayfaların gerçek bellekte kalmasının sistem başarımı için en iyi olduğunu yazılımcı sistemin istek halinde sayfalama ayırıcısından daha iyi bilir. Bu durumda sayfaların kilitlenmesi işe yarayabilir.

- Gizlilik. Bazı gizlilik gerektiren şeyleri sanal bellekte tutuyorsanız ve sanal bellek gerçek belleğe alınamazsa, gizli kalması gereken şeylerin açığa çıkma şansı artar. örneğin, bir parola disk takas alanına yazılmışsa, sanal ve gerçek bellekler tamamen temizlendikten sonra çok uzun sürelerle orada kalmaya devam edebilir.

Bir sayfayı kilitlediğiniz zaman, diğer sanal bellek kullanıcıları (aynı ya da başka süreçler) için daha az çerçeve kalabilir ve bu durum daha fazla sayfalama hatası oluşmasına ve hatta sistemin daha yavaş çalışmasına sebep olabilir. Hatta yeterince büyük belleği kilitlerseniz bazı yazılımlar gerçek bellek yokluğundan hiç çalışmayabilir.

4.2. Kilitli Bellekler Hakkında

Bir bellek kilidi bir gerçek bellek çerçevesi ile değil, bir sanal sayfa ile ilişkilidir. Sayfalama kuralı: Bir çerçeveye en azından bir kilitli sayfa kopyalanmışsa, geriye sanal belleğe kopyalanmaz.

Bellek kilitleri yığıta atılmaz. Örneğin, belli bir sayfayı iki kere kilitleyemezsiniz, yani bir sayfa ya kilitlidir ya da değildir.

Bir bellek, süreç tarafından doğrudan doğruya kilidi açılmadıkça kilitli kalmaya devam eder. (Ancak sürecin sonlandırılması ve çalıştırma işlemi sanal belleğin mevcudiyetinin sona ermesine sebep olur ve artık o kilitli değildir diyebilirsiniz).

Bellek kilitleri alt süreçler tarafından miras alınmaz. (Ancak, modern bir Unix sisteminde, bir çatallamanın hemen ardından asıl ve alt sürecin sanal adres alanı aynı çerçevelere kopyalanır ve böylece alt süreç kendini çatallayan atasının kilitlerine sahip olur.) Bkz. *Bir Sürecin Oluşturulması* (sayfa: 687).

Diğer süreçleri etkilemesi nedeniyle bir sayfayı sadece süper kullanıcı kilitleyebildiği halde her süreç kendi sayfasının kilidini kaldırabilir.

Bir sürecin kilitleyebileceği bellek miktarının ve ona adanabilen gerçek bellek miktarının sınırını sistem belirler. Bkz. *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

Linux'da kilitli sayfalar sizin düşündüğünüz gibi kilitlenmez. Belleği paylaşmayan iki sanal sayfa yine de aynı çerçeve tarafından kopyalanabilir. Her iki sayfanın aynı veriyi içerdiğini biliyorsa ve hatta sanal sayfalardan biri ya da her ikisi kilitli de olsa çekirdek bunu verimlilik adına yapar.

Ama bir süreç bu sayfalardan birini değiştirirse, çekirdek onu ayrı bir çerçeveye almalı ve çerçeveyi sayfanın verisi ile doldurmalıdır. Bu ***yazma sırasında kopyalanan sayfalama hatası*** olarak bilinir. Bu durum ve çerçevenin alınmasının G/Ç gerektirdiği marazi durumlarda biraz zaman alabilir.

Yazılımınızda bunun oluşmayacağından emin olmak için sayfaları kilitlemeyin. Onlara yazıp yazmayacacağınızı bilmeseniz bile onlara yazın. Ve yığıtınız için önceden ayrılmış çerçevelerin varlığından emin olmak için ihtiyaç

duyduğunuz en büyük yığıt boyundan daha büyük bir C özdevinimli değişkeninin bidirildiği bir karmaşık deyime girip değişkene bazı şeyler atayın ve onu döndürün.

4.3. Sayfaları Kilitleyen ve Kilitlerini Açan İşlevler

Bu bölümdeki işlevler `sys/mman.h` başlık dosyasında bildirilmiştir. Bu işlevler by POSIX.1b tarafından tanımlanmıştır ama onların kullanılabilirliği çekirdeğinize bağımlıdır. Çekirdeğiniz bu işlevlere izin vermiyorsa onlar var olsalar bile başarısız olacaklardır. Linux çekirdeği ile bu işlevleri *kullanabilirsiniz*.

Taşınabilirlik Bilgisi:

POSIX.1b, **mlock** ve **munlock** işlevleri kullanılabilir olduğunda, **_POSIX_MEMLOCK_RANGE** makrosunun `unistd.h` dosyasında ve bir bellek sayfasının bayt cinsinden boyu olan **PAGESIZE** makrosunun `limits.h` dosyasında tanımlanmasını gerektirir. Ayrıca, **mlockall** ve **munlockall** işlevleri kullanılabilir olduğunda, **_POSIX_MEMLOCK** makrosunun `unistd.h` dosyasında tanımlanmasını gerektirir. GNU C kütüphanesi bu gereksinimleri karşılar.

```
int mlock(const void *adres,                                    işlev
          size_t      uzunluk)
```

adres adresinden başlayan *uzunluk* bayt uzunluğundaki sanal sayfa aralığını kilitler. Aslında, sayfaları bir bütün olarak kilitlemeniz gerektiğinden belirtilen aralıktaki parçaları içeren sayfaların bir aralığıdır.

İşlev başarılı olarak dönerse, bu sayfaların her biri bir gerçek bellek çerçevesi tarafından (kalıcı olarak) kopyalanır (çerçevelenir) ve kalıcı olarak imlenir. Bu, işlevin sayfayı gerçek belleğe kopyalanmasına ve orada kalmasına sebep olması demektir.

İşlev başarısız olursa, sayfaların kilitlenme durumu etkin olmaz.

İşlev başarılı olduğunda dönen değer sıfırdır. Aksi takdirde, **-1** döner ve **errno** ilgili hata değerine ayarlanır. Bu işleve özel **errno** değerleri:

`ENOMEM`

- Belirtilen adres aralığının en azından bir kısmı çağrılan sürecin sanal adres aralığında mevcut değildir.
- Kilitleme, sürecin kilitli sayfa sınırının aşılmasına sebep olacaktır.

`EPERM`
: Süreci çağıran süper kullanıcı değildir.

`EINVAL`
: *uzunluk* pozitif değildir.

`ENOSYS`
: Çekirdek **mlock** yeteneği sağlamıyor.

Bir sürecin *tüm* belleğini **mlockall** ile kilitleyebilir ve **munlock** ya da **munlockall** ile kilitlerini açabilirsiniz.

Bir C yazılımında sayfalama hatalarından kaçınmak için **mlockall** işlevini kullanmalısınız, çünkü bir yazılımın kullandığı belleğin bir kısmı örneğin, yığıt ve özdevinimli değişkenler C kodundan gizlidir ve hangi adrese **mlock** deneceğini bilmeyeceksiniz.

```
int munlock(const void *adres,                                    işlev
            size_t      uzunluk)
```

munlock çağrılan sürecin sanal sayfalarının bir aralığının kilidini açar.

munlock işlevi **mlock** işlevinin tersidir ve **EPERM** başarısızlığı söz konusu değildir.

```
int mlockall(int bayraklar)                                       işlev
```

mlockall bir sürecin sanal adres alanındaki ve/veya ilerde eklenecek tüm sayfaları kilitler. Bu alan, kod sayfalarını, veri ve yığıt bölütünü, paylaşımlı kütüphaneleri, kullanıcı alanı çekirdek verisini, paylaşımlı belleği ve bellek eşlemli dosyaları içerir.

bayraklar aşağıdaki makrolarla ifade edilen tek bitlik bayrakların bir dizgesidir. Bunlar **mlockall**'dan istediğiniz işlevselliği belirtmek için kullanılır. Tüm diğer bitler sıfır olmalıdır.

MCL_CURRENT
: Çağrılan sürecin sanal adres alanında o an mevcut olan tüm sayfalar kilitlenir.

MCL_FUTURE
: Sürecin sanal adres alanına gelecekte doğumundan itibaren eklenecek sayfaların kilitlenmesini sağlayacak kipi etkinleştirir. Bu kip aynı süreç tarafından sahiplenilecek alt süreçlerin gelecekteki adres alanları üzerinde etkili değildir. Örneğin süreç bir exec çağrısı ile bir alt süreç çalıştırırsa bu kip, bu alt sürecin sayfaları için etkisizdir. Bkz. *Bir Dosyanın Çalıştırılması* (sayfa: 688).

İşlev **MCL_CURRENT** bayrağıyla çağrılır ve başarılı olarak dönerse, bu sayfaların her biri bir gerçek bellek çerçevesi tarafından (kalıcı olarak) kopyalanır (çerçevelenir) ve kalıcı olarak imlenir. Bu, işlevin sayfayı gerçek belleğe kopyalanmasına ve orada kalmasına sebep olması demektir.

Süreç **MCL_FUTURE** kipindeyse bu işlev başarılı olduğundan dolayıdır ve **MCL_CURRENT** belirtildiğinde, süreç tarafından sanal bellek alanına alan eklenmesini gerektiren bir sistem çağrısıyla ek alanın kiltlenmesi sürecin kilitli sayfa sınırının aşılmasına sebep oluyorsa işlev **errno**= **ENOMEM** hatasıyla başarısız olur. Bu durumda adres alanı eklemesi yığıt genişletmesi ile bağdaştırılamaz ve yığıt genişlemesi başarısız olur, bunun sonucunda da çekirdek sürece bir **SIGSEGV** sinyali gönderir.

İşlev başarısız olduğunda, kilitli sayfaların durumu ve gelecektekileri kilitleme kipi bundan etkinemez.

İşlev başarılı olduğunda dönen değer sıfırdır. Aksi takdirde, **-1** döner ve **errno** ilgili hata değerine ayarlanır. Bu işleve özel **errno** değerleri:

ENOMEM
- Belirtilen adres aralığının en azından bir kısmı çağrılan sürecin sanal adres aralığında mevcut değildir.
- Kilitleme, sürecin kilitli sayfa sınırının aşılmasına sebep olacaktır.

EPERM
: Süreci çağıran süper kullanıcı değildir.

EINVAL
: *bayraklar* içinde belirtilmeyen bitler sıfır değil.

ENOSYS
: Çekirdek **mlockall** yeteneği sağlamıyor.

Belirli sayfaları **mlock** ile kilitleyebilirsiniz. Kilitli sayfalardan ise **munlockall** ve **munlock** ile kilitleri kaldırabilirsiniz.

```
int munlockall(void)                                                    işlev
```

Çağrılan sürecin sanal adres alanındaki her sayfanın kilidini kaldırır ve **MCL_FUTURE** gelecekteki kilitleme kipini kapatır.

İşlev başarılı olduğunda dönen değer sıfırdır. Aksi takdirde, **-1** döner ve **errno** ilgili hata değerine ayarlanır. Sadece, tüm işlev ve sistem çağrılarının başarısız olabildiği soysal sebeplerle işlev başarısız olursa bu duruma özel bir **errno** değeri yoktur.

IV. Karakterle Çalışma

İçindekiler

Karakterler ve dizgelerle çalışan yazılımlar bir karakteri, alfabetik, rakam, boşluk, vs. olarak sınıflandırmayı ve karakterler üzerinde büyük–küçük harf dönüşümleri uygulamayı gerektirir. `ctype.h` başlık dosyasındaki işlevler bu amaç içindir.

Yerel ve karakter kümesi seçimi karakter kodlarının sınflandırılmalarını değiştirebildiğinden bu işlevlerin tümü o anki yerelden etkilenir. Daha teknik bir söylemle, onlar, *yerel kategorilerden* (sayfa: 165) karakter sınıflandırması için olan **`LC_CTYPE`** kategorisinden etkilenir.

ISO C standardı iki farklı işlev kümesi belirtir. Bu işlev kümelerinden biri **`char`** türünden karakterlerle diğeri **`wchar_t`** türünden geniş karakterlerle çalışır. (Bkz. *Genişletilmiş Karakterlere Giriş* (sayfa: 126)).

1. Karakterlerin Sınıflandırılması

Bu kısımda karakterlerin sınıflandırılmasında kullanılan işlevler açıklanmıştır. Örneğin **`isalpha`** işlevi bir karakterin alfabetik bir karakter olup olmadığını sınar. Argüman olarak bir karakter alır ve bir alfabetik bir karakterse sıfırdan farklı bir tamsayı, değilse sıfır ile döner. Onu aşağıdaki gibi kullanabilirsiniz:

```
if (isalpha (c))
  printf ("`%c' bir alfabetik karakterdir.\n", c);
```

Bu kısımdaki her işlev bir karakterin belli bir sınıfa üye olup olmadığına bakar ve bu işlevlerin isimleri daima **`is`** ile başlar. Herbiri argüman olarak bir karakter alır ve mantıksal bir değer olarak değerlendirilebilen**`int`** türünden bir değerle döner. Karakter argümanı **`int`** türünden olmalıdır. Bir gerçek karakter yerine **`EOF`** gibi bir sabit değer de verilebilir.

Her karakterin öznitelikleri yerele göre değişiklik gösterir. Yereller hakkında daha fazla bilgi edinmek isterseniz *Yereller ve Uluslararasılaştırma* (sayfa: 164) bölümüne bakınız.

Bu işlevler `ctype.h` başlık dosyasında bildirilmiştir.

`int` **`islower`**`(int c)` işlev

> c bir küçük harf ise sıfırdan farklı bir değerle döner. Harfin Latin alfabesinden olması şart değildir, herhangi bir alfabeden olabilir.

`int` **`isupper`**`(int c)` işlev

> c bir büyük harf ise sıfırdan farklı bir değerle döner. Harfin Latin alfabesinden olması şart değildir, herhangi bir alfabeden olabilir.

`int` **`isalpha`**`(int c)` işlev

> c bir alfabetik karakter (bir harf) ise sıfırdan farklı bir değerle döner. **`islower`** veya **`isupper`** işlevi doğru ile dönüyorsa **`isalpha`** işlevi de doğru ile döner.

Bazı yereller ne büyük ne de küçük harf olan ve **isalpha** işlevinin doğru ile döndüğü ek karakterler içerir. Standart **"C"** yerelinde böyle bir karakter yoktur.

`int isdigit(int c)` işlev

c değişkeninin değeri **0** ile **9** arasında bir rakam karakteri ise sıfırdan farklı bir değerle döner.

`int isalnum(int c)` işlev

c bir alfasayısal karakter (bir harf ya da rakam) ise sıfırdan farklı bir değerle döner. Başka bir ifade ile, **isalpha** ya da **isdigit** doğru ise **isalnum**'da doğrudur.

`int isxdigit(int c)` işlev

c değişkeninin değeri **0** ile **9** arasında bir rakam karakteri veya **A** ile **F** arasında bir büyük ya da küçük harf ise sıfırdan farklı bir değerle döner.

`int ispunct(int c)` işlev

c bir noktalama işareti ise sıfırdan farklı bir değerle döner. Noktalama işaretleri, alfasayısal ve boşluk olmayan basılabilir karakterlerdir.

`int isspace(int c)` işlev

c bir boşluk karakteri ise sıfırdan farklı bir değerle döner. **"C"** yerelinde **isspace** işlevi, sadece

```
' '  (boşluk)
'\f' (sayfa ileri)
'\n' (satırsonu)
'\r' (satırbaşı)
'\t' (yatay sekme)
'\v' (düşey sekme)
```

karakterleri için doğrudur.

`int isblank(int c)` işlev

c bir boşluk ya da sekme karakteri ise sıfırdan farklı bir değerle döner. Bu işlev bir GNU oluşumudur, fakat ISO C99'a eklenmiştir.

`int isgraph(int c)` işlev

c bir çizgesel karakter ise sıfırdan farklı bir değerle döner. Çizgesel karakterler, metin ekranında pencere çizmek amacıyla kullanılan karakterler gibi karakterlerdir. Boşluk karakterleri çizgesel karakter değildir.

`int isprint(int c)` işlev

c basılabilir bir karakter ise sıfırdan farklı bir değerle döner. Basılabilir karakterler, çizgesel karakterler ile boşluk (' ') karakterinden oluşur.

`int iscntrl(int c)` işlev

c bir denetim karakteri ise (basılabilir bir karakter değilse) sıfırdan farklı bir değerle döner.

`int isascii(int c)` işlev

c bir 7 bitlik **unsigned char** türünden ASCII karakter kümesinden karakter ise sıfırdan farklı bir değerle döner. Bu işlev bir BSD ve SVID oluşumudur.

2. Büyük–Küçük Harf Dönüşümleri

Bu kısımda karakterler üzerinde harf büyüklüklerine göre dönüşüm işlemlerinden bahsedilmiştir. Örneğin, **toupper** işlevi bir karakteri mümkünse büyük harfe dönüştürmek için kullanılır. Eğer karakter dönüştürülemiyosa **toupper** işlevi karakteri değiştirmez.

Bu işlevler karakteri **int** türünden bir argüman olarak alır ve dönüştürülen karakteri **int** türünden geridöndürür. Verilen argümana dönüşüm uygulanabilir değilse verilen argüman değiştirilmeden döner.

Uyumluluk Bilgisi:

ISO C öncesi oluşumlarda, bu işlevler başarısız olduklarında verilen argümanla dönmezler, sadece başarısız olurlardı. Uyumluluk açısından **toupper(c)** yerine **islower(c) ? toupper(c) : c** yazmanız gerekebilir.

Bu işlevler `ctype.h` başlık dosyasında bildirilmiştir.

`int` **tolower**`(int` *c*`)` işlev

c bir büyük harf ise, işlev ona karşılık gelen küçük harf ile döner. *c* bir büyük harf değise işlev *c* ile döner.

`int` **toupper**`(int` *c*`)` işlev

c bir küçük harf ise, işlev ona karşılık gelen büyük harf ile döner. *c* bir küçük harf değise işlev *c* ile döner.

`int` **toascii**`(int` *c*`)` işlev

Bu işlev karakterin yüksek seviyeli bitini temizleyerek 7 bitlik **unsigned char** türünde ASCII karakter kümesinden bir karakter döndürür. Bu işlev bir BSD ve bir SVID oluşumudur.

`int` **_tolower**`(int` *c*`)` işlev

Bu işlev **tolower** işlevi ile eşdeğerdedir ve *SVID* (sayfa: 22) ile uyumluluk için bulunmaktadır.

`int` **_toupper**`(int` *c*`)` işlev

Bu işlev **toupper** işlevi ile eşdeğerdedir ve *SVID* (sayfa: 22) ile uyumluluk için bulunmaktadır.

3. Geniş Karakterlerin Sınıflandırılması

ISO C90 standardının 1. düzeltmesi, geniş karakterleri sınıflandıran işlevler tanımlar. Özgün ISO C90 standardında **wchar_t** türü tanımlanmışsa da bu tür ile çalışan hçbir işlev tanımlanmamıştı.

Geniş karakterler için sınıflandırma işlevlerinin genel tasarımı daha geneldir. Daima kullanılabilir olanların dışında kullanılabilir sınıflandırmalar kümesi oluşumlarını mümkün kılar. POSIX standardı oluşumların nasıl yapılacağını belirtir ve bu zaten bir GNU C kütüphanesi gerçeklemesi olan **localedef** yazılımı ile gerçeklenmiştir.

Karakter sınıfı işlevleri normalde her karakter için bir bit kümesi olmak üzere bit kümeleri ile gerçeklenir. Verilen bir karakter için bit kümesi bir tablodan okunur ve anlamlı bitlerin bir olup olmadığına bakılır. Sınanacak bitler sınıf tarafından belirlenir.

Geniş karakter sınıflandırma işlevleri için bu görünür yapılır. Sınıflandırma türü için bir tür, bir verilmiş sınıf için bu değeri alan bir işlev ve verilen karakterin verilen sınıfa ait olup olmadığını sınıflandırma değerini kullanarak sınayan bir işlev daha vardır. En tepede, **char** türünden nesneler için kullanılan normal karakter sınıflandırma işlevleri gibi tanımlanmış olabilirler.

Bu kısımda bahsi geçen tüm işlevler aksi belirtilmedikçe `wctype.h` başlık dosyasında tanımlanmıştır.

wctype_t veri türü

wctype_t bir karakter sınıfına karşılık gelen bir değeri tutabilir. Böyle bir değeri üretmenin tanımlanmış tek yolu **wctype** işlevini kullanmaktır.

Bu tür wctype.h başlık dosyasında tanımlanmıştır.

```
wctype_t wctype(const char *özellik)                                    işlev
```

wctype işlevi *özellik* dizgesi ile belirtilen bir geniş karakter sınıfını ifade eden bir değer ile döner. Her yerelde tanımlanabilir standart özelliklerin bazıları aşağıda verilmiştir. *özellik* dizgesi **LC_CTYPE** kategorisi için seçilen yerelde tanımlanmış isimlerden biri değilse işlev sıfır değeriyle döner.

Her yerelde bilinen özellikler:

alnum	**alpha**	**cntrl**	**digit**
graph	**lower**	**print**	**punct**
space	**upper**	**xdigit**	

Bir karakterin standart dışı sınıflardan birine üyeliğini sınamak için ISO C standardı tamamen yeni bir işlev tanımlar.

```
int iswctype(wint_t   wc,                                               işlev
             wctype_t tanımlayıcı)
```

Bu işlev, *wc* karakteri *tanımlayıcı* ile belirtilen sınıfa ait bir karakterse sıfırdan farklı bir değerle döner. *tanımlayıcı* önceki bir **wctype** çağrısından dönen değer olmalıdır.

Çok kullanılan sınıflandırma işlevleri kullanımı kolaylaştırmak için C kütüphanesinde tanımlanmıştır. Özellik dizgesi bilinen karakter sınıflarından biriyse bu işlevler için **wctype** işlevini kullanmaya gerek yoktur. Ancak bazı durumlarda özellik dizgelerinden standart sınıfları elde etmekte önem kazanır.

```
int iswalnum(wint_t   wc)                                               işlev
```

wc bir alfasayısal karakter (bir harf ya da rakam) ise sıfırdan farklı bir değerle döner. Başka bir ifade ile, **iswalpha** ya da **iswdigit** doğru ise **iswalnum**'da doğrudur.

İşlev,

```
iswctype (wc, wctype ("alnum"))
```

kullanılarak da gerçeklenebilir.

```
int iswalpha(wint_t   wc)                                               işlev
```

wc bir alfabetik karakter (bir harf) ise sıfırdan farklı bir değerle döner. **iswlower** veya **iswupper** işlevi doğru ile dönüyorsa **iswalpha** işlevi de doğru ile döner.

Bazı yereller ne büyük ne de küçük harf olan ve **iswalpha** işlevinin doğru ile döndüğü ek karakterler içerir. Standart **"C"** yerelinde böyle bir karakter yoktur.

İşlev,

```
iswctype (wc, wctype ("alpha"))
```

kullanılarak da gerçeklenebilir.

```
int iswcntrl(wint_t   wc)                                               işlev
```

wc bir denetim karakteri ise (basılabilir bir karakter değilse) sıfırdan farklı bir değerle döner.

İşlev,

```
iswctype (wc, wctype ("cntrl"))
```

kullanılarak da gerçeklenebilir.

`int iswdigit(wint_t wc)` işlev

wc değişkeninin değeri **0** ile **9** arasında bir rakam karakteri ise sıfırdan farklı bir değerle döner.

Bilgi

Bu işlev sadece onluk sistemdeki rakamlar için değil, tüm rakam çeşitleri için sıfırdan farklı bir değerle döner. Bu nedenle aşağıdaki gibi bir kod koşulsuz olarak *çalışmayacaktır*:

```
n = 0;
while (iswdigit (*wc))
  {
    n *= 10;
    n += *wc++ - L'0';
  }
```

İşlev,

```
iswctype (wc, wctype ("digit"))
```

kullanılarak da gerçeklenebilir.

`int iswgraph(wint_t wc)` işlev

wc bir çizgesel karakter ise sıfırdan farklı bir değerle döner. Çizgesel karakterler, metin ekranında pencere çizmek amacıyla kullanılan karakterler gibi karakterlerdir. Boşluk karakterleri çizgesel karakter değildir.

İşlev,

```
iswctype (wc, wctype ("graph"))
```

kullanılarak da gerçeklenebilir.

`int iswlower(wint_t wc)` işlev

wc bir küçük harf ise sıfırdan farklı bir değerle döner. Harfin Latin alfabesinden olması şart değildir, herhangi bir alfabeden olabilir.

İşlev,

```
iswctype (wc, wctype ("lower"))
```

kullanılarak da gerçeklenebilir.

`int iswprint(wint_t wc)` işlev

wc basılabilir bir karakter ise sıfırdan farklı bir değerle döner. Basılabilir karakterler, çizgesel karakterler ile boşluk (`' '`) karakterinden oluşur.

İşlev,

```
iswctype (wc, wctype ("print"))
```

kullanılarak da gerçeklenebilir.

`int iswpunct(wint_t wc)` işlev

wc bir noktalama işareti ise sıfırdan farklı bir değerle döner. Noktalama işaretleri, alfasayısal ve boşluk olmayan basılabilir karakterlerdir.

İşlev,

```
iswctype (wc, wctype ("punct"))
```

kullanılarak da gerçeklenebilir.

`int iswspace(wint_t wc)` işlev

wc bir boşluk karakteri ise sıfırdan farklı bir değerle döner. `"C"` yerelinde `iswspace` işlevi, sadece

```
L' '   (boşluk)
L'\f'  (sayfa ileri)
L'\n'  (satırsonu)
L'\r'  (satırbaşı)
L'\t'  (yatay sekme)
L'\v'  (düşey sekme)
```

karakterleri için doğrudur. İşlev,

```
iswctype (wc, wctype ("space"))
```

kullanılarak da gerçeklenebilir.

`int iswupper(wint_t wc)` işlev

wc bir büyük harf ise sıfırdan farklı bir değerle döner. Harfin Latin alfabesinden olması şart değildir, herhangi bir alfabeden olabilir.

İşlev,

```
iswctype (wc, wctype ("upper"))
```

kullanılarak da gerçeklenebilir.

`int iswxdigit(wint_t wc)` işlev

wc değişkeninin değeri `0` ile `9` arasında bir rakam karakteri veya `A` ile `F` arasında bir büyük ya da küçük harf ise sıfırdan farklı bir değerle döner.

İşlev,

```
iswctype (wc, wctype ("xdigit"))
```

kullanılarak da gerçeklenebilir.

GNU C kütüphanesi bunlardan başka, ISO C standardında tek baytlık karakterler için de tanımlanmamış bir işlev daha içerir.

`int iswblank(wint_t wc)` işlev

wc bir boşluk ya da sekme karakteri ise sıfırdan farklı bir değerle döner. Bu işlev bir GNU oluşumudur, dakat ISO C99'a eklenmiştir. `wchar.h` başlık dosyasında bildirilmiştir.

4. Geniş Karakter Sınıflarının Kullanılması

İlk uyarı, şüphesiz şaşırtıcı değil ama hala arasıra sorun çıkarmaktadır. **isw***XXX* işlevleri makrolarla gerçeklenebilir ve GNU C kütüphanesi böyle yapar. Onlar yine de gerçek işlevler olarak kullanılabilirse de **wctype.h** başlık dosyası içerildiğinde makrolar kullanılacaktır. Bu durum, bu işlevlerin **char** türü olanları için de aynıdır.

İkincisi daha yeni bir bilgidir. En iyisi bir örnekle açıklamak. Kodun ilk parçasını özgün koddan (biraz kısaltarak) seçelim.

```
int is_in_class (int c, const char *class)
{
  if (strcmp (class, "alnum") == 0)
    return isalnum (c);
  if (strcmp (class, "alpha") == 0)
    return isalpha (c);
  if (strcmp (class, "cntrl") == 0)
    return iscntrl (c);
  …
  return 0;
}
```

Şimdi, **wctype** ve **iswctype** ile **if** merdiveninden kurtulabilirsiniz, ancak aşağıdaki gibi bir kod yanlış olacaktır:

```
int is_in_class (int c, const char *class)
{
  wctype_t desc = wctype (class);
  return desc ? iswctype ((wint_t) c, desc) : 0;
}
```

Burada sorun, bir tek baytlık karakterin geniş karakter karşılığının tür dönüşümü ile elde edilebilirliğinin garantili olmayışıdır. Doğru çözüm, kodu aşağıdaki gibi yazmaktır:

```
int is_in_class (int c, const char *class)
{
  wctype_t desc = wctype (class);
  return desc ? iswctype (btowc (c), desc) : 0;
}
```

btowc işlevi hakkında bilgi edinmek için *Bir Karakterin Dönüştürülmesi* (sayfa: 132) bölümüne bakınız. Burada, **wctype** işlevi hala bir dizge karşılaştırması yaptığından yazılımın başarımı arttırılmış olmayacaktır. **is_in_class** işlevi aynı sınıf ismi için defalarca çağrılırsa durum gerçekten ilginç olur. Bu durumda, *desc* değişkeni bir kere hesaplanıp tüm çağrılarda kullanılmalıydı. Bu nedenle işlevin yukardaki şekli şüphesiz işlevin son hali olmayacaktır.

5. Geniş Karakterlerde Büyük–küçük Harf Dönüşümleri

Sınıflandırma işlevleri ISO C standardı tarafından ayrıca genelleştirilmiştir. İki standardın eşleştirilmesi yerine bir yerelin diğerlerini içermesi sağlanabilir. **localedef** yazılımı bu yerel veri dosyalarının üretilmesini zaten desteklemektedir.

wctrans_t veri türü

Bu veri türü, yerele bağımlı karakter eşleşmelerini ifade eden bir değeri tutabilen ölçeklenebilir bir tür olarak tanımlanmıştır. Böyle bir değeri **wctrans** işlevinden dönen bir değer olarak yapılandırmaktan başka bir yol yoktur.

Bu tür `wctype.h` başlık dosyasında tanımlanmıştır.

`wctrans_t` **wctrans**`(const char *`*özellik*`)` işlev

wctrans işlevi, **LC_CTYPE** kategorisi için seçilen yerelde tanımlı, *özellik* ile belirtilen isimli bir eşleşme varsa bulmakta kullanılır. Dönen değer sıfırsa yerelde böyle bir eşleşmenin bulunmadığı anlaşılır. Sıfırdan farklı bir değer dönmüşse bu değer bir **towctrans** çağrısında kullanılabilir.

Her yerelde kullanılabilen iki eşleşme aşağıda verilmiştir:

tolower	**toupper**

Bu işlev `wctype.h` başlık dosyasında bildirilmiştir.

`wint_t` **towctrans**`(wint_t` *wc*`, wctrans_t` *tanımlayıcı*`)` işlev

Bu işlev, *wc* karakterinin *tanımlayıcı* ile belirtilen eşleşme kurallarına uygun karşılığını bulur ve bu değerle döner. *tanımlayıcı* önceki bir **wctrans** çağrısından dönen değer olmalıdır.

Bu işlev `wctype.h` başlık dosyasında bildirilmiştir.

Genel olarak kullanılabilen eşleşmeler için ISO C standardı birer **wctrans** çağrısı gerektirmeyen kısayollar tanımlar.

`wint_t` **towlower**`(wint_t` *wc*`)` işlev

wc bir büyük harf ise, işlev ona karşılık gelen küçük harf ile döner. *wc* bir büyük harf değise işlev *wc* ile döner.

İşlev,

```
towctrans (wc, wctrans ("tolower"))
```

kullanılarak da gerçeklenebilir. Bu işlev `wctype.h` başlık dosyasında bildirilmiştir.

`wint_t` **towupper**`(wint_t` *wc*`)` işlev

wc bir büyük harf ise, işlev ona karşılık gelen küçük harf ile döner. *wc* bir büyük harf değise işlev *wc* ile döner.

İşlev,

```
towctrans (wc, wctrans ("toupper"))
```

kullanılarak da gerçeklenebilir. Bu işlev `wctype.h` başlık dosyasında bildirilmiştir.

Geniş karakterlerin kullanımına ilişkin önceki bölümdeki uyarılar burada da geçerlidir. **towctrans** çağrılarında bir argüman olarak **char** türünden bir değer, tür dönüşümü ile **wint_t** türüne dönüştürülerek kullanılamaz.

V. Diziler ve Dizgeler

İçindekiler

Dizge (karakter dizisi) işlemleri birçok uygulamanın önemli bir parçasını oluşturur. GNU C kütüphanesi kopyalama ve birleştirme işlevleri de dahil olmak üzere geniş bir dizge işleme işlevleri kümesine sahiptir. Bu işlevlerin çoğu bellek erişimi ile ilgiliolarak da çalışır; örneğin `memcpy` işlevi her çeşit dizi içeriğinin kopyalanmasında kullanılabilir.

C yazılımcılığına yeni başlayanlar genelde bu işlevlerin benzerlerini yazarak "tekerleği yeniden icadederler". Halbuki bu zamanı kütüphane işlevlerini öğrenmeye ayırsalar ve bunları kullansalar verimlilik ve taşınabilirlik adına bir çok kazanç elde edeceklerdir.

Örneğin bir dizgeyi diğeriyle iki satır C kodu yazarak kolayca karşılaştırabilirsiniz, fakat yerleşik `strcmp` işlevini kullanırsanız daha az yanlış yapmış olacaksınız. Ve bu kütüphane işlevleri yüksek derecede eniyilendiklerinden yazılımınız daha hızlı çalışacaktır.

1. Dizgelerle İlgili Kavramlar

Bu kısım, C yazılımcılığına yeni başlayanlar için dizge kavramlarının kısa bir özetini barındırır. Karakter dizgelerinin C'de nasıl temsil edildiği ve bilinen bazı tuzaklar hakkında bilgi verilecektir. Bu bilgilere zaten sahipseniz bu bölümü atlayabilirsiniz.

Bir ***dizge***, `char` türünden nesnelerden oluşan bir dizidir. Fakat dizge değerli değişkenler genellikle `char *` türünden bir gösterici olarak bildirilirler. Bu tür değişkenler dizgenin metni için alan içermezler. Dizge bir dizi değişkeninde, bir dizge sabitinde ya da *özdevimli ayrılmış bir bellek bölgesinde* (sayfa: 49) olabilir. Gösterici değişkeninde seçilen bellek bölgesinin adresini saklarsınız. Ayrıca gösterici değişkeninde bir boş gösterici de saklayabilirsiniz. Boş gösterici hiçbir yeri göstermediğinden onu bir dizge olarak gösterirseniz bir hata alırsınız.

Bir dizge normalde geniş karakterli dizgenin tersine bir çokbaytlı karakter dizisidir. Geniş karakterli dizgeler `wchar_t` türünden dizilerdir ve çok baytlı karakter dizileri gibi genellikle `wchar_t *` türünden göstericilerle kullanılırlar.

Teamülen, bir ***boş karakter***, `'\0'` karakteridir ve bir çokbaytlı karakter dizisinin sonuna konur, ***boş geniş karakter*** ise `L'\0'` karakteridir ve bir geniş karakterli dizgenin sonuna konur. Örneğin, `char *` türünden p değişkeninin gösterdiği dizgenin sonunda bir boş karakter olup olmadığını `!*`p veya`*`p `== '\0'` yazarak sınayabilirsiniz.

Bir boş karakter ile bir boş gösterici arasındaki sadece kavramsal fark vardır, her ikisi de `0` tamsayısı tarafından temsil edilir.

Dizge sabitler C yazılımlarında çift tırnak içine alınmış karakterlerin büyük harf L ile öncelenmesi ile oluşurlar, `L"foo"` gibi. ISO C'de dizge sabitler kendiliğinden ***birleşik dizge*** oluştururlar, örneğin `"a" "b"` ile `"ab"` aynıdır. Geniş karakterli dizgeler için bu gösterimler `L"a" L"b"` veya `L"a" "b"` biçiminde olabilir. GNU C derleyicisi dizge sabitler üzerinde değişiklik yapılmasına izin vermez, çünkü sabitler salt–okunur bölgede tutulurlar.

Ayrıca `const` ile bildirilen karakter dizileri de değiştirilemezler. C derleyicisi `const char *` türünden bir değişiklik yapılamayan bir dizge göstericisi kullanıldığında istemdışı değişiklikleri saptayabildiğinden en iyisi dizge sabitleri `const char *` türünde bildirmektir.

Karakter dizisi için ayrılan belleğin miktarı dizgenin sonunu belirleyen boş karaktere kadar uzayabilir. Bu belgede ***ayrılan boyut*** denilince daima dizge için ayrılan belleğin toplam miktarından, ***uzunluk*** denilince dizgeyi sonlandıran boş karakter hariç dizgedeki toplam karakter sayısından bahsetmiş olacağız.

Namlı yazılım hatalarından biri bir dizgeye ayrılan yere sığacağından daha fazla karakter girilmeye çalışılmasıdır. Kodu yazarken, önceden ayrılmış bir dizi içine dizge veya karakterler taşınırken metnin uzunluğunu sürekli denetlemeli dizinin taşmamasına dikkat etmelisiniz. Bir çok kütüphane işlevi bu işlemi sizin yerinize *yapmaz*. Dizgenin sonunu belirtecek olan boş karakter için de yer ayırmayı unutmayın.

Genellikle, dizgeler, her baytı bir karakteri ifade eden bayt dizileridir. Bu özellik eğer dizge tek baytlık karakter kodlaması ile yazılmışsa geçerlidir. Eğer çok baytlı karakter kodlaması kullanılmışsa bu değişir. (Karakter kodlamaları için *Genişletilmiş Karakterlere Giriş* (sayfa: 126) bölümüne bakınız.) Yazılım geliştirme arayüzü bakımından bu iki dizge çeşidi bir fark oluşturmaz; yeterki yazılımcı bu farka dikkat ederek dizgeleri uygun yorumlasın.

Ancak arayüz için bir farklılık oluşturmayan bayt temelli işlevlerin kullanımı bazan zor olur. Bu işlevlerin baytları belirleyen sayaç parametreleri `strncpy` işlevini çağırarak bir çokbaytlı karakteri ortasından bölüp bir eksik (dolayısıyla kullanışsız) bayt dizisi olarak hedef tampona yerleştirir.

Bu sorunlardan kurtulmak için ISO C standardının daha sonraki sürümlerinde ***geniş karakterler*** (*Genişletilmiş Karakterlere Giriş* (sayfa: 126)) ile çalışan ikinci bir işlev kümesi tanımlandı. Her geniş karakter kurallara uygun ve yorumlanabilir olduğundan bu işlevlerle tek baytlık karakter dizileriyle çalışmada karşılaşılan sorunlar ortadan kalktı. Bu geniş karakterli dizgelerde kesme işleminin uygunsuz bir yerden yapılmayacağı anlamına gelmez. Normalde alfabe temelli diller (normalleştirilmemiş metinler hariç) için sorun yoktur, ancak hece temelli dillerde (Çince gibi) mantıksal birimleri çok sayıda geniş karakter oluşturduğundan bu sorun hala devam etmektedir. Bu kadarı bile yine de iyidir, çünkü en azından geçersiz bayt dizileri oluşmamaktadır. Ayrıca, daha yüksek seviyeli işlevler geniş karakterli dizgelerle çok baytlı karakter dizilerinden daha kolay çalışır. Bu bakımdan geniş karakterli dizgelerin kullanılması tercih edilmelidir.

Bu kısmın kalanında geniş karakterli dizgelerle çalışan işlevlere paralal olarak çokbaytlı karakter dizilerini de tartışacağız, çünkü onlar arasında hemen hemen bir tam eşdeğer kullanılabilirlik vardır.

2. Dizi ve Dizge Teamülleri

Bu kısımda diziler veya bellek blokları üzerinde çalışan işlevlerle normal karakterlerin ve geniş karakterlerin özellikle boş karakter sonlandırmalı dizileriyle çalışan işlevler açıklanmıştır.

Bellek blokları ile çalışan işlevlerin isimleri `mem` ve `wmem` ile başlar ve üzerinde çalışacakları bellek blokunun boyutunu (sırasıyla bayt ve geniş karakter sayısı olarak) belirten bir argüman alırlar. Bu işlevlerin dizi argümanları ve dönüş değerleri `void *` veya `wchar_t` türündendir. `mem` ile başlayan işlev isimlerine sahip işlevler için dizi elemanlarına konu olan baytlardır ve bu işlevlere her çeşit gösterici aktarabileceğiniz gibi, boyut argümanını

değerini hesaplarken **sizeof** işlecini kullanabilirsiniz. **wmem** işlevlerinin parametreleri **wchar_t *** türünde olmalıdır. Bu işlevlerde bu tür haricinde bir gösterici kullanamazsınız.

Özellikle normal dizgeler ve geniş karakterli dizgelerle çalışan işlevlerin isimleri sırasıyla **str** ve **wcs** ile başlar (**strcpy** ve **wcscpy** gibi) ve bir uzunluk argümanı almak yerine dizgeyi sonlandıran boş karaktere bakarlar. (Bu işlevlerin bazıları bir en büyük uzunluk değeri alırlar ve ayrıca bir boş karakterle vaktinden önce sonlandırma olup olmadığını sınarlar.) Bu işlevlerin dizi argümanları ve dönüş değerleri **char *** ve **wchar_t *** türünde olmalı ve dizi elemanları sırasıyla normal karakterlerden veya geniş karakterlerden oluşmalıdır.

Birçok durumda bir işlevin hem **mem** hem de **str**/**wcs** sürümleri vardır. Duruma bağlı olarak herbirinin kullanılması gerekebilir. Yazılımınız dizileri veya bellek bloklarını değiştiriyorsa daima **mem** işlevlerini kullanmalısınız. Diğer taraftan, boş karakter sonlandırmalı dizgelerde değişiklik yapacağınız zaman da, dizgenin uzunluğunu biliyor olmadıkça, **str**/**wcs** işlevlerini kullanmak daha uygun olur. Bilinen boyutlarla geniş karakter dizilerinde **wmem** işlevleri kullanılmalıdır.

Bellek ve dizge işlevlerinin bazıları argüman olarak tek karakter alır. İşlevdeki parametre **int** türüyle bildirildiğinden, **char** türünden bir değer parametre olarak kullanıldığında özdevinimli olarak **int** türünden bir değere terfi ettirilir. geniş karakterli dizge işlevleri için de benzer bir durum sözkonusudur; tek geniş karakter için parametre **wchar_t** değil **wint_t** türündedir. Ancak bir çok gerçekleme **wchar_t** türü yeterince geniş olduğundan özdevinimli terfiye gerek duymaz. ISO C standardı belli türün seçilmesini gerektirmediğinden **wint_t** türü kullanılmıştır.

3. Dizge Uzunluğu

Bir dizgenin uzunluğunu **strlen** işlevini kullanarak alabilirsiniz. Bu işlev `string.h` başlık dosyasında bildirilmiştir.

```
size_t strlen(const char *s)                                    işlev
```

strlen işlevi boş karakter sonlandırmalı *s* dizgesinin bayt cinsinden uzunluğu ile döner. (Başka bir deyişle dizi içindeki boş karakterin indisi ile döner.) Örneğin,

```
strlen ("hello, world")
    => 12
```

Bir karakter dizisine uygulandığında **strlen** işlevi burada saklanan dizgenin uzunluğu ile döner, dizi için ayrılan boyutla değil. Dizi için ayrılan boyutu **sizeof** işleci ile öğrenebilirsiniz:

```
char dizge[32] = "hello, world";
sizeof (dizge)
    => 32
strlen (dizge)
    => 12
```

Yalnız dikkatli olun, *dizge* bir karakter dizisi değil de bir gösterici ise bu çalışmaz. Örneğin,

```
char dizge[32] = "hello, world";
char *ptr = dizge;
sizeof (dizge)
    => 32
sizeof (ptr)
    => 4  /* (göstericilerin 4 bayt olduğu bir makina üzerinde) */
```

Dizge argüman alan işlevlerle çalışırken bu yanlışı yapmak çok kolaydır; çünkü bu işlevlerin argümanları daima göstericidir, dizi değildir.

Ayrıca belirtmek gerekir ki, çokbaytlı kodlanmış dizgeler için dönen değer dizgedeki karakterlerin sayısı değildir. Bu değeri almak isterseniz dizge önce geniş karakterli bir dizgeye dönüştürülmeli ve **wcslen** işlevi kullanılmalı ya da aşağıdakine benzer bir kod kullanılmalıdır:

```
/* Girdi string içinde.
   Uzunluk ise n karakter umuluyor.  */
{
  mbstate_t t;
  char *scopy = string;
  /* dahili durum.  */
  memset (&t, '\0', sizeof (t));
  /* Karakter sayısını alalım.  */
  n = mbsrtowcs (NULL, &scopy, strlen (scopy), &t);
}
```

Bunun böyle yapılması kullanışsızdır. Karakter sayısına (bayt sayısı değil) ihtiyacınız varsa geniş karakterleri kullanmak en iyisidir.

İşlevin geniş karakterli eşdeğeri `wchar.h` başlık dosyasında bildirilmiştir.

`size_t` **wcslen**`(const wchar_t *`*ws*`)` işlev

wcslen işlevi, **strlen** işlevinin geniş karakterli eşdeğeridir. Dönen değer, *ws* ile gösterilen geniş karakter dizgesinin geniş karakter sayısıdır. (Bu değer aynı zamanda boş geniş karakter sonlandırmalı *ws* dizisinin boş geniş karakteri içeren elemanının indisidir.)

Bir karakteri oluşturmak için çoklu geniş karakter sıraları olmadığından dönen değer sadece dizideki bir indis değil, ayrıca geniş karakterlerin de sayısıdır.

Bu işlev ISO C99 standardının 1. düzeltmesinde tanımlıdır.

`size_t` **strnlen**`(const char *`*s*`,` işlev
`size_t` *enbüyük–uzunluk*`)`

strnlen işlevi, *s* dizgesinin uzunluğu *enbüyük–uzunluk* bayttan kısa ise bayt cinsinden uzunluğu ile döner. Aksi takdirde *enbüyük–uzunluk* ile döner. Diğer taraftan bu işlev aşağıdaki ifadeye eşdeğerdir:

`(`**strlen** `(`*s*`) < n ? strlen (`*s*`) :` *enbüyük–uzunluk*`)`

Bu daha etkilidir ve *s* dizgesi boş karakter sonlandırmalı değilse bile çalışır.

```
char dizge[32] = "hello, world";
strnlen (dizge, 32)
    => 12
strnlen (dizge, 5)
    => 5
```

Bu işlev bir GNU oluşumudur ve `string.h` başlık dosyasında bildirilmiştir.

`size_t` **wcsnlen**`(const wchar_t *`*ws*`,` işlev
`size_t` *enbüyük–uzunluk*`)`

wcsnlen işlevi, **strnlen** işlevinin geniş karakterli eşdeğeridir. *enbüyük–uzunluk* parametresi dizgenin içerebileceği en çok geniş karakterler sayısıdır.

Bu işlev bir GNU oluşumudur ve `wchar.h` başlık dosyasında bildirilmiştir.

4. Kopyalama ve Birleştirme

Bu kısımda açıklanan işlevleri dizi ve dizgelerin içeriğini kopyalamakta veya bir dizini içerini diğerine eklemekte kullanabilirsiniz. **str** ve **mem** işlevleri `string.h` başlık dosyasında, **wstr** ve **wmem** işlevleri ise `wchar.h` başlık dosyasında bildirilmiştir. Bu kısımdaki işlevlerin argümanlarının sırası hedef, kaynak şeklindedir ve işlevler daima hedef dizinin adresi ile dönerler.

Bu işlevlerin çoğu kaynak ve hedef dizilerinin birbirinin üstüne binmesi durumunda düzgün çalışmaz. Örneğin, kaynak dizisinin sonu, hedef dizisinin başlangıcını aşarsa, kaynak dizisinin başlangıcı, kendi sonunun üzerine yazılır ve bu durumda kaynak dizisini sonlandıran boş karakter kaybolur. Sonlandırıcı boş karakter olmadığından kopyalama işlemi tüm belleğin ayrılmasına kadar gidebilir.

Dizilerin kopyalanmasında birbirinin üzerine binmesi sorunu olan tüm işlevler bu kılavuzda açıkça belirtilmiştir. Bu kısımdaki işlevlere ek olarak **sprintf** (*Biçimli Çıktı İşlevleri* (sayfa: 263)) ve **scanf** (*Biçimli Girdi İşlevleri* (sayfa: 284)) gibi başka işlevler de vardır.

```
void *memcpy(void *restrict       hedef,                                  işlev
             const void *restrict kaynak,
             size_t               boyut)
```

memcpy işlevi *boyut* baytı *kaynak* adresinden başlayan nesneden *hedef* adresinden başlayan nesneye kopyalar. Bu işlevde *kaynak* dizisinin *hedef* dizisinin üzerine binmesi durumundaki davranışı tanımsızdır; üstüste binme olasılığı varsa **memmove** kullanın.

İşlev, *hedef*'in değeri ile döner.

Bu örnekte bir dizinin içeriğini kopyalamak için **memcpy** işlevinin nasıl kullanılacağı gösterilmiştir:

```
struct foo *eskidizi, *yenidizi;
int diziboyu;
…
memcpy (yeni, eski, diziboyu * sizeof (struct foo));
```

```
wchar_t *wmemcpy(wchar_t *restrict       hedef–geniş,                     işlev
                 const wchar_t *restrict kaynak–geniş,
                 size_t                  boyut)
```

wmemcpy işlevi *boyut* geniş karakteri *kaynak–geniş* adresinden başlayan nesneden *hedef–geniş* adresinden başlayan nesneye kopyalar. Bu işlevde *kaynak–geniş* dizisinin *hedef–geniş* dizisinin üzerine binmesi durumundaki davranışı tanımsızdır; üstüste binme olasılığı varsa **wmemmove** kullanın.

Aşağıda **wmemcpy** işlevinin olası bir gerçeklemesi vardır. Başka olası eniyilemeleri de vardır.

```
wchar_t *
wmemcpy (wchar_t *restrict hedefg, const wchar_t *restrict kaynakg,
         size_t boyut)
{
  return (wchar_t *) memcpy (hedefg, kaynakg, boyut * sizeof (wchar_t));
}
```

İşlev, *hedef–geniş*'in değeri ile döner.

Bu işlev ISO C90 standardının 1. düzeltmesinde tanımlanmıştır.

```
void *mempcpy(void *restrict       hedef,                                  işlev
              const void *restrict kaynak,
              size_t               boyut)
```

mempcpy işlevi **memcpy** işlevi ile hemen hemen eşdeğerdir. İşlev, *boyut* baytı *kaynak* adresinden başlayan nesneden *hedef* adresine kopyalar. Fakat dönen değer *hedef*'in değeri değildir. *hedef* başlangıcından itibaren yazılan son baytı izleyen baytın adresi ile döner. Yani, dönen değer:

```
((void *) ((char *) hedef + boyut))
```

Bu işlev çok sayıda nesnenin ardışık bellek konumlarına kopyalanması durumunda kullanışlıdır.

```
void *
combine (void *o1, size_t s1, void *o2, size_t s2)
{
  void *result = malloc (s1 + s2);
  if (result != NULL)
    mempcpy (mempcpy (result, o1, s1), o2, s2);
  return result;
}
```

Bu işlev bir GNU oluşumudur.

```
wchar_t *wmempcpy(wchar_t *restrict       hedef–geniş,                    işlev
                  const wchar_t *restrict kaynak–geniş,
                  size_t                  boyut)
```

wmempcpy işlevi **wmemcpy** işlevi ile hemen hemen eşdeğerdir. İşlev, *boyut* geniş karakteri *kaynak–geniş* adresinden başlayan nesneden *hedef–geniş* adresine kopyalar. Fakat dönen değer *hedef–geniş*'in değeri değildir. *hedef–geniş* başlangıcından itibaren yazılan son geniş karakteri izleyen geniş karakterin adresi ile döner. Yani, dönen değer:

hedef–geniş + boyut

Bu işlev çok sayıda nesnenin ardışık bellek konumlarına kopyalanması durumunda kullanışlıdır.

Aşağıda **wmempcpy** işlevinin olası bir gerçeklemesi vardır. Başka olası eniyilemeleri de vardır.

```
wchar_t *
wmempcpy (wchar_t *restrict hedefg, const wchar_t *restrict kaynakg,
          size_t boyut)
{
  return (wchar_t *) mempcpy (hedefg, kaynakg, boyut * sizeof (wchar_t));
}
```

Bu işlev bir GNU oluşumudur.

```
void *memmove(void       *hedef,                                          işlev
              const void *kaynak,
              size_t      boyut)
```

memmove işlevi *boyut* baytı *kaynak* dan *hedef* e birbirlerinin üstüne binseler bile kopyalar. Üstüste bime durumunda, işlev, *kaynak* bloğundaki baytları *hedef* bloğuna dikkatle kopyalar.

İşlev, *hedef*'in değeri ile döner.

```
wchar_t *wmemmove(wchar          *hedef–geniş,                            işlev
                  const wchar_t *kaynak–geniş,
                  size_t         boyut)
```

memmove işlevi *boyut* geniş karakteri *kaynak–geniş* den *hedef–geniş* e birbirlerinin üstüne binseler bile kopyalar. Üstüste bime durumunda, işlev, *kaynak–geniş* bloğundaki geniş karakterleri *hedef–geniş* bloğuna dikkatle kopyalar.

İşlev, *hedef–geniş*'in değeri ile döner.

Bu işlev bir GNU oluşumudur.

```
void *memccpy(void *restrict       hedef,                                  işlev
              const void *restrict kaynak,
              int                  c,
              size_t               boyut)
```

Bu işlev *boyut* baytlık dizinin içindeki *c* karakterine kadar olan kısmı *kaynak* dan *hedef* e kopyalar. Dönen değer, *hedef* e kopyalanan kısmın *c* karakterini izleyen baytının adresidir, eğer *boyut* bayt içinde *c* karakteri bulunamamışsa işlev boş gösterici ile döner.

```
void *memset(void  *blok,                                                  işlev
             int    c,
             size_t boyut)
```

Bu işlev, *c* değerini (**unsigned char** türüne dönüştürerek) *blok* adresinden başlayan nesnenin ilk *boyut* baytının her birine kopyalar ve *blok* un değeri ile döner.

```
wchar_t *wmemset(wchar_t *blok,                                            işlev
                 wchar_t  wc,
                 size_t   boyut)
```

Bu işlev, *wc* değerini *blok* adresinden başlayan nesnenin ilk *boyut* geniş karakterinin her birine kopyalar ve *blok* un değeri ile döner.

```
char *strcpy(char *restrict       hedef,                                   işlev
             const char *restrict kaynak)
```

strcpy işlevi *kaynak* dizgesini (sonlandırıcı boş karaktere kadar) *hedef* dizgesine kopyalar. Bu işlevde *kaynak* dizgesinin *hedef* dizgesinin üzerine binmesi durumundaki davranışı tanımsızdır. İşlev, *hedef*'in değeri ile döner.

```
wchar_t *wcscpy(wchar_t *restrict       hedef–geniş,                       işlev
                const wchar_t *restrict kaynak–geniş)
```

wcscpy işlevi *kaynak–geniş* dizgesini (sonlandırıcı boş geniş karaktere kadar) *hedef–geniş* dizgesine kopyalar. Bu işlevde *kaynak–geniş* dizgesinin *hedef–geniş* dizgesinin üzerine binmesi durumundaki davranışı tanımsızdır. İşlev, *hedef–geniş*'in değeri ile döner.

```
char *strncpy(char *restrict       hedef,                                  işlev
              const char *restrict kaynak,
              size_t               boyut)
```

Bu işlev *boyut* sayıda karakteri *hedef* dizgesine kopyalamak dışında **strcpy** işlevi gibidir.

kaynak dizgesinin uzunluğu *boyut* bayttan uzunsa işlev, ilk *boyut* karakteri kopyalar. Bu durumda *hedef* dizgesi boş karakteri içermez.

kaynak dizgesinin uzunluğu *boyut* bayttan kısaysa *kaynak* dizgesi ile boş karakterden sonraki *boyut* bayta kadar olan baytlar boş karakterlerle doldurularak *hedef* dizgesine kopyalanır. Bu işlev az kullanışlıdır ama ISO C standardında belirtilmiştir.

Üstüste binme durumunda **strncpy** işlevinin davranışı tanımsızdır.

strncpy işlevinin kullanımı **strcpy** işlevinin aksine *hedef* için ayrılan alanın dışına taşan yazma ile ilgili yazılım hatalarından kaçınmak için bir yöntem sunar. Ancak, yazılımı yavaşlatır, çünkü büyük ihtimalle nispeten küçük bir dizgenin, oldukça büyük bir alana kopyalanması sözkonusu olacak ve dizgeden artan boş alanın boş karakterlerle doldurulması için boşuna zaman harcanacaktır.

```
wchar_t *wcsncpy(wchar_t *restrict        hedef–geniş,                    işlev
                 const wchar_t *restrict  kaynak–geniş,
                 size_t                   boyut)
```

Bu işlev *boyut* sayıda geniş karakteri *hedef–geniş* dizgesine kopyalamak dışında **wcscpy** işlevi gibidir.

kaynak–geniş dizgesinin uzunluğu *boyut* geniş karakterden uzunsa işlev, ilk *boyut* geniş karakteri kopyalar. Bu durumda *hedef–geniş* dizgesi boş geniş karakteri içermez.

kaynak–geniş dizgesinin uzunluğu *boyut* geniş karakterden kısaysa *kaynak–geniş* dizgesi ile boş geniş karakterden sonraki *boyut* geniş karaktere kadar olan kısım boş geniş karakterlerle doldurularak *hedef–geniş* dizgesine kopyalanır. Bu işlev az kullanışlıdır ama ISO C standardında belirtilmiştir.

Üstüste binme durumunda **wcsncpy** işlevinin davranışı tanımsızdır.

wcsncpy işlevinin kullanımı **wcscpy** işlevinin aksine *hedef–geniş* için ayrılan alanın dışına taşan yazma ile ilgili yazılım hatalarından kaçınmak için bir yöntem sunar. Ancak, yazılımı yavaşlatır, çünkü büyük ihtimalle nispeten küçük bir dizgenin, oldukça büyük bir alana kopyalanması sözkonusu olacak ve dizgeden artan boş alanın boş geniş karakterlerle doldurulması için boşuna zaman harcanacaktır.

```
char *strdup(const char *s)                                               işlev
```

Bu işlev boş karakter sonlandırmalı *s* dizgesini yeni bir bellek alanına kopyalar. Yeni dizge için yer ayırma işlemi **malloc** ile yapılır; bkz. *Özgür Bellek Ayırma* (sayfa: 50). **malloc** yeni dizge için yer ayıramazsa işlev boş gösterici ile döner, aksi takdirde yeni dizge için bir gösterici ile döner.

```
wchar_t *wcsdup(const wchar_t *ws)                                        işlev
```

Bu işlev boş geniş karakter sonlandırmalı *ws* dizgesini yeni bir bellek alanına kopyalar. Yeni geniş karakterli dizge için yer ayırma işlemi **malloc** ile yapılır; bkz. *Özgür Bellek Ayırma* (sayfa: 50). **malloc** yeni geniş karakterli dizge için yer ayıramazsa işlev boş gösterici ile döner, aksi takdirde yeni geniş karakterli dizge için bir gösterici ile döner.

Bu işlev bir GNU oluşumudur.

```
char *strndup(const char *s,                                              işlev
              size_t      boyut)
```

Bu işlev en fazla *boyut* karakteri kopyalamak dışında **strdup** işlevi gibidir.

s dizgesinin uzunluğu *boyut* bayttan uzun ise işlev ilk *boyut* baytı kopyalar ve sonuna bir boş karakter ekler, aksi takdirde tüm karakterler kopyalanır ve dizge sonlandırılır.

Bu işlev hedef dizgeyi daima bir boş karakter ile sonlandırması bakımından da **strncpy** işlevinden farklıdır.

strndup işlevi bir GNU oluşumudur.

```
char *stpcpy(char *restrict        hedef,                                 işlev
             const char *restrict  kaynak)
```

Bu işlev *hedef* dizgesinin sonunu gösteren bir gösterici ile dönmesi dışında **strcpy** işlevi gibidir.

Örneğin, bu yazılımda `foo` ve `bar` birleştirilerek `foobar` üretilmekte ve çıktılanmaktadır:

```
#include <string.h>
#include <stdio.h>

int
main (void)
{
  char tampon[10];
  char *hedef = tampon;
  hedef = stpcpy (hedef, "foo");
  hedef = stpcpy (hedef, "bar");
  puts (tampon);
  return 0;
}
```

Bu işlev ISO ya da POSIX standardının parçası değildir ve Unix sistemleri ile ilgili bir özel işlev de değildir. Tabii ki MS–DOS'dan geliyor.

Dizgelerin birbirinin üstüne binmesi durumundaki davranış tanımsızdır. İşlev `string.h` başlık dosyasında bildirilmiştir.

```
wchar_t *wcpcpy(wchar_t *restrict       hedef–geniş,                    işlev
                const wchar_t *restrict kaynak–geniş)
```

Bu işlev *hedef–geniş* dizgesinin sonunu gösteren bir gösterici ile dönmesi dışında **wcscpy** işlevi gibidir. İşlev kopyalanan dizgenin başlangıç adresi ile değil, dizgeyi sonlandıran boş geniş karakterin adresi ile döner.

Bu işlev bir ISO veya POSIX oluşumu değildir, ancak GNU C kütüphanesi geliştirilirken kullanışlı bulunmuştur.

Üstüste binme durumunda **wcpcpy** işlevinin davranışı tanımsızdır.

wcpcpy bir GNU oluşumudur ve `wchar.h` başlık dosyasında bildirilmiştir.

```
char *stpncpy(char *restrict       hedef,                               işlev
              const char *restrict kaynak,
              size_t               boyut)
```

Bu işlev, *hedef* e ilk *boyut* karakteri kopyalaması dışında **stpcpy** gibidir.

kaynak dizgesinin uzunluğu *boyut* bayttan uzunsa işlev, ilk *boyut* karakteri kopyalar ve kopyalanan son karakterden sonraki baytın adresi ile döner. Bu durumda *hedef* dizgesi boş karakteri içermez.

kaynak dizgesinin uzunluğu *boyut* bayttan kısaysa *kaynak* dizgesinin boş karakterinden sonraki *boyut* bayta kadar olan baytlar boş karakterlerle doldurularak *hedef* dizgesine kopyalanır ve yazılan **ilk** boş karakterin adresi ile döner. Bu işlev az kullanışlıdır ama **strncpy** işlevinin davranışı kullanışlı bulunduğundan gerçeklenmiştir.

Bu işlev bir ISO veya POSIX oluşumu değildir, ancak GNU C kütüphanesi geliştirilirken kullanışlı bulunmuştur.

Dizgelerin birbirinin üstüne binmesi durumundaki davranış tanımsızdır. İşlev `string.h` başlık dosyasında bildirilmiştir.

```
wchar_t *wcpncpy(wchar_t *restrict        hedef–geniş,                    işlev
                 const wchar_t *restrict  kaynak–geniş,
                 size_t                   boyut)
```

Bu işlev, *hedef–geniş* e ilk *boyut* karakteri kopyalaması dışında **wcpcpy** gibidir.

kaynak–geniş dizgesinin uzunluğu *boyut* bayttan uzunsa işlev, ilk *boyut* geniş karakteri kopyalar ve kopyalanan son boş olmayan geniş karakterden sonraki geniş karaktere bir gösterici ile döner. Bu durumda *hedef–geniş* dizgesi boş geniş karakteri içermez.

kaynak–geniş dizgesinin uzunluğu *boyut* bayttan kısaysa *kaynak–geniş* dizgesinin boş geniş karakterinden sonraki *boyut* bayta kadar olan alanlar boş geniş karakterlerle doldurularak *hedef–geniş* dizgesine kopyalanır ve yazılan **ilk** boş geniş karakterin adresi ile döner. Bu işlev az kullanışlıdır ama **wcsncpy** işlevinin davranışı kullanışlı bulunduğundan gerçeklenmiştir.

Bu işlev bir ISO veya POSIX oluşumu değildir, ancak GNU C kütüphanesi geliştirilirken kullanışlı bulunmuştur.

Üstüste binme durumunda **wcpncpy** işlevinin davranışı tanımsızdır.

wcpncpy bir GNU oluşumudur ve `wchar.h` başlık dosyasında bildirilmiştir.

```
char *strdupa(const char *s)                                              makro
```

Bu makro, **malloc** (*Değişken Boyutlu Özdevinimli Saklama* (sayfa: 75)) yerine **alloca** kullanarak yeni bir dizge ayırmak dışında **strdup** gibidir. Yani, **alloca** kullanarak ayrılan bellek blokları ile aynı sınırlamalara sahip bir dizi döner.

Çeşitli sebeplerle **strdupa** sadece bir makro olarak gerçeklenmiştir; yani bir işlev gibi adresini alamazsınız. Bu sınırlama dışında bir işlev olarak kullanışlıdır. Aşağıdaki kodda **malloc** kullanıldığında işlemin ne kadar pahalıya malolduğu gösterilmiştir.

```
#include <paths.h>
#include <string.h>
#include <stdio.h>

const char path[] = _PATH_STDPATH;

int
main (void)
{
  char *wr_path = strdupa (path);
  char *cp = strtok (wr_path, ":");

  while (cp != NULL)
    {
      puts (cp);
      cp = strtok (NULL, ":");
    }
  return 0;
}
```

strtok'un doğrudan *path* kullanılarak çağrılmasının geçersiz olduğuna dikkat edin. Ayrıca, **strdupa** parametre aktarımında girişime sebep olabilen **alloca**'yı (*Değişken Boyutlu Özdevinimli Saklama* (sayfa: 75)) kullandığından **strtok**'un argüman listesinde **strdupa** kullanımına izin verilmez.

Bu işlev sadece GCC ile kullanılabilir.

```
char *strndupa(const char *s,                                    makro
               size_t      boyut)
```

Bu işlev, **alloca** (sayfa: 75) kullanarak yeni bir dizge ayırmak dışında **strndup** gibidir. **strdupa** için geçerli olan yararlar ve sınırlamalar, **strndupa** için de geçerlidir.

Bu işlev de **strdupa** gibi sadece makro olarak gerçeklenmiştir. **strdupa** gibi bu makro da bir işlev çağrısının argüman listesinde kullanılamaz.

Bu işlev sadece GCC ile kullanılabilir.

```
char *strcat(char *restrict       hedef,                         işlev
             const char *restrict kaynak)
```

kaynak dizgesini *hedef* dizgesinin üzerine yazmak yerine *hedef* dizgesinin sonuna ekleyerek onunla birleştirmesi dışında **strcpy** işlevine benzer. *kaynak* dizgesinin ilk karakteri *hedef* dizgesini sonlandıran boş karakterin üzerine yazılır.

strcat işlevinin tanımı aşağıdakine eşdeğerdir:

```
char *
strcat (char *restrict hedef, const char *restrict kaynak)
{
  strcpy (hedef + strlen (hedef), kaynak);
  return hedef;
}
```

Dizgelerin üstüste binmesi durumu için bu işlevin davranışı tanımlanmamıştır.

```
wchar_t *wcscat(wchar_t *restrict       hedef–geniş,              işlev
                const wchar_t *restrict kaynak–geniş)
```

kaynak–geniş dizgesini *hedef–geniş* dizgesinin üzerine yazmak yerine *hedef–geniş* dizgesinin sonuna ekleyerek onunla birleştirmesi dışında **wcscpy** işlevine benzer. *kaynak–geniş* dizgesinin ilk geniş karakteri*hedef–geniş* dizgesini sonlandıran boş geniş karakterin üzerine yazılır.

wcscat işlevinin tanımı aşağıdakine eşdeğerdir:

```
wchar_t *
wcscat (wchar_t *hedefg, const wchar_t *kaynakg)
{
  wcscpy (hedefg + wcslen (hedefg), kaynakg);
  return hedefg;
}
```

Dizgelerin üstüste binmesi durumu için bu işlevin davranışı tanımlanmamıştır.

strcat ya da **wcscat** (ardından da **strncat** ya da **wcsncar**) işlevlerini kullanan yazılımcılar kolayca ihtiyatsız ve hata deli olarak tanımlanabilir. Hemen hemen her durumda, ortak dizge uzunlukları bilinir (hem, tampon boyutunun yeterli olacağını kim bilebilir ki?). Veya en azından, çeşitli işlev çağrılarının sonuçları izlenerek bilinir. Ama yine de **strcat**/**wcscat** işlevleri yetersizdir. Kopyalama başladığında hedef dizgenin sonunu bulmak için gereksiz zaman harcanacaktır. Aşağıdaki örneğe bakınız:

```
/* Bu işlev birçok dizgeyi birleştirir.
   Son parametre NULL olmalıdır.  */
char *
concat (const char *dizge, …)
{
```

```
  va_list ap, ap2;
  size_t toplam = 1;
  const char *s;
  char *sonuc;

  va_start (ap, dizge);
  /* Aslında va_copy, ama geç dönem gcc sürümleri
     bu ismi tanır.
  __va_copy (ap2, ap);

  /* Bakalım, ne kadar yer lazımmış.  */
  for (s = dizge; s != NULL; s = va_arg (ap, const char *))
    toplam += strlen (s);

  va_end (ap);

  sonuc = (char *) malloc (toplam);
  if (sonuc != NULL)
    {
      sonuc[0] = '\0';

      /* Dizgeleri kopyalayalım.  */
      for (s = dizge; s != NULL; s = va_arg (ap2, const char *))
        strcat (sonuc, s);
    }

  va_end (ap2);

  return sonuc;
}
```

Basit gibi görünüyor, özellikler dizgelerin kopyalandığı ikinci döngü. Her biri 100 baytlık on dizgenin birleştirildiğini varsayalım. Aradığımız ikinci dizge için dizgenin sonunda zaten 100 bayt ayrıldığından sonraki dizgeyi ekleyebiliriz. Toplamda tüm dizgeler için ara sonuçlarla birlikte uzunluk 5500! olur. Bellek ayırma için yapılan arama ile kopyalamayı birleştirisek bu işlevi daha verimli kılabiliriz:

```
char *
concat (const char *dizge, …)
{
  va_list ap;
  size_t tutulan = 100;
  char *sonuc = (char *) malloc (tutulan);

  if (sonuc != NULL)
    {
      char *yeniwp;
      char *wp;

      va_start (ap, dizge);

      wp = sonuc;
      for (s = dizge; s != NULL; s = va_arg (ap, const char *))
        {
          size_t uzunluk = strlen (s);

          /* Gerekiyorsa, tutulan belleği arttıralım.  */
          if (wp + uzunluk + 1 > sonuc + tutulan)
```

```
            {
              tutulan = (tutulan + uzunluk) * 2;
              yeniwp = (char *) realloc (sonuc, tutulan);
              if (newp == NULL)
                {
                  free (sonuc);
                  return NULL;
                }
              wp = yeniwp + (wp - sonuc);
              sonuc = yeniwp;
            }

          wp = mempcpy (wp, s, uzunluk);
        }

      /* sonuclanan dizgeyi sonlandıralım.  */
      *wp++ = '\0';

      /* Belleği tam istenen boyuta ayarlayalım.  */
      yeniwp = realloc (sonuc, wp - sonuc);
      if (yeniwp != NULL)
        sonuc = yeniwp;

      va_end (ap);
    }

  return sonuc;
}
```

Girdi dizgeleri hakkına biraz daha fazla bilgi birikimi kullanarak bellek ayırma işlemi daha hassas yapılabilirdi. Burada farklı olarak **strcat** işlevini asla kullanmadık. Ara sonuçların izini sürerek dizge sonlarının bulunmasını güvenceye aldık ve **mempcpy** işlevini kullandık. Ayrıca, dizgeleri elde etmek bakımından daha doğal görünen **stpcpy** işlevini de kullanmadık. Ama zaten, dizge uzunluğunu bildiğimizden bu gerekli değildi ve bu sebeple daha hızlı bellek kopyalaması yapabildik. Örneğimiz geniş karakterler için de aynı şekilde çalışırdı.

Bir yazılımcı **strcat** kullanmaya karar vermeden önce iki kere düşünmeli ve hesaplanmış sonuçların getirilerinden yararlanmak için kod tekrar yazılamaz mı acaba diye bakmalıdır. Tekrarlayalım: **strcat** işlevinin kullanılması hemen hemen daima gereksizdir.

```
char *strncat(char *restrict       hedef,                                  işlev
              const char *restrict kaynak,
              size_t               boyut)
```

Bu işlev, *kaynak* dizgesinin ilk *boyut* karakterini *hedef* dizgesinin sonuna eklemek dışında **strcat** gibidir. *hedef* dizgesinin sonuna daima bir boş karakter eklendiğinden *hedef* için ayrılan yer en azından *boyut* **+ 1** bayt olmalıdır.

strncat işlevi aşağıdaki gibi de gerçekleştirilebilirdi:

```
char *
strncat (char *hedef, const char *kaynak, size_t boyut)
{
  hedef[strlen (hedef) + boyut] = '\0';
  strncpy (hedef + strlen (hedef), kaynak, boyut);
  return hedef;
}
```

Dizgelerin üstüste binmesi durumu için bu işlevin davranışı tanımlanmamıştır.

```
wchar_t *wcsncat(wchar_t *restrict        hedef–geniş,                    işlev
                 const wchar_t *restrict  kaynak–geniş,
                 size_t                   boyut)
```

Bu işlev, *kaynak–geniş* dizgesinin ilk *boyut* geniş karakterini *hedef–geniş* dizgesinin sonuna eklemek dışında **wcscat** gibidir. *hedef–geniş* dizgesinin sonuna daima bir boş geniş karakter eklendiğinden *hedef–geniş* için ayrılan yer en azından *boyut* **+ 1** bayt olmalıdır.

wcscat işlevi aşağıdaki gibi de gerçekleştirilebilirdi:

```
wchar_t *
wcsncat (wchar_t *restrict hedefg, const wchar_t *restrict kaynakg,
         size_t boyut)
{
  hedefg[wcslen (hedefg) + boyut] = L'\0';
  wcsncpy (hedefg + wcslen (hedefg), kaynakg, boyut);
  return hedefg;
}
```

Dizgelerin üstüste binmesi durumu için bu işlevin davranışı tanımlanmamıştır.

Aşağıdaki örnekte **strncpy** ve **strncat** kullanımı gösterilmiştir (Geniş karakterli olarak benzeri yazılabilir). **strncat** çağrısına dikkat edin, *boyut* parametresi **tampon** karakter dizisinin üste binmesine karşı korunarak hesaplanmıştır.

```
#include <string.h>
#include <stdio.h>

#define BOYUT 12

static char tampon[BOYUT];

main ()
{
  strncpy (tampon, "herkese", BOYUT);
  puts (tampon);
  strncat (tampon, " merhaba", BOYUT - strlen (tampon) - 1);
  puts (tampon);
}
```

Kod aşağıdaki gibi bir çıktı üretir:

```
herkese
herkese mer
```

```
void bcopy(const void *kaynak,                                            işlev
           void       *hedef,
           size_t      boyut)
```

Bu işlev, BSD'den türetilen, **memmove**'un kısmen eskimiş bir alternafidir. Ancak **memmove** ile eşdeğer değildir, çünkü argümanları aynı sırada değildir ve dönen bir değer yoktur.

```
void bzero(void  *blok,                                                   işlev
           size_t boyut)
```

Bu işlev, BSD'den türetilen, **memset**'in kısmen eskimiş bir alternafidir. Ancak **memset** ile eşdeğer değildir, çünkü sakladığı tek değer sıfırdır.

5. Dizi/Dizge Karşılaştırması

Bu kısımdaji işlevleri dizi ve dizge içerikleri üzerinde karşılaştımalar yapmak için kullanabilirsiniz. Eşitliğin denetlenmesi kadar, ayrıca sıralama işlemleri için sıralama işlevleri olarak da kullanılabilirler. Bunun bir örneği olarak *Arama ve Sıralama* (sayfa: 203) bölümüne bakınız.

C'deki çoğu karşılaştırma işleminin aksine, dizge karşılaştırma işlevleri dizgelerin eşitliğinin varlığında değil *yokluğunda* sıfırdan farklı bir değer ile dönerler. İşaretin değeri eşdeğer olmayan dizgelerin ilk karakterlerinin birbirine göre sırasına bağlıdır: bir negatif değer ilk dizgenin ikincisinden "küçük", pozitif bir değer ise "büyük" olduğunu gösterir.

Bu işlevler çoğunlukla sadece eşitlik denetiminde kullanılırlar. Bu, ifade yazım kurallarına uygun olarak **! strcmp (s1, s2)** gibi bir ifade ile yapılır.

Bu işlevlerin tümü `string.h` başlık dosyasında bildirilmiştir.

```
int memcmp(const void *a1,                                          işlev
           const void *a2,
           size_t      boyut)
```

memcmp işlevi belleğin *a1* de başlayan *boyut* baytı ile *a2* de başlayan *boyut* baytını karşılaştırır. Dönen değer, ilk farklı bayt çiftleri arasındaki farkın işaretine bağlıdır (önce **unsigned char** türünden nesneler olarak yorumlanır, sonra da **int** türüne terfi ettirilirler).

İki bloğun içeriği aynı ise **memcmp** sıfır ile döner.

```
int wmemcmp(const wchar_t *a1,                                      işlev
            const wchar_t *a2,
            size_t         boyut)
```

wmemcmp işlevi başlangıcı *a1* de olan geniş karakterlerle başlangıcı *a2* de olan geniş karakterleri karşılaştırır. *a1* deki *a2* dekinden farklı ilk geniş karakterin *a2* dekinden küçük ya da büyük olmasına bağlı olarak sıfırdan küçük ya da büyük bir değerle döner.

İki bloğun içeriği aynı ise **wmemcmp** sıfır ile döner.

Keyfi değerli dizilerde **memcmp** işlevi çoğunlukla eşitlik denetiminde kullanılır. Genellikle, bayt dizileri dışındaki dizilerde bayt seviyesinde bir karşılaştırma anlamı değildir. Örneğin gerçek sayılardan oluşan dizilerin bayt seviyesinde karşılaştırılması gerçek sayıların değerleri arasındaki ilişki hakkında hiçbir şey söyleyemez.

wmemcmp işlevi, bir defada **sizeof (wchar_t)** bayta baktığından ve bunların sayısı sistem bağımlı olduğundan aslında sadece **wchar_t** türünde dizilerde kullanışlıdır.

Ayrıca **memcmp** işlevini, hizalama gereksinimlerinden dolayı aralarında boşluk bırakılmış yapı nesneleri, sonlarında fazladan boşluk bulunan birleşik yapılar ve ayrılan yere göre küçük olduklarından sonunda fazladan boşluk bulunan dizgeler gibi "delikler" içeren nesneleri karşılaştırırken dikkatli olmalısınız. Bu "delikler"lerin içerikleri düzensizdir ve bayt seviyesinden karşılaştırmalar tuhaf sonuçlara sebep olabilir.

Örneğin, aşağıdaki gibi bir yapı türü tanımında:

```
struct foo
  {
    unsigned char tag;
    union
```

```
      {
        double f;
        long i;
        char *p;
      } value;
  };
```

struct foo türündeki gibi nesneleri **memcmp** ile karşılaştırmak yerine özelleştirilmiş karşılaştırma işlevlerinin yazılmasını tercih etmelisiniz.

```
int strcmp(const char *s1,                                          işlev
           const char *s2)
```

strcmp işlevi *s1* ve *s2* dizgelerini karşılaştırır. Dönen değer, ilk farklı karakter çiftleri arasındaki farkın işaretine bağlıdır (önce **unsigned char** türünden nesneler olarak yorumlanır, sonra da **int** türüne terfi ettirilirler).

İki dizgenin içeriği aynı ise **strcmp** sıfır ile döner.

strcmp işlevi tarafından kullanılan sıralamanın bir sonucu olarak, eğer *s1* dizgesi *s2* dizgesinin bir alt dizgesi ise *s1* dizgesi, *s2* den "küçük" kabul edilir.

strcmp işlevi dizgelerin yazıldıkları dilin sıralama teamüllerini almaz. Bunlardan birini almak için **strcoll** kullanın.

```
int wcscmp(const wchar_t *ws1,                                      işlev
           const wchar_t *ws2)
```

wcscmp işlevi *ws1* ve *ws2* dizgelerini karşılaştırır. *ws1* deki *ws2* dekinden farklı ilk geniş karakterin *ws2* dekinden küçük ya da büyük olmasına bağlı olarak sıfırdan küçük ya da büyük bir değerle döner.

İki dizgenin içeriği aynı ise **strcmp** sıfır ile döner.

wcscmp işlevi tarafından kullanılan sıralamanın bir sonucu olarak, eğer *ws1* dizgesi *ws2* dizgesinin bir alt dizgesi ise *ws1* dizgesi, *ws2* den "küçük" kabul edilir.

wcscmp işlevi dizgelerin yazıldıkları dilin sıralama teamüllerini almaz. Bunlardan birini almak için **wcscoll** kullanın.

```
int strcasecmp(const char *s1,                                      işlev
               const char *s2)
```

Bu işlev harf büyüklükleri arasındaki farkları yoksayması dışında **strcmp** gibidir. Küçük ve büyük harfler arasındaki ilişki o an seçilmiş olan yerele bağlıdır. Standart **"C"** yerelinde Ä ve ä karakterleri eşleşmez ama, bu karakterleri alfabelerinde kullanan yerellerde eşleşirler.

strcasecmp BSD'den alınmıştır.

```
int wcscasecmp(const wchar_t *ws1,                                  işlev
               const wchar_T *ws2)
```

Bu işlev harf büyüklükleri arasındaki farkları yoksayması dışında **>wcscmp** gibidir. Küçük ve büyük harfler arasındaki ilişki o an seçilmiş olan yerele bağlıdır. Standart **"C"** yerelinde Ä ve ä karakterleri eşleşmez ama, bu karakterleri alfabelerinde kullanan yerellerde eşleşirler.

wcscasecmp işlevi bir GNU oluşumudur.

```
int strncmp(const char *s1,                                          işlev
            const char *s2,
            size_t      boyut)
```

Bu işlev ilk *boyut* karakterin karşılaştırılması dışında **strcmp** gibidir. Başka bir deyişle, iki dizgenin ilk *boyut* karakteri aynıysa işlev sıfırla döner.

```
int wcsncmp(const wchar_t *ws1,                                      işlev
            const wchar_t *ws2,
            size_t         boyut)
```

Bu işlev ilk *boyut* geniş karakterin karşılaştırılması dışında **wcscmp** gibidir. Başka bir deyişle, iki dizgenin ilk *boyut* geniş karakteri aynıysa işlev sıfırla döner.

```
int strncasecmp(const char *s1,                                      işlev
                const char *s2,
                size_t      n)
```

Bu işlev harf büyüklükleri arasındaki farkları yoksayması dışında **strncmp** gibidir. **strcasecmp** gibi küçük ve büyük harfler arasındaki ilişki o an seçilmiş olan yerele bağlıdır.

strncasecmp işlevi bir GNU oluşumudur.

```
int wcsncasecmp(const wchar_t *ws1,                                  işlev
                const wchar_t *s2,
                size_t         n)
```

Bu işlev harf büyüklükleri arasındaki farkları yoksayması dışında **wcsncmp** gibidir. **wcscasecmp** gibi küçük ve büyük harfler arasındaki ilişki o an seçilmiş olan yerele bağlıdır.

wcsncasecmp işlevi bir GNU oluşumudur.

Aşağıdaki örnkelerde **strcmp** ve **strncmp** kullanımı gösterilmiştir. Bu örneklerin benzerleri geniş karakterler için de yazılabilir. Bu örneklerde ASCII karakter ümesinin kullanıldığı varsayılmıştır. (Diğer karakter kümeleri — EBCDIC diyelim — kullanılırsa, harfler farklı sayısal değerlerle ilişkilendirileceğinden dönen değerler ve sıralama farklı olacaktır.)

```
strcmp ("hello", "hello";)
    => 0    /* Bu iki dizge aynıdır. */
strcmp ("hello", "Hello")
    => 32   /* Karşılaştırma harf büyüklüklerine duyarlıdır. */
strcmp ("hello", "world")
    => -15  /* 'h' karakteri 'w'den öncedir. */
strcmp ("hello", "hello, world")
    => -44  /* Bir boş karaktere karşılık virgül. */
strncmp ("hello", "hello, world", 5)
    => 0    /* İlk 5 karakter aynıdır. */
strncmp ("hello, world", "hello, stupid world!!!", 5)
    => 0    /* İlk 5 karakter aynıdır. */
```

```
int strverscmp(const char *s1,                                       işlev
               const char *s2)
```

strverscmp işlevi *s1* ve *s2* dizgelerini indis/sürüm numarası içerdiğini varsayarak karşılaştırır. Dönen değer **strcmp** işlevindeki gibidir. Aslında, dizgeler bir rakam içermiyorsa işlev, **strcmp** işlevi gibi davranır.

Temel olarak, dizgeler için normal karşılaştırma (karakter karakter) uygulanır, dizgelerden birinde bir rakama rastlandığı anda özel karşılaştırma kipine girilir ve birbirini izleyen rakamlar bir bütün olarak ele alınır, sayılar arasındabir fark tespit edilemezse tekrar normal karşılaştırma kipine dönülür. İki tür sayılar vardır: "tamsayı" ve "ondalık kısım" (bunlar '0' ile başlar). Bu sayı türleri sıralamayı aşağıdaki sırada etkiler:

- tamsayı/tamsayı: bildik şekilde karşılaştırılır.
- ondalık/tamsayı: ondalık sayı tamsayıdan küçüktür. Burada da bir sürpriz yok.
- ondalık/ondalık: burası biraz karışık. Ondalık kısımların baştan en uzun sıfırlı olanı diğerinden küçüktür; aksi takdirde normal karşılaştırma yapılır.

```
strverscmp ("rakam yok", "rakam yok")
    => 0    /* strcmp ile aynı davranış. */
strverscmp ("item#99", "item#100")
    => <0   /* önekleri aynı, ama 99 < 100. */
strverscmp ("alpha1", "alpha001")
    => >0   /* ondalık kısım tamsayıdan küçüktür. */
strverscmp ("part1_f012", "part1_f01")
    => >0   /* iki ondalık kısım. */
strverscmp ("foo.009", "foo.0")
    => <0   /* keza, ama sıfır sayısı öncelikli. */
```

Bu işlev özellikle dosya ismi sıralamasında kullanılır, çünkü dosya isimleri genellikle indis/sürüm numaraları içerir.

strverscmp işlevi bir GNU oluşumudur.

```
int bcmp(const void *a1,                                                    işlev
         const void *a2,
         size_t      boyut)
```

Bu işlev BSD'den alınmıştır ve **memcmp** için eskimiş bir isimdir.

6. Dizgeleri Yerele Özgü Karşılaştırma İşlevleri

Bazı yerellerde alfabetik sıralama karakter kodlarının sıralamasından farklıdır.[2]

strcoll, **strxfrm**, (`string.h` dosyasında bildirilmişlerdir) **wcscoll** ve **wcsxfrm** (`wchar.h` dosyasında bildirilmişlerdir) işlevlerini yerele özgü karakter sıralamasına uygun dizge karşılaştırmalarında kullanabilirsiniz. Bu işlevlerin kullanacağı yereli **LC_COLLATE** yerel kategorisine gerekli değeri atayarak belirtebilirsiniz. Daha ayrıntılı bilgi edinmek için *Yereller ve Uluslararasılaştırma* (sayfa: 164) kısmına bakınız.

Standart C yerelinde karakter sıralaması bakımından **strcoll** ile **strcmp** işlevlerinin davranışlarında bir fark yoktur. Benzer olarak **wcscoll** ve **wcscmp** işlevleri de bu bakımdan aynıdır.

Bu işlevleri etkin olarak çalıştırmanın yolu bir dizgenin içindeki karakterleri yerelin karakter sıralamasına uygun bir konumlamayla bir bayt sıralamasına dönüştürmektir. Böyle oluşturulmuş bayt sıralamaları ile yerelin karakter sıralamasına uygun olarak dizgeleri karşılaştırmak artık kolaydır.

strcoll ve **wcscoll** işlevleri bu dönüşümü karşılaştırma sırasından dolaylı olarak uygularlar. **strxfrm** ve **wcsxfrm** işlevleri ise tam aksine doğrudan doğruya karakter sıralaması/alfabetik sıralama eşleştirmesi yaparlar. Bir dizge kümesi üzerinde çok sayıda karşılaştırma yapacaksanız önce **strxfrm** veya **wcsxfrm** işlevlerini kullararak dizgeleri bir kerede dönüştürüp ardından **strcmp** veya **wcscmp** ile dönüştürülmüş dizgeleri karşılaştırmak daha verimli bir yöntemdir.

```
int strcoll(const char *s1,                                                 işlev
            const char *s2)
```

Bu işlev, karakter sıralaması için yerelin (**LC_COLLATE** yereli) karakter sıralamasını kullanması dışında **strcmp** işlevine benzer.

```
int wcscoll(const wchar_t *ws1,                                    işlev
            const wchar_t *ws2)
```

Bu işlev, karakter sıralaması için yerelin (**LC_COLLATE** yereli) karakter sıralamasını kullanması dışında **wcscmp** işlevine benzer.

Aşağıdaki örnekte bir dizge dizisi **strcoll** ile karşılaştırılarak sıralanmaktadır. Burada gerçek sıralama algoritması yazılmamıştır; bunun için **qsort** (*Dizi Sıralama İşlevi* (sayfa: 204)) kullanılmıştır. Buradaki kodun yaptığı iş dizgeler sıralanırken nasıl karşılaştırılacağını göstermektir. (Bu bölümü devamında **strxfrm** kullanarak bunun daha verimli olarak nasıl yapılacağından bahsedilecektir.)

```
/* Bu, qsort ile kullanılan bir karşılaştırma işlevidir. */

int
elemanlari_karsilastir (char **p1, char **p2)
{
  return strcoll (*p1, *p2);
}

/* Burası giriş noktası---yerelin karakter sıralaması
   kullanılarak dizgeleri sıralayan işlev. */

void
sort_strings (char **dizi, int dizge_sayisi)
{
  /* Dizgeleri karşılaştırarak diziyi sırala. */
  qsort (dizi, dizge_sayisi,
         sizeof (char *), elemanlari_karsilastir);
}
```

```
size_t strxfrm(char *restrict       hedef,                         işlev
               const char *restrict kaynak,
               size_t               boyut)
```

strxfrm işlevi, harf sıralaması için seçilmiş olan yerele göre saptanan karakter dönüşümünü kullanarak *kaynak* dizgesini dönüştürür ve dönüştürülen *boyut* karakterlik dizgeyi (sonlandırıcı boş karakter dahil) *hedef* dizisine kaydeder.

hedef ve *kaynak* birbirini eziyorsa işlevin davranışı tanımsızdır. Daha fazla bilgi için *Kopyalama ve Birleştirme* (sayfa: 94) bölümüne bakınız.

Dönen değer dönüştürülen dizgenin uzunluğudur. Bu değer *boyut* değerinden etkilenmez, ancak *boyut*'tan büyük ya da eşitse, dönüştürülen dizge *hedef* dizisine sığmamış demektir. Bu durumda, diziye sığdığı kadarıyla dizge kaydedilmiştir. Dönüştürülen dizgenin tamamını almak için işlevi daha büyük bir dizi ile tekrar çağırmalısınız.

Dönüştürülmüş dizge verilen dizgeden daha uzun olabileceği gibi daha kısa da olabilir.

boyut sıfırsa, *hedef* dizisine herhangi bir değer kaydedilmez. Bu durumda, **strxfrm** işlevi sadece dönüştürülmüş dizgenin uzunluğu ile döner. Bu ayrılacak dizinin boyunun ne olacağını saptamak açısından yararlıdır.*boyut* sıfır olduğunda *hedef*'in önemi yoktur, bir boş dizge bile olabilir.

```
size_t wcsxfrm(wchar_t *restrict hedef–geniş,                    işlev
               const wchar_t     *kaynak–geniş,
               size_t             boyut)
```

wcsxfrm işlevi, harf sıralaması için seçilmiş olan yerele göre saptanan karakter dönüşümünü kullanarak *kaynak–geniş* dizgesini dönüştürür ve dönüştürülen *boyut* geniş karakterlik dizgeyi (sonlandırıcı boş karakter dahil) *hedef–geniş* dizisine kaydeder.

hedef–geniş ve *kaynak–geniş* birbirini eziyorsa işlevin davranışı tanımsızdır. Daha fazla bilgi için *Kopyalama ve Birleştirme* (sayfa: 94) bölümüne bakınız.

Dönen değer dönüştürülen deniş karakterli dizgenin uzunluğudur. Bu değer *boyut* değerinden etkilenmez, ancak *boyut*'tan büyük ya da eşitse, dönüştürülen geniş karakterli dizge *hedef–geniş* dizisine sığmamış demektir. Bu durumda, geniş karakterli dizge, diziye sığdığı kadarıyla kaydedilmiştir. Dönüştürülen geniş karakterli dizgenin tamamını almak için işlevi daha büyük bir dizi ile tekrar çağırmalısınız.

Dönüştürülmüş geniş karakterli dizge verilen geniş karakterli dizgeden daha uzun olabileceği gibi daha kısa da olabilir.

boyut sıfırsa, *hedef–geniş* dizisine herhangi bir değer kaydedilmez. Bu durumda, **wcsxfrm** işlevi sadece dönüştürülmüş geniş karakterli dizgenin uzunluğu ile döner. Bu ayrılacak dizinin boyunun ne olacağını saptamak açısından yararlıdır (değeri **sizeof (wchar_t)** ile çarpmayı unutmayın). *boyut* sıfır olduğunda *hedef–geniş*'in önemi yoktur, bir boş dizge bile olabilir.

Buradaki örnekte, çok sayıda karşılaştırma yapmayı planladığınızda **strxfrm** işlevini nasıl kullanacağınız gösterilmiştir. Önceki örnekle aynı şeyi yapsa da daha hızlıdır, çünkü her dizgeye sadece bir kere dönüşüm uygular, diğer dizgelerle kaç defa karşılaştırma yapıldığının önemi yoktur. Çok sayıda dizge olduğunda kazanılan zaman, bellek ayırmak ve serbest bırakmak için harcanan zamandan bile daha büyüktür.

```
struct siralayici { char *girdi; char *donusmus; };

/* Bu karşılaştırma işlevi, struct siralayici
   dizisini sıralamak için qsort ile birlikte
   kullanılmıştır. */

int
elemanlari_karsilastir (struct sorter *p1, struct sorter *p2)
{
  return strcmp (p1->donusmus, p2->donusmus);
}

/* Burası giriş noktası---yerelin karakter sıralaması
   kullanılarak dizgeleri sıralayan işlev. */

void
dizgeleri_hizli_sirala (char **dizi, int dizge_sayisi)
{
  struct siralayici gecici_dizi[dizge_sayisi];
  int i;

  /* gecici_dizi yi ilklendirelim.
     Her eleman bir girdi dizgesi ve onun dönüştürülmüşünü
     içersin. */
  for (i = 0; i < dizge_sayisi; i++)
    {
      size_t uzunluk = strlen (dizi[i]) * 2;
```

```
      char *donusmus;
      size_t donusmus_uzunluk;

      gecici_dizi[i].girdi = dizi[i];

      /* Yeterince büyük bir tamponla önce deneyelim.  */
      donusmus = (char *) xmalloc (uzunluk);

      /* dizi[i]'yi dönüştürelim.  */
      donusmus_uzunluk = strxfrm (donusmus, dizi[i], uzunluk);

      /* Tampon yetersizse yeniden boyutlandırıp tekrar deneyelim.  */
      if (donusmus_uzunluk <= uzunluk)
        {
          /* Gerekli alanı ayıralım. sonlandırıcı boş karakter
             için +1'i unutmayalım.  */
          donusmus = (char *) xrealloc (donusmus,
                                        donusmus_uzunluk + 1);

          /* Dönen değerin önemi yok çünkü dönüşmüş dizgenin
             uzunluğunu biliyoruz.  */
          (void) strxfrm (donusmus, dizi[i],
                          donusmus_uzunluk + 1);
        }

      gecici_dizi[i].donusmus = donusmus;
    }

  /* Dönüştürülmüş dizgeleri karşılaştırarak gecici_dizi diziyi sıralayalım. */
  qsort (gecici_dizi, sizeof (struct siralayici),
         dizge_sayisi, elemanlari_karsilastir);

  /* Elemanları geri, geçici diziye sıralı olarak yerleştirelim. */
  for (i = 0; i < dizge_sayisi; i++)
    dizi[i] = gecici_dizi[i].girdi;

  /* Ayrılan dizgeleri serbest bırakalım. */
  for (i = 0; i < dizge_sayisi; i++)
    free (gecici_dizi[i].donusmus);
}
```

Bu kodun geniş karakterli sürümünü ilgilendiren parçası şöyle olurdu:

```
void
dizgeleri_hizli_sirala (wchar_t **dizi, int dizge_sayisi)
{
  …
      /* dizi[i]'yi dönüştürelim.  */
      donusmus_uzunluk = wcsxfrm (donusmus, dizi[i], uzunluk);

      /* Tampon yetersizse yeniden boyutlandırıp tekrar deneyelim.  */
      if (donusmus_uzunluk <= uzunluk)
        {
          /* Gerekli alanı ayıralım. sonlandırıcı boş karakter
             için +1'i unutmayalım.  */
          donusmus = (wchar_t *) xrealloc (donusmus,
                                           (donusmus_uzunluk + 1)
```

```
                                                * sizeof (wchar_t));

          /* Dönen değerin önemi yok çünkü dönüşmüş dizgenin
             uzunluğunu biliyoruz.  */
          (void) wcsxfrm (donusmus, dizi[i],
                          donusmus_uzunluk + 1);
        }
  ...
```

realloc çağrısındaki ek olarak yapılan **sizeof (wchar_t)** ile çarpma işlemine dikkat edin.

Uyumluluk Bilgisi

Dizgelerin yerele özgü harf sıralama işlevleri ISO C90'nın yeni bir özelliğidir. Daha eski C oluşumlarında buna eşdeğer bir özellik yoktur. Geniş karakteri sürümü ise ISO C90'nın 1. düzeltmesinde yer almıştır.

7. Arama İşlevleri

Bu kısımda dizgeler ve diziler üzerinde çeşitli aramalar yapan kütüphane işlevleri açıklanmıştır. Bu işlevler `string.h` başlık dosyasında bildirilmiştir.

```
void *memchr(const void *blok,                                          işlev
             int         c,
             size_t      boyut)
```

Bu işlev *blok* adresinden başlayan nesnenin ilk *boyut* baytı içindeki ilk *c* (bir **unsigned char**'a dönüştürülmüş) baytını bulur. Dönen değer baytın konumunu içeren bir göstericidir. *c* baytı bulunamazsa bir boş gösterici döner.

```
wchar_t *wmemchr(const wchar_t *blok,                                   işlev
                 wchar_t        wc,
                 size_t         boyut)
```

Bu işlev *blok* adresinden başlayan nesnenin ilk *boyut* geniş karakteri içindeki ilk *wc* geniş karakterini bulur. Dönen değer geniş karakterin konumunu içeren bir göstericidir. *wc* baytı bulunamazsa bir boş gösterici döner.

```
void *rawmemchr(const void *blok,                                       işlev
                int         c)
```

memchr işlevi çoğunlukla, parametrelerle belirtilen bellek bloğunda bir *c* baytının bulunduğu bilinerek kullanılır. Ancak bu, *boyut* parametresinin gerçekte gerekmediği anlamına gelir ve bu işlevle uygulanan testler çalışma anında yapıldığından (blok sonunun aşılıp aşılmadığına bakılması gibi) gerekli değildir.

rawmemchr işlevi şaşırtıcı sıklıkla karşılaşılan bu durum için vardır. Arayüzü, *boyut* parametresinin bulunmayışı dışında **memchr** işlevine benzer. Yazılımcı, *c* karakterinin *blok* içinde mevcut olduğu kabulünde bir hataya düşerse, işlev bloğun sonunu aşacaktır. Bu durum için sonuç belirsizdir. Aksi takdirde bayt konumuna bir gösterici döner.

Bu işlev özellikle bir dizgenin sonuna bakılmak istendiğinde kullanılır. Tüm dizgeler bir boş karakterle sonlandırılmış olduğundan,

```
rawmemchr (str, '\0')
```

gibi bir çağrı dizgenin sonunu asla aşmayacaktır.

Bu işlev bir GNU oluşumudur.

```
void *memrchr(const void *blok,                                    işlev
              int          c,
              size_t       boyut)
```

Bu işlev, *blok* ve *boyut* ile belirtilen bloğu sondan başa doğru araması dışında, aramayı baştan sona doğru yapan **memrchr** gibidir.

Bu işlev bir GNU oluşumudur.

```
char *strchr(const char *dizge,                                    işlev
             int         c)
```

strchr işlevi, boş karakter sonlandırmalı *dizge* dizgesi içinde *c* karakterini (**char** türüne çevirerek) bulursa ilk bulduğu karakter için bir gösterici ile döner, bulamazsa boş gösterici ile döner.

Örnek:

```
strchr ("hello, world", 'l')
    => "llo, world"
strchr ("hello, world", '?')
    => NULL
```

Sonlandırıcı boş karakter dizgenin bir parçası olarak ele alındığından, *c* argümanı olarak boş karakter vererek dizgenin sonuna bir gösterici alabilirsiniz. Böyle durumlarda **strchrnul** kullanmak daha iyi olur (ama daha az taşınabilirdir).

```
wchar_t *wcschr(const wchar_t *dizge–geniş,                        işlev
                int            wc)
```

wcschr işlevi, *dizge–geniş* dizgesi içinde *wc* geniş karakterini bulursa ilk bulduğu karakter için bir gösterici ile döner, bulamazsa boş gösterici ile döner.

Sonlandırıcı boş karakter dizgenin bir parçası olarak ele alındığından, *wc* argümanı olarak boş karakter vererek dizgenin sonuna bir gösterici alabilirsiniz. Böyle durumlarda **wcschrnul** kullanmak daha iyi olur (ama daha az taşınabilirdir).

```
char *strchrnul(const char *dizge,                                 işlev
                int         c)
```

strchrnul işlevi, karakteri bulamadığı takdirde dizgeyi sonlandıran boş karaktere bir gösterici döndürmesi dışında **strchr** ile aynıdır.

Bu işlev bir GNU oluşumudur.

```
wchar_t *wcschrnul(const wchar_t *dizge–geniş,                     işlev
                   wchar_t        wc)
```

wcschrnul işlevi, geniş karakteri bulamadığı takdirde dizgeyi sonlandıran boş karaktere bir gösterici döndürmesi dışında **wcschr** ile aynıdır.

Bu işlev bir GNU oluşumudur.

strchr işlevinin kullanışlı ama anlamsız bir kullanımı da dizgeyi sonlandıran boş karaktere bir gösterici istenmesi durumudur. Bunu yapmanın kolay bir yolu vardır:

```
s += strlen (s);
```

Bu oldukça iyi bir çözüm gibi görünürse de toplama işlemi, **strlen** ile zaten yapılan işlemin tekrarlanmasını sağlar. Daha iyi bir çözüm şöyle olurdu:

```
s = strchr (s, '\0');
```

strchr işlevinin ikinci parametresi için bir sınırlama olmadığından burada bir boş karakter belirtilebilir. Şimdi düşüneceksiniz; **strchr** çıkışta iki kritere baktığından **strlen** işlevine göre daha çok işlem yapacak. Bu doğru. Ama GNU C kütüphanesinde **strchr** işlevi özel olarak daha hızlı olması için eniyilenerek gerçekleştirilmiştir.

```
char *strrchr(const char *dizge,                                    işlev
              int         c)
```

strrchr işlevi, *dizge* dizgesini sondan başa doğru araması dışında **strchr** işlevine benzer.

Örnek:

```
strrchr ("hello, world", 'l')
    => "ld"
```

```
wchar_t *wcsrchr(const wchar_t *dizge–geniş,                        işlev
                 wchar_t        c)
```

wcsrchr işlevi, *dizge–geniş* dizgesini sondan başa doğru araması dışında **wcschr** işlevi gibidir.

```
char *strstr(const char *dizge,                                     işlev
             const char *altdizge)
```

Bu işlev, *dizge* içinde bir karakter yerine *altdizge* dizgesini araması dışında **strchr** gibidir. *dizge* dizgesi içinde *altdizge* dizgesinin, bulduğunda ilk karakterine bir gösterici ile döner, bulamazsa boş gösterici ile döner. *altdizge* bir boş dizge olarak verilirse işlev *dizge* ile döner.

Örnek:

```
strstr ("hello, world", "l")
    => "llo, world"
strstr ("hello, world", "wo")
    => "world"
```

```
wchar_t *wcsstr(const wchar_t *dizge,                               işlev
                const wchar_t *altdizge)
```

Bu işlev, *dizge* içinde bir geniş karakter yerine *altdizge* dizgesini araması dışında **wcschr** gibidir. *dizge* dizgesi içinde *altdizge* dizgesinin, bulduğunda ilk geniş karakterine bir gösterici ile döner, bulamazsa boş gösterici ile döner. *altdizge* bir boş dizge olarak verilirse işlev *dizge* ile döner.

```
wchar_t *wcswcs(const wchar_t *dizge,                               işlev
                const wchar_t *altdizge)
```

wcswcs işlevi, **wcsstr** işlevinin eski ve artık kullanılmayan bir benzeridir. İşlevin ismi ilk olarak, ISO C90 1. düzeltmesinden önce X/Open Taşınabilirlik Kılavuzunda kullanılmıştı.

```
char *strcasestr(const char *dizge,                                 işlev
                 const char *altdizge)
```

Arama yaparken harf büyüklüğünü gözardı etmesi dışında **strstr** işlevi gibidir. **strcasecmp** işlevindeki gibi büyük ve küçük harflerin birbirleriyle ilişkileri yerele bağlıdır.

Örnek:

```
strcasestr ("hello, world", "L")
    => "llo, world"
strcasestr ("hello, World", "wo")
    => "World"
```

```
void *memmem(const void *dizge,                                    işlev
             size_t      dizge–uzunluğu,
             const void *altdizge,
             size_t      altdizge–uzunluğu)
```

Bu işlev **strstr** işlevi gibidir, ama *altdizge* ve *dizge* birer boş karakter sonlandımalı dizge değil birer bayt dizisidir, *dizge–uzunluğu* ve *altdizge–uzunluğu* da bunların uzunluklarıdır.

Bu işlev bir GNU oluşumudur.

```
size_t strspn(const char *dizge,                                   işlev
              const char *arananlar)
```

strspn ("string span" kısaltması) işlevi, *dizge* içinde *arananlar* dizgesi ile belirtilen karakterlerden birinin bulunması durumunda, bulunan karakterin *arananlar* içinde, bulunduğu ilk altdizgenin uzunluğu ile döner. *arananlar* dizgesindeki karakterlerin sırasının önemi yoktur.

Örnek:

```
strspn ("hello, world", "abcdefghijklmnopqrstuvwxyz")
    => 5
```

Burada karakter bayt anlamındadır. Çok baytlı karakter kodlaması kullanılan dizgelerde bir karakter birden fazla bayttan oluştuğundan bu işlevde her bayt ayrı ayrı değerlendirilir. Bu işlev yerele bağımlı değildir.

```
size_t wcsspn(const wchar_t *dizge–geniş,                          işlev
              const wchar_t *arananlar)
```

wcsspn ("wide character string span" kısaltması) işlevi, *dizge–geniş* içinde *arananlar* dizgesi ile belirtilen geniş karakterlerden birinin bulunması durumunda, bulunan geniş karakterin *arananlar* içinde, bulunduğu ilk altdizgenin uzunluğu döner. *arananlar* dizgesindeki geniş karakterlerin sırasının önemi yoktur.

```
size_t strcspn(const char *dizge,                                  işlev
               const char *arananlar)
```

strcspn ("string complement span" kısaltması) işlevi, *dizge* içinde *arananlar* dizgesi ile belirtilen karakterlerden birinin bulunması durumunda, *dizge* içinde, bulunan karakteri içermeyen ilk alt dizgenin uzunluğu ile döner.

Örnek:

```
strcspn ("hello, world", " \t\n,.;!?")
    => 5
```

Burada karakter bayt anlamındadır. Çok baytlı karakter kodlaması kullanılan dizgelerde bir karakter birden fazla bayttan oluştuğundan bu işlevde her bayt ayrı ayrı değerlendirilir. Bu işlev yerele bağımlı değildir.

```
size_t wcscspn(const wchar_t *dizge–geniş,                         işlev
               const wchar_t *arananlar)
```

wcscspn ("wide character string complement span" kısaltması) işlevi, *dizge–geniş* içinde *arananlar* dizgesi ile belirtilen geniş karakterlerden birinin bulunması durumunda, *dizge–geniş* içinde, bulunan karakteri içermeyen ilk alt dizgenin uzunluğu ile döner.

```
char *strpbrk(const char *dizge,                                    işlev
              const char *arananlar)
```

strpbrk ("string pointer break" kısaltması) işlevi, **strcspn** işlevine benzer, farklı olarak, *dizge* içinde, bulunan karakterle başlayan dizgeye bir gösterici ile döner. *arananlar* içindeki karakterlerden biri bulunamazsa boş gösterici döner.

Örnek:

```
strpbrk ("hello, world", " \t\n,.;!?")
    => ", world"
```

Burada karakter bayt anlamındadır. Çok baytlı karakter kodlaması kullanılan dizgelerde bir karakter birden fazla bayttan oluştuğundan bu işlevde her bayt ayrı ayrı değerlendirilir. Bu işlev yerele bağımlı değildir.

```
wchar_t *wcspbrk(const wchar_t *dizge–geniş,                        işlev
                 const wchar_t *arananlar)
```

wcspbrk ("wide character string pointer break" kısaltması) işlevi, **wcscspn** işlevine benzer, farklı olarak, *dizge–geniş* içinde, bulunan geniş karakterle başlayan dizgeye bir gösterici ile döner. *arananlar* içindeki geniş karakterlerden biri bulunamazsa boş gösterici döner.

7.1. Uyumluluk için Varolan Dizge Arama İşlevleri

```
char *index(const char *dizge,                                      işlev
            int         c)
```

index işlevi **strchr** işlevinin diğer ismidir; yani ikiside aynıdır. **index** işlevi BSD'dan gelir ve System V'den türetilmiş sistemlede hiç kullanılmamıştır. ISO C bu isim yerine **strchr** ismini içerdiğinden daima yeni ismi kullanmalısınız.

```
char *rindex(const char *dizge,                                     işlev
             int         c)
```

rindex işlevi **strrchr** işlevinin diğer ismidir; yani ikiside aynıdır. **rindex** işlevi BSD'dan gelir ve System V'den türetilmiş sistemlede hiç kullanılmamıştır. ISO C bu isim yerine **strrchr** ismini içerdiğinden daima yeni ismi kullanmalısınız.

8. Bir Dizgeyi Dizgeciklere Ayırma

Hemen tüm uygulamalarda, bir komut dizgesini dizgeciklere ayırmak gibi bazı basit ayrıştırma işlemlerine ihtiyaç duyulur. Bunu `string.h` başlık dosyasında bildirilmiş olan **strtok** işlevi ile yapabilirsiniz.

```
char *strtok(char *restrict       yeni–dizge,                       işlev
             const char *restrict ayraçlar)
```

Bir dizge **strtok** işlevini peşpeşe çağırarak dizgeciklerine bölünebilir.

Bölünecek dizge ilk çağrıda *yeni–dizge* argümanı ile işleve aktarılır. **strtok** işlevi bunu bazı dahili durum bilgilerini ayarlamakta kullanır. Sonraki çağrılarda, *yeni–dizge* argümanında bir boş gösterici aktarılması aynı dizgeden başka dizgeciklerin alınacağını belirtir. *yeni–dizge* argümanında boş gösterici olmayan bir gösterici belirterek yapılan her **strtok** çağrısı durum bilgilerini yeniden ilklendirir. Hiçbir kütüphane işlevinin sizin haberiniz olmadan (bu dahili durum bilgisini karıştıran) **strtok** çağrısı yapmayacağı garanti edilmiştir.

ayraçlar argümanı çıkarılacak dizgecikleri belirlemede kullanılan ayraçlardan oluşan bir dizgedir. Bu ayraçlardan birine rastlandığında bu karakter bir boş karakterle değiştirilir ve dizgenin başlangıcı *yeni–dizge* argümanında döndürülür. Dizgecikler daima dizgeciğin sonunda bir ayracın varlığına göre ayrılır.

Sonraki **strtok** çağrılarında arama önceki dizgeciği sonlandıran boş karakterden sonraki karakterden başlar. Sonraki **strtok** çağrılarında hep aynı *ayraçlar* dizgesini kullanmak zorunluluğu yoktur.

yeni–dizge dizgesinin sonuna gelindiğinde ya da kalan dizge sadece ayraç karakterlerinden oluşuyorsa işlev boş gösterici ile döner. If the end of the string *yeni–dizge* is reached, or if the remainder of string consists only of delimiter characters, **strtok** returns a null pointer.

Burada karakter bayt anlamındadır. Çok baytlı karakter kodlaması kullanılan dizgelerde bir karakter birden fazla bayttan oluştuğundan bu işlevde her bayt ayrı ayrı değerlendirilir. Bu işlev yerele bağımlı değildir.

```
wchar_t *wcstok(wchar_t       *yeni–dizge,                                  işlev
                const char *ayraçlar)
```

Bir dizge **wcstok** işlevini peşpeşe çağırarak dizgeciklerine bölünebilir.

Bölünecek dizge ilk çağrıda *yeni–dizge* argümanı ile işleve aktarılır. **wcstok** işlevi bunu bazı dahili durum bilgilerini ayarlamakta kullanır. Sonraki çağrılarda, *yeni–dizge* argümanında bir boş gösterici aktarılması aynı dizgeden başka dizgeciklerin alınacağını belirtir. *yeni–dizge* argümanında boş gösterici olmayan bir gösterici belirterek yapılan her **wcstok** çağrısı durum bilgilerini yeniden ilklendirir. Hiçbir kütüphane işlevinin sizin haberiniz olmadan (bu dahili durum bilgisini karıştıran) **wcstok** çağrısı yapmayacağı garanti edilmiştir.

ayraçlar argümanı çıkarılacak geniş karakterli dizgecikleri belirlemede kullanılan ayraçlardan oluşan bir geniş karakterli dizgedir. Bu ayraçlardan birine rastlandığında bu geniş karakter bir boş karakterle değiştirilir ve dizgenin başlangıcı *yeni–dizge* argümanında döndürülür. Dizgecikler daima dizgeciğin sonunda bir ayracın varlığına göre ayrılır.

Sonraki **wcstok** çağrılarında arama önceki dizgeciği sonlandıran boş karakterden sonraki geniş karakterden başlar. Sonraki **wcstok** çağrılarında hep aynı *ayraçlar* dizgesini kullanmak zorunluluğu yoktur.

yeni–dizge geniş karakterli dizgesinin sonuna gelindiğinde ya da kalan geniş karakterli dizge sadece ayraç karakterlerinden oluşuyorsa işlev boş gösterici ile döner.

Uyarı

strtok ve **wcstok** işlevleri ayrıştırma sırasında dizgeyi değiştirdiğinden, **strtok**/**wcstok** çağrısından önce dizgeyi daima geçici bir tampona kaydetmelisiniz (Bkz. *Kopyalama ve Birleştirme* (sayfa: 94)). **strtok** veya **wcstok** işlevinin yazılımınızın başka bir parçasından gelen bir dizgeyi değiştirmesine izin verirseniz, **strtok** veya **wcstok** çağrısından sonra dizge değişmiş olacağından dizgenin başka amaçlar için kullanılması gerektiğinde umduğunuz değerde olmayacağından sorunlarla karşılaşabilirsiniz.

İşleme soktuğunuz dizge bir sabit olduğu takdirde, **strtok** veya **wcstok** işlevi onu değiştirmeye kalktığında yazılımınız salt–okunur belleğe yazacağından bir ölümcül hata alacaktır Bkz. *Yazılım Hatalarının Sinyalleri* (sayfa: 604). **strtok** veya **wcstok** işlevinin yaptığı işlemin dizgeyi değiştirmeyeceğini (mesela sadece bir dizgecik vardır) düşünseniz bile dizge değiştirilebilir (GNU libc değiştirecektir).

Bu, bir genel prensibin özel durumudur: Eğer bir yazılım parçası belli bir veri yapısını güncelleme ile görevlendirilmemişse sözkonusu veri yapısının bu yazılım tarafından geçici olarak güncellenmesi hatalara yol açabilir.

strtok ve **wcstok** işlevleri evresel değildir. Evreselliğin nerede ve niçin önemli olduğu *Sinyal İşleme ve Evresel Olmayan İşlevler* (sayfa: 623) bölümünde açıklanmıştır.

strtok işlevinin kullanımına bir örnek:

```
#include <string.h>
#include <stddef.h>

…

const char
  dizge[] = "noktalama isaretleri -- ve bosluklarla ayrilmis kelimeler!";
const char ayraclar[] = " .,;:!-";
char *dizgecik, *kopya;

…

kopya = strdupa (dizge);              /* Yazılabilir kopya.  */
dizgecik = strtok (kopya, ayraclar); /* dizgecik => "noktalama" */
dizgecik = strtok (NULL, ayraclar);  /* dizgecik => "isaretleri" */
dizgecik = strtok (NULL, ayraclar);  /* dizgecik => "ve" */
dizgecik = strtok (NULL, ayraclar);  /* dizgecik => "bosluklarla" */
dizgecik = strtok (NULL, ayraclar);  /* dizgecik => "ayrilmis" */
dizgecik = strtok (NULL, ayraclar);  /* dizgecik => "kelimeler" */
dizgecik = strtok (NULL, ayraclar);  /* dizgecik => NULL */
```

GNU C kütüphanesi bir dizgeyi dizgeciklerine bölmek için, tek katılışlı olmanın sınırlamalarını aşan iki işlev daha içerir. Bu işlevler sadece çok karakterli karakter dizgeleri için kullanılabilir.

```
char *strtok_r(char        *yeni–dizge,                                    işlev
               const char *ayraçlar,
               char       **sonraki)
```

Bu işlev, **strtok** gibi, ardışık olarak çağrılarak bir dizgeyi dizgeciklerine ayırmakta kullanılır. Farkı, sonraki dizgecik hakkındaki bilgilerin, bir dizge göstericisine gösterici olan *sonraki* argümanı ile gösterilen alanda saklanmasıdır. İşlevin, *yeni–dizge* argümanının boş gösterici ile ve çağrılar arasında *sonraki* argümanının değişmeden bırakılarak çağrılması, çok katılışlılığı engellemeksizin işlemin yapılmasını sağlar.

Bu işlev POSIX.1 içinde tanımlıdır ve çok evreliliği destekleyen çoğu sistemde bulunur.

```
char *strsep(char       **sonraki,                                         işlev
             const char *ayraç)
```

Bu işlev, **strtok_r** işlevinden *yeni–dizge* argümanının yerini *sonraki* argümanını alması dışında **strtok_r** işlevine benzer. Taşıyıcı göstericinin ilklendirilmesi yazılımcı tarafından yapılır. Ardışık çağrılarla *ayraç* ile ayrılan dizgeciğin adresi ile dönerken, sonraki dizgeciğin başlangıcını gösteren *sonraki* argümanını günceller.

strsep ile **strtok_r** arasındaki bir diğer fark da, eğer girdi dizgesi içinde *ayraç* karakterlerinden biri birden fazla içeriliyorsa, her *ayraç* karakteri çifti için bir boş dizge döndürmesidir. Bu demektir ki, bir yazılım normalde işlevi çalıştırmadan önce bir boş dizge döndürüp döndürmediğini sınamalıdır.

Bu işlev 4.3 BSD içinde tanımlıdır ve genişçe bir kullanım alanı vardır.

Yukarıdaki örneği **strsep** için uyarlarsak:

```
#include <string.h>
```

```
#include <stddef.h>

…

const char
  dizge[] = "noktalama isaretleri -- ve bosluklarla ayrilmis kelimeler!";
const char ayraclar[] = " .,;:!-";
char *dizgecik, *kopya;

…

kopya = strdupa (dizge);
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "noktalama" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "isaretleri" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "isaretleri" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "ve" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "bosluklarla" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "ayrilmis" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "kelimeler" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => "" */
dizgecik = strsep (&kopya, ayraclar);    /* dizgecik => NULL */
```

`char *basename(const char *dosyaismi)` işlev

basename işlevinin GNU sürümü, *dosyaismi* ile belirtilen dosya yolunun son elemanı ile döner. *dosyaismi* argümanını içerdiği bölü çizgileri bakımından değiştirmediğinden kullanımı tercih edilir. İşlevi prototipi `string.h` dosyasında bulunabilir. Eğer `libgen.h` dosyası da içerilmişse bu işlevin XPG sürümü ile değiştirileceğini aklınızdan çıkarmayın.

GNU **basename** işlevinin kullanım örneği:

```
#include <string.h>

int
main (int argc, char *argv[])
{
  char *prog = basename (argv[0]);

  if (argc < 2)
    {
      fprintf (stderr, "Kullanimi: %s <arg>\n", prog);
      exit (1);
    }

  …
}
```

Taşınabilirlik Bilgisi

Bu işlev, farklı sistemlerde farklı sonuçlar üretebilir.

`char *basename(char *dosyayolu)` işlev

basename işlevinin XPG sürümüdür ve ruhen GNU sürümüne benzer. Fakat, *dosyayolu* içindeki '/' karakterleri silinerek değişikliğe uğratılır. Eğer argüman sadece '/' karakterinden oluşuyorsa '/' karakteri döner. Ayrıca, **NULL** ise ya da bir boş dizge ise "." döner. İşlevin prototipi `libgen.h` dosyasında bulunabilir.

XPG **basename** kullanım örneği:

```
#include <libgen.h>

int
main (int argc, char *argv[])
{
  char *prog;
  char *path = strdupa (argv[0]);

  prog = basename (path);

  if (argc < 2)
    {
      fprintf (stderr, "Kullanimi: %s <arg>\n", prog);
      exit (1);
    }

  …

}
```

`char *`**dirname**`(char *`*dosyayolu*`)` işlev

dirname işlevi, **basename** işlevinin XPG sürümünün tümleyenidir. *dosyayolu* ile belrtilen dosyayı içeren dizin ile döner. Argümanın değeri **NULL** ise veya bir boş dizge ise ya da hiç '/' karakteri içermiyorsa, "." döner. İşlevin prototipi `libgen.h` dosyasında bulunabilir.

9. strfry

Aşağıdaki işlev yazılım geliştirmenin uzun süredir devam eden bir ikilemine karşılıktır: "Veriyi doğru olarak dizge biçiminde nasıl alırım ve daha sonra onu nasıl kolayca bozabilirim?" Bu, GNU C kütüphanesini kullanmayan yazılımcılar için oldukça basittir. GNU C kütüphanesini kullanan yazılımlarda dizge verisinin yok edilmesi için tercih edilen yöntem **strfry** işlevinin kullanılmasıdır.

İşlevin prototip `string.h` başlık dosyasındadır.

`char *`**strfry**`(char *`*dizge*`)` işlev

strfry işlevi bir dizgeyi rasgele evirmece ile dönüştürür. İşlev dizgenin içeriğini kendi içinde rasgele konumlar arasında takaslar, bunu yaparken bu iki konum atnı olabilir.

İşlev daima *dizge* ile döner.

Taşınabilirlik Bilgisi

İşlev GNU C kütüphanesine özeldir.

10. Bayağı Şifreleme

memfrob işlevi bir veri dizisini tanınmayacak hale getirmek daha sonra tekrar eski haline getirmek için kullanılır. Şifrelenmiş veri herkes tarafından kolayca normal veriye dönüştürülebildiğinden tam bir şifreleme sayılmaz.

Dönüşüm, eşek şakasından hoşlananlara karşı kullanılan Usenet'in "Rot13" şifreleme yöntemine eşdeğerdir. Rot13'ün aksine **memfrob** işlevi sadece metin üzerinde değil ikilik verilerle de çalışır. Gerçek şifrelemeyle iligileniyorsanız, *Şifrelemeyle İlgili İşlevler* (sayfa: 803) bölümüne bakınız. İşlev `string.h` başlık dosyasında bildirilmiştir.

```
void *memfrob(void  *bellek,                                          işlev
              size_t uzunluk)
```

memfrob işlevi, *uzunluk* baytlık *bellek* veri yapısını her baytla oynayarak dönüştürür (her bayta ikilik 00101010 ile bit bit ayrıcalıklı VEYA uygulanarak). İşlev, dönüşüm için ayrı bir bellek alanı kullanmaz ve daima *bellek* ile döner.

memfrob işlevini aynı veri ile tekrar çağırdığınızda veriyi eski özgün haliyle alırsınız.

Verinin bazıları tarafından görülmesi istenmiyorsa ya da veriye erişimin zorlaşması isteniyorsa, bu işlev bilgiyi gizlemek için yararlıdır. Bilginin başkaları tarafından görülmesi gerçekten istenmiyorsa, *Şifrelemeyle İlgili İşlevler* (sayfa: 803) bölümüne bakınız.

Taşınabilirlik Bilgisi

İşlev GNU C kütüphanesine özeldir.

11. İkilik Verinin Kodlanması

Sadece metin türü verilerin saklanabildiği ya da aktarılabildiği ortamlarda saklama ya da aktarma öncesi ikilik verinin baytlarının karakterlere dönüştürülmesi gerekir. SVID sistemleri (ve günümüzde XPG uyumlu sistemler) bu işlem için çok az destek sağlar.

```
char *l64a(long int n)                                                işlev
```

Bu işlev, temel karakter kümesindeki karakterleri kullanarak 32 bitlik girdiyi kodlar. İşlev n sayısının kodlanmış halini içeren 7 karakterlik bir tampona gösterici ile döner. Yazılımcı bir sayı dizisini kodlamak istiyorsa, ikinci bir tampona dönen veriyi kopyalamalıdır. n sıfırsa boş dizge döner, biraz tuhaf ama standart böyle.

Uyarı

Tamponu durağan olduğundan işlev çok evreli yazılımlarda kullanılmamalıdır. Bu işlevin evreli yazılımlarca kullanılabilecek bir eşdeğeri C kütüphanesinde yoktur.

Uyumluluk Bilgisi

XPG standardında negatif n değerleri için **l64a** işlevinin dönüş değeri anlamlı değildir. GNU gerçeklemesinde, işlev argümanını işaretsiz olarak ele alır, böylece sıfırdan farklı n değerleri negatif olsalar bile anlamlı bir değer döner. Taşınabilir yazılımlar geliştiriyorsanız bu durumu dikkate almalısınız.

Büyük bir tamponu kodlamak isterseniz, her seferinde 32 bitlik bir tamponu dönüştürecek şekilde bir döngü kullanmalısınız. Örnek:

```
char *
encode (const void *buf, size_t len)
{
  /* Ne kadar uzunlukta bir tampon gerektiğini biliyoruz */
```

```
  unsigned char *in = (unsigned char *) buf;
  char *out = malloc (6 + ((len + 3) / 4) * 6 + 1);
  char *cp = out, *p;

  /* Uzunluğu kodlayalım. */
  /* 'htonl' kullanarak farklı bayt sıralaması kullanan makinalarda
      bile doğru çözümleme yapılmasını garanti edelim.
      'l64a' b bayttan daha kısa bir dizge döndürebilir,
      bu durumda genişliği tamamlamak için boşluğu 0
      karakterleriyle dolduralım.  */

  p = stpcpy (cp, l64a (htonl (len)));
  cp = mempcpy (p, "......", 6 - (p - cp));

  while (len > 3)
    {
      unsigned long int n = *in++;
      n = (n << 8) | *in++;
      n = (n << 8) | *in++;
      n = (n << 8) | *in++;
      len -= 4;
      p = stpcpy (cp, l64a (htonl (n)));
      cp = mempcpy (p, "......", 6 - (p - cp));
    }
  if (len > 0)
    {
      unsigned long int n = *in++;
      if (--len > 0)
        {
          n = (n << 8) | *in++;
          if (--len > 0)
            n = (n << 8) | *in;
        }
      cp = stpcpy (cp, l64a (htonl (n)));
    }
  *cp = '\0';
  return out;
}
```

Kütüphanenin gereken tam işlevselliği sağlamaması garip ama elden ne gelir.

l64a ile kodlanmış veri aşağıdaki işlev kullanılarak eski haline getirilebilir.

`long int` **a64l**`(const char *`*dizge*`)` işlev

dizge parametresi, **l64a** çağrısı ile elde edilmiş en az 6 karakterlik bir dizge olmalıdır. İşlev karakterleri aşağıdaki tabloya göre çözümler. Tabloda bulunmayan bir karaktere rastlanırsa işlev, **atoi** işlevinin tersine çözümlemeyi durdurur. Satırlara ayrılmış bir tampon kullanıyorsanız, satırsonu karakterlerine dikkat etmelisiniz.

Kodu çözülen sayı bir **long int** değer olarak döndürülür.

l64a ve **a64l** işlevleri base 64 kodlaması kullanır. Yani bir kodlanmış dizgenin her karakteri bir girdi sözcüğünün (16 bitlik alan) altı biti ile ifade edilir. Base 64 sayılar için kullanılan semboller:

	0	1	2	3	4	5	6	7
0	**.**	**/**	**0**	**1**	**2**	**3**	**4**	**5**
8	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**

16	E	F	G	H	I	J	K	L
24	M	N	O	P	Q	R	S	T
32	U	V	W	X	Y	Z	a	b
40	c	d	e	f	g	h	i	j
48	k	l	m	n	o	p	q	r
56	s	t	u	v	w	x	y	z

Bı kodlama şeması standart değildir. Daha geniş çapta kullanılan başka kodlama yöntemleri de (UU kodlaması, MIME kodlaması gibi) vardır. Genelde bu kodlamalardan birini kullanmak daha iyidir.

12. Argz ve Envz Vektörleri

Bir ***vektör*** boş gösterici ile sonlandırılmış gösterici dizisidir.

Bir ***argz vektörleri***, her elemanı boş karakter (`'\0'`) ile ayrılmış kesintisiz bir bellek bloğu içindeki dizgeler dizisidir.

Bir ***envz vektörü*** her elemanı bir isim–değer çifti olan ve bu çiftlerin `'='` karakteri ile ayrıldığı (en azından Unix ortamında) bir argz vektörüdür.

12.1. Argz İşlevleri

Her argz vektörü ilk elemanı gösteren **`char *`** türünde bir gösterici ve dizi boyutunu belirten **`size_t`** türünde bir gösterici ile temsil edilir. Her ikisi de bir boş argz vektörünü gösteren **`0`** ile ilklendirilir. Tüm argz işlevleri ya bir gösterici ile bir boyut argümanı ya da her ikisinin de değişebileceği öngörüldüğü durumlarda her ikisi de bir gösterici olan argümanlar alırlar.

Argz işlevleri dizge dizilerini ayırmak ya da büyütmek için **`malloc`**/**`realloc`** işlevlerini kullanır. Böylece bu işlevler tarafından oluşturulan tüm argz vektörleri **`free`** işlevi ile serbest bırakılabilir. Buna karşın, bir dizgeyi büyütebilen argz işlevlerinin dizgeyi **`malloc`** kullanarak ayırdığı umulur (bu tür argz işlevleri, bellek sıralaması ne olursa olsun, sadece kendi argümanlarını saptar ve onları değiştirmek için aynı alanı kullanır). Bakınız: *Özgür Bellek Ayırma* (sayfa: 50).

Bellek ayırmada yapan tüm argz işlevleri **`error_t`** türünde bir değer ile döner. Başarı durumundan bu değer **`0`** dır. Aksi takdirde, bir ayırma hatası oluşursa **`ENOMEM`** ile döner.

Bu işlevler standart bir başlık dosyası olan `argz.h` içinde bildirilmiştir.

```
error_t argz_create(char *const argv[],                                  işlev
                    char      **argz,
                    size_t     *argz_boyu)
```

`argz_create` işlevi *argv* Unix tarzı argüman vektörünü (normal C dizgelerine göstericilerden oluşmuş, **`(char *)0`** ile sonlandırılmış bir diziye gösterici; bkz, *Yazılım Argümanları* (sayfa: 645)) aynı elemanlarla argz vektörüne dönüştürür ve *argz* ve *argz_boyu* ile döndürür.

```
error_t argz_create_sep(const char *dizge,                               işlev
                        int         ayraç,
                        char      **argz,
                        size_t     *argz_boyu)
```

`argz_create_sep` işlevi, boş karakter sonlandırmalı *dizge* dizgesini her *ayraç* karakteri için bir eleman olmak üzere bir argz vektörüne (*argz* ve *argz_boyu* içinde döndürerek) dönüştürür.

```
size_t argz_count(const char *argz,                                    işlev
                  size_t      argz_boyu)
```

argz ve *argz_boyu* ile belirtilen argz vektörünün eleman sayısı ile döner.

```
void argz_extract(char  *argz,                                         işlev
                  size_t argz_boyu,
                  char **argv)
```

argz_extract işlevi, *argz* ve *argz_boyu* ile belirtilen argz vektörünü, (*argz* içindeki her elemana bir göstericiyi **0** ile sonladırılmış olarak *argv* içindeki kendi konumuna yerleştirerek) *argv* içinde saklanan Unix tarzı argüman vektörüne dönüştürür. *argv*, *argz* dizisinin tüm elemanları artı sonlandırıcı **(char *)0** için yeterince yer bulunan önceden ayrılmış bir alanı göstermelidir (**(argz_count (*argz*, *argz_boyu*) + 1) * sizeof (char *)** bayt yeterlidir). *argv* içindeki dizge göstericileri, *argz* içindeki alanları gösterdiğinden (kopyalama yapılmaz), *argz*'nin *argv* etkinken değişmemesi isteniyorsa ayrıca kopyalanmalıdır. Bu işlev *argz* içindeki elemanların bir *exec işlevine* (sayfa: 688) aktarılması için kullanışlıdır.

```
void argz_stringify(char  *argz,                                       işlev
                    size_t uzunluk,
                    int    ayraç)
```

argz_stringify işlevi, *argz* içindeki **'\0'** karakterlerini *ayraç* karakteri ile değiştirerek (dizgeyi sonlandıracak olan sonuncu boş karakter hariç) elemanları birleştirip normal bir dizgeye çevirir. Bu işlev, *argz*'nin okunabilir olarak basılması gibi işlemler için faydalıdır.

```
error_t argz_add(char       **argz,                                    işlev
                 size_t      *argz_boyu,
                 const char *dizge)
```

argz_add işlevi, *dizge* dizisini **argz* argz vektörünün sonuna ekler ve **argz* ile **argz_boyu* argümalarını yeni duruma uygun olarak günceller.

```
error_t argz_add_sep(char       **argz,                                işlev
                     size_t      *argz_boyu,
                     const char *dizge,
                     int         ayraç)
```

argz_add_sep işlevi **argz_add** işlevine benzer. Farklı olarak, *dizge* dizisini *ayraç* karakterlerine göre alt dizgelerine ayırarak ekler. Bu işlev, PATH değişkeni gibi bir değişkenin değerini oluşturan tüm elamanların ayrı birer dizge olarak eklenmek istenmesi durumunda yararlıdır (*ayraç* karakteri olarak **':'** verilerek).

```
error_t argz_append(char       **argz,                                 işlev
                    size_t      *argz_boyu,
                    const char *tampon,
                    size_t      tampon_boyu)
```

argz_append işlevi, *tampon_boyu*nu **argz_boyu*na ve *tampon* ile başlayan dizgeyi **argz* vektörüne ekler ve bunları birarada tutan yeni bir **argz* alanına yerleştirir (eskisi serbest bırakılır).

```
void argz_delete(char  **argz,                                         işlev
                 size_t *argz_boyu,
                 char   *girdi)
```

Eğer *girdi*, **argz* vektörünün elemanlarından birinin başlangıcını gösteriyorsa, **argz_delete** işlevi bu girdiyi kaldıracak ve **argz* ile **argz_boyu*nu güncelleyerek **argz* vektörüne yeniden yer ayıracaktır. Eleman silen argz işlevleri genelde argz argümanları için bellekte yeniden yer ayırdıklarından buradaki *girdi* gibi kaldırılan elemanın göstericisi geçersiz olur.

```
error_t argz_insert(char       **argz,                                    işlev
                    size_t      *argz_boyu,
                    char        *önce,
                    const char  *girdi)
```

argz_insert işlevi, *girdi* dizgesini **argz* vektörünün *önce* ile gösterilen elemanının öncesine ekler ve **argz* ile **argz_boyu*nu güncelleyerek **argz* için yeniden yer ayırır. Eğer *önce* **0** ise işlev **argz_add** işlevinin yaptığı gibi elemanı sona ekler. İlk eleman **argz* ile aynı olduğundan *önce* için **argz* verilirse, *girdi* dizgesi başa eklenir.

```
char *argz_next(char        *argz,                                        işlev
                size_t       argz_boyu,
                const char  *girdi)
```

argz_next işlevi *argz* vektöründeki elemanları yinelemek için uygun bir yol sağlar. *argz* vektöründe *girdi* elemanından sonraki elemana ya bir gösterici ile döner ya da *girdi* elemanını izleyen bir eleman yoksa **0** ile döner. *girdi* olarak **0** verilirse, *argz* vektörünün ilk elemanı döner.

Bu davranış iki yineleme tarzı akla getirir:

```
    char *girdi = 0;
    while ((girdi = argz_next (argz, argz_boyu, girdi)))
      eylem;
```

(çift parantez kullanımı bazı C derleyicileri için gereklidir) ve:

```
char *girdi;
for (girdi = argz;
     girdi;
     girdi = argz_next (argz, argz_boyu, girdi))
  eylem;
```

İkincisi *argz* vektörünün boş olması durumunda bir boş bellek bloğuna gösterici değil, **0** değerinde olması gerekliliğine bağlıdır; bu sabit, buradaki işlevler tarafından oluşturulan argz işlevleri için sağlanmıştır.

```
error_t argz_replace(char       **argz,                                   işlev
                     size_t      *argz_boyu,
                     const char  *dizge,
                     const char  *ile,
                     unsigned    *yineleme_sayısı)
```

argz içindeki *dizge* dizgelerini *ile* dizgesi ile değiştirir ve *argz* için gerekliyse yeniden yer ayırır. *yineleme_sayısı* sıfırdan farklı ise **yineleme_sayısı* uygulanan yer değiştirmenin sayısı kadar arttırılır.

12.2. Envz İşlevleri

Envz vektörleri her elemanının üzerindeki ek kısıtlamalar dışında argz vektörleri gibidir; örneğin, gerekirse, üzerlerinde argz işlevleri de kullanılabilir.

Envz vektörünün her elemanı **'='** işareti ile ayrılmış bir isim–değer çiftidir; eğer bir eleman içinde birden fazla **'='** karakteri varsa bu karakterlerden ilkinden sonraki dizge değer olarak kabul edilir ve hepsi **'\0'** dan farklı bir karakter olarak ele alınır.

Eleman bir **'='** karakteri içermiyorsa, eleman bir "null" girdinin ismi kabul edilir. Bu girdiyi bir boş değerli girdiden ayırmak için: **envz_get** işlevi isim bir null girdinin ismi ise **0** ile döner, aksine girdi bir boş değer içeriyor **""** döner; **envz_entry** işlevi bu tür girdileri nasıl olursa olsun bulacaktır. Null girdiler **envz_strip** işlevi ile kaldırılabilir.

Argz işlevlerindeki gibi, envz işlevlerinin de bellek ayırabildiğinden **error_t** türünde dönüş değeri vardır ve ya **0** ya da **ENOMEM** ile döner.

Bu işlevler standat bir başlık dosyası olan `envz.h` dosyasında bildirilmiştir.

```
char *envz_entry(const char *envz,                                    işlev
                 size_t      envz_boyu,
                 const char *isim)
```

envz_entry işlevi *envz* içinde *isim* isimli girdiyi bulur ve bu elemana (eleman *isim* ile başlar ve **'='** karakteri ile devam eder) bir gösterici ile döner. Belirtilen ismi içeren bir eleman yoksa **0** döner.

```
char *envz_get(const char *envz,                                      işlev
               size_t      envz_boyu,
               const char *isim)
```

envz_get işlevi *envz* içinde *isim* isimli girdiyi bulur ve bu elemanın değer parçasına (**'='** karakterinden sonrası) bir gösterici ile döner. Eğer bu ismi içeren bir eleman yoksa ya da eleman sadece isim parçasından oluşuyorsa (null girdi), **0** döner.

```
error_t envz_add(char       **envz,                                   işlev
                 size_t      *envz_boyu,
                 const char *isim,
                 const char *değer)
```

envz_add işlevi, *isim* isimli ve *değer* değerli bir girdiyi **envz* vektörüne ekler ve **envz* ile **envz_boyu*nı güncelleyerek **envz* için tekrar yer ayırır. *envz* içinde aynı isimli bir girdi varsa önce bu girdi kaldırılır. Eğer *değer* **0** ise, yeni girdi özel null türü girdi (yukarıda bahsedilmişti) olacaktır.

```
error_t envz_merge(char       **envz,                                 işlev
                   size_t      *envz_boyu,
                   const char *envz2,
                   size_t      envz2_boyu,
                   int         üsteyaz)
```

envz_merge işlevi, *envz2* içindeki girdileri *envz* vektörüne ekler ve **envz_add** işlevindeki gibi **envz* ve **envz_boyu*nu günceller. Eğer *üsteyaz* doğru ise, *envz2* içindeki girdilerden *envz* içinde olanlar varsa *envz2* içindekiler *envz* içindekilerin üstüne yazılır, değilse üste yazılmaz.

Null girdiler de diğer girdiler gibi ele alınır. Yani, her iki vektörde de aynı isimde iki girdi varsa, *envz* içindeki bir null girdi ise ve *üsteyaz* yanlışsa, *envz2* içindeki girdi *envz* vektörüne eklenmez.

```
void envz_strip(char  **envz,                                         işlev
                size_t *envz_boyu)
```

envz_strip işlevi *envz* içindeki null girdileri **envz* ve **envz_boyu*nu güncelleyerek kaldırır.

VI. Karakter Kümeleriyle Çalışma

İçindekiler

Bilgisayarların ilk görüldüğü zamanlarda karakter kümelerinde her karakter için altı, yedi ya da sekiz bit kullanılmıştı: sekiz bitten (bir bayt) daha geniş bir karakter hiç yoktu. Bu yaklaşımın sınırlamaları latin karakter kümelerini kullanmayanlar nezdinde daha belirgin hale geldi, bunlarda dilin karakter kümesindeki karakter sayısı için genellikle 2^8 yeterli değildir. Bu kısımda çokbaytlı karakter kümelerini desteklemek üzere C kütüphanesine eklenmiş işlevsellikten söz edilecektir.

1. Genişletilmiş Karakterlere Giriş

Karakterlerle baytlar arasında 1:1 ilişki olan karakter kümeleri ile 1:2 den 1:4 oranlarına kadar ilişki olan karakter kümeleri arasındaki farkları aşacak çok çeşitli çözümler vardır. Bu bölümün devamında C kütüphanesinin işlevselliğini geliştirirken verilen tasarım kararlarını anlamaya yardımcı olacak bir kaç örneğe yer verilmiştir.

Önce dahili ve harici gösterimler arasında bir ayrım yapmalıyız. ***Dahili gösterim*** bir yazılım tarafından bellekte tutulan metnin gösterimini anlıyoruz. ***Harici gösterim*** deyince ise bazı iletişim kanalları üzerinden aktarımda ya da bunlar üzerinde saklanacak metinlerin gösterimlerini anlıyoruz. Harici gösterime örnek vermek gerekirse, bir dizinde bulunan ve okunacak ya da çözümlenecek dosyaları gösterebiliriz.

Geleneksel olarak iki gösterim arasında bir fark yoktur. Tek baytlık dahili ve harici gösterim aynıdır ve eşit kullanılabilirliktedir. Bu kullanılabilirlik karakter kümeleri genişledikçe ve sayıları arttıkça azalır.

Dahili gösterimle ilgili aşılacak sorunlardan biri farklı karakter kümeleriyle harici olarak kodlanmış metinlerin elde edilmesidir. İki metni okuyup bazı ölçütleri kullanarak karşılaştıran bir yazılım varsayalım. Karşılaştırma sadece metinler dahili olarak bir ortak biçimde tutulabiliyorsa yapılabilir.

Böyle bir ortak biçim (= karakter kümesi) için sekiz bit elbette artık yeterli değildir. Öyleyse en küçük öğe büyütülmelidir: artık ***geniş karakterler*** kullanılmalıdır. Karakter başına bir bayt yerine iki hatta dört bayt kullanılması sözkonusu olacaktır. (Üç, bellek adreslemesi açısından iyi bir değer değildir ve dört bayttan fazlası da gerekmemektedir).

Bu kılavuzun bazı bölümlerinde görüleceği gibi bellekte geniş karakterli metinlerle çalışabilen işlevlerle tamamen yeni bir işlev ailesi oluşturulmuştur. Bu tür geniş karakter gösterimleri için kullanılan karakter kümelerinin çoğu Unicode ve ISO 10646 (UCS olarak da bilinir. UCS: Universal Character Set – Evrensel Karakter Kümesi) kullanır. Unicode (yunikod diye okunur) bir 16 bitlik karakter kümesi olarak tasarlandı; ISO 10646 ise 31 bitlik dev bir kod uzayı olarak tasarlandı. Uygulamada her iki standart eşdeğerdir. Aynı karakter listesini ve aynı kod tablosunu kullanırlar. Fakat Unicode ek anlamsallık belirtir. Bu noktada, sadece ilk **0x10000** kodluk karakter (BMP: Basic Multilingual Plane – "Temel Çokdilli Seviye" olarak da bilinir) atanmıştır. Unicode ve ISO 10646 karakterleri için tanımlanmış kodlamalardan bazıları: UCS–2 16 bitliktir ve sadece BMP'deki karakterleri içerir. UCS–4 32 bitliktir ve Unicode ve ISO 10646 karakterlerini içerir. UTF–8 tek baytla gösterilen ASCII karakter kümesine ek olarak ASCII olmayan 2 ilâ 6 karakterlik dizilimlerle ifade edilen karakterleri içerir. Son olarak UTF–16, UCS–2'nin içerdiklerine ek olarak **0x10ffff** e kadar BMP olmayan karakterleri içerir.

Geniş karakterleri göstermek için **char** türü yeterli değildir. Bu sebeple ISO C standardı bir geniş karakterli dizgenin bir karakterini tutmak için tasarlanmış yeni bir veri türünden bahseder. Benzerliği sağlamak için tek bir geniş karakter alan işlevlerde kullanılacak ve **int** türüne karşı düşen bir tür de vardır.

wchar_t veri türü

Bu veri türü geniş karakterli dizgelerin temel türü olarak kullanılır. Başka bir deyişle, bu tür nesne dizileri, çokbaytlı karakterlerin **char[]** dizisine eşdeğerdir. Tür, `stddef.h` başlık dosyasında tanımlanmıştır.

ISO C90 standardı, **wchar_t**'den bahsederken gösterimi hakkında belirgin hiçbir şey söylemez. Sadece temel karakter kümesinin tüm elemanlarını saklama yeteneğinde olması gerektiğini belirtir. Diğer taraftan, gömülü sistemlere uyarlanabilirlik bakımından **wchar_t** türünün **char** olarak tanımlanması meşru olmalıdır.

Fakat GNU sistemleri için **wchar_t** daima 32 bit genişliktedir ve tüm USC–4 değerleri gösterebilme yeteneğine sahiptir, böylece ISO 10646'nın tümü kapsama dahil olur. Bazı Unix sistemlerinde **wchar_t** 16 bitlik olarak tanımlanır ve sadece Unicode'u kapsar. Bu tanımlama standart açısından geçerli olmakla birlikte ISO 10646 ve UCS–2 deki karakterlerinin tümü ile UTF–16'nın 16 biti aşan karakterlerini fiilen çoklu geniş karakter kodlaması olarak gösterebilir. Fakat çoklu geniş karakterli kodlamaya başvurulması **wchar_t** türünün kullanım amacıyla çelişir.

wint_t veri türü

wint_t, tek bir geniş karakter içeren değişkenler ve parametreler için kullanılan bir veri türüdür. Normal **char** dizgeler kullanılırken isim olarak **int** türüne eşdeğer olan bu türün kullanılması önerilir. **wchar_t** ve **wint_t** türleri 32 bit genişlikte olduklarında çoğunlukla aynı gösterime sahiptir ancak, **wchar_t** **char** olarak tanımlanmışsa, parametre terfilerinden dolayı **wint_t** de **int** olarak tanımlanmalıdır.

Bu veri türü `wchar.h` başlık dosyasında tanımlanmış ve ISO C90'ın 1. düzeltmesinde bahsedilmiştir.

char veri türü için var olan makrolar gibi **wchar_t** türündeki bir nesnenin gösterebileceği azami ve asgari değerleri belirten makrolar da vardır.

`wint_t` **WCHAR_MIN** makro

wint_t türünde bir nesne tarafından tutulabilecek en küçük değerdir.

Bu makro ISO C90 standardının 1. düzeltmesinde bulunur.

wint_t **WCHAR_MAX** makro

wint_t türünde bir nesne tarafından tutulabilecek en büyük değerdir.

Bu makro ISO C90 standardının 1. düzeltmesinde bulunur.

Diğer bir geniş karakterlere özel değer **EOF**'a eşdeğerdir.

wint_t **WEOF** makro

WEOF makrosu genişletilmiş karakter kümesindeki herhangi bir üyeden farklı olan **wint_t** türünde bir değer olarak değerlendirilir.

WEOF, **EOF** ile aynı değerde olmakla birlikte onun aksine negatif olmaması gereklidir. Başka bir deyişle, aşağıdaki küçük kod,

```
{
  int c;
  …
  while ((c = getc (fp)) < 0)
    …
}
```

geniş karakterler için doğrudan **WEOF** kullanılarak aşağıdaki gibi yazılır:

```
{
  wint_t c;
  …
  while ((c = wgetc (fp)) != WEOF)
    …
}
```

Bu makro wchar.h başlık dosyasında tanımlanmıştır ve ISO C90'ın 1. düzeltmesinde bulunur.

Bu dahili gösterimler saklama ve aktarım sırasına sorunlara yol açarlar. Çünkü her geniş karakter çok sayıda bayttan oluşur ve bunlar bayt sıralamasından etkilenirler. Makinaların farklı bayt sıralamasına (endianess) sahip olmaları aynı verinin farklı değerlendirilmesine sebep olur. Bu bayt sıralaması ayrıca tamamı bayt temelli olan iletişim protokollerinden de etkilenir. Çoğunlukla gönderici geniş karakterleri baytlarına ayırmak konusunda bir karar vermek durumunda kalır. Bir son (ama en az önemli) husus da geniş karaktelerin, özelleştirilmiş tek baytlı karakter kümelerine göre daha fazla saklama alanı gerektirmesidir.

Yukarıdaki sebeplerden dolayı, dahili kodlama UCS–2 ya da UCS–4 ise çoğunlukla bir harici kodlama dahili kodlamadan farklı olur. Harici kodlama bayt temellidir ve ortama ve elde edilecek metine uygun olarak seçilebilir. Bu harici kodlama için farklı karakter kümeleri kullanılabilir. ASCII temelli tüm karakter kümeleri bir gereksinimi tamamen karşılar: dosyasistemi bakımından yeterlilik (filesystem safe); yani '/' karakteri kodlama içinde sadece kendi olarak kullanılır. Bazı şeyler EBCDIC (Extended Binary Coded Decimal Interchange Code – Genişletilmiş İkilik kodlu Ondalık Değişim Kodu; IBM tarafından kullanılan bir karakter kümesi ailesidir) gibi karakter setleri için biraz farklıdır, ama işletim sistemi EBCDIC'i doğrudan anlayamıyorsa sistem çağrılarının parametrelerinde kullanmadan önce işletim sisteminin anlayabileceği kodlamaya dönüştürülmüş olmalıdır.

- En basit karakter kümeleri tek baytlık karakter kümeleridir. Sadece 256 karakter içerebilir ve tüm dilleri kapsamak açısından yetersizdir. 8 bitlik karakter kümeleri ile çalışmak basittir. Daha sonra gösterileceği gibi bu doğru değildir, uygulamalar 8 bitlik karakter kümelerini kullanmaları gerektiği için kullanırlar.
- ISO 2022 standardı, bir baytttan daha fazla baytla gösterilebilen karakterlerin olduğu genişletilmiş karakter kümeleri için bir mekanizma tanımlar. Bu, bir metni bir durumla ilişkilendirerek yapılır. Karakterler, metin içinde bulunabildikleri durumu değiştirmekte kullanılabilirler. Metindeki her bayt, her durum için farklı yorumlanmalıdır. Bir baytın kendisi olarak mı yoksa bir karakteri oluşturan çok sayıda bayttan biri olarak mı yorumlanacağı duruma bağlıdır.

ISO 2022'nin çoğu kullanımlarında tanımlı karakter kümeleri bir sonraki karakterden fazlasını kapsayan durum değişikliklerine izin vermez. Bir karakterin bayt sırasının başlangıcı bulunduktan sonra metin doğru olalak yorumlanabildiğinden bu büyük yarar sağlar. Bu kuralı kullanan karakter kümelerine örnek olarak çeşitli EUC karakter kümeleri (Sun'ın işletim sistemlerinde kullanılan, EUC–JP, EUC–KR, EUC–TW ve EUC–CN) veya Shift_JIS (SJIS, bir Japonca kodlama) verilebilir.

Ancak bir karakterden daha fazlası için geçerli olan ve diğer bir bayt sıralaması tarafından değiştirilen bir durumu kullanan karakter kümeleri de vardır. Bunlara örnek olarak ISO–2022–JP, ISO–2022–KR ve ISO–2022–CN verilebilir.

- Latin alfabesini kullanan dillerin 8 bitlik karakter kümelerini düzeltmek için başlatılan çalışmalar ISO 6937 benzeri karakter kümeleri ile sonuçlandı. Burada **±** gibi karakterleri oluşturan baytlar kendileri olarak bir çıktı üretmez: istenen sonucu üretmek için birtakım karakterler birlikte kullanılır. Örneğin **0xc3 0xbc** bayt sıralaması (8 bitlik **Ã¼** karakterleri) **ü** karakterini oluşturur. **±** karakterini elde etmek için ise **0xc2 0xb1** bayt sıralaması (8 bitlik **Â±** karakterleri) kullanılır.

  ISO 6937 benzeri karakter kümeleri teletex gibi bazı gömülü sistemlerde kullanılır.

- Dahili olarak kullanılan Unicode veya ISO 10646 metinlerini dönüştürmek yerine UCS–2/UCS–4 den farklı bir kodlamanın kullanılması çoğunlukla yeterli olur. Unicode ve ISO 10646 nın her ikisi de böyle bir kodlamayı belirtirler: UTF–8. Bu kodlama, 1 bayttan 6 bayta kadar uzunluklarda bayt dizgelerini 31 bit genişlikle, ISO 10646 daki tüm karakterleri gösterebilmektedir.

  ISO 10646'yı UTF–7 olarak kodlamak üzere bazı çalışmalarda yapılmıştır, ancak günümüzde kullanılması gereken tek kodlama UTF–8 dir. Aslında, UTF–8 gelecekte desteklenen tek harici kodlama olacaktır. Evrensel olarak kullanılabilirliği anlaşılmıştır. Tek olumsuz yanı, bazı dillerin karakterlerini oluştururken kullanılan bayt dizgelerinin uzunluğu bu diller için kullanılan özel kodlamalara göre daha büyük yer harcanmasına sebep olmasıdır. Ancak Unicode sıkıştırma şeması gibi yönemlerle bu sorunlar da giderilecektir.

Sona kalan soru şudur: kullanılacak kodlama ya da karakter kümesi nasıl seçilecektir? Yanıt: Buna siz kendi kendinize karar veremezsiniz, bunu sistem geliştiricileri ile kullanıcıların çoğunluğunun yaptığı tercih belirler. Amaç birlikte çalışabilirlik olunca birinin kullandığını bir diğeri birlikte çalışabilmek için kullanacaktır. Bir kısıtlama yoksa seçim kullanıcıların ortak gereksinimlerine göre şekillenecektir. Başka bir deyişle örneğin, bir projenin sadece Rusya'da kullanılacağı düşünülüyorsa KOI8–R ya da benzeri bir karakter kümesi kullanmak gerekir. Ama aynı proje örneğin Yunanistan'da da kullanılabilecekse, karakter kümesi seçimi herkesin gereksinimlerine yanıt verebilecek şekilde seçilmelidir.

En geniş çözümü sağlayan, en genel karakter kümesi hangisi diye baktığımızda bunun ISO 10646 olduğunu görürüz. Harici kodlama olarak UTF–8 kullanılır ve geçmişte kendi dillerini kullanmakta sorunları olan kullanıcıların sorunları kalmaz.

Geniş karakter gösteriminin seçilmesi ile ilgili olarak son bir açıklama daha yapmak gerekir. Yukarıdaki açıklamaların ışığında doğal seçim Unicode veya ISO 10646 kullanmaktır dedik. Bu gerekli değildir ama en azından ISO C standardı tarafından cesaretlendiriliyoruz. Standart en azından **__STDC_ISO_10646__** diye bir makro tanımlar ve bu makro sadece **wchar_t** türünün kodladığı ISO 10646 karakterlerinin kullanıldığı sistemlerde tanımlıdır. Bu sembolü tanımlamayarak geniş karakterli gösterimlerle ilgili kabuller yapılmasından kaçınılmalıdır. Yazılımcılar sadece C kütüphanesi tarafından sağlanan bu işlevleri kullandıklarında geniş karakterle ilgili olarak diğer sistemlerle bir uyumluluk sorunu yaşamazlar.

2. Karakter Kümesi İşlevlerine Bakış

Bir Unix C kütüphanesi karakter kümesi dönüşümleri için iki aile içinde toplanan üç farklı işlev kümesi içerir. İlk aile (en çok kullanılanı) ISO C90 standardında belirtilmiştir ve bundan dolayı Unix dünyasında taşınabilirdir.

Maalesef bu aile en az kullanışlı olanıdır. Özellikle kütüphane geliştirirken (uygulamaların aksine) bu işlevlerden mümkün olduğunca kaçınılmalıdır.

İkinci işlev ailesi, erken dönem Unix standartlarında (XPG2) görülür ve hala en son ve en büyük Unix standardı olan Unix 98'in de parçasıdır. Ayrıca en güçlü ve en kullanışlı işlevler kümesidir. Fakat biz, ISO C90 1. düzeltmesinde tanımlanan işlevlerle başlayacağız.

3. Geridönüşümlü Çok Baytlı Dönüşüm

ISO C standardı dizgeleri çokbaytlı gösterimden geniş karakterli dizgelere dönüştürecek işlevler tanımlar. Bunların bir takım tuhaf özellikleri vardır:

- Çok baytlı kodlama için varsayılan karakter kümesi işlevlere argüman olarak belirtilmez. Bunun yerine yerelin **LC_CTYPE** kategorisi tarafından belirtilen karakter kümesi kullanılır; bkz. *Yerellerin Etkilediği Eylemlerin Sınıflandırılması* (sayfa: 165).
- Bir defada bir karakterden fazlası ile çalışan işlevler argüman olarak boş karakter sonlandırmalı dizgeleri gerektirirler (örneğin, metin bloklarının dönüşümü uygun bir yere bir boş karakter eklenmedikçe yapılmaz). GNU C kütüphanesi bir boyut belirtmeye imkan veren oluşumları içeriyor olsa da bunlar yine de genellikle sonlandırılmış dizgeler beklerler.

Bu sınırlamalara rağmen ISO C işlevleri çoğu bağlamda kullanılabilir. Grafik kullanıcı arayüzlerinde örneğin, metin basit ASCII değilse bir geniş karakterli dizge olarak gösterilmesinin gerektiği durumlar için kullanılacak işlevlerin bulunması gerekir. Metin, çevirileri içeren bir dosyadan gelmeli, kullanıcı çeviriyi kullanabileceği yerele ve dolayısıyla ayrıca kullanacağı harici kodlamaya karar vermelidir. Böyle bir durumda (ve birçok başka durumda), burada açıklanan işlevler çok uygundur. Dönüşümleri uygularken daha özgür olmak isterseniz **iconv** işlevlerine de bir bakın: *Soysal Karakter Kümesi Dönüşümü* (sayfa: 145).

3.1. Dönüşüm Seçimi

Burada açıklayacağımız işlevler tarafından uygulanan dönüşümlerde önceki bölümde bahsettiğimiz gibi seçilen yerelin **LC_CTYPE** kategorisi tarafından belirlenen karakter kümesi kullanılır. Her yerelin (**localedef**'e argüman olarak verilen) kendi karakter kümesi vardır ve bunun harici çokbaytlı kodlamalardan biri olduğu kabul edilir. Geniş karakterli karakter kümesi daima UCS–4'tür, eninde sonunda GNU sistemlerinde böyledir.

Her çokbaytlı karakter kümesinin karakteristik özelliklerinden biri, bir karakteri göstermek için gereken en fazla bayt sayısıdır. Bu, dönüşüm işlevlerinin kullanıldığı bir kodu yazarken oldukça önemli bir bilgidir (örnekleri aşağıda görülebilir). Bu bilgiyi sağlamak için ISO C standardı iki makro tanımlar.

`int` **MB_LEN_MAX** makro

MB_LEN_MAX desteklenen yerellerin hepsi için tek bir karakterin çokbaytlı gösteriminde olabilecek en fazla bayt sayısını belirtir. Bir derleme zamanı sabitidir ve `limits.h` başlık dosyasında tanımlanmıştır.

`int` **MB_CUR_MAX** makro

MB_CUR_MAX o an geçerli olan yereldeki bir çokbaytlı karakterde olabilecek en fazla bayt sayısı olan bir pozitif tamsayı ifadeye genişletilir. Değeri hiçbir zaman **MB_LEN_MAX**'dan büyük olamaz. **MB_LEN_MAX**'ın tersine bu makronun bir derleme zamanı sabiti olması gerekmez ve GNU C kütüphanesinde de değildir.

MB_CUR_MAX `stdlib.h` başlık dosyasında tanımlanmıştır.

ISO C derleyicileri değişken uzunluklu dizi tanımlarına kesinlikle izin vermediğinden iki farklı makro gereklidir, ancak yine de özdevimli bellek ayırmadan kaçınılması istenir. Kodun bu eksik parçası aşağıdaki soruna yol açar.

```
{
  char tampon[MB_LEN_MAX];
  ssize_t uzunluk = 0;

  while (! feof (fp))
    {
      fread (tampon[uzunluk], 1, MB_CUR_MAX - uzunluk, fp);
      /* … tamponu işle */
      uzunluk -= kullanilan;
    }
}
```

İç döngüdeki kod tek bir çokbaytlı karaktere çevrilmesi için *tampon* dizisinde daima yeterli bayt bulunduğu varsayımına dayanır. Çoğu derleyici değişken uzunluktaki dizilere izin vermediğinden *tampon* dizisi sabit uzunluktadır. **fread** çağrısı *tampon* dizisinde daima **MB_CUR_MAX** bayt olduğundan emin olarak yapılır. Burada **MB_CUR_MAX** bir derleme zamanı sabiti değilse bir sorun çıkmaz.

3.2. Durumun saklanması

Bu kısmın başlarında ***durumsal*** kodlama kullanılan karakter kümelerinden bahsedilmişti. Bunların metnin içindeki kodlanmış değerleri öncekilere bir şekilde bağımlıdır.

Dönüşüm işlevleri bir metni birden fazla adımda dönüştürebildiklerinden bu bilgiyi işlevlerin bir çağrısından diğerine aktarmamız gerekir.

mbstate_t veri türü

> **mbstate_t** türünde bir değişken bir dönüşüm işlevi çağrısından diğerine aktarılması gerekli olan ***öteleme durumu*** hakkında tüm bilgiyi içerir.
>
> **mbstate_t** türü `wchar.h` başlık dosyasında tanımlanmıştır ve ISO C90 standardının 1. düzeltmesinde bulunur.

mbstate_t türünde nesneleri kullanırken yazılımcı bu nesneleri tanımlamalı (normalde yığıt üzerinde yerel değişkenler olarak) ve dönüşüm işlevine göstericisi ile aktarmalıdır. O anki çokbaytlı karakter durumsal ise dönüşüm işlevi bu yolla nesneyi güncelleyebilir.

Durum nesnesini belirli bir duruma koyacak bir ilklendirici ya da bu işleme özel bir işlev yoktur. Kurallar gereğince nesne daima ilk kullanımdan önce ilk durumu göstermeli ve bu aşağıdaki gibi bir kodla değişkeni tamamen temizleyerek yapılmalıdır:

```
{
  mbstate_t durum;
  memset (&durum, '\0', sizeof (durum));
  /* bundan sonra durum kullanılabilir.  */
  …
}
```

Çıktıyı üretecek dönüşüm işlevlerini kullanırken çoğunlukla o anki durumun ilk durum olup olmadığına bakılması gerekir. Örneğin, dizilimin bir noktasında durumu ilk duruma ayarlayacak önceleme dizilimlerinin kullanılıp kullanılmayacağına karar vermek için bu gereklidir. İletişim protokolleri genellikle bunu gerektirir.

`int` **mbsinit**`(const mbstate_t *`*ps*`)` işlev

> **mbsinit** işlevi göstericisi *ps* olan nesnenin ilk durumda olup olmadığını saptamakta kullanılır. *ps* bir boş göstericiyse ya da nesne ilk durumdaysa dönen değer sıfırdan farklı olur. Aksi takdirde sıfır döner.

mbsinit işlevi wchar.h başlık dosyasında tanımlanmıştır ve ISO C90 standardının 1. düzeltmesinde bulunur.

mbsinit işlevi kullanılan bir kod çoğunlukla aşağıdakine benzer:

```
{
  mbstate_t durum;
  memset (&durum, '\0', sizeof (durum));
  /* Burada durum kullanılır.  */
  …
  if (! mbsinit (&durum))
    {
      /* ilk duruma döndürecek kod.  */
      const wchar_t bos[] = L"";
      const wchar_t *kaynak = bos;
      wcsrtombs (cikistamponu, &kaynak, ciktampuzun, &durum);
    }
  …
}
```

İlk duruma geri dönmeyi sağlayacak olan önceleme dizilimini çıktılayacak kod ilginçtir. **wcsrtombs** işlevi gerekli çıktılama kodunu (bkz. *Dizge Dönüşümleri* (sayfa: 137)) saptamakta kullanılabilir.

Bilgi

Geniş karakterli kodlama durumsal olmadığından GNU sistemlerinde çokbaytlı metni geniş karakterli metne dönüştürmek için bu ek eylemin uygulanması gerekli değildir. Ancak bir durumsal kodlama kullanarak **wchar_t** yapılmasının yasaklanmasına hiçbir standartta değinilmemiştir.

3.3. Bir Karakterin Dönüştürülmesi

Çok temel dönüşüm işlevlerinin çoğu tek karakter ile çalışır. Lütfen aklınızdan çıkarmayın, tek karakter her zaman tek bayt anlamına gelmez. Ancak çoğunlukla çokbaytlı karakter kümeleri tek baytlık karakterler içerdiğinden, baytları dönüştürmeye yarayan işlevler vardır. Sıklıkla, ASCII çokbaytlı karakter kümesinin bir alt kümesidir. Böyle bir senaryoda, her ASCII karakter kendini temsil eder, tüm diğer karakterlerin en azından ilk baytı 0 ile 127 arasındaki karakterlerden biri olur.

`wint_t` **btowc**`(int` *c*`)` işlev

btowc ("byte to wide character" kısaltması) işlevi, ilk öteleme durumundaki tek baytlık geçerli bir *c* karakterini o an geçerli olan **LC_CTYPE** yerelindeki dönüşüm kurallarına uygun olarak geniş karakter eşdeğerine dönüştürür.

Eğer `(unsigned char)` *c* geçerli olmayan tek baytlık bir çokbaytlı karakter ise ya da *c* karakteri **EOF** ise işlev **WEOF** ile döner.

c karakterinin geçerliliğinin sadece ilk öteleme durumu için sınandığını lütfen unutmayın. Durum bilgisinin alınmasında kullanılacak bir **mbstate_t** nesnesi yoktur ve ayrıca işlev herhangi bir sabit durum kullanmaz.

btowc işlevi ISO C90 standardının 1. düzeltmesinde tanımlanmış ve wchar.h başlık dosyasına bildirilmiştir.

Tek baytlık değerlerin daima ilk durumuna göre yorumlanması sınırlamasına rağmen bu işlev aslında çoğu zaman oldukça kullanışlıdır. Karakterlerin çoğu ya tamamen tek baytlık karakter kümelerindendir ya da ASCII'ye göre bir genişletmedir. Bu bilgilerden sonra aşağıdaki gibi bir kod yazmak mümkündür (bu özel örnek çok kullanışlıdır):

```
wchar_t *
itow (unsigned long int val)
{
  static wchar_t buf[30];
  wchar_t *wcp = &buf[29];
  *wcp = L'\0';
  while (val != 0)
    {
      *--wcp = btowc ('0' + val % 10);
      val /= 10;
    }
  if (wcp == &buf[29])
    *--wcp = L'0';
  return wcp;
}
```

Böylesine karmaşık bir gerçeklemeyi kullanmak neden gerekli ve neden basitçe **'0' + val % 10** bir geniş karaktere dönüştürülmüyor? Yanıtı, **wchar_t** türünde karakterler üzerinde bu çeşit aritmetik işlemler uygulandığında sonuç garanti değildir de ondan. Diğer durumlarda ise baytlar derleme zamanında sabit değildir, bu nedenle derleyici işlem yapamaz. Bu gibi durumlarda **btowc** kullanmak gereklidir.

Aksi yönde dönüşüm için de bir işlev vardır.

`int` **wctob**`(wint_t` *wc*`)` işlev

wctob ("wide character to byte" kısaltması) işlevi parametre olarak geçerli bir geniş karakter alır. Bu karakterin ilk durumu bir bayt uzunlukta ise işlevin dönüş değeri karakterin kendisi olacaktır. Aksi takdirde **EOF** döner.

wctob işlevi ISO C90 standardının 1. düzeltmesinde tanımlanmış ve `wchar.h` başlık dosyasına bildirilmiştir.

Karakterleri tek tek çokbaytlı gösterimden geniş karakterli gösterime ya da tersine dönüştürecek daha genel amaçlı işlevler de vardır. Bu işlevler çokbaytlı gösterimin uzunluğu ile ilgili bir sınırlamaya sahip değillerdir ve ayrıca ilk durumda olmayı gerektirmez.

```
size_t mbrtowc(wchar_t *restrict    pwc,                    işlev
               const char *restrict dizge,
               size_t               n,
               mbstate_t *restrict  ps)
```

mbrtowc ("multibyte restartable to wide character" kısaltması) işlevi, *dizge* ile gösterilen dizgedeki sonraki çokbaytlı karakteri geniş karaktere dönüştürür ve *pwc* ile gösterilen geniş karakterli dizge içinde saklar. Dönüşüm o an geçerli olan **LC_CTYPE** ile belirtilen yerele bağlı olarak gerçekleşir. Yerelde kullanılan karakter kümesi dönüşüm için bir durum bilgisi gerektiriyorsa, bu bilgi *ps* ile gösterilen nesne ile belirtilebilir. *ps* bir boş gösterici ise, işlev tarafından bir durağan dahili durum değişkeni kullanılır.

Sonraki çokbaytlı karakter bir boş geniş karakter ise, işlevin dönüş değeri sıfırdır ve durum nesnesi sonrasında ilk durumdadır. Eğer sonraki *n* veya daha az bayt doğru çokbaytlı karakter biçimindeyse, dönüş değeri, çokbaytlı karakter biçimindeki *dizge*den başlayan baytların sayısıdır. Dönüşüm durumu dönüşümde tüketilen baytlara göre güncellenir. Her iki durumda da geniş karakter (ya **L'\0'** ya da dönüşümde bulunan) *pwc* bir boş gösterici değilse, *pwc* ile gösterilen dizgede saklanır.

Çok baytlı dizgenin ilk *n* baytının geçerli bir çokbaytlı karakter olduğu varsayılmış ama dönüşüm için *n* den daha fazla bayt gerekiyorsa işlevin dönüş değeri **(size_t) -2** dir ve hiçbir değer saklanmaz. Girdi

gereğinden fazla öteleme durumu içerebildiğinden *n*, **MB_CUR_MAX**'dan büyük ya da ona eşit bir değer içerdiğinde bile bu durumun oluşabileceğini unutmayınız.

Çokbaytlı dizgenin ilk *n* baytının geçerli bir çokbaytlı karakter olduğu varsayılmamışsa, hiçbir değer saklanmaz, **errno** genel değişkenine **EILSEQ** değeri atanır ve işlev **(size_t) -1** ile döner. Dönüşüm durumu bundan sonra tanımsızdır.

mbrtowc işlevi ISO C90 standardının 1. düzeltmesinde tanımlanmış ve `wchar.h` başlık dosyasına bildirilmiştir.

mbrtowc işlevinin kullanımı basittir. Bir çokbaytlı dizgeyi bir geniş karakterli dizgeye kopyalarken küçük harfleri büyük harfe çeviren bir işlev şöyle olur (bu tam bir uygulama değildir; sadece örnektir; hata denetimi yapılmaz ve bellek kaçağı olabilir):

```
wchar_t *
mbstouwcs (const char *s)
{
  size_t uzunluk = strlen (s);
  wchar_t *sonuc = malloc ((uzunluk + 1) * sizeof (wchar_t));
  wchar_t *wcp = sonuc;
  wchar_t tmp[1];
  mbstate_t durum;
  size_t bayt_sayisi;

  memset (&durum, '\0', sizeof (durum));
  while ((bayt_sayisi = mbrtowc (tmp, s, uzunluk, &durum)) > 0)
    {
      if (bayt_sayisi >= (size_t) -2)
        /* Geçersiz girdi dizgesi.  */
        return NULL;
      *wcp++ = towupper (tmp[0]);
      uzunluk -= bayt_sayisi;
      s += bayt_sayisi;
    }
  return souc;
}
```

mbrtowc kullanımı temizdir. Tek bir geniş karakter *tmp*`[0]` içinde saklanır ve tüketilen baytların sayısı *bayt_sayisi* değişkeninde saklanır. Eğer dönüşüm başarılı olursa, geniş karakterin büyük harf karşılığı *sonuc* dizisinde saklanır ve girdi dizgesinin göstericisi ile kullanılabilir baytların sayısı ayarlanır.

mbrtowc hakkında belirsiz kalan tek şey sonuç için belleği ayırmakta kullanılan yöntem olabilir. Yukarıdaki kod, çokbaytlı girdi dizgesindeki baytların dönüşüm sonrası elde edilen geniş karakterli dizgenin bayt sayısından büyük ya da eşit olduğu varsayımına dayandırılmıştır. Bu yöntem sonucun uzunluğu hakkında iyimserdir ve çok sayıda geniş karakterli dizge ya da çok uzun bir dizge bu yöntemle oluşturulmaya çalışılırsa ek bellek ayrılması gerekebilecektir. Ayrılan bellek bloğu döndürülmeden önce doğru boyuta ayarlanabilir, fakat en doğrusu sonucun gerektirdiği kadar belleği baştan ayırmaktır. Umulanın aksine, elde edilecek geniş karakterli dizgenin boyunu çokbaytlı dizgenin boyutlarına bakarak elde edebilecek bir işlev yoktur. Yine de, işlemin bir parçası olabilecek bir işlev vardır.

```
size_t mbrlen(const char *restrict dizge,                    işlev
              size_t               n,
              mbstate_t           *ps)
```

mbrlen ("multibyte restartable length" kısaltması) işlevi, *dizge* ile başlayan en fazla *n* baytlık sonraki geçerli ve çokbaytlı tam karakterin bayt sayısını hesaplar.

Sonraki çokbaytlı karakter boş geniş karaktere karşılıksa, dönüş değeri sıfırdır. Sonraki *n* bayt, geçerli bir çokbaytlı karakter biçimindeyse, bu çokbaytlı karakteri oluşturan baytların sayısı ile döner.

Eğer ilk *n* geçerli bir çokbaytlı karakter için yetersizse, dönüş değeri **(size_t) -2** dir. Aksi halde, çokbaytlı karakter dizilimi geçersizdir ve dönüş değeri **(size_t) -1** dir.

Çokbaytlı dizilim *ps* ile gösterilen nesne ile belirtilen duruma göre yorumlanır. *ps* bir boş gösterici ise **mbrlen**'e özgü bir dahili durum nesnesi kullanılır.

mbrlen işlevi ISO C90 standardının 1. düzeltmesinde tanımlanmış ve `wchar.h` başlık dosyasına bildirilmiştir.

Dikkatli okuyucular **mbrlen** işlevinin şöyle gerçeklenebileceğini farkedecektir:

```
mbrtowc (NULL, s, n, ps != NULL ? ps : &dahili)
```

Bu doğrudur ve aslında resmi belirtimde bahsedilendir. Şimdi, bir çokbaytlı karakter dizisinden bir geniş karakterli dizgenin uzunluğunun nasıl saptanabileceğine bakalım. İşlev doğrudan kullanılabilir değildir, fakat **mbslen** isimli bir işlevi onu kullanarak tanımlayabiliriz:

```
size_t
mbslen (const char *s)
{
  mbstate_t durum;
  size_t sonuc = 0;
  size_t bayt_sayisi;
  memset (&durum, '\0', sizeof (durum));
  while ((bayt_sayisi = mbrlen (s, MB_LEN_MAX, &durum)) > 0)
    {
      if (bayt_sayisi >= (size_t) -2)
        /* Birşeyler yanlış.  */
        return (size_t) -1;
      s += bayt_sayisi;
      ++sonuc;
    }
  return sonuc;
}
```

Bu işlev, dizgedeki her çokbaytlı karakter için basitçe **mbrlen** çağrısı yapar ve işlev çağrılarını sayar. Burada **MB_LEN_MAX**'ı **mbrlen**'in boyut argümanı olarak kullandığımıza dikkat edin. Bu kabul edilebilir, çünkü;

a. Bu değer en uzun çokbaytlı karakter diziliminden büyüktür.
b. *dizge* dizgesinin boş bayt ile bittiğini biliyoruz. Bu sonlandırıcı bayt başka bir çokbaytlı karakter diziliminin parçası olamaz ama bir geniş boş karakteri temsil edebilir.

Diğer taraftan, **mbrlen** işlevi geçersiz belleği asla okumaz.

Şimdi bu işlev kullanılabilir (daha temiz olarak, bu işlev GNU C kütüphanesinin bir parçası *değildir*). Çok baytlı karakterli *dizge* dizgesinden dönüştürülerek elde edilecek geniş karakterli dizgenin saklanacağı alanın genişliğini hesaplayabiliriz:

```
wcs_bytes = (mbslen (s) + 1) * sizeof (wchar_t);
```

mbslen işlevinin verimsiz olduğunu unutmayın. **mbstouwcs**'nin **mbslen** ile gerçeklenmesi çokbaytlı karakterli girdi dizgesine iki defa dönüşüm uyguladığından bu dönüşüm masraflıdır. Bu durumda, kullanımı daha kolay ama işlemi iki defa yapmayan bir yöntem düşünmek gerekir.

```
size_t wcrtomb(char *restrict      dizge,                                    işlev
               wchar_t             wc,
               mbstate_t *restrict ps)
```

wcrtomb ("wide character restartable to multibyte" kısaltması) işlevi tek bir geniş karakteri, bunun karşılığı olan çokbaytlı karakter dizgesine dönüştürür.

dizge bir boş gösterici ise, işlev, *ps* ile gösterilen nesnedeki durum bilgisini (ya da işlevin dahili durum bilgisini) başlangıç durumuna sıfırlar. *dizge* boş bir gösterici olduğundan, bu aşağıdaki gibi bir çağrı ile aşılabilir:

```
wcrtombs (temp_buf, L'\0', ps)
```

wcrtomb yeterince büyük olduğu garanti edilen bir dahili tampona yazabiliyorsa bunu yapar.

Eğer *wc* bir boş geniş karakter ise, **wcrtomb** bunu yoksayar, eğer gerekliyse, *ps* durumunu ilk duruma getirecek bir öteleme diziliminden ardından tek bir boş karakter gelir ve *dizge* dizgesinde saklanır.

Aksi takdirde, bir bayt dizilimi (öteleme dizilimlerini de içerebilir) *dizge* dizgesine yazılır. Bu sadece *wc* geçerli bir geniş karakterse mümkündür (örneğin, yereli **LC_CTYPE** kategorisine göre seçilen karakter kümesindeki bir çokbaytlı gösterim). *wc* geçerli bir geniş karakter değilse, *dizge* dizgeside hiçbir şey saklanmaz, **(size_t) -1** döner.

Bir hata oluşmazsa, işlev *dizge* dizgesinde saklanan baytların sayısı ile döner. Bu, öteleme durumlarını gösteren bütün baytları içerir.

İşlevin arayüzünden biraz bahsetmek gerekirse: *dizge* dizgesinin uzunluğunu belirten bir parametre yoktur. Bunun yerine, işlev en azından **MB_CUR_MAX** baytın varlığını kabul eder. Çünkü tek bir karakterin ifade edilebileceği azami bayt sayısı budur. Bu durumda, çağrıcı yeterli yerin mevcut olduğuna emindir, aksi takdirde tampon taşması oluşabilirdi.

wcrtomb işlevi ISO C90 standardının 1. düzeltmesinde tanımlanmış ve `wchar.h` başlık dosyasına bildirilmiştir.

wcrtomb işlevinin kullanımı **mbrtowc** işlevinin kullanımına göre daha kolaydır. Aşağıdaki örnekte, bir geniş karakterli dizge bir çokbaytlı dizgeye eklenmektedir. Tekrar belirtelim; kod kullanılabilir (veya doğru) değildir, bazı sorunlara ve kullanıma bir örnektir.

```
char *
mbscatwcs (char *s, size_t len, const wchar_t *ws)
{
  mbstate_t state;
  /* Mevcut dizgenin uzunluğunu bulalım.  */
  char *wp = strchr (s, '\0');
  len -= wp - s;
  memset (&state, '\0', sizeof (state));
  do
    {
      size_t nbytes;
      if (len < MB_CUR_LEN)
        {
          /* Sonraki karakterin tampona sığacağını garanti etmiyoruz.
             Bu durumda bir hata dönebilir.  */
          errno = E2BIG;
          return NULL;
        }
      nbytes = wcrtomb (wp, *ws, &state);
```

```
      if (nbytes == (size_t) -1)
        /* Dönüşümde hata.  */
        return NULL;
      len -= nbytes;
      wp += nbytes;
    }
  while (*ws++ != L'\0');
  return s;
}
```

İlk işlevde *dizge* dizisi içindeki dizgenin sonu bulunmaktadır. **strchr** işlevi bunu çok iyi yapar, çünkü çokbaytlı karakter gösterimlerinde bir gereklilik olarak boş bayt kendisini temsil etmesi (bu bağlamda dizgenin sonu) dışında bir amaçla asla kullanılmaz.

Durum nesnesini ilklendirip döngüye girilince ilk iş olarak *dizge* dizisinde yeterince yer olup olmadığına bakıyoruz. **MB_CUR_LEN** bayttan daha az yer varsa işlevden çıkıyoruz. Bu daima en iyi seçim olmasa da bizim başka bir şansımız yok. **MB_CUR_LEN** bayttan daha az yer olabilir ve sonraki çokbaytlı karakter sadece bir bayt uzunlukta olabilirdi. Bu durumda tamponda yeterince yerin varlığına karar verecek ek kod nedeniyle işlev çok geç dönerdi. Bu çözüm pek kullanışlı değildir, daha doğru ama çok yavaş bir çözüm olurdu.

```
...
if (len < MB_CUR_LEN)
{
  mbstate_t temp_state;
  memcpy (&temp_state, &state, sizeof (state));
  if (wcrtomb (NULL, *ws, &temp_state) > len)
  {
    /* Sonraki karakterin tampona sığacağını garanti etmiyoruz.
             Bu durumda bir hata dönebilir.  */
    errno = E2BIG;
    return NULL;
  }
}
...
```

Burada tamponun taşabileceği bir dönüşüm uyguluyoruz, yani tamponun boyutu hakkındaki kararı işlemi yaptıktan sonra veriyoruz. **wcrtomb** çağrısındaki hedef tampon için **NULL** argümanına dikkat edin; bu noktada dönüştürülen metinle ilgilenmediğimizden, bu, bu sorunu aşmak için iyi bir yöntemdir. Bu kod parçasındaki en lüzumsuz şey dönüşüm durum nesnesinin yinelenmesidir; ancak eğer *uzunluk*durumda sonraki çokbaytlı karakteri yoksayacak bir değişiklik gerekliyse gerçek dönüşümde aynı öteleme durum değişikliğinin uygulanmasını isteriz. Bunun yanında, ilk öteleme durum bilgisini korumak zorundayız.

Bu soruna çok sayıda ve çok daha iyi çözümler vardır. Bu örnek sadece öğrenim amacıyla hazırlanmıştır.

3.4. Dizge Dönüşümleri

Önceki bölümde bahsedilen işlevler bir defada sadece tek bir karakteri dönüştürmek içindi. Gerçekte uygulanan çoğu işlem dizgeler üzerinde yapılır ve ISO C standardı dizgelerin tamamının dönüşümlerini de tanımlamıştır. Bununla birlikte, tanımlı işlevler bazı sınırlamalara sahiptir; ancak, GNU C kütüphanesi bazı önemli durumlarda yardımcı olabilecek bazı genişletmeler içerir.

```
size_t mbsrtowcs(wchar_t *restrict      hedef,                    işlev
                 const char **restrict kaynak,
                 size_t                uzunluk,
                 mbstate_t *restrict   ps)
```

mbsrtowcs işlevi ("multibyte string restartable to wide character string" kısaltması) **kaynak* içindeki boş karakter sonlandırmalı çokbaytlı karakter dizgesini eşdeğer geniş karakter dizgesine dönüştürür. Dönüşüm, *ps* ile gösterilen nesnedeki ya da *ps* bir boş gösterici ise **mbsrtowcs** içindeki dahili durum bilgisi kullanılarak başlatılır. İşlev dönmeden önce, durum nesnesi son karakter dönüştürüldükten sonraki durumla güncellenir. Sonlandırıcı boş karakter işlenmişse durum, ilk durum olur.

hedef bir boş gösterici değilse, sonuç **mbsrtowcs** ile gösterilen dizide saklanır; aksi takdirde, dönüşüm sonucu bir iç tamponda saklanmış olacağından sonuç kullanılabilir olmayacaktır.

hedef dizgesi *uzunluk* geniş karakterlik olarak belirlenmişse, girdi dizgesinin dönüştürülen kısmı *uzunluk* geniş karakterlik olduğunda dönüşüm durdurulur ve *uzunluk* döner. *hedef* bir boş dizge ise *uzunluk* anlamlı değildir.

Eksik dönüşümlü bir dönüşün sebeplerinden biri de girdi dizgesinin geçersiz çokbaytlı dizilim içermesidir. Bu durumda **errno** genel değişkenine **EILSEQ** değeri atanrak işlev, **(size_t) -1** ile döner.

Tüm diğer durumlarda işlev, bu çağrı ile dönüştürülmüş geniş karakterlerin sayısı ile döner. *hedef* boş değilse, **mbsrtowcs** işlevi *kaynak* ile ya bir boş gösterici (girdi dizgesinde boş karaktere erişilmişse) ya da son dönüştürülen çokbaytlı karakterden sonraki baytın adresini döndürür.

mbsrtowcs işlevi ISO C90 standardının 1. düzeltmesinde tanımlanmış ve `wchar.h` başlık dosyasına bildirilmiştir.

mbsrtowcs işlevinin tanımı önemli bir sınırlama içerir. *hedef*'in bir boş karakter sonlandırmalı dizge içermesi gereksinimi metin içeren tamponlarda dönüştürme yapmak isterseniz sorunlara sebep olur. Bir tampon normalde, boş karakter sonlandırmalı dizgeler değil, herbiri satırsonu karakteri ile biten satırlar içerir. Şimdi bir işlevin tampon içindeki satırlardan birini dönüştürmesini istediğimizi varsayalım. Satır boş karakterle bitmediğinden kaynak gösterici değiştirilmemiş metin tamponunu doğrudan gösteremez. Bunun anlamı: ya **mbsrtowcs** çağrısı sırasında uygun bir yere boş karakter yerleştirilmeli (bu, salt okunur tamponlarda ve çok evreli uygulamalarda yapılamaz) ya da satır, boş karakterle sonlandırılmış olarak başka bir tampona kopyalanmalıdır. *uzunluk* parametresine belli bir değer atayarak dönüştürülecek karakterlerin sayısını sınırlamak genellikle mümkün değildir. Her çokbaytlı karakterin kaç bayttan oluştuğu bilinemediğinden sadece tahmin yapılabilir.

Satırsonu karakterinden sonra bir boş karakter koyarak satırın sonlandırılması ile ilgili olarak çok tuhaf sonuçlara sebep olan bir sorun hala mevcuttur. Yukarıda **mbsrtowcs** işlevinin açıklamasında bahsedildiği gibi girdi dizgesinin sonundaki boş karakter işlendikten sonra durumun ilk öteleme durumu olacağı garanti edilmiştir. Fakat bu boş karakter gerçekte metnin bir parçası değildir (yani, orjinal metinde satırsonu karakterinden sonraki dönüşüm durumu ilk öteleme durumundan farklı olacaktı ve sonraki satırın ilk karakteri bu durum kullanarak kodlanacaktı). Diğer taraftan, dönüşüm boş baytta duracağından (öteleme durumu sıfırlanacağından) gerçekte olması gereken durum bilgisi kaybolacaktır. Günümüzdeki çoğu karakter kümesi bir satırsonu karakterinden sonraki öteleme durumunun ilk durum olmasını gerektirse de bu kesin bir garanti değildir. Basitçe, metnin üzerinde çalışılan parçasının boş karakterle sonlandırılması her zaman uygun bir çözüm değildir ve genellikle hiç kullanılmaz.

Soysal dönüşüm arayüzü (*Soysal Karakter Kümesi Dönüşümü* (sayfa: 145)) bu sınırlamaya sahip değildir (basitçe tamponlarla çalışır, dizgelerle değil) ve GNU C kütüphanesi girdi dizgesinde işlenecek baytların azami sayısının belirtilebildiği ek parametreler alan işlevler içerir. Bu yolla, **mbsrtowcs** işlevinin yukarıdaki örneğindeki sorun, satır uzunluğu saptanıp bu uzunluk işleve aktarılarak çözülebilir.

```
size_t wcsrtombs(char *restrict           hedef,                    işlev
                 const wchar_t **restrict kaynak,
                 size_t                   uzunluk,
                 mbstate_t *restrict      ps)
```

wcsrtombs işlevi ("wide character string restartable to multibyte string" kısaltması) boş karakter sonlandırmalı bir geniş karakter dizgesi olan **kaynak*'ı eşdeğeri olan çokbaytlı karakter dönüştürür ve sonucu *hedef* ile gösterilen dizide saklar. Boş geniş karakter ayrıca dönüştürülür. Dönüşüm, *ps* ile gösterilen nesnedeki ya da *ps* bir boş gösterici ise **wcsrtombs** içindeki dahili durum bilgisi kullanılarak başlatılır. *hedef* bir boş gösterici ise dönüşüm yine yapılır ama sonuç döndürülmez. Girdi dizgesindeki tüm karakterler dönüşütürülmüşse ve *hedef* bir boş gösterici değilse *kaynak* göstericisi bir boş gösterici olarak döner.

Geniş karakterlerden birinin geçerli bir çokbaytlı karakter eşdeğeri yoksa dönüşüm durdurulur ve **errno** genel değişkenine **EILSEQ** değeri atanarak işlev **(size_t) -1** ile döner.

İşlemin eksik kalmasına başka bir sebep de *hedef*'in bir boş karakter olmadığı durumda dönüştürülecek karakterin *uzunluk* bayttan fazlasını gerektirmesidir. Bu durumda *kaynak* başarıyla dönüştürülmüş son karakterden sonraki karakteri gösterecek bir gösterici olarak döner.

Bir kodlama hatası dışında **wcsrtombs** işlevinin dönüş değeri *hedef* içinde saklanan çokbaytlı karakterlerin toplam bayt sayısıdır. İşlev dönmeden önce, durum nesnesi son karakter dönüştürüldükten sonraki durumla güncellenir. Sonlandırıcı boş geniş karakter işlenmişse durum, ilk durum olur.

The **wcsrtombs** işlevi ISO C90 standardının 1. düzeltmesinde tanımlanmış ve `wchar.h` başlık dosyasına bildirilmiştir.

Yukarıda **mbsrtowcs** işlevi ile ilgili sınırlama bu işlevde de geçerlidir. Girdi karakterlerinin sayısını doğrudan belirlemek mümkün değildir. Ya doğru yere bir boş geniş karakter yerleştirilmeli ya da girdi uzunluğu dolaylı olarak çıktı dizgesinin uzunluğu (*uzunluk* parametresi) sınanarak saptanmalıdır.

```
size_t mbsnrtowcs(wchar_t *restrict      hedef,                    işlev
                  const char **restrict kaynak,
                  size_t                nmc,
                  size_t                uzunluk,
                  mbstate_t *restrict   ps)
```

İşlev, **mbsrtowcs** işlevine çok benzer. *nmc* parametresi dışında tüm parametreler ve işlevin dönüş durumu aynıdır.

nmc parametresi ile çokbaytlı karakter dizgesinin en çok kaç baytının kullanılacağı belirtilir. Başka bir deyişle, **kaynak* çokbaytlı karakter dizisinin boş karakter sonlandırmalı olması gerekli değildir. Fakat dizge içinde ilk *nmc* karakter içinde bir boş karaktere rastlanırsa dönüşüm orada durur.

Bu işlev bir GNU oluşumudur. Amacı yukarıda belirtilen sorunun etrafından dolanmaktır. Böylece çokbaytlı karakterler içeren bir tamponun dönüştürülmesi boş karakter yerleştirmeden ve boş karakter dönüşüm durumunu etkilemeksizin mümkün olur.

Bir çokbaytlı karakter dizgesini bir geniş karakter dizgesine dönüştürecek ve gösterecek bir işlev şöyle yazılabilir (bu kullanılabilir bir örnek değildir):

```
void
showmbs (const char *src, FILE *fp)
{
  mbstate_t state;
  int cnt = 0;
  memset (&state, '\0', sizeof (state));
  while (1)
  {
    wchar_t linebuf[100];
    const char *endp = strchr (src, '\n');
```

```
    size_t n;

    /* Başka satır yoksa çık.  */
    if (endp == NULL)
      break;

    n = mbsnrtowcs (linebuf, &src, endp - src, 99, &state);
    linebuf[n] = L'\0';
    fprintf (fp, "line %d: \"%S\"\n", linebuf);
  }
}
```

mbsnrtowcs çağrısından sonra durumla ilgili bir sorun çıkmaz. Dizgelere dışarıdan bir karakter yerleştirmediğimizden *state* durumunu sadece verilen tamponu dönüştürmek için kullandık. Durumun değişmesiyle ilgili bir sorun yaşamadık.

```
size_t wcsnrtombs(char *restrict            hedef,
                  const wchar_t **restrict  kaynak,
                  size_t                    nwc,
                  size_t                    uzunluk,
                  mbstate_t *restrict       ps)
```
işlev

wcsnrtombs işlevi geniş karakter dizgesini çokbaytlı karakter dizgesine dönüştürmekte kullanılır. **wcsrtombs** işlevine benzemekle birlikte **mbsnrtowcs** işlevi gibi girdi dizgesinin uzunluğunun belirtilebildiği bir ek parametre alır.

**kaynak* dizgesinin sadece ilk *nwc* geniş karakteri dönüştürülür. Bu parça içinde bir boş karaktere rastlanırsa dönüşüm bu karakterde durdurulur.

wcsnrtombs işlevi bir GNU oluşumudur ve **mbsnrtowcs** işlevi gibi boş karakter içermeyen dizgelerin dönüştürülmesinde yararlı olur.

3.5. Çokbaytlı Dönüşüm Örneği

Bundan önceki bölümlerde verilen örnekler birer özetti ve hata denetimlerini içermiyordu. Burada tam ve belgelemesi yapılmış bir örneğe yer verilmiştir. **mbrtowc** işlevi kullanılmışsa da diğer sürümleri başka işlevlerin yardımıyla gerçeklenmiştir.

```
int
file_mbsrtowcs (int input, int output)
{
  /* MB_LEN_MAX kullanmalıyız.
     MB_CUR_MAX urada taşınabilir değildir.  */
  char buffer[BUFSIZ + MB_LEN_MAX];
  mbstate_t state;
  int filled = 0;
  int eof = 0;

  /* state'i ilklendirelim.  */
  memset (&state, '\0', sizeof (state));

  while (!eof)
    {
      ssize_t nread;
      ssize_t nwrite;
      char *inp = buffer;
      wchar_t outbuf[BUFSIZ];
```

```
      wchar_t *outp = outbuf;

      /* buffer'ı girdi dosyasından dolduralım.  */
      nread = read (input, buffer + filled, BUFSIZ);
      if (nread < 0)
        {
          perror ("read");
          return 0;
        }
      /* Dosya sonuna geldiğimizde bunu belirleyelim. */
      if (nread == 0)
        eof = 1;

      /* tampona aktarılan baytları sayalım */
      filled += nread;

      /* Bu baytları mümkün olduğunda geniş karakterlere dönüştürelim. */
      while (1)
        {
          size_t thislen = mbrtowc (outp, inp, filled, &state);
          /* Geçersiz karaktere rastlandığında dur;
             böylece son geçerli karaktere kadar olan parçayı okumuş olacağız  */
          if (thislen == (size_t) -1)
            break;
          /* Dizge içinde boş karakter bulursak 0 değeri ile döneceğiz.
             Bu bir düzeltme gerektirir.  */
          if (thislen == 0)
            thislen = 1;
          /* Bu karakterle işimiz bitti, sonrakine geçelim. */
          inp += thislen;
          filled -= thislen;
          ++outp;
        }

      /* Dönüştürülen geniş karakterleri yazalım.  */
      nwrite = write (output, outbuf,
                      (outp - outbuf) * sizeof (wchar_t));
      if (nwrite < 0)
        {
          perror ("write");
          return 0;
        }

      /* Gerçekten bir geçersiz karakterimiz var mı bakalım. */
      if ((eof && filled > 0) || filled >= MB_CUR_MAX)
        {
          error (0, 0, "cokbaytli karakter gecersiz");
          return 0;
        }

      /* karakterleri tamponun başından itibaren yerleştirelim. */
      if (filled > 0)
        memmove (inp, buffer, filled);
    }

  return 1;
}
```

4. Evresel Olmayan Dönüşümler

Önceki kısımda bahsedilen dönüşüm işlevler ISO C90'ın 1. düzeltmesinde tanımlanmış işlevlerdi. Fakat ISO C90 standardı da karakter kümesi dönüşümlerini yapmak için işlevler içermektedir. Önce bu orjinal işlevleri açıklamadık, çünkü hemen hemen tamamen kullanışsızdırlar.

ISO C90'da tanımlanan dönüşüm işlevlerinin tümü sadece işlev içindeki durumu kullanırlar. Bu dahili durum kullanılarak çoklu dönüşümler aynı anda (sadece evreler kullanıldığında değil) yapılamaz ve dönüşüm işlevlerine hangi durumu kullanacakları belirtilemediğinden tek karakter ve dolayısıyla dizge dönüşümleri yapılamaz.

Bu orjinal işlevler sadece çok sınırlı durumlarda kullanılabilir. Yeni bir dizgenin dönüştürülmesine başlamadan önce mevcut dizge tamamen dönüştürülmüş olmalı ve her dizge aynı işlevle dönüştürülmelidir (bu noktada kütüphanenin kendisi bile ilgili bir sorun yoktur; bir işlevin kullandığı durumun başka bir işlev tarafından değiştirilmeyeceği garanti edilmiştir). *Bu sebeplerle, evresel olmayan dönüşüm işlevleri yerine önceki kısımda bahsedilen işlevlerin kullanılması tercih edilmelidir.*

4.1. Evresel Olmayan Karakter Dönüşümleri

```
int mbtowc(wchar_t *restrict     sonuç,                                  işlev
           const char *restrict  dizge,
           size_t                boyut)
```

mbtowc ("multibyte to wide character" kısaltması) işlevi boş olmayan *dizge* ile çağrıldığında *dizge* dizgesinin ilk çokbaytlı karakteri karşılığı olan geniş karakter koduna dönüştürülür ve sonuç **sonuç*'da saklanır.

mbtowc işlevi *boyut* bayttan fazlasına bakmaz.

mbtowc işlevi boş olmayan *dizge* için üç olasılığı ayrımsar: *dizge*'nin ilk *boyut* baytı geçerli çokbaytlı karakterlerle başlar, geçersiz bir bayt dizilimi ile ya da bir karakterin parçası olarak başlar; ya da *dizge* bir boş dizgeyi (bir boş karakter) gösterir.

Geçerli bir çokbaytlı karakteri **mbtowc** bir geniş karaktere dönüştürüp **sonuç*'da saklar ve bu karakterin bayt sayısı ile döner (daima en az 1 dir ve asla *boyut* bayttan büyük değildir).

Geçersiz bir bayt diziliminde **mbtowc** –1 ile döner. Boş bir dizge için 0 ile döner ve **sonuç* içinde **'\0'** saklanır.

Çokbaytlı karakter kodu ötelenmiş karakterleri kullanıyorsa, **mbtowc** bunu düzeltir ve bir öteleme durumuna günceller. **mbtowc** boş gösterici içeren *dizge* ile çağrılırsa öteleme durumunu kendi standart ilk durumuyla ilklendirir. Ayrıca kullanılan çokbaytlı karakter kodu için bir öteleme durumu varsa sıfırdan farklı bir değerle döner. Bkz. *Öteleme Durumu* (sayfa: 144).

```
int wctomb(char    *dizge,                                               işlev
           wchar_t wchar)
```

wctomb ("wide character to multibyte" kısaltması) işlevi *wchar* geniş karakter kodunu karşılığı olan çokbaytlı karaktere dönüştürür ve sonucu bayt cinsinden *dizge* içinde saklar. En çok **MB_CUR_MAX** karakter saklanır.

wctomb işlevi boş olmayan *dizge* ile çağrılırsa *wchar* için üç olasılığı ayrımsar: geçerli bir geniş karakter kodu, geçersiz bir kod ve **L'\0'**.

Geçerli bir kod verilmişse, **wctomb** bunu çokbaytlı karaktere dönüştürüp sonucu *dizge* içinde saklar ve karakterdeki bayt sayısı ile döner (daima en az 1'dir ve **MB_CUR_MAX**'dan büyük değildir).

wchar geçersiz bir geniş karakter kodu ise **wctomb** –1 ile döner. **L'\0'** ise **0** ile döner, ayrıca **dizge* içinde **'\0'** saklar.

Çokbaytlı karakter kodu ötelenmiş karakterler içeriyorsa **wctomb** bunu düzeltir ve bir öteleme durumuna günceller. Ayrıca kullanılan çokbaytlı karakter kodu bir öteleme durumuna sahipse sıfırdan farklı bir değerle döner. Bkz. *Öteleme Durumu* (sayfa: 144).

Bu işlev *wchar* argümanı ile sıfır aktarılarak çağrılması *dizge*'in boş olmaması halinde hem saklanmış öteleme durumunun yeniden ilklendirilmesi hem de **'\0'** çokbaytlı karakterinin saklanması ve işlevin 0 ile dönmesi ile ilgili bir yan etkiye sahiptir.

mbrlen'e benzer olarak bir çokbaytlı karakterin uzunluğunu hesaplayan ve evresel olmayan bir işlev de vardır. **mbtowc** kurallarıyla tanımlanabilir.

```
int mblen(const char *dizge,                                          işlev
          size_t      boyut)
```

mblen işlevi boş olmayan *dizge* argümanı ile çağrıldığında *dizge*'nin başındaki çokbaytlı karakterin *boyut* bayttan daha fazla karakterine bakmaksızın bayt sayısı ile döner.

mblen'in dönüş değeri üç olasılığa bağlıdır: *dizge*'nin ilk *boyut* baytı geçerli çokbaytlı karakterlerle başlar, geçersiz bir bayt dizilimi ile ya da bir karakterin parçası olarak başlar; ya da *dizge* bir boş dizgeyi (bir boş karakter) gösterir.

Geçerli bir çokbaytlı karakter için **mblen** karakterin bayt sayısı ile döner (en az 1, en çok *boyut* bayt). Geçersiz bir bayt dizilimi için –1 ile döner. Bir boş dizge için ise sıfır ile döner.

Çokbaytlı karakter kodu ötelenmiş karakterleri kullanıyorsa, **mblen** bunu düzeltir ve bir öteleme durumuna günceller. **mblen** boş gösterici içeren *dizge* ile çağrılırsa öteleme durumunu kendi standart ilk durumuyla ilklendirir. Ayrıca kullanılan çokbaytlı karakter kodu için bir öteleme durumu varsa sıfırdan farklı bir değerle döner. Bkz. *Öteleme Durumu* (sayfa: 144).

mblen işlevi `stdlib.h` dosyasında bildirilmiştir.

4.2. Evresel Olmayan Dizge Dönüşümleri

ISO C90 standardında tek karakterlerden başka dizgeleri dönüştürecek işlevler de tanımlanmıştır. Bu işlevler ISO C90 standardının 1. düzeltmesinde tanımlanmış olan benzerleriyle aynı sorunlardan muzdariptir; bkz. *Dizge Dönüşümleri* (sayfa: 137).

```
size_t mbstowcs(wchar_t     *gdizge,                                  işlev
                const char *dizge,
                size_t      boyut)
```

mbstowcs ("multibyte string to wide character string" kısaltması) işlevi *dizge* boş karakter sonlandırmalı çokbaytlı karakter dizgesini geniş karakter kodlu bir diziye dönüştürür ve sonucun ilk *boyut* geniş karakterini *gdizge* içinde saklar. Sonlandıran boş karakter boyuta dahildir. Bu durumda *dizge*'den elde edilen geniş karakterlerin sayısı *boyut*'dan fazla ise sonuç sonlandırıcı boş karakteri içermeyecektir.

dizge'deki karakterlerin dönüşümü ilk öteleme durumuyla başlar.

Eğer geçersiz bir çokbaytlı karakter dizgesi bulunursa, **mbstowcs** işlevi –1 değeri ile döner. Aksi takdirde *gdizge* dizisi içinde saklanan geniş karakterlerin sayısı ile döner. Bu sayı *boyut*'tan küçükse sonlandırıcı boş karakter dahil değildir.

Aşağıdaki örnekte bir çokbaytlı karakter dizisinin dönüşümü ve elde edilen sonuç için nasıl yer ayrılacağı gösterilmiştir.

```
wchar_t *
mbstowcs_alloc (const char *string)
{
  size_t size = strlen (string) + 1;
  wchar_t *buf = xmalloc (size * sizeof (wchar_t));

  size = mbstowcs (buf, string, size);
  if (size == (size_t) -1)
    return NULL;
  buf = xrealloc (buf, (size + 1) * sizeof (wchar_t));
  return buf;
}
```

```
size_t wcstombs(char          *dizge,                                    işlev
                const wchar_t *gdizge,
                size_t         boyut)
```

wcstombs ("wide character string to multibyte string" kısaltması) işlevi boş karakter sonlandırmalı *gdizge* geniş karakter dizgesini çokbayktlı karakter dizgesine dönüştürür ve sonucun *boyut* baytını *dizge* içinde saklar. Yeterince yer yoksa sonuç sonlandırıcı boş karakteri içermez. Karakter dönüşümü ilk öteleme durumuyla başlar.

Sonlandırıcı boş karakter boyuta dahildir. Bu bakımdan *boyut*, *gdizge* içindeki baytların sayısına eşit ya da daha küçükse sonlandırıcı boş karakter saklanmaz.

Bir kod için geçerli bir çokbaytlı karakter bulunamazsa işlev –1 ile döner. Aksi takdirde, *dizge* dizisi içinde saklanan baytların sayısı ile döner. Bu sayı *boyut*'tan küçükse sonlandırıcı boş karakter dahil değildir.

4.3. Öteleme Durumu

Bazı çokbaytlı karakterlerin bayt dizilimleri tek başlarına anlamlı değildir; dizge içindeki kendilerinden önce gelen karakterlere bağlı olarak değerlendirilir. Kendinden sonraki dizilimlerin anlamını değiştiren belli başlı birkaç dizilim vardır. Bu dizilimlere ***öteleme dizilimleri*** ve bunların hepsine birden ***öteleme durumu*** denir.

Öteleme dizilimlerini ve öteleme durumunu bir örnekle açıklayalım. **0200** dizilimi (bir baytlık) Japonca kipini başlatır ve bu dizilimden sonra gelen **0240** ile **0377** arasındaki bayt çiftleri birer karakter olarak yorumlanır. Aynı şeklide, **0201** ile Latin–1 kipine girilir ve **0240** ile **0377** arasındaki karakterler Latin–1 karakter kümesindeki karakterler olarak yorumlanır. Burada iki öteleme durumu (Japonca ve Latin–1 kipi) olan bir çokbaytlı kod ile belli öteleme durumlarını belirten iki öteleme diziliminden bahsetmiş olduk.

Kullanımdaki çokbaytlı karakter kodu için öteleme durumları varsa, **mblen**, **mbtowc** ve **wctomb** işlevleri taradıkları dizgenin mevcut öteleme durumlarını düzenlemeli ve güncellemelidir. Bunun olması için bu kurallara uymalısınız:

- Bir dizgeyi taramaya başlamadan önce, çokbaytlı karakter adresi için işlevi **mblen (NULL, 0)** gibi bir boş gösterici ile çağırın. Bu öteleme durumunu işlevin kendi standart ilk değeriyle ilklendirecektir.
- Dizgeyi sırayla bir defada bir karakter işlenecek şekilde tarayın. Taradığınız karakterleri yedeklemeyin ve yeniden taramayın ve bu arada başka dizgeleri işleme sokmayın.

mblen işlevinin bu kurallara uygun olarak kullanımına bir örnek:

```
void
scan_string (char *s)
```

```
{
  int length = strlen (s);

  /* Öteleme durumunu ilklendirelim.  */
  mblen (NULL, 0);

  while (1)
    {
      int thischar = mblen (s, length);
      /* Dizge sonunu ve geçersiz bir karakteri yakalayalım.  */
      if (thischar == 0)
        break;
      if (thischar == -1)
        {
          error ("çokbaytli karakter gecersiz");
          break;
        }
      /* Sonraki karaktere geçelim.  */
      s += thischar;
      length -= thischar;
    }
}
```

mblen, **mbtowc** ve **wctomb** işlevleri öteleme durumu kullanan bir çokbaytlı kod için evresel değildir. Bununla birlikte, bu işlevleri çağıran bir kütüphane işlevi olmadığından öteleme durumunun bilginiz dışında değişmesi mümkün değildir.

5. Soysal Karakter Kümesi Dönüşümü

Bu kısma kadar bahsedilen dönüşüm işlevlerinin hepsi işlem yaptıkları karakter kümelerinin işlevin bir argümanı olarak belirtilmediği tüm karakter kümeleri için kullanılabilen işlevlerdi. Çokbaytlı kodlama **LC_CTYPE** kategorisi için seçilmiş yerel tarafından belirtilen karakter kümesini kullanır. Geniş karakter kümesi gerçeklemeyle sabittir (GNU C kütüphanesi için daima ISO 10646 kodlu UCS–4'tür.).

Bu, genel karakter dönüşümlerinde bir takım sorunlar içerir:

- Her dönüşüm için karakter kümesi ne kaynağın ne de hedefin **LC_CTYPE** kategorisinde belirtilmiş yerel karakter kümesidir. **LC_CTYPE** yereli **setlocale** işlevi kullanılarak değiştirilebilir.

  LC_TYPE yerelinin değiştirilmesi, **LC_CTYPE** kategorisini kullanan işlevlerden (örn, *karakter sınıflandırma işlevleri* (sayfa: 82)) dolayı yazılımın kalanında büyük sorunlara yolaçar.

- **LC_CTYPE** seçimi genel ve tüm evreler tarafından paylaşılan bir seçim olduğundan farklı karakter kümeleri arasındaki dönüşümler mümkün olmaz.

- **wchar_t** gösterimi için kaynağın ve hedefin karakter kümesi kullanılamadığına göre önceki işlevleri kullanarak bir metni dönüştürmek istersek iki kademeli bir işlem gerekir. Birinci adımda çokbaytlı kodlama için kaynak karakter kümesi seçilip metin **wchar_t** metnine dönüştürülecek, ikinci adımda ise hedefin karakter kümesi seçilip geniş karakterli metin bu karakter kümesindeki çok baytlı karakterlere dönüştürülecektir.

  Bu mümkün olsa bile (garanti değildir) çok yorucu bir çalışmadır. Bundan başka, yerelin iki defa değişmesine bağlı olarak bazı sorunlara yol açabilecektir.

XPG2 standardı bu sınırlamaları olmayan tamamen yeni bir işlev ailesi tanımlar. Bunlar seçili yerele bağlı olmadıkları gibi kaynak ve hedef için seçilmiş karakter kümeleriyle ilgili bir sınırlama da getirmezler. Tek sınır dönüşüm için kullanılabilecek karakter kümeleridir. Böyle bir kullanışlılık gerçeklemenin kalitesinin bir ölçüsüdür.

Bundan sonraki bölümlerde önce **iconv** arayüzünden ve dönüşüm işlevinden söz edilecektir. Son olarak gelişkin dönüşüm yeteneklerinden yararlanmak isteyen ileri düzey yazılımcıları ilgilendireceği umularak gerçeklemeden bahsedilecektir.

5.1. Soysal Dönüşüm Arayüzü

Bu işlev ailesi bir özkaynağın geleneksel kullanım yöntemini kullanır: aç–kullan–kapa. Arayüz her biri bir adımı gerçekleştiren üç işlevden oluşur.

Arayüzün açıklanmasına geçmeden önce bir veri türünden bahsetmemiz gerekiyor. Diğer aç–kullan–kapa arayüzler gibi burada bahsedilen işlevler de tanımlayıcıları kullanarak çalışır. Bu tanımlayıcıların kullandığı özel veri türü `iconv.h` başlık dosyasında tanımlanmıştır.

iconv_t veri türü

Bu veri türü `iconv.h` dosyasında tanımlanmış bir soyut türdür. Yazılımcı bu veri türünün tanımıyla ilgili hiçbir kabul yapmamalıdır; şeffaf olmayan bir türdür.

Bu türdeki nesneler **iconv** işlevleri kullanılan dönüşümler için atanmış tanımlayıcıları alabilir. Tanımlayıcıyı kullanan dönüşümlerin nesneyi serbest bırakması gerektiğinden nesnelerin kendi kendilerini serbest bırakmaması gerekir.

İlk adım bir eylemci oluşturan işlevdir.

```
iconv_t iconv_open(const char *hedef_kod,                                    işlev
                   const char *kaynak_kod)
```

iconv_open işlevi bir dönüşüm başlatmadan önce kullanılması gereken işlevdir. İşlev, dönüşüm için kullanılacak karakter kümelerinin belirtildiği iki argüman alır. Gerçekleme böyle bir dönüşümü yapabilecek yeterlilikteyse işlev bir tanımlayıcı ile döner.

İstenen dönüşüm mümkün değilse işlev, **(iconv_t) -1** ile döner. Bu durumda **errno** genel değişkenine şu değerlerden biri atanır:

`EMFILE`
: Süreç zaten **OPEN_MAX** dosya tanımlayıcı açmış.

`ENFILE`
: Sisemin açık dosya sayısı ile ilgili sınırı aşıldı.

`ENOMEM`
: İşlemi gerçekleştirecek kadar bellek yok.

`EINVAL`
: *kaynak_kod* ile *hedef_kod* arasında dönüşüm desteklenmiyor.

Bağımsız dönüşümleri yapacak farklı evrelerde aynı tanımlayıcıyı kullanmak mümkün değildir. Tanımlayıcı ile ilişkili veri yapıları dönüşüm durumu hakkında bilgi içerir. Bu farklı dönüşümlerde kullanılarak altüst edilmemelidir.

Bir **iconv** tanımlayıcı bir dosya tanımlayıcı gibi her kullanım için yeniden oluşturulmalıdır. Tanımlayıcı *kaynak_kod* ile *hedef_kod* arasındaki tüm dönüşümler içindir şeklinde düşünülmemelidir.

GNU C kütüphanesinin **iconv_open** gerçeklemesi diğer gerçeklemelerden farklı olarak önemli bir özelliğe sahiptir. Dönüşüm için desteklenen tüm karakter kümeleri ile ilgili verileri ve kodları içeren dosyalar belli başlı dizinlerde tutulur. Bu özelliğin nasıl gerçeklendiği *glibc iconv Gerçeklemesi* (sayfa: 152) bölümünde açıklanmıştır. Burada üstünde duracağımız tek şey, GCONV_PATH ortam değişkeninde ismi bulunan dizinlerin sadece bir gconv-modules dosyası içermesi halinde geçerli olduğudur. Bu dizinlerin sistem yönetici tarafından oluşturulması şart değildir. Aslında bu özellik kullanıcıların kendilerine özel dönüşümleri yazabilmelerine ve kullanabilmelerine imkan vermek için tasarlanmıştır. Şüphesiz, güvenlik kaygıları nedeniyle SUID çalıştırılabilirlerle bu çalışmaz; bu durumda sadece sistem dizinlerine bakılır. Bunun yeri de normalde *önek*/lib/gconv dizinidir. GCONV_PATH ortam değişkenine sadece **iconv_open** işlevi ilk çağrıldığında bakılır. Değişkende sonradan yapılan bir değişikliğin etkisi yoktur.

iconv_open işlevi ilk olarak X/Open Portability Guide, 2. sürümünde kullanıldı. Daha sonra tüm ticari Unix'ler tarafından desteklendi. Bununla birlikte, gerçeklemenin kalitesi ve bütünlüğü aynı değildir. **iconv_open** işlevi iconv.h dosyasında bildirilmiştir.

iconv gerçeklemesinin geniş veri yapısı **iconv_open** tarafından döndürülen tanımlayıcı ile ilişkilendirilebilir. Diğer taraftan, tüm dönüşümler yapıldıktan sonra ve başka dönüşüm kalmamışsa tüm özkaynakların hemen serbest bırakılması önemlidir.

```
int iconv_close(iconv_t dt)                                                    işlev
```

iconv_close işlevi önceki bir **iconv_open** çağrısından dönen *dt* tanımlayıcısı ile ilişkili tüm özkaynakları serbest bırakır.

İşlev hatasız bir işlem yürütmüşse 0 ile döner, aksi takdirde **errno** değişkenine hata durumunu kaydederek –1 ile döner. İşlev için tanımlı hatalar:

EBADF
: Dönüşüm tanımlayıcı geçersiz.

iconv_close işlevi diğer **iconv** işlevleri ile birlikte XPG2 içinde tanımlanmış ve iconv.h dosyasında bildirilmiştir.

Standart aslında tek bir dönüşüm işlevi tanımlar. Bu çok genel bir arayüz tanımlandığı için böyledir: dönüşümün bir tampondan diğerine yapılacağı öngörülmüştür. Bir dosyadan tampona ve tersi ya da dosyadan dosyaya dönüşümler standarttaki bu tanımı genişleterek gerçekleştirilebilir.

```
size_t iconv(iconv_t dt,                                                       işlev
             char  **girdi_tamponu,
             size_t *girdi_uzunluğu,
             char  **çıktı_tamponu,
             size_t *çıktı_uzunluğu)
```

iconv işlevi *dt* dönüşüm tanımlayıcısı ile ilişkilendirilmiş kurallara uygun olarak girdi tamponundaki metni dönüştürerek sonucu çıktı tamponunda saklar. Gerekli durum bilgisi tanımlayıcı ile ilişkilendirilmiş veri yapıları içinde tutulduğundan aynı metin için işlev defalarca çağrılabilir.

girdi_tamponu* ile belirtilen girdi tamponu **girdi_uzunluğu* bayt içerir. Kullanılan girdinin çağrıcı ile iletişimi için ek olarak bir dolaylı işlem gerekir. Tampon göstericisinin **char türünde olduğuna dikkat edin. Uzunluk, tampondaki karakterler geniş karakterler olsa bile bayt cinsindendir.

Çıktı tamponu da benzer şekilde sonucun saklanacağı **çıktı_tamponu* göstericisinin belirtiği adresten başlayan en az **çıktı_uzunluğu* baytlık bir alan olarak belirtilir. Yine benzer şekilde tampon göstericisi **char** türündedir ve tampon uzunluğu bayt cinsindendir. *çıktı_tamponu* veya **çıktı_tamponu* bir boş gösterici ise dönüşüm yine yapılır ama bir çıktı üretilmez.

girdi_tamponu bir boş gösterici ise, dönüşüm durumunu ilk duruma getiren bir eylem gerçekleştirilir. Bu, durumsal olmayan kodlamalarda bir eylemle sonuçlanmaz, ama eğer, kodlama bir duruma sahipse böyle bir işlev çağrısı gerekli durum değişikliklerini gerçekleştirecek bazı bayt dizilimlerini çıktı tamponuna koyar. Normal bir *girdi_tamponu* ile yapılan bir sonraki işlev çağrısında dönüşüm ilk durumdan başlar. Yazılımcının dönüşümün durumsallığı ile ilgili bir önkabul yapmaktan kaçınması gerekir. Girdi ve çıktı karakterleri durumsal olmasa bile gerçekleme yine de durumları tutmak zorunda olabilir. Aşağıda açıklanan sebeplerden dolayı GNU C kütüphanesinde bu böyledir. Bu nedenle, durumu sıfırlayacak bir **iconv** çağrısı, bir protokol, çıktı metni için bunu gerektiriyorsa daima bunu yapacaktır.

Dönüşüm üç sebepten birine bağlı olarak durabilir. İlki girdi tamponundaki tüm karakterlerin dönüştürüldüğü durumdur. Bu aslında iki farklı anlama gelir: ya girdi tamponundaki tüm karakterler tüketilmiştir ya da tamponun sonunda girdide eksik ama tamponda tam biçimde olan bazı baytlar bulunmuştur. Durmanın ikinci sebebi çıktı tamponunun dolmasıdır. Üçüncüsü ise girdinin geçersiz karakterler içermesidir.

Bu durumların hepsinde sorunsuz yapılan son dönüşümün ardından girdi ve çıktı tamponları *girdi_tamponu* ve *çıktı_tamponu*'nda ve her tamponun uzunluğu sırayla *girdi_uzunluğu* ve *çıktı_uzunluğu* göstericileriyle döndürülür.

iconv_open çağrısında seçilen karakter kümeleri hemen hemen tamamen isteğe bağlı olduğundan girdi karakter kümesinde, çıktı karakter kümesinde karşılığı olmayan karakterler bulunabilir. Bu durumda işlevin davranışı tanımsızdır. GNU C kütüphanesinin bu durum karşısında **şimdiki** davranışı bir hata durumu ile dönmektir. Bu olması gereken bir çözüm değildir; bu nedenle, daha iyi bir çözüm için çalışmalar sürmektedir ve kütüphanenin gelecek sürümlerinde daha iyi bir çözüm sunulacaktır.

Girdi tamponundaki karakterlerin tümü değiştirilip sonuç çıktı tamponunda saklanmışsa, işlev tersinir olmayan dönüşümlerin sayısı ile döner. Bunun dışındaki tüm durumlarda işlev **(size_t) -1** ile döner ve **errno** değişkenine hata durumunu kaydeder. Böyle durumlarda *girdi_uzunluğu* ile gösterilen değer sıfırdan farklıdır.

`EILSEQ`
: Girdi geçersiz bir bayt dizilimi içerdiğinden dönüşüm durdu. Çağrıdan sonra, **girdi_tamponu* geçerli bayt dizilimlerinin ilk baytını gösterir.

`E2BIG`
: Çıktı tamponu yetersiz olduğundan dönüşüm durdu.

`EINVAL`
: Girdi tamponunun sonundaki bayt dizilimi eksik olduğundan dönüşüm durdu.

`EBADF`
: *dt* argümanı geçersiz.

iconv işlevinden XPG2 standardında bahsedilmiş ve işlev `iconv.h` başlık dosyasında bildirilmiştir. **iconv** işlevi, benzerlerinden daha iyi gerçeklenmiştir. Daha esnek bir işlevsellik sunar. Sorunlu kısımlar, geçersiz girdi ve girdi tamponunun sonundaki geçersiz bayt dizilimlerinin olduğu durumlardır. Üçüncü bir sorun varsa da, bu bir tasarım sorunu değildir,dönüşümlerin seçimi ile ilgilidir. Standart meşru isimler hakkında birşey içermez. Bunun diğer gerçeklemeleri nasıl olumsuz etkilediğini sonraki bölümde bir örnekle göstereceğiz.

5.2. **iconv** Örnekleri

Aşağıdaki örnekte ortak bir soruna bir çözüm sunulmaktadır. **wchar_t** dizgeleri için sistem tarafından bilinen kodlamalar verilmiştir. Metin bir dosyadan okunmakta ve geniş karakter tamponunda saklanmaktadır. Dönüşüm **mbsrtowcs** kullanılarak da yapılabilirdi fakat evvelce basettiğimiz sorunlar oluşurdu.

```
int
file2wcs (int fd, const char *charset, wchar_t *outbuf, size_t avail)
{
  char inbuf[BUFSIZ];
  size_t insize = 0;
  char *wrptr = (char *) outbuf;
  int result = 0;
  iconv_t cd;

  cd = iconv_open ("WCHAR_T", charset);
  if (cd == (iconv_t) -1)
    {
      /* Yanlış giden birşeyler olabilir.  */
      if (errno == EINVAL)
        error (0, 0, "'%s' için wchar_t cinsinden bir karşılık yok",
               charset);
      else
        perror ("iconv_open");

      /* Çıktı dizgesini sonlandıralım.  */
      *outbuf = L'\0';

      return -1;
    }

  while (avail > 0)
    {
      size_t nread;
      size_t nconv;
      char *inptr = inbuf;

      /* Girdinin kalanını okuyalım.  */
      nread = read (fd, inbuf + insize, sizeof (inbuf) - insize);
      if (nread == 0)
        {
          /* Buraya gelimişsek okuma tamamlanmıştır. Ama hala
             inbuf içinde kullanılmamış
             karakterler kalmış olabilir. Onları geri koyalım.  */
          if (lseek (fd, -insize, SEEK_CUR) == -1)
            result = -1;

          /* Gerekliyse, ilk duruma girecek bayt dizilimini yazalım. */
          iconv (cd, NULL, NULL, &wrptr, &avail);

          break;
        }
      insize += nread;

      /* Dönüşümü yapalım.  */
      nconv = iconv (cd, &inptr, &insize, &wrptr, &avail);
      if (nconv == (size_t) -1)
        {
          /* Herşey yolunda gitmez. Tamponun sonunda bitmemiş
             bir bayt dizilimi olabilir. Bazan da gerçekten
             önemli bir sorun olabilir.
          if (errno == EINVAL)
```

```
            /* Bu zararsız. Kullanılmayan baytları tamponun başına
               taşıyalım ki, sonraki adımda onlar kullanılabilsin.  */
            memmove (inbuf, inptr, insize);
          else
            {
              /* Bu gerçekten bir sorun. Ya çıktı tamponu yetersiz
                 ya da girdi geçersizdir. Her durumda, dosya
                 göstericisini işlenen son baytın konumuna
                 ayarlayacağız.  */
              lseek (fd, -insize, SEEK_CUR);
              result = -1;
              break;
            }
        }
    }

  /* Çıktı dizgesini sonlandıralım.  */
  if (avail >= sizeof (wchar_t))
    *((wchar_t *) wrptr) = L'\0';

  if (iconv_close (cd) != 0)
    perror ("iconv_close");

  return (wchar_t *) wrptr - outbuf;
}
```

Bu örnek, **iconv** işlevlerinin en önemli kullanım şekillerini göstemektedir. Büyük boyutta bir metnin **iconv** çağrılarıyla nasıl dönüştürülebileceği gösterilmiştir. Kullanıcı durumsal kodlamalar hakkında birşey bilmek zorunda değildir.

iconv işlevinin **errno** değişkenine **EINVAL** değerini atayarak döndüğü durum ilginçtir. Bu gerçekte dönüşümdeki bir hata değildir. Girdi karakter kümesinin tek parça olarak işlenemeyen metinler ve bir bayttan uzun bayt dizilimleri içermesine bağlı olarak oluşur. Bu durumda çokbaytlı dizilimin bölünmesi olasılığı vardır. Çağrıcı basitçe bu sorunlu baytları atlayıp girdiden kalan baytları okuyup **iconv**'ye aktararak dönüşümü devam ettirebilir. ISO C standardında böyle bir olay sonrasındaki duruma bağlı olarak dönüşüm işlevlerinin tanımlayıcıda tutulan dahili durum bilgisinin ne olacağı belirtilmemiştir.

Ayrıca, örnekte **iconv** işlevinde geniş karakterli dizgelerin kullanımına bağlı sorunlar da gösterilmiştir. **iconv** işlevinin açıklamasında belirtildiği gibi, işlev daima bir **char** dizisi alır, dolayısıyla bu dizinin boyutu bayt cinsindendir. Örnekte ise çıktı tamponu bir geniş karakter tamponudur; bu nedenle **iconv** çağrısında **char*** türünde olan *wrptr* yerel değişkenini kullandık.

Bu oldukça masum görünür ama hizalama konusunda sıkı sınırlamaları olan platformlarda sorunlara yol açabilir. Bu nedenle bu tür **iconv** çağrılarında ilgili karakter kümesindeki karakterlere erişmek için uygun bir gösterici kullanılmalıdır. İşlevin girdi parametresi bir **wchar_t** göstericisi olduğundan bu sağlanmıştır (aksi takdirde parametre hesaplanırken hizalama bozulacaktı). Fakat diğer bir durumda, özellikle kullanılan karakter kümesi türünün bilinmediği soysal işlevler yazılırken metinler bayt dizilimleri olarak ele alındığından bu çok zor olabilir.

5.3. Diğer iconv Gerçeklemeleri

Burada diğer sistemlerdeki **iconv** gerçeklemelerini tartışmayacağız, ancak taşınabilir uygulamalar yazılırken onlar hakkında biraz birşeyler bilmek gerekir. Önceki bölümlerde **iconv** işlevinin belirtimi ile ilgili olarak taşınabilirlik sorunlarından bahsetmiştik.

İlk uyarı, kullanılabilecek karakter kümesi sayısının çokluğu ile ilgilidir. Dönüşümlerin doğrudan C

kütüphanesinde kodlanması elbette uygulanabilir değildir. Bu nedenle, dönüşüm bilgisi C kütüphanesi dışından bazı dosyalarla sağlanmalıdır. Bu, aşağıdaki yöntemlerden biri ya da her ikisi birden kullanılarak yapılır:

- C kütüphanesi gerekli dönüşüm tablolarını ve diğer bilgileri veri dosyalarından okuyabilen işlevler ailesi içerir. Bu dosyalar gerektikçe yüklenir.

  Bu çözüm tüm karakter kümelerine (teorik olarak sonsuz sayıda) uygulanması büyük bir çaba gerektirdiğinden oldukça sorunludur. Farklı karakter kümelerinin yapısal farkları çok çeşitli tablo işleme işlevlerinin geliştirilmesini gerektirir.Bu işlevler doğaları gereğince özellikle gerçekleştirilmiş işlevlerden daha yavaştırlar.

- C kütüphanesi özdevimli yüklenebilen nesne dosyaları içerir ve dönüşüm işlevleri bu nesnelerin içeriğini icra eder.

  Bu çözüm çok büyük esneklik sağlar. C kütüphanesinin kendisi çok az kod içerir ve bu nedenle genel bellek ihtiyacı çok azalır. Ayrıca, C kütüphanesi ile yüklenebilir modüller arasındaki belgelemesi yapılmış bir arayüz ile kullanılabilecek dönüşüm modüllerinin üçüncü parti modüllerle genişletilebilmesi mümkündür. Bu çözümün uygulanabilir olması için özdevimli yükleme mümkün olmalıdır.

Ticari Unix'lerdeki bazı gerçeklemeler bu yöntemlerin bir karışımını kullanırken çoğunluğu ikinci çözümü kullanır. Yüklenebilir modüllerin kullanımı kodu kütüphanenin dışına taşır ve genişletmeler ve eklentiler için kapıyı açık tutar. Ancak, durağan olarak ilintilenmiş yazılımlarda özdevimli yükleme desteği olmayan bazı platformda bu tasarım bir sınırlama haline gelebilmektedir. Bu yeteneği olmayan platformlarda bu arayüzün durağan olarak ilintilenmiş yazılımlarda kullanılması bu nedenle mümkün değildir. GNU C kütüphanesinin, ELF platformlarında, bu gibi durumlarda özdevimli yükleme ile ilgili sorunları yoktur; bu nedenle bu konu tartışma götürür. Tehlike bu durumla ilgili bilgi sahibi olunduğu halde diğer sistemlerin sınırlamalarını unutmaktır.

Diğer **iconv** gerçeklemeleri hakkında bilmemiz gereken ikinci şey kullanılabilecek dönüşümlerin genellikle çok sınırlı kalmasıyla ilgilidir. Bazı gerçeklemeler standart dağıtımlarında (geliştirici ve özel uluslararası dağıtımlar değil) en fazla 100 bilemedin 200 dönüşüm olasılığı sağlar. Bu 200 farklı karakter kümesinin desteklendiği anlamına gelse iyi; örneğin, bir karakter kümesinin on farklı karakter kümesine dönşümünü on olarak sayarlar. Bununla birlikte bu karakter kümesinin diğer yöndeki dönüşümü de eklenerek bu sayı 20 yapılır. Bu platformlarda yapılan ince hesabı varın siz düşünün. Bazı Unix şirketleri ise sadece bir dönüşümü diğer dönüşümleri kullanışsız olsa bile türetmek için kullanır.

Bu doğrudan üçüncü ve oldukça büyük bir soruna yol açar. Bu yolla gerçeklenmiş **iconv** dönüşüm işlevlerini kullanan Unix sistemlerinde A kümesinden B kümesine ve B kümesinden C kümesine dönüşüm yapılabilmesi A kümesinden C kümesine dönüşüm yapılabileceği anlamına gelmez.

İlk bakışta bir sorun yokmuş gibi görünse de basit bir uygulama sorunun farkedilmesini sağlar. Sorunu göstermek için A karakter kümesinden C karakter kümesine dönüşüm yapacak bir kod yazdığımızı varsayalım.

```
cd = iconv_open ("C", "A");
```

çağrısı yukarıda belirttiğimiz nedenle başarısız olacaktır. Şimdi yazılımımızın geleceği ne olacak? Bu dönüşüm gerekli...

Tam baş belası. **iconv** işlevi bunu yapmalıydı. Kodu nasıl yazmalı? Önce B karakter kümesine dönüşüm yapmayı deneyebiliriz:

```
cd1 = iconv_open ("B", "A");
```

ve

```
cd2 = iconv_open ("C", "B");
```

Bu çalışacaktır, ama B'nin hangi karakter kümesi olacağını nasıl bileceğiz?

Yanıtı ne yazık ki, genel bir çözümün olmadığıdır. Yine de bazı sistemler bize yardımcı olabilir. Bu sistemlerde çoğu karakter kümesi UTF–8 kodlu ISO–10646 veya Unicode metinlere ve tersine dönüştürülebilir. Bundan başka sisteme çok bağlı bir yöntem de yardımcı olabilir. Bu sistemlerde, dönüşüm işlevleri yüklenebilir modüllerde gelir ve bu modüller dosya sisteminde belirli bir yerde bulunurlar. Bu dosyalara bakarak A kümesinden C kümesine dönüşüm yaparken kullanılacak ortak ara dönüşüm kümesi saptanabilir.

Bu örnek, yukarıda bahsedilen **iconv** tasarım hatalarından birinin örneğidir. Kullanılabilecek dönüşümlerin listesini yazılımsal olarak elde etmek en azından mümkün olmalıdır. **iconv_open** işlevi böyle bir dönüşümün olamayacağını söylerse bu yolla bu liste bulunabilir.

5.4. glibc iconv Gerçeklemesi

Bir önceki bölümde **iconv** gerçeklemelerinin sorunlarını okuduktan sonra GNU C kütüphanesindeki gerçeklemenin bu sorunlardan hiçbirine yol açmadığını söylemek elbette iyi olacaktır. Geliştirme şimdiki durumuna göre (Ocak 1999) genişletilerek sürdürülmektedir. **iconv** işlevlerini geliştirmesi henüz bitmemiş olmakla birlikte temel işlevsellik değişmeyecektir.

GNU C kütüphanesinin **iconv** gerçeklemesi dönüşümleri gerçekleştirmek için paylaşımlı yüklenebilir modülleri kullanır. Kütüphanenin kendi içinde yerleşik olan dönüşümlerin sayısı çok azdır fakat bunlar oldukça önemsiz dönüşümlerdir.

Yüklenebilir modüllerin tüm faydalarından GNU C kütüphanesindeki gerçeklemede yararlanılmıştır. Arayüz iyi belgelendiğinden (aşağıya bakınız), bu özellikle cezbedicidir ve bundan dolayı yeni dönüşüm modüllerini yazmak kolaydır. Yüklenebilir modüllerin kullanımından kaynaklanan sakıncalar GNU C kütüphanesinde en azından ELF sistemlerde sorun çıkarmaz. Kütüphane paylaşımlı nesneleri, durağan olarak ilintili çalıştırılabilirler olsalar bile yükleyebildiğinden durağan ilintilemenin **iconv** kullanmak istendiğinde yasak olmaması gerekir.

Bahsi geçen sorunlardan ikicisi desteklenen dönüşümlerin sayısı ile ilgiliydi. Şu anda GNU C kütüphanesi yüzelliden fazla karakter kümesini desteklemektedir. Bu karakter kümeleri arasında deteklenen dönüşümlerin sayısı ise 22350'den (150 çarpı 149) fazladır. Bir karakter kümesinden diğerinde dönüşüm eksikse kolayca eklenebilir.

Bu yüksek sayıdan dolayı kısmen etkileyici olmakla birlikte GNU C kütüphanesindeki **iconv** gerçeklemesi önceki bölümde bahsedilen üçüncü sorundan da muaftır (A dan B ye ve B den C ye dönüşüm mümkünse A dan C ye doğrudan dönüşüm de daima mümkün olmalıdır da mümkün mü acaba sorunu). **iconv_open** işlevi bir hata ile dönerse ve **errno** değişkenine **EINVAL** değerini atarsa bu, istenen dönüşümün doğrudan ya da dolaylı olarak mümkün olmadığı anlamına gelir.

Üçgenlere bölme her karakter kümesinin diğerine dönüşümü arada UCS–4 kodlu ISO–10646 dönüşümü kullanılarak gerçekleştirilir. Ara dönüşüm için ISO 10646 karakter kümesinin kullanılmasıyla üçgenlere bölmek mümkün olur.

Yeni bir karakter kümesi için ISO 10646 kümesinde dönüşümün gerekliliğinden bahsetmek mümkün olmadığı gibi diğer karakter kümelerine dönüşümde ISO 10646'nın ne kaynak ne de hedef karakter kümesi olarak kullanılmasının gerekliliğinden bahsetmek mümkündür. Mevcut dönüşümlerin tamamı basitçe birbiriyle alakalı dönüşümlerdir.

Şu an mevcut olan tüm dönüşümler dönüşümü gereksiz yere yavaşlatan yukarıda bahsedilen üçgenlere bölme yöntemini kullanırlar. Eğer örneğin, ISO–2022–JP ve EUC–JP gibi bazı karakter kümeleri arasındaki dönüşümün hızlı olması istenirse aradan ISO 10646 çıkarılıp iki karakter kümesi arasında doğrudan bir dönüşüm yaptırılabilir. Bu iki karakter kümesi ISO 10646 ya nazaran birbirine daha benzer karakter kümeleridir.

Böyle bir durumda yeni bir dönüşümü yazmak ve daha iyi bir seçenek üretmek kolaydır. GNU C kütüphanesinin **iconv** gerçeklemesi eğer daha verimli olacağı belirtilirse dönüşümü gerçekleştiren modülü özdevinimli olarak

kullanacaktır.

5.4.1. `gconv-modules` dosyalarının biçimi

Kullanılabilir dönüşümler hakkındaki bilgilerin tamamı **`gconv-modules`** adı verilen bir dosyanın içinde yeralır. Bu dosyanın yeri **`GCONV_PATH`** ortam değişkeninde kayıtlıdır. **`gconv-modules`** dosyaları satırlardan oluşan metin dosyalarıdır. Dosyadaki her satır şöyle yorumlanır:

- Bir satırdaki boşluk olmayan ilk karakter bir **#** karakteri ise bu satırın sadece açıklamaları içerdiği varsayılır ve içeriği yorumlanmaz.

- **`alias`** ile başlayan satırlar karakter kümesi için bir takma ad tanımlar. Bu satırlar üzerinde iki sözcük olması beklenir. İlk sözcük takma adı, ikinci sözcük ise karakter kümesinin gerçek adını belirtir. Bu satırlarda belirtilen takma adları **`iconv_open`** işlevinin *kaynak_kod* veya *hedef_kod* parametrelerinde kullanırsanız gerçek karakter kümesi ismi kullanılmış gibi işlem yapılır.

  Bir karakter kümesinin birçok farklı isim aldığına sıkça rastlanır. Genelde karakter kümesinin resmi ismi yerine halk dilindeki ismi kullanılır. Bundan başka bir karakter kümesinin çeşitli nedenlerle oluşturulmuş özel isimleri de olabilir. Örneğin ISO tarafından belirtilen tüm karakter kümeleri *nnn* kayıt numarası olmak üzere **`ISO-IR-`***nnn* biçiminde bir takma isme sahiptir. Bu, yazılımlarca karakter kümesi isimlerini oluşturan kayıt numaralarının bilinmesini ve bunların **`iconv_open`** çağrılarında kullanılabilmesini mümkün kılar. Bir karakter kümesi ile ilintili tüm takma isimler altalta ayrı satırlarda belirtilebilir.

- **`module`** ile başlayan satırlar mevcut dönüşüm modüllerini içerir. Bu satırlar üç veya dört sözcük içermelidir.

  İlk sözcük kaynak karakter kümesini, ikincisi hedef karakter kümesini, üçüncüsü ise bu dönüşümde yüklenecek olan modülün dosya ismini belirtir. Dosya ismi paylaşımlı nesne dosyalarının uzantısını (normalde **`.so`**) içermemelidir. Bu dosyaların **`gconv-modules`** dosyasının bulunduğu dizinde olduğu varsaylır. Satırdaki son sözcük isteğe bağlıdır. Dönüşüm bedelini gösteren bir tamsayıdır ve belirtilmemişse öntanımlı değeri olan 1 olduğu varsayılır. Dönüşümü gerçekletirecek alt dönüşümlerin sayısını belirtir. Bedel değerinin kullanımını bir örnekle açıklayalım.

Yukarıdaki örneğe dönersek, ISO–2022–JP ile EUC–JP karakter kümeleri arasında doğrudan dönüşüm için bir modülün varlığından sözetmiştik. Her iki yönde de dönüşüm tek bir modül tarafından yapılır ve bu modülün dosya ismi ISO2022JP–EUCJP.so'dur. Dosyanın bulunduğu dizindeki **`gconv-modules`** dosyasının içeriğinde bu modülün tanımı şöyle olurdu:

```
module  ISO-2022-JP//   EUC-JP//        ISO2022JP-EUCJP     1
module  EUC-JP//        ISO-2022-JP//   ISO2022JP-EUCJP     1
```

Bu iki satırın neden yeterli olduğunu anlamak için **`iconv`** tarafından bunların nasıl kullanıldığını anlamak gerekir. Bu soruna yaklaşım oldukça basittir.

`iconv_open` işlevinin ilk çağrısında yazılım tüm **`gconv-modules`** dosyalarını okur ve iki tablo oluşturur: biri bilinen takma adları diğeri dönüşümler ve bunları gerçekleştiren paylaşımlı nesneyi içerir.

5.4.2. `iconv`'de dönüşüm yolunun bulunması

Olası dönüşümlerin listesi önemli kenarların oluşturduğu bir çizge şeklinde tarif edilebilir. Kenarların önemini **`gconv-modules`** dosyalarında belirtilen bedeller belirler. **`iconv_open`** işlevi kaynak ve hedef karakter kümeleri arasındaki en kısa yolu bulmak için böyle bir çizgeyle çalışan bir algoritma kullanır.

`iconv` gerçeklemesinin neden örneğin ISO–2022–JP ve EUC–JP karakter kümeleri arasındaki dönüşümlerde kütüphane ile gelen dönüşümler yerine **`gconv-modules`** dosyalarında belirtilen dönüşüm modülünü kul-

landığını açıklamak artık kolaydır. Kütüphane ile gelen dönüşümler, dönüşümü iki adımda gerçekleştirir (ISO–2022–JP –> ISO 10646 ve ISO 10646 –> EUC–JP). Bu durumda bedel= 1 + 1= 2 olur. Yukarıda ise **gconv-modules** dosyasında bu dönüşümün bedeli 1 olan bir dönüşüm modülü ile gerçekleştirilebileceği belirtilmiştir.

A mysterious item about the **gconv-modules** file above (and also the file coming with the GNU C library) are the names of the character sets specified in the **module** lines. Why do almost all the names end in **//**? And this is not all: the names can actually be regular expressions. At this point in time this mystery should not be revealed, unless you have the relevant spell–casting materials: ashes from an original DOS 6.2 boot disk burnt in effigy, a crucifix blessed by St. Emacs, assorted herbal roots from Central America, sand from Cebu, etc. Sorry! *The part of the implementation where this is used is not yet finished. For now please simply follow the existing examples. It'll become clearer once it is. –drepper*

A last remark about the **gconv-modules** is about the names not ending with **//**. A character set named **INTERNAL** is often mentioned. From the discussion above and the chosen name it should have become clear that this is the name for the representation used in the intermediate step of the triangulation. We have said that this is UCS–4 but actually that is not quite right. The UCS–4 specification also includes the specification of the byte ordering used. Since a UCS–4 value consists of four bytes, a stored value is effected by byte ordering. The internal representation is *not* the same as UCS–4 in case the byte ordering of the processor (or at least the running process) is not the same as the one required for UCS–4. This is done for performance reasons as one does not want to perform unnecessary byte–swapping operations if one is not interested in actually seeing the result in UCS–4. To avoid trouble with endianess, the internal representation consistently is named **INTERNAL** even on big–endian systems where the representations are identical.

5.4.3. `iconv` modülü veri yapıları

Bu bölüme kadar modüllerin yerlerinin kullanımı açıklandı. Burada yeni bir modül yazmak için kullanılacak arayüz açıklanacaktır. Bu bölümde açıklanacak arayüz Ocak 1999'dan beri kullanılmaktadır. Arayüz gelecekte biraz değişecekse de bu değişiklik uyumluluk korunarak yapılacaktır.

Yeni bir modülü yazmak için gerekli tanımlar standartdışı bir başlık dosyası olan `gconv.h` içindedir. Aşağıda, bu dosyada bulunan tanımlar şimdilik sadece bir önbilgi verecek kadar açıklanmıştır.

iconv kullanıcısının bakış açısından arayüz basittir: **iconv_open** işlevi **iconv** çağrılarında kullanılabilen bir tanıtıcı ile döner. Bu tanıtıcının görevi sona erdiğinde bir **iconv_close** çağrısı ile tanıtıcı serbest bırakılır. Burada sorun, tanıtıcı **iconv** işlevine aktarılan herşey olduğundan tanıtıcının tüm dönüşüm adımlarını ve ayrıca her dönüşümün durum bilgisini tutması zorunluluğudur. Bu nedenle, gerçeklemeyi anlamak için en önemi elemanlar aslında bu veri yapılarıdır.

İki farklı veri yapısına ihtiyacımız var. İlki dönüşümü, ikincisi da durumu, v.s. yi açıklamak için. Aslında `gconv.h` dosyasında bunun gibi iki tanım vardır.

`struct __gconv_step` veri türü

Bu veri yapısı bir modülün gerçekleştirdiği bir dönüşümü açıklar. Dönüşüm işlevleri ile yüklenen bir modüldeki her işlev için bu türde tek bir nesne vardır (bu nesne asıl dönüşüme ilişkin bir bilgi içermez, sadece dönüşümün kendisini tanımlar).

`struct __gconv_loaded_object *__shlib_handle`
`const char *__modname`
`int __counter`

Yapının bu elemanları C kütüphanesi tarafından dahili olarak kullanılır ve paylaşımın yüklenmesini ve kaldırılmasını yönetirler. Birinin kullanılmış olması diğerlerinin kullanılmasını ya da ilklendirilmesini gerektirmez.

`const char *__from_name`
`const char *__to_name`
: **__from_name** ve **__to_name** alanları kaynak ve hedef karakter kümelerinin isimlerini içerir. Bir modül birden fazla karakter kümesi ve yönde dönüşüm için kullanılabildiğinden asıl dönüşümü tanımlamakta kullanılır.

`gconv_fct __fct`
`gconv_init_fct __init_fct`
`gconv_end_fct __end_fct`
: Bu alanlar yüklenebilir modüldeki işlevlere göstericileri içerir. Arayüz aşağıda açıklanacaktır.

`int __min_needed_from`
`int __max_needed_from`
`int __min_needed_to`
`int __max_needed_to;`
: Bu değerler modülün ilklendirme işleviyle atanmalıdır. **__min_needed_from** değeri kaynak karakter kümesinin en az kaç bayt gerektirdiğini belirtir. **__max_needed_from** değeri azami değer ile ayrıca olası öteleme dizilimlerini belirtir.

  The **__min_needed_to** ve **__max_needed_to** ise benzer değerleri hedef karakter kümesi için içerir.

  bu değerlerin doğruluğu son derece önemlidir, çünkü aksi takdirde dönüşüm işlevleri sorun çıkaracak ve bekleneni yapmayacaktır.

`int __stateful`
: Bu eleman da ilklendirme işlevi ile ilklendirilmelidir. Kaynak karakter kümesi durumsal ise bu elemanın değeri sıfırdan farklı olacaktır.

`void *__data`
: Bu eleman modüldeki dönüşüm işlevleri tarafından serbestçe kullanılabilir. **void *__data** bir çağrıdan diğerine fazladan bilgi aktarmak için kullanılabilir. Gerekmedikçe ilklendirilmesi gerekmez. Elemana özdevimli ayrılmış bir bellek alanının göstericisi atanmışsa (büyük ihtimalle ilklendirme işlevi tarafından), kullanılacak son işlevin bu alanı serbest bırakması sağlanmalıdır. Aksi takdirde uygulama bellek kaçağına sebep olur.

  Bu veri yapısının bu belirtim dönüşümünün tüm kullanıcıları tarafından paylaşılmasının sağlanması önemlidir. Bu nedenle, **__data** elemanının belli bir dönüşüm işlevinin kullanımına özel veri içermemesi gerekir.

`struct __gconv_step_data` veri türü

Bu veri türü dönüşüm işlevlerinin kullanımına özel bilgileri içerir.

`char *__outbuf`
`char *__outbufend`
: Bu elemanlar dönüşüm adımında kullanılan çıktı tamponuna ilişkin veriyi içerir. **__outbuf** elemanı tamponun başlangıcını, **__outbufend** elemanı ise tamponun son baytını belirtir. Dönüşüm işlevi tamponun boyutunun herşeye yeterli olduğunu varsaymamalı, ancak en azından tam bir karakter için tamponda yeterli yer olduğunu varsaymalıdır.

  Dönüşüm bittikten sonra, eğer dönüşüm son adımdaysa, son bayt tampona yazıldıktan sonra ne kadar mevcut çıktı olduğunu belirtmek için **__outbuf** elemanı değiştirilmelidir, **__outbufend** elemanı değiştirilmemelidir.

`int` **`__is_last`**
Dönüşüm işlemi son adımdaysa bu elemanın değeri sıfırdan farklıdır. Bu bilgi yineleme için gereklidir. Aşağıdaki dönüşüm işlevlerinin iç yapıları ile ilgili açıklamalara bakınız. Bu eleman asla değiştirilmemelidir.

`int` **`__invocation_counter`**
Dönüşüm işlevi bu elemanı kaç defa çağrıldığı bilgisini tutmak için kullanabilir. Bazı karakter kümeleri dönüşüm işlevinin ilk adımında bazı çıktılar üretirler. Bu alan bu ilk adımın belirlenmesi için kullanılabilir. Bu eleman asla değiştirilmemelidir.

`int` **`__internal_use`**
Bu eleman belli bazı durumlar için kullanılabilen alanlardan biridir. Dönüşüm işlevleri **`mbsrtowcs`** işlevini gerçeklemekte kullanılmışsa (yani, işlev **`iconv`** gerçeklemesi üzerinden doğrudan kullanılmamışsa) bu alana sıfırdan farklı bir değer atanır.

`mbsrtowcs` işlevlerinin normalde bütün metni dönüştürmek için tek tek dizgeleri dönüştürmek üzere defalarca çağrılmasına karşın **`iconv`** işlevlerinin metnin tamamını dönüştürmekte kullanılması gibi bir fark oluşur.

Fakat bu durumda karakter kümesinin belirtimindeki bazı kuralların yerine getirilmesiyle ilgili bazı sorunlarla karşılaşılır. Bazı karakter kümeleri metnin tamamı için bir defalığına ilk adımda bir çıktı verilmesini gerektirir. Eğer metni dönüştürme işlemi **`mbsrtowcs`** işlevinin defalarca çağrılmasını gerektiriyorsa, ilk çağrıda bu çıktı verilmelidir. Bununla birlikte, **`mbsrtowcs`** işlevinin çağrıları arasında iletişim olmadığından dönüşüm işlevlerinin bu çıktıyı vermesi mümkün olmaz. Bu durum tanıtıcı sayesinde bu bilgiye erişebilen **`iconv`** çağrılarından farklı bir durumdur.

`int __internal_use` elemanı çoğunlukla **`__invocation_counter`** elemanı ile birlikte ağağıdaki gibi kullanılır:

```
if (!data->__internal_use
      && data->__invocation_counter == 0)
  /* İlk adım çıktısını bas.  */
  …
```

Bu elemanın değeri asla değiştirilmemelidir.

`mbstate_t *` **`__statep`**
`__statep` elemanı **`mbstate_t`** (bkz. *Durumun saklanması* (sayfa: 131)) türünde bir nesneye göstericidir. Durumsal bir karakter kümesi dönüşüm durumu hakkındaki bilgileri saklamak için **`__statep`** ile gösterilen alanı kullanmalıdır. **`__statep`** elemanı kendini asla değiştirmemelidir.

`mbstate_t` **`__state`**
Bu eleman *asla* doğrudan değiştirilmemelidir. Yapının kullandığı alanın belirtildiği bir elemandır.

5.4.4. `iconv` modül arayüzleri

Veri yapıları hakkında bilgi edindikten sonra dönüşüm işlevlerinin açıklamalarına girebiliriz. Arayüzü anlayabilmek için dönüşüm nesnelerini yükleyen C kütüphanesindeki işlevsellik hakkında biraz önbilgi vermek gerekir.

Bir dönüşümün defalarca kullanıldığı duruma sıklıkla rastlanır (yazılımın çalışması esnasında aynı karakter kümesi için çok sayıda **`iconv_open`** çağrısı yapılması). GNU C kütüphanesindeki **`mbsrtowcs`** ve benzeri işlevler de aynı işlevin kullanım sayısını arttıran **`iconv`** işlevselliğini kullanırlar.

Dönüşümlerin böyle defalarca kullanılabilmesinden dolayı modüller her dönüşüm için tekrar yüklenmezler. Modül bir kere yüklendikten sonra aynı **iconv** veya **mbsrtowcs** çağrıları defalarca yapılabilir. Bilgilerin işleve özel dönüşüm bilgileri ve dönüşüm verisi bilgileri olarak ayrılabilmesi sayesinde bu mümkün olur. Bunun yapılmasını sağlayan iki veri yapısından bir önceki bölümde bahsedilmişti.

Bu ayrıca, modül tarafından sağlanan işlevlerin sözdiziminde ve arayüzün kendisinde de yansıtılır. Aşağıdaki isimlere sahip olması zorunlu üç işlev vardır:

gconv_init
: **gconv_init** işlevi, dönüşüm işlevine özel veri yapısını ilklendirir. Bu nesne bu dönüşümü kullanan tüm dönüşümlerce paylaşılır ve bu nedenle, dönüşümün kendisi ile ilgili durum bilgisi burada saklanmaz. Bir modül birden fazla dönüşümü gerçekleştirebiliyorsa **gconv_init** işlevi defalarca çağrılabilir.

gconv_end
: **gconv_end** işlevi, **gconv_init** işlevi ile ayrılan tüm özkaynakları serbest bırakmak için kullanılır. Böle bir işlem gerekmiyorsa bu işlev tanımlanmayabilir. Modül birden fazla dönüşümü gerçekleştirebiliyorsa ve **gconv_init** işlevi her dönüşüm için aynı özkaynakları ayırmıyorsa bu işlevi tanımlarken dikkatli olunmalıdır.

gconv
: Asıl dönüşüm işlevidir. Bir metin bloğunu dönüştürmekte kullanılır. Dönüşüm işlevlerinin bu amaca yönelik dönüşüm verisi ve **gconv_init** ile ilklendirilen dönüşüm adım bilgisi işleve aktarılmalıdır.

Bu üç modül arayüz işlevi için tanımlanmış üç veri türü vardır ve bunlar arayüzü tanımlar:

```
int (*__gconv_init_fct)(struct __gconv_step *)                    işlev
```

Modülün gerçekleştirdiği her dönüşüm için sadece bir kere çağrılan ilklendirme işlevinin arayüzünü belirler.

struct __gconv_step veri yapısının açıklamasında değinildiği gibi ilklendirme işlevi bu veri yapısının elemanlarını ilklendirir.

__min_needed_from
__max_needed_from
__min_needed_to
__max_needed_to
: Bu elemanlar kaynak ve hedef karakter kümelerindeki bir karakteri oluşturan bayt sayısının azami ve asgari değerleri ile ilklendirilmelidir. Karakaterlerin hepsi aynı bayt sayısı ile ifade ediliyorsa azami ve asgari değerler aynı olacaktır.

__stateful
: Bu eleman kaynak karaker kümesinin durumsal olması halinde sıfırdan farklı, aksi takdirde sıfır olmalıdır.

İlklendirme işlevinin dönüşüm işlevi ile bilgi alışverişi yapması gerekliyse, bu iletişim **__gconv_step** yapısının **__data** elemanı kullanılarak yapılabilir. Ancak, bu veri tüm dönüşümlerce paylaşılacağından dönüşüm işlevleri bu veriyi değiştirmemelidir. Bunun yapılışına bir örnek:

```
#define MIN_NEEDED_FROM         1
#define MAX_NEEDED_FROM         4
#define MIN_NEEDED_TO           4
#define MAX_NEEDED_TO           4

int
gconv_init (struct __gconv_step *step)
```

```
{
  /* Dönüşüm yönünü belirleyelim.  */
  struct iso2022jp_data *new_data;
  enum direction dir = illegal_dir;
  enum variant var = illegal_var;
  int result;

  if (__strcasecmp (step->__from_name, "ISO-2022-JP//") == 0)
    {
      dir = from_iso2022jp;
      var = iso2022jp;
    }
  else if (__strcasecmp (step->__to_name, "ISO-2022-JP//") == 0)
    {
      dir = to_iso2022jp;
      var = iso2022jp;
    }
  else if (__strcasecmp (step->__from_name, "ISO-2022-JP-2//") == 0)
    {
      dir = from_iso2022jp;
      var = iso2022jp2;
    }
  else if (__strcasecmp (step->__to_name, "ISO-2022-JP-2//") == 0)
    {
      dir = to_iso2022jp;
      var = iso2022jp2;
    }

  result = __GCONV_NOCONV;
  if (dir != illegal_dir)
    {
      new_data = (struct iso2022jp_data *)
        malloc (sizeof (struct iso2022jp_data));

      result = __GCONV_NOMEM;
      if (new_data != NULL)
        {
          new_data->dir = dir;
          new_data->var = var;
          step->__data = new_data;

          if (dir == from_iso2022jp)
            {
              step->__min_needed_from = MIN_NEEDED_FROM;
              step->__max_needed_from = MAX_NEEDED_FROM;
              step->__min_needed_to = MIN_NEEDED_TO;
              step->__max_needed_to = MAX_NEEDED_TO;
            }
          else
            {
              step->__min_needed_from = MIN_NEEDED_TO;
              step->__max_needed_from = MAX_NEEDED_TO;
              step->__min_needed_to = MIN_NEEDED_FROM;
              step->__max_needed_to = MAX_NEEDED_FROM + 2;
            }

          /* Evet, bu durumsal bir kodlama.  */
```

```
          step->__stateful = 1;

          result = __GCONV_OK;
        }
    }

  return result;
}
```

İşlev önce hangi dönüşümün istendiğine bakar. Bu işlev ile ele alınan modül dört farklı dönüşümü gerçekleştirmektedir; hangisinin kullanılmak istendiği isimlere bakarak saptanabilir. Karşılaştırma daima harf büyüklüğünden bağımsız olarak yapılmalıdır.

Sonra, seçilen dönüşüm için gerekli bilgileri içeren veri yapısına sıra geliyor ve buna yer ayrılıyor. **struct iso2022jp_data** yerel olarak modülün dışında tanımlandığından bu veri işlev dışında kullanılamaz. Eğer modülün desteklediği dört dönüşümün tamamı için dönüşüm istenseydi, dört veri bloğu olacaktı.

İlginç olan veri nesnesinin **__min_** ve **__max_** elemanlarının ilklendirme adımıdır. Tek bir ISO–2022–JP karakteri bir bayttan dört bayta kadar uzunlukta olabilir. Bundan dolayı burada **MIN_NEEDED_FROM** ve **MAX_NEEDED_FROM** makroları kullanılmıştır. Çıktı daima dahili karakter kümesi (UCS–4) olacağından her karakter daima dört bayt uzunlukta olacaktır. Dahili karakter kümesinden ISO–2022–JP karakter kümesine dönüşüm için önceleme dizilimlerinin karakter kümeleri arasında geçiş yapmak için gerekli olduğunu hesaba katmak zorundayız. Bu nedenle, bu yön için **__max_needed_to** elemanına **MAX_NEEDED_FROM + 2** değeri atanmaktadır. Böylece diğer karakter kümesine geçiş için gekeli olan önceleme dizilimleri hesaba katılmış olur. İki yöndeki azami değerler arasındaki dengesizlik kolayca açıklanabilir: ISO–2022–JP metin okunurken önceleme dizilimleri tek başlarına elde edilebilir (yani, önceleme diziliminin etkisi durum bilgisi içinde kaydedilmiş olacağından önceleme dizilimi bir gerçek karakterin işlenmesinde gerekli değildir). Diğer yönde durum farklıdır. Hangi karakter kümesinin sonra geleceği genelde bilinmediğinden durumu değiştirecek önceleme dizilimleri ileriye dönük hesaba katılamaz. Bu, önceleme dizilimlerinin sonraki karakter ile birlikte ele alınması zorunluluğu demektir. Bu nedenle karakterin gerektirdiğinden daha fazla alan gereklidir.

İlklendirme işlevinin olası dönüş değerleri şunlardır:

__GCONV_OK
: İlklendirme başarılı.

__GCONV_NOCONV
: İstenen dönüşümü bu modül desteklemiyor. Bu durum, **gconv-modules** dosyası hatalıysa oluşabilir.

__GCONV_NOMEM
: Ek bilginin saklanacağı bellek ayrılamadı.

Modül yüklenmeden işlevin çağrılması önemce daha kolaydır. Çoğunlukla hiçbir şeye sebep olmaz; tamamen ihmal edilebilir.

```
void (*__gconv_end_fct)(struct gconv_step *)                    işlav
```

Bu işlevin görevi ilklendirme işlevinin ayırdığı tüm özkaynakları serbest bırakmaktır. Bu nedenle argüman olarak nesnenin sadece **__data** elemanını kullanır. İlklendirme işlevi ile ilgili örneğe devam edersek dönüşümü sonlandıran işlev şöyle olurdu:

```
void
gconv_end (struct __gconv_step *data)
{
  free (data->__data);
}
```

En önemli işlev, karmaşık karakter kümeleri için oldukça karışık olabilen dönüşüm işlevinin kendisidir. Daha fazlası gerekli olmadığından burada işlevin sadece iskeletinden bahsedilecektir.

```
int (*__gconv_fct)(struct __gconv_step *,                                    işlev
                   struct __gconv_step_data *,
                   const char **,
                   const char *,
                   size_t *,
                   int)
```

Dönüşüm işlevinin çağrılmasını gerektiren iki sebep olabilir: metni dönüştürmek ya da durumu sıfırlamak. **iconv** işlevinin açıklamasına bakılırsa boşaltma kipinin neden gerekli olduğu görülebilir. Hangi kipin seçilmiş olduğu bir tamsayı olan altıncı argümana bakılarak saptanır. Boşaltma kipi seçilmişse bu argümanın değeri sıfırdan farklı olacaktır.

Çıktı tamponunun yerinin her iki kip için ortak olduğu görülür. Bu tampon hakkındaki bilgi dönüşüm adım verisinde saklanır. Bu bilgiye ilişkin gösterici işleve ikinci argüman olarak aktarılır. **struct __gconv_step_data** yapısının açıklaması dönüşüm adım verisi hakkında daha fazla bilgi içerir.

Boşaltma için ne yapılacağı kaynak karakter kümesine bağlıdır. Eğer kaynak karakter kümesi durumsal değilse birşey yapmak gerekmez. Aksi takdirde, işlev durum nesnesini ilk duruma getirecek bir bayt dizilimini göndermek zorundadır. Bu yapıldıktan sonra dönüşüm zincirindeki diğer dönüşüm işlevlerine de bu imkan tanınmalıdır. Bu işlevi başka bir adımın izleyip izlemeyeceği ilk argümana adım verisinin **__is_last** elemanındaki bilgi aktarılarak saptanabilir.

Metnin dönüştürüldüğü kip daha ilginçtir. Bu kipteki ilk adımda girdi tamponundaki metnin olabildiğince büyük bir kısmı dönüştürülür ve sonuç çıktı tamponunda saklanır. Girdi tamponunun başlangıcı, tamponun başlangıcını gösteren bir göstericiye gösterici olan üçüncü argümandan saptanır. Dördüncü argüman tampondaki son bayttan sonraki bayta bir göstericidir.

Dönüşüm, eğer karakter kümesi durumsal ise mevcut duruma bağlı olarak uygulanır. Durum bilgisi adım verisinin **__statep** elemanı tarafından gösterilen bir nesnede saklanır (ikinci argüman). Girdi tamponu boşsa ya da çıktı tamponu dolmuşsa dönüşüm durdurulur. Bu durumda üçüncü parametredeki gösterici değişkeni son işlenen bayttan sonraki baytı göstermelidir (eğer tüm girdi tüketilmişse, bu gösterici ve dördüncü parametre aynı değerde olur).

Sonra ne yapılacağı bu adımın son adım olup olmamasına bağlıdır. Eğer bu adım son adımsa yapılacak tek şey, adım verisi yapısının **__outbuf** elemanının son yazılan bayttan sonraki baytı gösterecek şekilde güncellenmesidir. Ek olarak, beşinci parametre tarafından gösterilen **size_t** türündeki değişken geri dönüşümsüz olarak dönüştürülen karakter sayısı (bayt sayısı değil) kadar arttırılmalıdır. Bundan sonra işlev dönebilir.

Eğer bu adım son adım değilse, sonraki dönüşüm işlevlerine kendi görevlerini yerine getirebilmeleri imkanı sağlanmalıdır. Bu nedenle uygun dönüşüm işlevi çağrısı yapılmalıdır. İşlevler hakkındaki bilgiler dönüşüm veri yapılarında saklanır ve ilk parametre olarak aktarılır. Bu bilgi ve adım verisi dizilerde saklanır, bu durumda sonraki eleman her iki halde de basit olarak gösterici aritmetiği ile bulunabilir:

```
int
gconv (struct __gconv_step *step, struct __gconv_step_data *data,
       const char **inbuf, const char *inbufend, size_t *written,
       int do_flush)
{
  struct __gconv_step *next_step = step + 1;
  struct __gconv_step_data *next_data = data + 1;
  …
```

next_step göstericisi sonraki adım bilgisini içerirken, **next_data** sonraki veri kaydını içerir. Sonraki işlev çağrısı bu nedenle şöyle görünecektir:

```
  next_step->__fct (next_step, next_data, outerr, outbuf,
                    written, 0)
```

Fakat henüz bu yeterli değildir. İşlev çağrısı döndükten sonra dönüşüm işlevi biraz daha işlem yapmak zorunda kalabilir. İşlevin dönüş değeri **__GCONV_EMPTY_INPUT** ise, çıktı tamponunda hala yer var demektir. Girdi tamponu boş olmadıkça dönüşüm, işlevi girdi tamponunun kalanını işlemek üzere tekrar çağırır. Eğer dönüş değeri **__GCONV_EMPTY_INPUT** değilse bazı şeyler yanlış gitmiştir ve bunun kurtarılması gerekir.

Dönüşüm işlevi için bir gereklilik de, girdi tamponu göstericisinin (üçüncü argüman) daima çıktı tamponuna konulan dönüştürülmüş son karakteri göstermesidir. Dönüşümün uygulandığı adımda eğer daha alt adımları gerçekleştiren dönüşüm işlevleri hata verip durmazsa, çıktı tamponundaki karakterlerin tümü tüketilmemişse ve bu nedenle girdi tamponu göstericileri doğru konumu gösterecek duruma getirilmemişse bu zaten gerçekleşir.

Girdi tamponunun düzeltilmesi, eğer girdi ve çıktı karakter kümelerindeki tüm karakterler sabit genişlikteyse kolaydır. Bu durumda, çıktı tamponunda kaç karakter kaldığını heesaplayabilir ve bu sonuçtan hareketle girdi tampon göstericisini benzer bir hesaplamayla elde edebiliriz. Zor olan, karakter kümelerinin değişken genişlikte karakterler içermesi ve dönüşüm bir de durumsal ise işlemin daha da karmaşıklaşmasıdır. Bu durumlarda dönüşüm, ilk dönüşümden önceki bilinen durumdan bir daha başlatılır (gerekliyse, dönüşüm durumu sıfırlanmalı ve dönüşüm döngüsü tekrar çalıştırılmalıdır). Burada farklı olan ne kadar girdi oluşturulması gerektiğinin bilinmesi ve dönüşümün ilk işe yaramaz karakterden önce durdurulabilmesidir. Bu yapıldıktan sonra girdi tamponu göstericileri tekrar güncellenmelidir. Bundan sonra işlev dönebilir.

Üstünde durulması gereken son bir şey daha var. Dönüşümün ilk çağrısında bir iletinin çıktılanması gerektiği durumlar için çağrının ilk çağrı olup olmadığının bilinmesi için dönüşüm işlevi adım verisi yapısının **__invocation_counter** elmanının değerini çağrıcıya dönmeden önce arttırmalıdır. Bunun nasıl kullanıldığı hakkında daha fazla bilgi edinmek isterseniz **struct __gconv_step_data** yapısının açıklamasına bakınız.

Dönüş değeri şunlardan biri olmalıdır:

__GCONV_EMPTY_INPUT
: Tüm girdi tüketildi ve çıktı taponunda yer kaldı.

__GCONV_FULL_OUTPUT
: Çıktı tamponunda yer kalmadı. Bu değer son adımda alınmamışsa zincirdeki sonraki işlev çağrısında bu değere uygun işlem yapılmalıdır.

__GCONV_INCOMPLETE_INPUT
: Bozuk bir karakter dizilimi içerdiğinden girdi tamponu tüketilememiştir.

Aşağıda bir dönüşüm işlevinin çerçevesi çizilmeye çalışılmıştır. Yeni bir dönüşüm işlevi yazılacaksa, burada boş bırakılmış yerler doldurulmalıdır.

```
int
gconv (struct __gconv_step *step, struct __gconv_step_data *data,
       const char **inbuf, const char *inbufend, size_t *written,
       int do_flush)
{
  struct __gconv_step *next_step = step + 1;
  struct __gconv_step_data *next_data = data + 1;
  gconv_fct fct = next_step->__fct;
  int status;

  /* İşlev girdisiz çağrılmışsa bu ilk duruma getirme anlamındadır.
     Girdinin bir kısmı işlendikten sonra bu yapılmışsa girdi atlanır. */
  if (do_flush)
    {
      status = __GCONV_OK;

      /* Durum nesnesini ilk duruma getiren bayt dizilimi gerekebilir. */

      /* Call the steps down the chain if there are any but only
        if we successfully emitted the escape sequence.  */
      if (status == __GCONV_OK && ! data->__is_last)
        status = fct (next_step, next_data, NULL, NULL,
                      written, 1);
    }
  else
    {
      /* Gösterici değişkenlerinin ilk değerlerini saklayalım.  */
      const char *inptr = *inbuf;
      char *outbuf = data->__outbuf;
      char *outend = data->__outbufend;
      char *outptr;

      do
        {
          /* Bu adımın başlangıç değerini hatırlayalım.  */
          inptr = *inbuf;
          /* The outbuf buffer is empty.  */
          outptr = outbuf;

          /* Durumsal kodlamalar için durum burada güvenceye alınmalı.  */

          /* Dönüşüm döngüsü çalıştır ve durumu uygun değere ayarla.  */

          /* Bu son adımsa, döngüden çık. Yapacak birşey kalmamış.  */
          if (data->__is_last)
            {
              /* Kullanılabilir kaç bayt kaldı bilgisini sakla.  */
              data->__outbuf = outbuf;

            /* geri dönüşümsüz dönüşüm yapılmışsa numarasını
               *written'a ekle.  */

            break;
          }
```

```
          /* Üretilen tüm çıktıyı yaz.  */
          if (outbuf > outptr)
            {
              const char *outerr = data->__outbuf;
              int result;

              result = fct (next_step, next_data, &outerr,
                            outbuf, written, 0);

              if (result != __GCONV_EMPTY_INPUT)
                {
                  if (outerr != outbuf)
                    {
                      /* Girdi tampon göstericisini sıfırla.
                         Burada karmaşık durumu belgeleyelim.  */
                      size_t nstatus;

                      /* Göstericileri yeniden yükle.  */
                      *inbuf = inptr;
                      outbuf = outptr;

                      /* Durum sıfırlanacaksa sıfırla.  */

                      /* Dönüşümü tekrar yap, ama bu sefer çıktı
                         tamponunun sonu outerr de.  */
                    }

                  /* urumu değiştir.  */
                  status = result;
                }
              else
                /* Çıktı tamamlandı, herşey tamamsa
                   sonraki adıma geçelim.  */
                if (status == __GCONV_FULL_OUTPUT)
                  status = __GCONV_OK;
            }
        }
      while (status == __GCONV_OK);

      /* Bu adımdaki işimiz bitti.  */
      ++data->__invocation_counter;
    }

  return status;
}
```

Yeni modül yazmak için bu kadar bilgi yeterlidir. Bunu yapmak isteyenler GNU C kütüphanesinin kaynak koduna da bakabilirler. Pek çok çalışan ve eniyilenmiş örnek bulunmaktadır.

VII. Yereller ve Uluslararasılaştırma

İçindekiler

Farklı ülkeler ve kültürlerin kendi içlerinde iletişim kurma konusunda farklı uzlaşımları vardır. Bu uzlaşımlar tarih ve zaman gösterimleri gibi basit uzlaşımlardan konuşulan dil gibi karmaşık uzlaşımlara kadar değişiklik gösterir.

Yazılımın *uluslararasılaştırma*sı denince yazılımın kullanıcının tercih ettiği uzlaşımlara uyarlanması anlaşılır. ISO C'de, uluslararasılaştırma yerini *yerelle*re bırakır. Her yerel her bir uzlaşımın başka bir amaca yönelik olduğu bir uzlaşımlar bütünüdür. Kullanıcı uzlaşım kümesini bir yerel belirterek (ortam değişkenleri üzerinden) seçer.

Bütün yazılımlar seçili yereli kendi ortamlarının bir parçası olarak miras alırlar. Bu yazılımlar yerel seçimine riayet edecek şekilde yazıldıklarında kullanıcı tarafından tercih edilen uzlaşımlara uyacaklardır.

1. Yerelin Etkisi

Her yerel çeşitli amaçlara yönelik uzlaşımlar içerir:

- Geçerli çokbaytlı karakter dizilimleri ve bunların yorumlanması (*Karakter Kümeleriyle Çalışma* (sayfa: 126)).
- Yerel karakter kümesindeki karakterlerin sınıflandırılması; alfabetik, büyük ve küçük harfler ile bunlar arasındaki dönüşümlerle ilgili uzlaşımlar (*Karakterle Çalışma* (sayfa: 82)).
- Yerel dil ve karakter kümesi için karşılaştırma dizilimi (*Dizgeleri Yerele Özgü Karşılaştırma İşlevleri* (sayfa: 107)).
- Sayıların ve parasal büyüklüklerin biçimlenmesi (*Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169)).
- Tarih ve zamanın biçimlenmesi (*Zaman Değerlerinin Biçimlenmesi* (sayfa: 550)).
- Çıktı ve hata iletileri için kullanılacak dil (*İleti Çevirileri* (sayfa: 181)).
- Evet ve hayır yanıtları verilecek sorularda kullanılacak dil (*Evet/Hayır Yanıtları* (sayfa: 179)).
- Daha karmaşık kullanıcı girdilerinde kullanılacak dil (C kütüphanesi bunun gerçekleştirilmesinde şimdilik size yardımcı olamaz).

Belirtilen yerele uyarlanma ile ilgili bazı şeyler kütüphane yordamlarınca kendiliğinden gerçekleştirilir. Örneğin, yazılımınızda metin karşılaştırmalarının seçili yerele göre yapılması gerekliyse dizgeleri karşılaştırmak için **`strcoll`** veya **`strxfrm`** işlevini kullanılmalıdır.

Bazı şeyler de kütüphanenin kapsamı dışında bırakılmıştır. Örneğin, kütüphane, yazılımınızın çıktıladığı iletileri kendiliğinden çeviremez. Bunu yapmanın tek yolu çıktının kullanıcının diline çevrilmesinin az ya da çok elle yapılmasıdır. C kütüphanesi farklı dillerdeki çevirilerin çıktıya uygulanabilmesini kolaylaştıran işlevlere sahiptir.

Bu kısımda kullanımdaki yerelin değiştirilebilmesi için kullanılan mekanizmalardan bahsedilecektir. Yerelin bazı kütüphane işlevlerine etkilerine yeri geldikçe bu işlevlerin açıklamalarında ayrıntılı olarak yer verilmiştir.

2. Yerelin Seçimi

Kullanıcının yerel seçimini belirtmesinin en basit yolu **LANG** ortam değişkeninde bunu belirtmesidir. Bu değişken tüm amaçlar için tek bir yerel belirtir. Örneğin, kullanıcı İspanya'nın çoğunluğunun standart uzlaşımlarını kapsayan **espana-castellano** isimli bir varsayımsal yereli belirtebilirdi.

Desteklenen yereller kullandığınız işletim sistemine bağlıdır ve bu sistemdeki isimlerle seçim yapılır. Biz bir tek **C** veya **POSIX** olarak adlandırılan standart yerel dışında hangi yerellerin bulunacağına ilişkin herhangi bir taahhüde giremeyiz. Yerellerin nasıl oluşturulacağına daha sonra değineceğiz.

Ayrıca, bir kullanıcı farklı amaçlar için farklı yereller belirtebilme (çok sayıda yerelin bir karışımı olarak) seçeneğine de sahiptir.

Örneğin, kullanıcı İspanya'da çalışan, para birimi olarak dolar kullanan ve İspanyolca konuşan bir amerikalı olabilirdi. Bu kullanıcı paraların biçimlenmesi için **usa-english** yerelini diğer tüm amaçlar için **espana-castellano** yerelini belirtebilirdi.

espana-castellano ve **usa-english** yerellerinin her ikisi de tüm yereller gibi her amaca uygun uzlaşımları içerir. Bu bakımdan, kullanıcı kendi amaçlarına uygun olan yerelleri karışık olarak seçebilir.

3. Yerellerin Etkilediği Eylemlerin Sınıflandırılması

Yereller, kullanım amaçlarına göre ***kategoriler*** halinde gruplanmıştır. Böylece kullanıcı veya yazılım yereli bu kategorilere bağlı olarak seçebilir. Aşağıda bu kategoriler listelenmiştir. Her kategori ismi kullanıcının bir değer atayabileceği bir ortam değişkeni ismi ve **setlocale** işlevinde bir makro ismi olarak kullanılabilir.

LC_COLLATE
: Bu kategori dizgelerin karşılaştırılmasında kullanılan uzlaşımları içerir. **strcoll** ve **strxfrm** işlevleri bu amaçla kullanılır; bkz. *Dizgeleri Yerele Özgü Karşılaştırma İşlevleri* (sayfa: 107).

LC_CTYPE
: Bu kategori çokbaytlı ve geniş karakterlerin sınıflandırılması ve dönüşümleri ile ilgili uzlaşımları içerir; bkz. *Karakterle Çalışma* (sayfa: 82) ve *Karakter Kümeleriyle Çalışma* (sayfa: 126).

LC_MONETARY
: Bu kategori parasal gösterimlerin biçimlenmesi ile ilgili uzlaşımı içerir; bkz. *Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169).

LC_NUMERIC
: Bu kategori parasal değil, sayısal gösterimlerin biçimlenmesi ile ilgili uzlaşımı içerir; bkz. *Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169).

LC_TIME
: Bu kategori tarih ve zaman gösterimlerin biçimlenmesi ile ilgili uzlaşımı içerir; bkz. *Zaman Değerlerinin Biçimlenmesi* (sayfa: 550).

LC_MESSAGES
: Bu kategori kullanıcı arayüzünü oluşturan iletilerin çevirilerine ve düzenli ifadelere uygulanacak dilin seçimi için kullanılır. Bkz. *İleti Çevirilerinde Uniforum Yaklaşımı* (sayfa: 189) ve *X/Open İleti Kataloglarının İşlenmesi* (sayfa: 181)

LC_ALL

Bu bir ortam değişkeni değildir; sadece **setlocale** işlevinde tüm uzlaşımlar için tek bir yerelin belirtilmesi amacıyla kullanılabilen bir makrodur. Bu makroya atanan değer, tüm **LC_*** ortam değişkenleri ile **LANG** ortam değişkenine atanmış olur.

LANG
: Bu ortam değişkeni tanımlıysa, yukarıda belirtilen ortam değişkenleri ile sonradan değiştirilmedikçe tüm uzlaşımlar için kullanılacak yereli belirtir.

İleti çevirileri ile ilgili işlevler geliştirilirken yukarıdaki değişkenlerce sağlanan işlevselliğin yetersizliği hissedildi. Örneğin, birden fazla yerel ismi belirtilebilmeliydi. İngilizceyi almancadan daha iyi konuşan bir isviçreli ve yazılımların öntanımlı olarak iletileri ingilizce çıktıladığını varsayalım. İleti çıktılama dili olarak, İlk seçim isviçrece, ikinci almanca, bunların başarısızlığı halinde ingilizce belirtilebilmesi mümkün olmalıydı. Bu **LANGUAGE** değişkeni ile mümkündür. Bu GNU oluşumu ile ilgili daha fazla bilgi edinmek için *gettext kullanan yazılımların kullanımı* (sayfa: 199) bölümüne bakınız.

4. Yazılımlarda Yerelin Belirtilmesi

Bir C yazılımı yerele ilişkin ortam değişkenlerini başlatıldığında miras alır. Bu işlem kendiliğinden gerçekleşir. Ancak bu değişkenler işlevlerde kullanılan yereli belirtmezler. Çünkü ISO C tüm yazılımların öntanımlı olarak standart **C** yereli ile başlatılması gerektiğinden bahseder. Ortam tarafından belirtilen yerelleri kullanmak için şöyle bir **setlocale** çağrısı yapmanız gerekir:

```
setlocale (LC_ALL, "");
```

Bu çağrı ilgili ortam değişkenleri ile kullanıcı tarafından belirtilen uzlaşımların kütüphane tarafından gözönüne alınmasını sağlar.

setlocale işlevini belli bir kategoriyi ya da genel amaçlı olarak belli bir yereli belirtmek için de kullanabilirsiniz.

Bu bölümde bahsedilen tüm semboller `locale.h` başlık dosyasında tanımlanmıştır.

```
char *setlocale(int          kategori,                                    işlev
                const char *yerel)
```

setlocale işlevi yerel *kategori* katgorisini *yerel* yereline ayarlar. Sistem tarafından desteklenen tüm yerellerin listesini kabukta

```
locale -a
```

komutunu vererek alabilirsiniz.

kategori için **LC_ALL** belirtilmişse, tüm uzlaşımlar için tek bir yerelin belirtilmesini sağlar. Diğer *kategori* değerleri tek bir amaca uygun uzlaşımı belirtir (bkz. *Yerellerin Etkilediği Eylemlerin Sınıflandırılması* (sayfa: 165)).

Bu işlevi ayrıca, *yerel* argümanında boş gösterici aktararak belirtilen kategoriye atanmış değeri öğrenmek için de kullanabilirsiniz. Bu durumda **setlocale** işlevi, *kategori* kategorisi için seçilmiş olan yerel ismini içeren bir dizge döndürür.

setlocale işlevi ile döndürülen bu dizge işlevin sonraki çağrıları ile değiştirilen yerel kategorisinin bu ilk duruma getirilmesinde kullanmak üzere saklanabilir (*Kopyalama ve Birleştirme* (sayfa: 94)). (Kütüphanede kendiliğinden bir **setlocale** çağrısının asla yapılmayacağı garanti edilmiştir.)

setlocale ile döndürülen dizgede değişiklik yapmamalısınız. İlk duruma getirmek için yapacağınız çağrıda kullanacağınız dizge işlev tarafından döndürülen dizgenin aynısı olmak zorundadır. Ayrıca bu işlemi yaparken yerel dizgesinin alındığı kategoriye atanmasına da dikkat etmelisiniz.

Bu sorguyu **LC_ALL** kategorisi için yaparsanız, dönen değer, tüm kategoriler için seçilmiş yerellerin birleşimi olacaktır. Bu durumda dönen değer tek bir yerel ismi içermeyebilir. Aslında değerin nasıl görüneceği ile ilgili bir varsayım yapmadık. Ancak işlevin sonraki çağrısında **LC_ALL** makrosu için aynı değeri aktarırsanız, aynı yerel birleşimi elde edilecektir.

Dönen dizgenin sonradan kullanmak istediğinizde aynı kodlamada kalmasını istiyorsanız dizgenin bir kopyasını saklamalısınız. Dönen göstericinin daima geçerli kalacağı garanti edilmemiştir.

yerel argümanı bir boş gösterici değilse, **setlocale** tarafından döndürülen dizge kategoriye atanan yeni yerele ilişkin dizge olacaktır.

yerel argümanında bir boş dizge belirtirseniz, ilgili ortam değişkeni okunacak ve değeri belirtilen *kategori* kategoroisine atanacaktır.

yerel argümanına boş olmayan bir dizge belirtirseniz, mümkünse bu ismin yereli kullanılacaktır.

Geçersiz bir yerel ismi belirtirseniz, işlev bir boş gösterici döndürür ve o anki yereli değiştirmez.

Bu örnekte **setlocale** işlevinin başka bir yereli geçici olarak etkin kılmak için kullanılması gösterilmiştir:

```
#include <stddef.h>
#include <locale.h>
#include <stdlib.h>
#include <string.h>

void
with_other_locale (char *new_locale,
                   void (*subroutine) (int),
                   int argument)
{
  char *old_locale, *saved_locale;

  /* O anki yerelin ismini alalım.  */
  old_locale = setlocale (LC_ALL, NULL);

  /* setlocale çağrılarıyla bozulmadan önce ismi kopyalayalım. */
  saved_locale = strdup (old_locale);
  if (saved_locale == NULL)
    fatal ("Bellek yetersiz");

  /* Şimdi yereli değiştirelim ve onunla ilgili işlemleri yapalım. */
  setlocale (LC_ALL, new_locale);
  (*subroutine) (argument);

  /* Yereli eski değerine getirelim. */
  setlocale (LC_ALL, saved_locale);
  free (saved_locale);
}
```

Taşınabilirlik Bilgisi

Bazı C sistemleri ek yerel kategorileri tanımlayabilir; kütüphanenin ileri sürümlerinde bu yapılabilir. Dolayısıyla **LC_** ile başlayan bu ek sembollerin `locale.h` başlık dosyasında tanımlanabileceği varsayılmalıdır.

5. Standart Yereller

Tüm işletim sistemlerinde bulabileceğiniz standart yerel isimleri sadece üç tanedir:

`"C"`
: Bu standart C yerelidir. Öznitelikleri ve davranışları ISO C standardı ile belirlenmiştir. Yazılımınız çalıştırıldığında, kendi içinde öntanımlı olarak bu yereli kullanır.

`"POSIX"`
: Bu standart POSIX yerelidir. Şimdilik standart C yereli için bir takma addır.

`""`
: Boş isim yerel seçiminin ortam değişkenlerine bakarak yapılacağını belirtir. Bkz. *Yerellerin Etkilediği Eylemlerin Sınıflandırılması* (sayfa: 165).

İsimli yerellerin tanımlanması ve kurulması normalde sistem yöneticisinin (veya GNU C kütüphanesini kuran kişinin) işidir. Bundan başka her kullanıcının kendine özgü yerellerini belirtmesi de mümkündür. Tüm bunlar araçları tanıtırken daha sonra açıklanacaktır.

Yazılımınız **`C`** yerelinden farklı birşeylere ihtiyaç duyarsa, standart olabileceği tartışmalı bir isim yerine, taşınabilirlik açısından kullanıcının ortam değişkenleri ile belirttiği yerelin kullanılması daha iyidir. Farklı makinelerin aynı yerel için farklı yerel isimleri içerebileceğini aklınızdan çıkarmayın.

6. Yerel Bilgisine Erişim

Yerel bilgisine erişmenin çeşitli yolları vardır. En basiti C kütüphanesinin kendisinden istemektir. Bu kütüphanedeki çeşitli işlevlerle yerel bilgisine doğrudan erişilebilir ve o an seçili olan yerelle sağlanan bilgiler kullanılabilir. Burada yerel modelinin normalde nasıl anlamlandırıldığından bahsedilecektir.

Bir örnek olarak tarih ve zaman gösterimlerini biçimlendiren (*Zaman Değerlerinin Biçimlenmesi* (sayfa: 550)) **`strftime`** işlevini ele alalım. **`LC_TIME`** kategorisinin standart içeriğinin bir kısmı ay isimlerinden oluşur. Yazılımcı her ay ismi için çevrilmek üzere bir ay isimleri listesi yapmak yerine bu işi zaten yapan **`strftime`** işlevini kullanabilir. Biçim dizgesindeki **`%A`**, **`LC_TIME`** tarafından seçilen yerele özgü gün ismi ile değiştirilir. Bu basit bir örnekti, bu tür işleri yapan başka işlevlerde benzer bir yöntem kullanır.

Fakat çoğunlukla, bir görevi kendiliğinden yerine getirecek bir işlev bulunmaz. Bu durmlarda yerel bilgisine doğrudan erişebilmek önem kazanır. Bunun için C kütüphanesi iki işlev içerir: **`localeconv`** ve **`nl_langinfo`**. İlki ISO C standardının bir parçasıdır ve taşınabilirdir, ancak kafa karıştırıcı bir arayüzü vardır. İkincisi ise Unix arayüzünün bir parçasıdır ve Unix standardına uyumlu sistemlerde taşınabilirdir.

6.1. `localeconv`: Taşınabilirdir ama ...

ISO C topluluğu **`localeconv`** işlevini **`setlocale`** işlevi ile birlikte anar. **`localeconv`** işlevi, biçare tasarımın anaparçasıdır. Genişletilebilir olmaması ve **`LC_MONETARY`** ve **`LC_NUMERIC`** kategorileri ile ilgili bilgileri sağlamanın dışında bir bilginin gerekmediği durumlarda kullanımı tercih edilir. Buna rağmen, çok taşınabilir olduğundan sadece bu durumda özellikle kullanılmalıdır. **`strfmon`** işlevi seçili yerele göre bu bilgiyi parasal gösterimleri biçimlemekte kullanılır.

```
struct lconv *localeconv(void)                                    işlev
```

`localeconv` işlevi sayısal ve parasal değerlerin o anki yerele göre biçimlenmesi için gereken bilgiyi içeren elemanlardan oluşan bir yapının göstericisi ile döner.

Yapıda ve yapının içeriğinde değişiklik yapmamalısınız. Yapı **`localeconv`** veya **`setlocale`** işlevlerinin sonraki çağrıları ile tekrar düzeltilir. Fakat bunlar dışında kütüphanede bu yapıyı düzelten başka işlev yoktur.

```
struct lconv                                                      veri türü
```

Bu, **localeconv** işlevinin dönüş değerinin türüdür. Elemanları bundan sonraki alt bölümlerde açıklanmıştır.

struct lconv yapısının **char** türünde bir elemanı varsa ve değeri **CHAR_MAX** ise, bu, yerelin bu parametre ile ilgili bir değer içermediği anlamına gelir.

6.1.1. Soysal Sayısal Biçimleme Parametreleri

Bunlar **struct lconv** yapısının standart elemanlarıdır; başkaları da olabilir:

`char` ***decimal_point**
`char` ***mon_decimal_point**
Bunlar sayısal ve parasal gösterimler için ondalık ayraçlardır. **C** yerelinde **decimal_point** değeri **"."** iken **mon_decimal_point** değeri **""**'dir.

`char` ***thousands_sep**
`char` ***mon_thousands_sep**
Bunlar sayısal ve parasal gösterimlerde ondalık ayracın solundaki sayının genellikle binlik gruplara ayrılmasında kullanılan ayraçlardır. **C** yerelinde her iki üyeninde değeri **""** (boş dizge) dir.

`char` ***grouping**
`char` ***mon_grouping**
Bunlar sayısal ve parasal gösterimlerde ondalık ayracın solundaki sayının kaç rakamlık gruplara ayrılacağını belirtmekte kullanılır. **grouping** sayısal, **mon_grouping** ise parasal gösterimlere uygulanır.

Bu dizgelerin içindeki ayrı ayrı **char** türünde tanımlanabilecek her sayı **char** türünde bir tamsayı olarak yorumlanır. Dizgenin içindeki her sayı (soldan sağa) grupların (ondalık ayraçtan itibaren sağdan sola) rakam sayısını verir. Son sayı **0** ise kalan gruplar önceki sayıya göre gruplanır; **CHAR_MAX** ise kalan sayıya gruplama uygulanmaz, başka bir deyişle kalan sayı ayraçsız olabildiğince geniş bir grup olur.

Örneğin, **grouping** için **"\04\03\02"** değeri belirtilmişse, **123456787654321** sayısı **12**, **34**, **56**, **78**, **765**, **4321** şeklinde gruplanır. Yani sondaki 4'lü bir grup, ondan önceki 3'lü bir grup, ondan öncekiler de 2'li gruplar halinde ayrılır. Gruplama ayracı olarak **,** belirilmişse sayı **12,34,56,78,765,4321** olarak basılır.

"\03" değeri tekrarlanan 3'lü gruplarla gruplama yapılacağını belirtir. Normalde ABD'de böyle bir gruplama kullanılır.

Standart **C** yerelinde **grouping** ve **mon_grouping** için her ikisine de gruplama yapılmayacağı anlamına gelen **""** değeri belirtilmiştir.

`char` **int_frac_digits**
`char` **frac_digits**
Bunlar parasal gösterimin sırayla uluslararası ve yerel biçimlendirmesinde ondalık ayracın sağında kaç rakamın gösterileceğini belirtmekte kullanılır. (Çoğunlukla ikisine de aynı değer atanır.)

Standart **C** yerelinde, her iki üye de "belirtilmemiş" anlamına gelen **CHAR_MAX** değerini içerir. ISO C standardı bu değere rastlandığında ne yapılacağını belirlememiştir; bizim tavsiyemiz ondalık ayraçtan sonra hiçbir rakam gösterilmeyeceği şeklinde yorumlanmasıdır. (C yerelinin **mon_decimal_point** değeri olarak boş dizge belirttiğini gözönüne alırsanız bir rakamın gösterilmesi sayının değerinin değişmesi demektir!)

6.1.2. Para sembolünün Basılması

struct lconv yapısının para sembolü ile ilgili üyeleri para birimini tanımlayan sembolü basmak için kullanılır. Amerikan doları için uluslararası ve yerel sembol aynıdır ve bu sembol $'dır.

Her ülkenin iki standart para birimi sembolü vardır. ***Yerel para sembolü*** ülke içinde, ***uluslararası para sembolü*** ise uluslararası kullanımda yerel sembolün kullanımının sorun yaratmaması için kullanılan para sembolüdür.

Örneğin, birçok ülke kendi para birimi olarak dolar kullanır. Yerel kullanımda sorun çıkarmayan bu sembol, uluslararası kullanımda amerikan dolarından ve diğer dolarlardan ayrılması gerekir. Kanada ve Avustralya için böyledir.

`char *currency_symbol`
: Yerelin yerel para birimi sembolüdür.

  Standart **C** yerelinde, bu üyenin değeri "belirtilmemiş" anlamına gelen **""** boş dizgesidir. ISO C standardı bu değerin nasıl yorunlanacağını belirlememiştir; bizim tavsiyemiz boş dizge olarak yorumlanmasıdır.

`char *int_curr_symbol`
: Yerelin uluslararası para birimi sembolüdür.

  int_curr_symbol değeri normalde uluslarası standart tarafından belirlenen üç harfli bir kısaltmadır [ISO 4217 Codes for the Representation of Currency and Funds] (Para ve Fonların gösterimi için ISO 4217 kodları) ve bu kısaltmadan sonra tek karakterlik bir ayraç gelir (çoğunlukla boşluk).

  Standart **C** yerelinde, bu üyenin değeri "belirtilmemiş" anlamına gelen **""** boş dizgesidir; bizim tavsiyemiz boş dizge olarak yorumlanmasıdır.

`char p_cs_precedes`
`char n_cs_precedes`
`char int_p_cs_precedes`
`char int_n_cs_precedes`
: Bu üyelerin değeri **1** ise, para sembolleri para değerinin önüne, **0** ise para değerinin sonuna konur. **p_cs_precedes** ve **int_p_cs_precedes** değerleri pozitif miktarlara, **n_cs_precedes** ve **int_n_cs_precedes** değerleri ise negatif miktarlara uygulanır.

  Standart **C** yerelinde bu üyelerin hepsine "belirtilmemiş" anlamına gelen **CHAR_MAX** değeri atanmıştır. ISO C standardı bu değerin nasıl yorumlanacağını belirtmemiştir. Bizim tavsiyemiz çoğu ülkede para sembolünün para miktarının önüne konulmasından hareketle para sembolünün önce basılacağı biçimde yorumlanmasıdır. Başka yönden bakarsak değerin sıfırdan farklı bir değer olarak (1 olarak) yorumlanması önerilir.

  İsimleri **int_** ile başlayan üyeler **int_curr_symbol**'e, diğerleri **currency_symbol**'e uygulanır.

`char p_sep_by_space`
`char n_sep_by_space`
`char int_p_sep_by_space`
`char int_n_sep_by_space`
: Bu üyelerin değeri **1** ise, para sembolü ile para miktarı arasına bir boşluk konur, **0** ise boşluk konmaz. **p_sep_by_space** ve **int_p_sep_by_space** üyeleri pozitif (veya sıfır) para miktarlarına, **n_sep_by_space** ve **int_n_sep_by_space** üyeleri ise negatif para miktarlarına uygulanır.

  Standart **C** yerelinde, bu üyelerin hepsine "belirtilmemiş" anlamına gelen **CHAR_MAX** değeri atanmıştır. ISO C standardı bu değerin nasıl yorumlanacağını belirtmemiştir. Tavsiyemiz değerin sıfırdan farklı bir değer olarak (1 olarak) yorumlanmasıdır (yani boşluk konması).

İsimleri **int_** ile başlayan üyeler **int_curr_symbol**'e, diğerleri **currency_symbol**'e uygulanır. **int_curr_symbol** için özel bir durum vardır. Standart uluslararası değerler sembolden sonra bir boşluk içerir (Para sembolünün para miktarından önce kullanılması ve arada boşluk bırakılması için). Bu durumda bu boşluğun basılmasının önlenmesine (bilhassa para sembolünün para miktarından sonra basıldığı durumda) dikkat edilmelidir.

6.1.3. Para Miktarına İşaret Basılması

struct lconv yapısının bu üyelerinde parasal gösterimde (varsa) işaretin nasıl basılacağını belirtilir.

`char *`**positive_sign**
`char *`**negative_sign**
Bunlar pozitif (veya sıfır) ve negatif para miktarlarını belirtmekte kullanılacak işaretleri içeren dizgelerdir.

Standart **C** yerelinde, bu üyelerin her ikisine de "belirtilmemiş" anlamına gelen **""** boş dizgesi atanmıştır.

ISO C standardı bu değerlerin nasıl yorumlanacağını belirtmemiştir. Tavsiyemiz pozitif işareti **positive_sign** boş dizge olarak belirtilse dahi bulduğunuz gibi, negatif işareti de **negative_sign** boş dizge olarak belirtilse bile **–** olarak basılmasıdır.

`char` **p_sign_posn**
`char` **n_sign_posn**
`char` **int_p_sign_posn**
`char` **int_n_sign_posn**
Bu üyeler pozitif ve negatif miktarların işaretlerinin yerini belirtmede kullanılan küçük tamsayılar içerir. Olası değerler şunlardır:

`0`
Para sembolü ve miktar parantez içine alınır.

`1`
İşaret, para sembolü ve para miktarından önce basılır.

`2`
İşaret, para sembolü ve para miktarından sonra basılır.

`3`
İşaret, para sembolünden hemen önce basılır.

`4`
İşaret, para sembolünden hemen sonra basılır.

`CHAR_MAX`
"Belirsiz". Standart **C** yerelinde her iki üye de bu değere sahiptir.

ISO standardı değerin **CHAR_MAX** olması halinde ne yapılacağını belirlememiştir. Tavsiyemiz, işaretin para sembolünden sonra basılmasıdır.

İsimleri **int_** ile başlayan üyeler **int_curr_symbol** ile diğerleri **currency_symbol** ile uygulanır.

6.2. Yerel Verisine Noktasal Erişim

X/Open Taşınabilirlik Kılavuzu yazılırken yazarlar yerele özgü bilgilere erişmek anlamında **localeconv** işlevinin yetersizliğinde anlaştılar. Yerelde kullanılabilir bilgilere (daha sonra POSIX.1 standardında belirtildiği gibi) daha çok yoldan erişmek gerekir. **nl_langinfo** işlevi bunun için tasarlanmıştır.

`char *`**nl_langinfo**`(nl_item` *öğe*`)` işlev

nl_langinfo işlevi yerel kategorilerindeki her elemana tek tek erişmek için kullanılabilir. Tüm bilgiyi döndüren **localeconv** işlevinin tersine, **nl_langinfo** işlevi çağrıcının istediği bilgiyi belirtebilmesini sağlar. Bu çok hızlı yapıldığından işlevin defalarca çağrılması bir soruna yol açmaz.

İkinci bir getirisi de sayısal ve parasal gösterim bilgilerine ek olarak **LC_TIME** ve **LC_MESSAGES** kategorileriyle ilgili bilgilerinde istenebilmesidir.

nl_type türü `nl_types.h` dosyasında tanımlanmıştır. *öğe* argümanı **langinfo.h** dosyasında tanımlı sayısal değerlerden biri olmalıdır. X/Open standardı şu değerleri tanımlar:

CODESET
: **nl_langinfo** seçili yerelin karakter kodlaması için kullanılan karakter kümesinin ismini içeren bir dizge ile döner.

ABDAY_1
ABDAY_2
ABDAY_3
ABDAY_4
ABDAY_5
ABDAY_6
ABDAY_7
: **nl_langinfo** kısaltılmış gün ismi ile döner. **ABDAY_1** Pazar gününe karşılıktır.

DAY_1
DAY_2
DAY_3
DAY_4
DAY_5
DAY_6
DAY_7
: **ABDAY_1** ve benzerleri gibidir, farklı olarak kısaltılmamış gün ismi ile döner.

ABMON_1
ABMON_2
ABMON_3
ABMON_4
ABMON_5
ABMON_6
ABMON_7
ABMON_8
ABMON_9
ABMON_10
ABMON_11
ABMON_12
: İşlev kısaltılmış ay ismi ile döner. **ABMON_1** Ocak ayına karşılıktır.

MON_1
MON_2
MON_3
MON_4
MON_5
MON_6
MON_7

MON_8
MON_9
MON_10
MON_11
MON_12
: **ABMON_1** ve benzerleri gibidir, farklı olarak kısaltılmamış ay ismi ile döner.

AM_STR
PM_STR
: İşlev, 12 saatlik zaman gösteriminde kullanılan (sırayla) öğleden önce ve öğleden sonra kısaltmaları (öö/ös) olarak belirtilmiş dizge ile döner.

  12 saatlik zaman gösterimi kullanılmayan yerellerde bu dizgeler boş olabilir, bu takdirde 12 saatlik gösterim seçilse bile bu kısaltmalar gösterilmeyecektir.

D_T_FMT
: Yerele özgü tarih ve zaman gösterimi olarak **strftime** işlevinde kullanılabilecek biçim dizgesi ile döner.

D_FMT
: Yerele özgü tarih gösterimi olarak **strftime** işlevinde kullanılabilecek biçim dizgesi ile döner.

T_FMT
: Yerele özgü zaman gösterimi olarak **strftime** işlevinde kullanılabilecek biçim dizgesi ile döner.

T_FMT_AMPM
: Yerele özgü öö/ös gösterimi olarak **strftime** işlevinde kullanılabilecek biçim dizgesi ile döner.

  öö/ös biçimi seçilmiş yerelde belirlenmemişse dönen değer **T_FMT** için dönen değer ile aynı olabilir.

ERA
: İşlev seçili yerelde kullanılan çağ ile döner.

  Çoğu yerelde bu değer atanmamıştır. Bu değerin atandığı yarellerden biri olarak japonca gösterilebilir. Japonya'da geleneksel tarih gösterimleri imparatorun saltanat dönemine karşı düşen çağ ismini de içerir.

  Normalde bu değeri doğrudan kullanmak gerekmez. **strftime** işlevinin biçim dizgesinde **E** değiştiricisini belirterek bu bilginin kullanılması sağlanabilir. Dönen dizgenin biçimi belirlenmemiştir, bu bakımdan başka sistemlerde aynı dizgenin elde edileceği varsayımında bulunmayın.

ERA_YEAR
: Yerelde belirtilen çağa göre belirtilen yol döner. **ERA** gibi bu değeri de doğrudan kullanmak gerekmez.

ERA_D_T_FMT
: Yerele özgü çağa göre tarih ve zaman gösterimi olarak **strftime** işlevinde kullanılabilecek biçim dizgesi ile döner.

ERA_D_FMT
: Yerele özgü çağa göre tarih gösterimi olarak **strftime** işlevinde kullanılabilecek biçim dizgesi ile döner.

ERA_T_FMT
: Yerele özgü çağa göre zaman gösterimi olarak **strftime** işlevinde kullanılabilecek biçim dizgesi ile döner.

ALT_DIGITS
: 0–99 arasındaki sayıların gösterimi için kullanılan 100 değerle döner. Bu değer de **ERA** gibi doğrudan kullanmak için tasarlanmamıştır. **strftime** işlevinde **O** değiştiricisi belirtilerek kullanılır. Bu takdirde biçim dizgesindeki saat, dakika, saniye, ayın günü, aylar ve haftalar gibi sayısal değerler buna göre gösterilir.

INT_CURR_SYMBOL
: **localeconv** işlevi ile dönen **struct lconv** yapısının **int_curr_symbol** elemanındaki değer ile döner.

CURRENCY_SYMBOL
CRNCYSTR
: **localeconv** işlevi ile dönen **struct lconv** yapısının **currency_symbol** elemanındaki değer ile döner.

  CRNCYSTR, Unix98 tarafından hala ihtiyaç duyulan artık kullanılmayan bir takma addır.

MON_DECIMAL_POINT
: **localeconv** işlevi ile dönen **struct lconv** yapısının **mon_decimal_point** elemanındaki değer ile döner.

MON_THOUSANDS_SEP
: **localeconv** işlevi ile dönen **struct lconv** yapısının **mon_thousands_sep** elemanındaki değer ile döner.

MON_GROUPING
: **localeconv** işlevi ile dönen **struct lconv** yapısının **mon_grouping** elemanındaki değer ile döner.

POSITIVE_SIGN
: **localeconv** işlevi ile dönen **struct lconv** yapısının **positive_sign** elemanındaki değer ile döner.

NEGATIVE_SIGN
: **localeconv** işlevi ile dönen **struct lconv** yapısının **negative_sign** elemanındaki değer ile döner.

INT_FRAC_DIGITS
: **localeconv** işlevi ile dönen **struct lconv** yapısının **int_frac_digits** elemanındaki değer ile döner.

FRAC_DIGITS
: **localeconv** işlevi ile dönen **struct lconv** yapısının **frac_digits** elemanındaki değer ile döner.

P_CS_PRECEDES
: **localeconv** işlevi ile dönen **struct lconv** yapısının **p_cs_precedes** elemanındaki değer ile döner.

P_SEP_BY_SPACE
: **localeconv** işlevi ile dönen **struct lconv** yapısının **p_sep_by_space** elemanındaki değer ile döner.

N_CS_PRECEDES

localeconv işlevi ile dönen **struct lconv** yapısının **n_cs_precedes** elemanındaki değer ile döner.

N_SEP_BY_SPACE
: **localeconv** işlevi ile dönen **struct lconv** yapısının **n_sep_by_space** elemanındaki değer ile döner.

P_SIGN_POSN
: **localeconv** işlevi ile dönen **struct lconv** yapısının **p_sign_posn** elemanındaki değer ile döner.

N_SIGN_POSN
: **localeconv** işlevi ile dönen **struct lconv** yapısının **n_sign_posn** elemanındaki değer ile döner.

INT_P_CS_PRECEDES
: **localeconv** işlevi ile dönen **struct lconv** yapısının **int_p_cs_precedes** elemanındaki değer ile döner.

INT_P_SEP_BY_SPACE
: **localeconv** işlevi ile dönen **struct lconv** yapısının **int_p_sep_by_space** elemanındaki değer ile döner.

INT_N_CS_PRECEDES
: **localeconv** işlevi ile dönen **struct lconv** yapısının **int_n_cs_precedes** elemanındaki değer ile döner.

INT_N_SEP_BY_SPACE
: **localeconv** işlevi ile dönen **struct lconv** yapısının **int_n_sep_by_space** elemanındaki değer ile döner.

INT_P_SIGN_POSN
: **localeconv** işlevi ile dönen **struct lconv** yapısının **int_p_sign_posn** elemanındaki değer ile döner.

INT_N_SIGN_POSN
: **localeconv** işlevi ile dönen **struct lconv** yapısının **int_n_sign_posn** elemanındaki değer ile döner.

DECIMAL_POINT
RADIXCHAR
: **localeconv** işlevi ile dönen **struct lconv** yapısının **decimal_point** elemanındaki değer ile döner.

  RADIXCHAR Unix98 tarafından hala ihtiyaç duyulan artık kullanılmayan bir takma addır.

THOUSANDS_SEP
THOUSEP
: **localeconv** işlevi ile dönen **struct lconv** yapısının **thousands_sep** elemanındaki değer ile döner.

  THOUSEP Unix98 tarafından hala ihtiyaç duyulan artık kullanılmayan bir takma addır.

GROUPING

localeconv işlevi ile dönen **struct lconv** yapısının **grouping** elemanındaki değer ile döner.

YESEXPR
: Dönen değer, evet/hayır sorusuna olumlu yanıt olarak **regex** işlevinde kullanılabilen bir düzenli ifadedir. GNU C kütüphanesi uygulamalarda bunu daha da kolaylaştıran **rpmatch** işlevini içerir.

NOEXPR
: Dönen değer, evet/hayır sorusuna olumsuz yanıt olarak **regex** işlevinde kullanılabilen bir düzenli ifadedir.

YESSTR
: Evet/hayır sorusuna olumlu yanıt olarak yerel özgü dizge ile döner.

  İleti çevirilerinde çok özel bir durum olduğundan bu değer artık kullanılmamaktadır. İleti çeviri işlevleri ile elde edilmesi daha iyidir (bkz. *İleti Çevirileri* (sayfa: 181)).

  Bu sembol artık kullanılmamaktadır. Onun yerine ileti çevirileri kullanılmalıdır.

NOSTR
: Evet/hayır sorusuna olumsuz yanıt olarak yerel özgü dizge ile döner. **YESSTR** için bahsedilen herşey bunun için de geçerlidir.

  Bu sembol artık kullanılmamaktadır. Onun yerine ileti çevirileri kullanılmalıdır.

langinfo.h dosyası bunlardan başka semboller de içerir ama bunların pek kullanım alanı yoktur. Onların kullanılması taşınabilirliği ortadan kaldırır. Bu nedenle kullanılmaları tavsiye edilmez.

Her geçerli argüman için dönen değer tüm durumlarda kullanılabilir (öö/ös biçim kodlarının olası olağandışılığı dahil). Eğer kullanıcı ilgili kategori için bir yerel seçimi yapmamışsa **nl_langinfo** işlevi **"C"** yerelindeki bilgi ile döner. Bundan dolayı işlevi aşağıdaki örnekte gösterildiği gibi kullanmak mümkündür.

öğe argümanı geçersizse, boş dizge içeren bir gösterici döner.

Bir **nl_langinfo** kullanım örneği olarak bir işlev tarih ve zamanı yerele özgü biçimde basmak için kullanılmıştır. Burada, dikkat ederseniz, **strftime** işlevi yerel bilgisini zaten dahili olarak kullandığından biçim dizgesinde yeteri kadar değiştirici kullanılmıştır.

```
size_t
i18n_time_n_data (char *s, size_t len, const struct tm *tp)
{
  return strftime (s, len, "%X %D", tp);
}
```

Biçim ne haftanın günü ne de ay ismi içerir, bu nedenle uluslararası olarak kullanılabilir. Yanlış!. Çıktı **"hh:mm:ss AA/GG/YY"** biçiminde üretilir. Bu biçim sadece ABD'de geçerlidir. Diğer ülkeler farklı biçim kullanır. Bu nedenle işlev şöyle yazılmalıydı:

```
size_t
i18n_time_n_data (char *s, size_t len, const struct tm *tp)
{
  return strftime (s, len, nl_langinfo (D_T_FMT), tp);
}
```

Artık yazılım çalıştırıldığında seçili yerelin tarih ve zaman biçimi kullanılacaktır. Kullanıcı doğru yereli seçmişse tarih ve zaman ile ilgili yanlış anlamalar olmayacakır.

7. Sayıların Biçimlenmesi

Gerek **localeconv** ile dönen yapı ile gerekse **nl_langinfo** işlevine sembol belirterek sayısal ve parasal gösterimleri biçimlemekte kullanılan yerele özgü bilgilerin çeşitli parçalarının elde edilebileceğini görmüştük. Ayrıca temel kuralların oldukça karmaşık olduğunu da görmüştük.

Bu nedenle, X/Open standartları bu tür yerel bilgisini kullanan ve sayıları bu kurallara göre biçimlemeyi kullanıcı açısından kolaylaştıran bir işlev tanımlamıştır.

```
ssize_t strfmon(char        *tampon,                                    işlev
                size_t       azamiboyut,
                const char *biçim,
                …)
```

strfmon işlevi **strftime** işlevine benzer. İşlev bir tampon, tampon boyu ve biçim dizgesi alır ve çıktı biçim dizgesi ile belirtildiği gibi biçimlenerek metin olarak tampona yazılır. İşlev, **strftime** gibi tampona yazılan baytların sayısı ile döner.

İki fark vardır: **strfmon** birden fazla argüman alır ve şüphesiz biçim belirtimi farklıdır. **strftime**'a benzer olarak, biçim dizgesi çıktıda olduğu gibi normal metindir ve biçim belirteçleri **%** işareti ile belirtilir. **printf** işlevinde olduğu gibi, **%** işaretinin ardından, biçim karakterinden önce isteğe bağlı olarak çeşitli seçenekler ve biçimleme bilgileri belirtilebilir:

- **%** işaretinin hemen ardından bu seçeneklerden biri veya birkaçı belirtilebilir:

  =*f*
  : *f* karakteri, sayısal dolgu karakteri olarak bu alanda kullanılacak tek baytlık karakterdir. Bu karakterin öntanımlı değeri boşluk karakteridir. Bu karakterin dolgu karakteri olarak kullanılabilmesi için ondalik ayracın solundaki bölüm için genişlik belirtilmiş olmalıdır. Belirtilen alan genişliği dolgu karakteri sayısı değildir.

  ^
  : Seçili yerelin kurallarına bağlı rakam gruplaması yapılmaz. Öntanımlı olarak yerelde belirtilmiş gruplama yapılır.

  +
  (
  : Bu seçeneklerden sadece biri kullanılabilir. Para miktarında işaretin nasıl belirtileceğini belirlerler. Öntanımlı olarak ve **+** belirtilmişse +/– için yerelde belirtilen kullanılır. **(** belirtilmişse, negatif miktarlar parantez içine alınır. Tam biçim yazılımın çalışması sırasında seçili yerelin **LC_MONETARY** kategorisindeki değerlere göre saptanır.

  !
  : Çıktı para sembolünü içermeyecektir.

  –
  : Çıktı eğer alan genişliğinde doldurma yapılacağı belirtilmemişse sağa değil sola yanaşık biçimlenecektir.

  Belirtimin sonraki parçası isteğe bağlı olan alan genişliğidir. Bir genişlik belirtilmemişse 0 verilmiş kabul edilir. Çıktılama sırasında, işlev önce ne kadar alan gerektiğini saptar. Eğer gerektiği kadar genişlik belirtilmişse çıktı bu alanı kullanır. Aksi takdirde, belirtilen genişlik dolgu karakteri ile doldurularak çıktılama yapılır. **–** iminin varlığına vaya yokluğuna bağlı olarak işaretin bulunduğu tarafın boşluğu saptanır. Varsa, çıktı sola yanaşık yapılarak boşluk sağa eklenir ya da tersi yapılır.

Şimdiye kadar **printf** ve **strftime** biçimlerine benzerliğinden dolayı biçim oldukça bildik göründü. Ancak sonraki iki alan biraz yeni. İlki **#** karakteri ile başlayan bir ondalık sayı dizgesidir. Bu dizge ondalık ayracın solunda kalan rakam (karakter sayısını değil) sayısını belirtir. Bu miktara gruplama karakterleri dahil değildir. Basılacak sayının rakamları bu genişliği dolduramazsa, boş kalan alan **=** imi ile belirtilmişse o karakterle aksi takdirde boşluk karakteri ile doldurulur. Örneğin alan genişliği 6 ve dolgu karakteri ***** olarak verilmişse, 123 sayısı, *****123** olarak biçimlenecektir.

İsteğe bağlı ikinci alan ise **.** karakteri ile başlayan bir ondalık sayı dizgesidir. Bu dizge ondalık ayracın sağındaki rakam sayısını belirtir. Öntanımlı değer yerelin **frac_digits** ve **int_frac_digits** değerleridir (bkz. *Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169)). Eğer belirtilen genişlik bütün haneleri göstermek için yeterli değilse, yuvarlama yapılır. Alan genişliği sıfır olarak belirtilmişse ondalık ayraç basılmaz.

Bir GNU oluşumu olarak GNU C kütüphanesindeki **strfmon** gerçeklemesi bu seçeneklerden sonra isteğe bağlı **L** biçim belirtecini kabul eder. Eğer bu belirteç varsa, belirtilen argüman bir **double** değer değil **long double** değer kabul edilir.

Son eleman bir biçim belirtecidir. Üç belirteç belirtilebilir:

i
: Bir uluslararası parasal değerin biçimlenmesi için yerelin kuralları kullanılır.

n
: Bir ulusal parasal değerin biçimlenmesi için yerelin kuralları kullanılır.

%
: Çıktıya **%** işareti basılır. Sadece **%%** belirtilebilir, bir im, seçenek ya da belirteç belirtilmesine izin verilmemiştir.

printf işlevinin yaptığı gibi, işlev biçim dizgesini soldan sağa okur ve biçim dizgesinden sonra verilmiş olan değerleri bu dizgeye göre biçimlendirir. Değerler **L** belirteci varsa **long double**, aksi takdirde **double** türünde kabul edilir. Sonuç, *tampon* ile gösterilen tamponda en fazla *azamiboyut* karakterlik saklanır.

İşlevin dönüş değeri *tampon*'da saklanan boş karakter sonlandırmalı dizgenin karakter sayısıdır. Eğer *tampon* içinde saklanacak karakterlerin sayısı *azamiboyut*'tan büyükse tamponun yetersiz kaldığını belirtmek için işlev –1 ile döner. Bu durumda **errno** değişkenine **E2BIG** değeri atanır.

Küçük bir kaç örnekle işlevin nasıl çalıştığını gösterebiliriz. Örneklerin ABD yerelinde (**en_US**) çalıştırıldığı varsayılmıştır. En basit biçim:

```
strfmon (buf, 100, "@%n@%n@%n@", 123.45, -567.89, 12345.678);
```

için şu çıktı üretilir:

```
"@$123.45@-$567.89@$12,345.68@"
```

Burada birkaç şeyden bahsedebiliriz. İlki çıktılanan sayılardaki genişlik farklarıdır. Biçim dizgesinde bir genişlik belirtmedik, yani bunun önemi yok. İkincisi, üçüncü sayı binlik gruplama ayracı ile çıktılandı. Binlik gruplama ayracı **en_US** yereli için virgüldür. Bundan başka sayının yuvarlatıldığını görüyoruz. Biz ondalık ayracın sağındaki hane sayısını belirtmedik ama yerel bunu 2 olarak belirlediğinden .678 –> .68 olarak yuvarlandı. Son olarak **i** değil **n** belirttiğimizden ulusal para sembolü görüntülendi.

Bu örnekte çıktının nasıl hizalandığı gösterilmiştir:

```
strfmon (buf, 100, "@%=*11n@%=*11n@%=*11n@", 123.45, -567.89, 12345.678);
```

Çıktısı şöyle olur (@ karakterlerine dikkat):

```
"@    $123.45@   -$567.89@ $12,345.68@"
```

Burada iki şeyden söz edilebilir: İlki, biçim dizgesinde alan genişliği belirtildiğinden ve bu genişlikten daha geniş bir sayı olmadığından tüm alanlar aynı genişlikte çıktılandı (onbir karakter). İkinci önemli nokta ise dolgu karakteri belirtildiği halde çıktıda kullanılmamış olmasıdır. Bunun sebebi, **#** karakteri ile ondalık noktanın solundaki alan genişliğinin verilmemiş olmasıdır; bu nedenle öntanımlı olarak boşluk kullanılmıştır. Aşağıdaki örnekte bu genişlik belirtilmiştir:

```
strfmon (buf, 100, "@%=*11#5n@%=*11#5n@%=*11#5n@",
         123.45, -567.89, 12345.678);
```

Çıktısı:

```
"@ $***123.45@-$***567.89@ $12,456.68@"
```

Görüldüğü üzere para sembolleri hizalandı ve para sembolü ile sayı arasındaki alan dolgu karakteri ile doldurudu. Genişlik 5 seçildiğinden 123.45 için ondalik ayracın solundaki rakam sayısı bu değerden küçük olduğundan (5 e değil 11 karaktere tamamlanacak şekilde) artan alan yıldızlarla doldurulmuştur. Son sayıda ise binler ayracının genişliğe dahil olmadığı kanıtlanmaktadır. Son bir örnekle kalan işlevselliği açıklayalım:

```
strfmon (buf, 100, "@%=0(16#5.3i@%=0(16#5.3i@%=0(16#5.3i@",
         123.45, -567.89, 12345.678);
```

Bu karmaşık biçim dizgesi şu çıktıyı üretir:

```
"@ USD 000123,450 @(USD 000567.890)@ USD 12,345.678 @"
```

Burada en önemli değişiklik negatif sayıları göstermekte kullanılan yöntemdir. Finansal çevrelerde bu çoğunlukla parantezler kullanılarak yapılır ve bu **(** imi ile seçilir. **0** karakterleri burada sayı için anlamlı haneler olmadıklarından bunlar arasında binler ayracı kullanılmamıştır. Biçim belirteci olarak **n** değil, **i** kullandığımızdan uluslararası para sembolü görüntülendi. Bu dört karakterlik (**"USD "**) bir dizgedir. Son nokta ise, ondalik ayracın sağındaki hane sayısı 3 olarak belirtildiğinden ilk iki sayıda sağa birer sıfır eklenirken, 3. sayıda yuvarlama yapılmamıştır.

8. Evet/Hayır Yanıtları

Grafiksel arayüzü olmayan bazı uygulamalarda evet ya da hayır olarak yanıtlanması gereken sorular olur. Eğer iletiler yabacı dillere çevriliyorsa, yanıtlarında yerelleştirilmesi gerekir. Soruyu bir dilde sorup, başka bir dilde (genellikle ingilizce) yanıt istemek tuhaf kaçar.

GNU C kütüphanesi uygulamaların ilgili yerel tanımına kolayca erişebilmesini mümkün kılan **rpmatch** işlevini içerir.

```
int rpmatch(const char *yanıt)                                    işlev
```

rpmatch işlevi *yanıt* dizgesinde evet ya da hayır dizgelerinden birinin varlığını arar. Bu işlem seçili yerelin **LC_MESSAGES** kategorisindeki **YESEXPR** ve **NOEXPR** düzenli ifadeleri ile *yanıt* dizgesi karşılaştırılarak yapılır. İşlev şu değerlerden biri ile döner:

1
: Kullanıcı olumlu yanıt verdi.

0
: Kullanıcı olumsuz yanıt verdi.

-1
: Yanıt ne **YESEXPR** ne de **NOEXPR** düzenli ifadesine uyuyor.

Bu işlev standartlaşmamıştır ama GNU C kütüphanesinde ve IBM AIX kütüphanesinde bulunmaktadır.

Bu işlev normalde şöyle kullanılabilir:

```
...
/* Öntanımlı bir değerimiz olsun.  */
_Bool doit = false;

fputs (gettext ("Do you really want to do this? "), stdout);
fflush (stdout);
/* getline çağrısını hazırlayalım.  */
line = NULL;
len = 0;
while (getline (&line, &len, stdout) >= 0)
  {
    /* Yanıta bakalım.  */
    int res = rpmatch (line);
    if (res >= 0)
      {
        /* Yanıt olumluysa.  */
        if (res > 0)
          doit = true;
        break;
      }
  }
/* Yanıtı aldığımız diziyi serbest bırakalım.  */
free (line);
```

Döngünün bir hata ya da olumlu veya olumsuz bir yanıt alınıncaya kadar sürdürüldüğüne dikkat edin.

VIII. İleti Çevirileri

İçindekiler

Bir yazılımın kullanıcı arayüzü kullanıcının işini yapmasını kolaylaştıracak şekide tasarlanmalıdır. Arayüzü oluşturan iletilerin kullanıcının tercih ettiği dilde olması da bu kolaylıklardan biridir.

İletilerin farklı dillerde gösterilmesi farklı yollarla gerçekleştirilebilir. Bunlardan biri tüm dilleri ve her iletinin bu dillerdeki çevirilerini yazılıma eklemektir. Bu pek de iyi bir çözüm değildir, çünkü dil kümesini genişletmek gerektiğinde işler zorlaşır (kodun değişmesi gerekir). Kodun kendisi ileti kümeleri arttıkça aşırı büyür.

Daha iyi bir çözüm ileti kümelerini her dil için ayrı dosyalarda tutmak, kullanıcının seçimine bağlı olarak yazılımın çalışması esnasında iletileri bu dosyalardan yüklemektir.

GNU C Kütüphanesi ileti çevirilerini destekleyen iki işlev ailesi içerir. Sorun bu amaca uygun bir arayüzün POSIX standardında resmen tanımlanmamış olmasıdır. **catgets** ailesi işlevler X/Open standardında tanımlanmış, ancak bu endüstriyel kararlarla oluşturulmuş ve bu bakımdan kabul edilebilir kararlara dayanması gerekmemiştir.

Yukarıda kısaca bahsedildiği gibi ileti çevirilerini içeren harici veri dosyaları kullanılarak ileti kataloglarının kolayca genişletilebilmesi sağlanmıştır. Yani, bu dosyalar yazılımdaki her ileti için ilgili olduğu dilde birer çeviri içerir. Bu yapıya göre ileti işleme işlevlerinin görevleri:

- uygun çevirileri içeren harici dosyayı bulmak
- veriyi yüklemek ve çevirilerin adreslenebilmesini sağlamak.
- belirtilen bir anahtarı çevrilmiş iletiye eşlemek

Son adımın gerçeklenmesi açısından iki yaklaşım farklı davranır. Kalan her şey için tasarım kararları bunun etkisinde verilmiştir.

1. X/Open İleti Kataloglarının İşlenmesi

catgets işlevleri basit bir şemaya dayanır:

Kaynak koddaki her ileti eşsiz bir belirteç kullanılarak çeviriyle ilişkilendirilir. İletinin katalog dosyasından alınmasında sadece bu belirteç kullanılır.

Bunun yazılımın geliştiricisi açısından anlamı, yazılımın kodunda ve ileti kataloğunda aynı olan her belirtecin anlamını bilmek zorunda kalmaktır.

Bir ileti çevrilmeden önce katalog dosyası yüklenmiş olmalıdır. Yazılımın kullanıcısının işlevin kullanıcının istediği kataloğu bulmasına yardımcı olması gerekir. Bu yazılımcının yapması gereken işlerden değildir.

catgets işlevleri ile ilgili veri türleri, sabitler ve işlevler `nl_types.h` başlık dosyasında tanımlanmış ya da bildirilmiştir.

1.1. catgets İşlevleri

```
nl_catd catopen(const char *katalog,                                    işlev
                int         seçenek)
```

catopen işlevi *katalog* ile belirtilen ileti dosyalarını bulmaya çalışır ve bulduğunda yükler. Dönüş değeri geçirimsiz türdedir ve diğer işlev çağrılarında yüklü kataloğu belirtmek için kullanılabilir.

İşlev başarısız olursa katalog yüklenmemiş demektir ve dönüş değeri **(nl_catd) -1**'dir. *errno* genel değişkeni başarısızlığın sebebini belirten bir hata kodu içerir. İşlev çağrısının başarılı olması katalogdaki tüm iletilerin çevrilmiş olduğu anlamına gelmez.

Katalog dosyasının yeri yazılımın kullanıcısının kararını açıklaması sağlanarak belirlenir. Kullanılacak dile karar vermek kullanıcıya bırakılır ve bazan başka bir katalog dosyaları kullanmak yararlı olur. Bunun tamamı kullanıcı tarafından bazı ortam değişkenleri kullanılarak belirtilebilir.

İlk sorun ileti kataloglarının bulunduğu yeri bulmaktır. Her yazılımın farklı dosyalar için kendi yeri olabileceği gibi, katalog dosyaları dillere göre gruplanarak tüm yazılımlar için aynı bir yerde de tutulabilir.

catopen işlevine yazılımın katalog dosyalarının nerede bulacağı kullanıcı tarafından **NLSPATH** ortam değişkenine yazılarak belirtilebilir. Bu değerin farklı dil ve yereller için kullanılabilir olması gerektiğinden, değer basit bir dizge olamayacaktır. Dizgenin kendine özgü bir biçimi vardır (**printf**'in biçim dizgesi gibi). Bir örnek:

```
/usr/share/locale/%L/%N:/usr/share/locale/%L/LC_MESSAGES/%N
```

Görüleceği üzere, iki nokta üstü üste işaretleri ile ayırarak birden fazla dizin belirtilebilir. Biçim dizgesinde dikkati çeken ikinci durum **%L** ve **%N** belirteçleridir. **catopen** işlevi bunların çeşitlerini ve yerlerine neleri yerleştireceğini bilir.

%N
: Bu biçim elemanı katalog dosyasının isminin yerine geçer. Değeri, **catopen** işlevine belirtilen *katalog* argümanının değeridir.

%L
: Bu biçim elemanı ileti çevirileri için seçili yerelin isminin yerine geçer. Bunun nasıl saptanacağı aşağıda anlatılmıştır.

%l
: (Küçük L harfi.) Bu biçim elemanı yerel isminin dil elemanının yerine geçer. Seçili yereli betimleyen bu dizgenin şu biçimde olması beklenir:

  dil`[`*_ülke*`[`*.karakter_kümesi*`]]`

  Bu biçim elemanı bu dizgenin *dil* parçasını ifade eder.

%t
: Bu biçim elemanı yerel isminin *ülke* elemanının yerine geçer. Yerel isminin biçimi yukarda verilmişti.

%c
: Bu biçim elemanı yerel isminin *karakter_kümesi* elemanının yerine geçer. Yerel isminin biçimi yukarda verilmişti.

%%
: **%** karakteri bir önceleme karakteri gibi kullanıldığından kendisini de öncelemesi gerekir. **%%** belirteci aynen **printf** işlevindeki gibi çalışır.

NLSPATH kullanılarak, kullanılacak farklı dillerin ileti kataloglarının aranacağı dizinler keyfi olarak belirlenebilir. Eğer **NLSPATH** ortam değişkeni ortamda mevcut değilse, öntanımlı değeri:

önek`/share/locale/%L/%N:`*önek*`/share/locale/%L/LC_MESSAGES/%N`

Burada *önek*, GNU C kütüphanesi derlenirken **configure** betiğine **--prefix** seçeneğinde belirtilen dizindir (bu dizin çoğu zaman ya **/usr** ya da boş dizgedir).

Sorun artık hangisinin kullanılacağına karar vermektir. Karar, yukarıda bahsedilen biçim elemanlarının nelerle ikame edileceğidir. Herşeyden önce, ileti kataloğunun ismi içinde bir dosya yolu belirtebilir (yani dosya ismi bölü işaretleri içerir). Bu durumda **NLSPATH** ortam değişkeni kullanılmaz. Katalog yazılımda belirtildiği gibi mevcut olmalıdır; şüphesiz, bu çalışma dizinine görelidir. Bu istenen bir durum değildir ve katalog isimleri bu yöntemle yazılmamalıdır. Bununla beraber, **catgets** arayüzünü kullanan platformlar bakımından bu davranış taşınabilir olmayacaktır.

Aksi takdirde, *standart ortamdaki* (sayfa: 678) ortam değişkenlerinin değerlerine bakılır. Hangi değişkenlere bakılacağına **catopen** işlevinin *seçenek* parametresine bakarak karar verilir. Eğer bu değer **NL_CAT_LOCALE** (`nl_types.h` dosyasında tanımlıdır) ise, **catopen** işlevi **LC_MESSAGES** kategorisi için seçili yerelin ismini kullanır.

Eğer *seçenek* sıfırsa **LANG** ortam değişkenine bakılır. Bu davranış, yerel kavramının POSIX yerelleri seviyesinde henüz aşılamadığı erken dönemlerden günümüze ulaşmıştır.

Ortam değişkeninin içerdiği yerel ismi yukarıda açıklandığı gibi *dil*`[_`*ülke*`[.`*karakter_kümesi*`]]` biçiminde olmalıdır. Hiç ortam değişkeni bulunamazsa, çevirilerin kullanılmayacağı anlamına gelen **"C"** yereli kullanılır.

Bir hata oluştuğunda *errno* genel değişkenine şunlar atanabilir:

`EBADF`
: Katalog mevcut değil.

`ENOMSG`
: Set/ileti demeti ileti kataloğundaki mevcut bir elemanın ismi değil.

Bazan hatalara karşı sınama yapmak faydalı olduğu halde yazılımlar normalde herhangi bir sınamadan kaçınacaktır. Bir çevirinin yokluğu özgün ileti varsa sorun olmaz, çevrilmemiş ileti basılır. Bu durumu kullanıcı ya olduğu gibi kabul eder ya da iletinin neden çevrilmediğini araştırır.

Lütfen dikkat edin, o an seçili yerelin saptanması ve çeviri kataloğunun bulunması için **setlocale** çağrısı yapmak gerekmez. **catopen** işlevi bu değerleri ortam değişkenlerinden doğrudan kendisi okur.

```
char *catgets(nl_catd      katalogtanıtıcı,                                  işlev
              int          set,
              int          ileti,
              const char *dizge)
```

catgets işlevi önceki bir **catopen** çağrısı ile açılmış ileti kataloğuna erişmekte kullanılır. *katalogtanıtıcı* parametresi önceki bir **catopen** çağrısının dönüş değeri olmalıdır.

Sonraki iki parametre, *set* ve *ileti*, ileti kataloğu dosyalarının iç düzeni ile ilgilidir. Bu aşağıda ayrıntılı olarak açıklanmıştır. Şimdilik, kataloğun çeşitli set ve iletiler içeren herbiri numaralanmış evrelerden oluştuğunu bilmemiz yeterli olacaktır. Ne set numarasının ne de ileti numarasının bir önemi vardır. Keyfi olarak seçilmiş olabilirler. Fakat her iletinin (başka biriyle aynı olmadıkça) kendine özel eşsiz bir set/ileti çifti ve ileti numarası olmalıdır.

Kullanıcı tarafından seçilen dil için ileti kataloğunun bulunacağı garanti olmadığından son parametre olan *dizge* ile bu duruma bir çözüm getirilmiştir. Eğer ileti ile eşleşen bir dizge bulunamazsa *dizge* döndürülür. Bunu yazılımcı açısından anlamı:

- *dizge* parametresi kabul edilebilir bir metin içermelidir (bu ayrıca umulan dizge dönmediğinde bu duruma dair bir ipucu elde edilmesini sağlayabilmelidir).
- tüm *dizge* argümanları aynı dilde yazılmış olmalıdır.

catgets işlevinin dönüş değeri her durumda geçerli bir dizgedir. Ya ileti kataloğundaki bir çeviridir ya da *dizge* parametresi ile belirtilendir. Çevirinin kullanılıp kullanılmayacağına karar veren kod şöyledir:

```
{
  char *trans = catgets (desc, set, msg, input_string);
  if (trans == input_string)
    {
      /* Birşeyler yanlış gitmiş.  */
    }
}
```

Destekleyici bir işlevsellik olmadıkça **catgets** işlevlerini kullanarak yazılım geliştirmek biraz tatsız olur. Her set/ileti numarası demetinin eşsiz olması gerektiğinden yazılımcı kodu yazarken aynı zamanda iletilerin listesini de tutmalı ve aynı proje üzerinde çalışan yazılımcılar arasında koordinasyon sağlanmış olmalıdır. Bunun nasıl biraz daha esnetilebileceğini *catgets Kullanımı* (sayfa: 187) bölümünde göreceğiz.

`int` **catclose**`(nl_catd` *katalogtanıtıcı*`)` işlev

catclose işlevi önceki bir **catopen** çağrısı ile açılmış bir ileti kataloğu ile ilgili özkaynakları serbest bırakmakta kullanılır. Eğer özkaynaklar başarıyla serbest bırakılmışsa işlev **0** ile aksi takdirde **−1** ile döner ve *errno* değişkenine hata durumu atanır. Eğer *katalogtanıtıcı* ile belirtilen katalog tanıtıcı geçersizse, *errno* değişkenine **EBADF** atanır.

1.2. İleti Kataloğu Dosyalarının Biçimi

Bir yazılımın tüm iletilerinin çevrilmesinin kabul edilebilir tek yolu, iletileri bir ileti kataloğu dosyasında toplamak ve bunu çevirmenin çevirmesini sağlamaktır. Yani, belli bir çeviri ile set/ileti çiftini ilişkilendiren girdilerden oluşan bir dosyamız olmalıdır. Bu dosyanın biçimi X/Open standardında şöyle belirlenmiştir:

- Boş satırlar ve sadece boşluk karakterleri içeren satırlar yoksayılır.
- Satırın başında bir **$** karakterinden sonra boşluk geldiğinde satırın kalanının bir açıklama içerdiği varsayılır ve bu satırlar da yoksayılır.
- Satır **$set** ile başlıyorsa bunu bir boşluk karakteri ile bir argüman izlemelidir. Bu argüman şunlardan biri olabilir:
  - Bir sayı. Bu sayının değeri altındaki iletiler bir küme oluşturur.

  - Alfanümerik karakterlerle alt çizgi karakterinden oluşabilen bir belirteç. Bu durumda ileti kümesinin numarası özdevinimli olarak atanır. Bu değer o kümeye kadar belirtilenlerden büyük bir sayı olmalıdır.

    Sembolik isimlerin nasıl kullanıldığı *catgets Kullanımı* (sayfa: 187) bölümünde açıklanmıştır.

    Bir sembol ismi birden fazla varsa bu bir hatadır. Bir kümenin içerdiği tüm iletiler bu numara altında toplanır.

- Satır **$delset** ile başlıyorsa bunu bir boşluk karakteri ile bir argüman izlemelidir. Bu argüman şunlardan biri olabilir:

  - Bir sayı. Bu sayının değeri altındaki iletilerin oluşturduğu kümenin silineceğini belirtir.
  - Alfanümerik karakterlerle alt çizgi karakterinden oluşabilen bir belirteç. Bu sembolik belirteç evvelce tanımlanmış bir küme ismi olmalıdır. İsim bilinmiyorsa bu bir hatadır.

  Her iki durumda da belirtilen kümelerdeki tüm iletiler silinir ve çıktıda görünmezler. Fakat bu küme daha sonra tekrar **$set** komutu ile seçilerek içerdiği iletilerin tekrar çıktılanması sağlanabilir.

- Satır **$quote** ile başlıyorsa bunu bir boşluk karakteri ile girdi dosyasında tırnak karakteri olarak kullanılacak karakter izlemelidir. Satır boşluktan farklı bir karakter ile bitmiyorsa dosyada tırnaklar iptal edilir.

  Öntanımlı olarak bir tırnak karakteri yoktur. Bu kipte dizgeler öncelenmemiş ilk satırsonu karakterinde biter. Eğer bir **$quote** belirtimi varsa satırsonu karakterlerinin öncelenmesi gerekmez. Bu durumda dizge öncelenmemiş ilk tırnak karakterinde biter.

  Bu özelliğin çok kullanılan bir kullanımı tırnak karakteri olarak **"** belirtmektir. Dizge içinde bir **"** karakteri kullanmak gerekirse **\"** şeklinde tırnak işareti tersbölü ile öncelenmelidir.

- Bunların dışında satırlar bir sayı veya bir alfanümerik sözcük (alt çizgi dahil) ile başlayabilir. İlk boşluk karakterinden hemen sonra ileti dizgesi gelir. Satırın başındaki sözcük ya da sayı iletinin numarasını ifade eder.

  Eğer satır bir sayı ile başlıyorsa bu açıkça iletinin numarasıdır. İleti kümesi içinde aynı numaralı ileti birden fazla varsa bu bir hatadır.

  Eğer satır bir sözcük ile başlıyorsa ileti numarası özdevinimli olarak atanır. Değeri iletiyi içeren kümedeki en büyük numaralı iletinin numararasının bir fazlasıdır. Bir ileti kümesi içinde aynı sözcük birden fazla ileti için kullanılmışsa bu bir hatadır. Ama aynı sözcük başka bir küme içinde kullanılabilir. Sembolik belirteçlerin nasıl kullanıldığı *catgets Kullanımı* (sayfa: 187) bölümünde anlatılmıştır. Belirteç ile ilgili bir sınırlama vardır: bu, **Set** olmamalıdır. Bunun sebebi ilerde açıklanacaktır.

  İleti metni öncelenmiş karakterleri içerebilir. Bunlar ISO C dilinde tanınan karakterlerdir (**\n**, **\t**, **\v**, **\b**, **\r**, **\f**, **** ve *****nnn*, burada *nnn* karakterin sekizlik kodudur).

Önemli

Set ve iletiler için sayılar yerine belirteçlerin belirtilebilmesi bir GNU oluşumudur. X/Open standardına sıkı sıkıya bağlı sistemlerde bu özellik yoktur. Bir ileti kataloğu dosyası örneği:

```
1 $ Bu bir açıklamadır.
2 $quote "
3
4 $set SetOne
5 1 Numarası 1 olan ileti.
6 two "   Kimliği \"two\" olan ileti, buna 2 değeri atanacaktır"
```

```
7
8 $set SetTwo
9 $ Önceki Set'in numarası 1 oldu, bununki 2 olacaktır.
10 4000 "Numaralar keyfidir, 1'den başlamaları gerekmez."
```

Bu küçük örnekte görünenler:

- 1. ve 9. satır **$** işaretinden sonra bir boşluk içerdiğinden birer açıklamadır.
- Tırnak karakteri olarak **"** atanmıştır. Aksi takdirde ileti tırnak içine alınamayacağından **two** kimlikli iletinin başındaki boşluklar gösterilemeyecekti.
- Numaralı iletilerle sembolik isimli iletilerin karışık kullanılması bir sorun oluşturmaz, numaralama özdevinimli olarak yapılacaktır.

Bu dosya biçimi çok kolay görünmesine rağmen bir çalışan yazılımda kullanmak için yeterli değildir. **catopen** işlevi dosyayı çözümlemeli ve sözdizimsel hataları en iyi şekilde elde etmelidir. Bu öyle kolay değildir ve işlem oldukça yavaştır. Bu bakımdan **catgets** işlevleri verinin daha bütünleşik ve kullanıma hazır bir dosya biçiminde olmasını umar. Bunun için ayrıntıları sonraki bölümde açıklanacak olan özel bir uygulama, **gencat** vardır.

Bu biçimdeki dosyalar insanlarca okunabilir değildir. Yazılımlar tarafından kolayca kullanılabilmeleri için ikilik biçimdedirler. Fakat biçimin bayt sıralaması dosyaların bulunduğu sisteme bağlıdır.

İkilik dosyanın biçimi hakkında ayrıntıya girmeyeceğiz, çünkü bu dosyalar daima **gencat** uygulaması tarafından oluşturulurlar. GNU C Kütüphanesinin kaynak kodunda **gencat** uygulamasının kaynak kodu da bulunduğundan, çok meraklıysanız, dosyanın biçimi hakkında fikir edinmek için kaynak kodunu inceleyebilirsiniz.

1.3. İleti Kataloğu Dosyalarının Üretilmesi

gencat uygulaması X/Open standardında belirtilmiştir ve GNU gerçeklemesi bu belirtime uygundur. Fazladan, **catgets** işlevlerinin daha makul şekilde çalışmasına yardımcı olacak ek oluşumlarla gerçeklenmiştir.

gencat yazılımı iki yolla çalıştırılabilir:

`**gencat** [*seçenek*]… [*çıktı–dosyası* [*girdi–dosyası*]…]`

Bu, X/Open standardında tanımlı arayüzdür. *çıktı–dosyası* parametresi verilmezse girdi standart girdiden okunur. Çok sayıda girdi dosyası belirtilebilir. Eğer *çıktı–dosyası* verilmemişse çıktılama için standart çıktı kullanılır. Başka yazılımlardan kullanmak için ikinci bir arayüz sağlanmıştır.

`**gencat** [*seçenek*]… **-o** *çıktı–dosyası* [*girdi–dosyası*]…`

-o seçeneği çıktı dosyasını belirtmek için kullanılır ve belirtilen diğer bütün dosyalar girdi dosyası olarak kullanılır.

Bunlardan başka, *girdi–dosyası* olarak **-** veya **/dev/stdin** belirtilirse, girdi standart girdiden alınır. Aynı şekilde, *çıktı–dosyası* olarak **-** veya **/dev/stdout** belirtilirse, çıktılama standart çıktıya yapılır. **-** kullanımı X/Open standardında bir dosya ismi olarak ele alındığı halde, aygıt ismi olarak kullanımı bir GNU oluşumudur.

gencat uygulaması tüm girdi dosyalarını bibirine ekleyerek çalışır ve bu iletileri çıktı dosyasındakilerle biraraya getirir. Bu işlem çıktı dosyasında zaten mevcut olan Set/İleti demetlerini silip, olmayanları çıktıya ekleyerek yapılır. Bu işlem, eski eşleşen Set/ileti demetleri kaldırılarak yapıldığından çıktı dosyasının tamamen boşalmasına bile sebep olabilir. Çıktılama standart çıktıya yapılıyosa bu biraraya getirme işlemi yapılmaz.

Burada **gencat** uygulamasının tanıdığı seçeneklere yer verilmiştir. X/Open standardında uygulama için herhangi bir seçenek tanımlanmadığından bu seçeneklerin hepsi GNU oluşumudur.

-V
--version
Sürüm bilgilerini basar ve çıkar.

-h
--help
Bir kullanım iletisi basar ve çıkar.

--new
Girdi dosyalarındaki yeni iletileri çıktı dosyasındaki eski iletilerle birleştirmez. Çıktı dosyasının eski içeriği silinir.

-H *name*
--header=*isim*
Bu seçenekle, yazılımda kullanmak için girdi dosyalarındaki Set'lerin ve iletilerin sembolik isimleri çıktılanır. Bu isimlerin nasıl kullanıldığı sonraki bölümde açıklanmıştır. Seçenekte *name* parametresi ile çıktı dosyasının ismi belirtilir. Çıktı dosyasında isimleri numaralarla eşleyen **#define** yönergeleri bulunur.

Üretilen dosyanın sadece girdi dosyalarındaki sembolleri içereceğine dikkat edin. Eğer, çıktı dosyası önceki çıktı dosyasındakilerle katıştırılarak elde ediliyorsa, hem girdi dosyalarında hem de eski çıktı dosyasında bulunan iletilerin sembolleri üretilen başlık dosyasında bulunmayacaktır.

1.4. **catgets** Kullanımı

catgets işlevleri iki farklı yolla kullanılabilir: X/Open belirtiminden hiç sapmadan ve GNU oluşumlarını kullanarak. Önce ilk yöntemin nasıl kullanıldığına bakacağız, böylece ikinci yöntemin yani GNU oluşumlarının getirilerini daha iyi anlayabileceğiz.

1.4.1. Sembolik isimleri kullanmadan

İleti X/Open biçimi katalog dosyalarında sembolik isimlere izin verilmediğinden bu seferlik sadece numaralarla çalışacağız. Yazılımı geliştirmeye başlarken çevrilebilir dizgelerin hepsini şu biçimde bir kodla değiştirmelisiniz:

```
catgets (catdesc, set, msg, "dizge")
```

catdesc parametresi, yazılım başında normalde bir kere yapılan bir **catopen** çağrısından alınır. *"dizge"* çevirlecek dizgedir. Set ve ileti numaralarının başlatılması sorun olur.

Büyükçe bir yazılımda aynı anda çok sayıda geliştirici çalışırsa numara ayırma işleminin eşgüdümü önem kazanır. Aynı demet numarası ile iki farklı dizge indislenemediğinden, birbirinin aynı dizgeler ile birbirinin aynı çeviriler için numaraların yeniden kullanılması tercih edilir. (Bir dilde aynı olan dizgeler başka bir dilde bağlama bağlı olarak farklı dizgeler olabilir; bir de bu var...)

Numara ayırma işlemi yazılımın farklı parçalarında farklı Set numaraları kullanılarak biraz olsun rahatlatılabilir. Böylece eşgüdümlenecek geliştirici sayısı düşürülebilir. Yine de ayırma ile ilgili listeler tutulabilir ve hataların giderilmesi kolaylaştırılabilir. Bu hatalar derleyici ya da **catgets** işlevleri tarafından saptanamaz. Sadece yazılımın kullanıcısı yanlış iletinin basıldığını görür. En kötü durumda, yanlış olarak tanımlanamayan iletiler çok sinir bozucu olabilir. Doğrularla yanlışlar birbirine karışır ve bu bir felaketle sonuçlanır.

1.4.2. Sembolik isimleri kullanarak

Önceki bölümde bahsedilen sorunlardan şu sonuçları çıkarabiliriz:

1. Numaralar bir kere ayrıldıktan sonra onları değiştirmek gerekirse bu çok zor olur.
2. Numaralar dizgenin içeriği hakkında hiçbir fikir vermediğinden karışıklıklar olabilir.

Sürekli olarak sembolik isimleri kullanarak ve dizge içeriğini bir sembolik isimle eşleyen bir yöntem sağlanarak bu iki sorundan kurtulmak mümkündür. Bu yöntemim yazılımcıya maliyeti yazılımı geliştirirken bir yandan da ileti kataloğunu yazmasıdır.

Yazılım derlenmeden önce sembolik isimlerin numaralarla eşleştirilmesi gerektiğinden bunun böyle olması gerekir. Önceki bölümde **gencat** uygulaması tanıtılırken isimlerle eşleşmeleri içeren bir başlık dosyasının nasıl üretileceğinden bahsedilmişti. Örneğin, önceki örnekteki katalog dosyasının isminin `ex.msg` olduğu kabul ederek,

```
gencat -H ex.h -o ex.cat ex.msg
```

komutuyla şu içeriğe sahip bir başlık dosyası üretilir:

```
#define SetTwoSet 0x2   /* ex.msg:8 */

#define SetOneSet 0x1   /* ex.msg:4 */
#define SetOnetwo 0x2   /* ex.msg:6 */
```

Görüldüğü gibi, kaynak dosyasında verilen sembollerden eşsiz belirteçler üretmek için yeni semboller elde edilmiş ve bunlar numaralara atanmıştır. Kaynak dosyayı okuyarak ve kuralları bilerek başlık dosyasının içeriğini tahmin etmek mümkündür ama bu gerekli değildir. **gencat** uygulaması herşeyi dikkatlice yapabilir. Bütün geliştiriciler, projenin kaynak dosyalarının bağımlılık listesine üretilen başlık dosyasını koymalı ve girdi dosyaları değiştiğinde başlığı üretecek kuralları eklemelidir.

Sembollerden makro isimlerinin elde edilmesi ile ilgili olarak da bir kaç laf etmek lazım. Her makro ismi iki parçadan oluşur: İleti kümesinin ismi artı iletinin ismi ya da özel bir dizge olarak **Set**. Yani, **SetOnetwo** makrosu ile ileti katalog dosyasındaki **SetOne** ileti kümesinin **two** isimli iletisine erişilir.

Diğer isimler ileti kümelerinin isimlerini belirtir. Özel **Set** dizgesi ileti belirtecinin yerine kullanılır.

Eğer koddaki **SetOne** kümesinin ikinci dizgesi C kodunda kullanılmak istenirse şöyle yapılırdı:

```
catgets (catdesc, SetOneSet, SetOnetwo,
         "   Kimliği \"two\" olan ileti, buna 2 değeri atanacaktır")
```

İşlevin bu yöntemle yazılması C kaynak kodunda herhangi bir değişiklik gerekmeksizin ileti numarasını hatta Set numarasını değiştirmek mümkün olacaktır. (Dizgenin metni normalde aynı görünür; bu sadece bu örnek içindir.)

1.4.3. Bunları yazılım geliştirirken nasıl kullanacağız?

Sembolik sürüm numaraları ile çalışma yöntemini göstermek için burada küçük bir örneğe yer verdik. Çook karmaşık ve ünlü bir selamlama yazılımını yazmak istediğimizi varsayalım. Koda şöyle başlardık:

```
#include <stdio.h>
int
main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

Şimdi iletileri uluslararasılaştıralım ve böylece kullanıcı iletiyi kendi diline çevirebilsin.

```
#include <nl_types.h>
#include <stdio.h>
#include "msgnrs.h"
int
```

```
main (void)
{
  nl_catd catdesc = catopen ("hello.cat", NL_CAT_LOCALE);
  printf (catgets (catdesc, SetMainSet, SetMainHello,
                   "Hello, world!\n"));
  catclose (catdesc);
  return 0;
}
```

Katalog nesnesinin nasıl açıldığını ve diğer işlev çağrılarında kullanılacak tanımlayıcının nasıl döndürüldüğünü görüyoruz. Buradaki işlevler makul davranacağından işlevlerde herhangi bir başarısızlık sınaması yapmamız aslında gerekmiyor. Sadece bir çeviri döndürecekler.

Burada **SetMainSet** ve **SetMainHello** sabitleri belirsiz kaldı. Bunlar iletiyi tanımlayan sembolik isimlerdir. Katalog dosyasındaki bilgiyle eşleşen gerçek tanımlarını elde etmek için ileti kataloğunu oluşturup onu **gencat** uygulaması ile işlemeliyiz.

```
$ Turkish messages for the famous greeting program.
$quote "

$set Main
Hello "Merhaba Dünyalı!\n"
```

Yazılımımızı artık derleyebiliriz. Katalog kaynak dosyasının isminin `hello.msg` ve yazılım kaynak dosyasının isminin `hello.c` olduğunu varsayıyoruz:

```
$ gencat -H msgnrs.h -o hello.cat hello.msg
$ cat msgnrs.h
#define MainSet 0x1     /* hello.msg:4 */
#define MainHello 0x1   /* hello.msg:5 */
$ gcc -o hello hello.c -I.
$ cp hello.cat /usr/share/locale/tr/LC_MESSAGES
$ echo $LC_ALL
tr
$ ./hello
Merhaba Dünyalı!
$
```

gencat çağrısı ile ileti kataloğunun ikilik biçimi ile birlikte `msgnrs.h` başlık dosyasını da elde ettik. İlk olarak `hello.c`'yi derledik, sonra da `hello.cat`'i **catopen** işlevinin bulabileceği yere yerleştirdik. Katalog dosyasını doğru yere yerleştirip yerleştirmediğimizi **LC_ALL** ortam değişkeninin değerine bakarak sınamayı unutmuyoruz. (Ç.N. — Özgün metinde "tr" değil "de" var ama "de" değerli bir "LC_ALL" ne kadar mümkün?; yazar burada biraz dikkatsizmiş sanırım. Değişken LC_ALL değil LANGUAGE olmalıydı, çünkü LC_ALL ortam değişkeninin değeri normalde bir boş dizgedir; ortam değişkeni olarak anlamlı değildir ama bir C makrosu olarak anlamlıdır.)

2. İleti Çevirilerinde Uniforum Yaklaşımı

Sun Microsystems, Uniforum grubunda ileti çevirilerine farklı bir yaklaşımı standarlaştırmaya çalıştı. Gerçek bir standart hiç tanımlanmamıştı, ama bu arayüz hala Sun'ın işletim sistemlerinde kullanılıyordu. Açık kaynak kod geliştirenlere bu yaklaşım daha iyi bir geliştirme ortamı sağladığı için GNU projelerinde kullanılmaya başlandı ve GNU C kütüphanesi dışında buna destek olmak için GNU **gettext** paketi oluşturuldu.

GNU **gettext**'teki **libintl** kodu, GNU C kütüphanesindeki kod ile aynıdır. Bu bakımdan GNU **gettext** kılavuzundaki bilgiler ayrıca buradaki işlevsellik için de geçerlidir. Bu kısımda kütüphane işlevlerini ayrıntılı olarak

açıklayacağız, ama çok sayıdaki yardımcı uygulamayı bu kılavuzda açıklamayacağız. Ayrıntılarla ilgilenenlerin GNU **gettext** kılavuzuna bakmalarını öneririz. Burada bunların sadece kısa bir tanıtımını yapacağız.

catgets işlevleri çoğu sistemde öntanımlı olarak bulunmasına rağmen **gettext** arayüzü de en azından onun kadar taşınabilirdir. İşlevlerin bulunmadığı yerlerde GNU **gettext** paketi kullanılabilir.

2.1. gettext İleti Katalogları

İleti çevirilerine **gettext** yaklaşımının altında yatan paradigmalar, temel işlevsellik bakımından eşdeğeri olan **catgets** işlevlerinden farklıdır.

2.1.1. gettext ile Çeviri

gettext işlevlerinin çok basit bir arayüzü vardır. En temel işlev argüman olarak sadece çevrilecek dizgeyi alır ve onun çevirisi ile döner. Bu, özgün dizgenin sadece hata durumunda kullanıldığı, normalde ek bir anahtar gerektiren **catgets** yaklaşımından en temel farktır.

Çevrilecek dizgenin tek argüman olması şüphesiz dizgenin kendisinin bir anahtar olduğu anlamına gelir. Bundan dolayı ileti kataloglarının hem özgün dizgeyi hem de çeviriyi içermesi gerekir. **gettext** işlevinin görevi argüman olarak verilen dizgeyi ileti kataloğundaki dizgelerle karşılaştırıp o dizgeye ait çeviriyi döndürmektir. Şüphesiz bu işlem eniyilenmiştir, dolayısıyla bu işlem **catgets**'deki gibi bir atomik anahtarla erişim sağlamaktan daha pahalıya malolmaz.

gettext yaklaşımının getirileri olduğu gibi götürüleri de vardır. Bu konuda daha fazla ayrıntı için GNU **gettext** kılavuzuna bakınız.

gettext ile ilgili tüm tanım ve bildirimler **libintl.h** başlık dosyasında bulunabilir. Bu işlevlerin C kütüphanesinin bir parçası olmayan sistemlerde **libintl.a** isimli ayrı bir kütüphane olarak (ya da paylaşımlı bir kütüphane olarak) bulunabilir.

```
char *gettext(const char *msgid)                                    işlev
```

gettext işlevi o an seçili olan ileti kataloglarında *msgid* ile belirtilen dizgeye eşdeğer çeviriyi arar. Eğer böyle bir dizge varsa onunla döner. Aksi takdirde, *msgid* ile belirtilen dizgeyi döndürür.

Dönüş değeri **char *** türünde olduğundan dönen dizgenin değiştirilmemesi gerektiğine dikkat edin. Bu bozuk türdeki sonuçlar işlevin geçmişinden kaynaklanır ve işlevin kullandığı yöntemi yansıtmaz.

Dikkar ederseniz, yukarıda ileti katalogları (çoğul) yazdık. Bu, bu işlevlerin GNU gerçeklemesinin bir özelliğidir ve ileti kataloglarının seçim yöntemlerindan bahsederken bundan daha ayrıntılı sözedeceğiz (bkz. *gettext kataloğunun yeri* (sayfa: 191)).

gettext işlevi *errno* genel değişkeninin değerini değiştirmez. Şöyle bir kodu yazabilmek için bu gereklidir:

```
printf (gettext ("Operation failed: %m\n"));
```

Burada *errno* değeri, **%m** biçim belirtimiyle işlenerek **printf** işlevinde kullanılmıştır. Eğer **gettext** işlevi bu değeri değiştirmiş olsaydı, (**printf** işlevinden önce çağrılacağından) biz yanlış ileti alacaktık.

Bu durumda, bir ileti kataloğunun mevcut olup olmadığını saptamanın bir yolu yoktur. Normalde kataloglarn bulunmasını sağlamak kullanıcının görevidir. Çünkü yazılım çalışması için normalde kullanıcının dilinde bir ileti kataloğuna ihtiyaç duymaz, bu bakımdan ileti kataloğunun bulunması için kullanıcıya yardımcı olması beklenmez.

İleti kayaloğuna erişmekte kullanılan diğer iki işlev öntanımlı olmayan bir ileti kataloğunun seçilmesine yardımcı olur. Eğer yazılım birbirinden bağımsız olarak geliştirilmiş parçalardan oluşuyorsa, her parçanın kendi ileti kataloğu olabilir ve bunlar aynı anda kullanılabilir. C kütüphanesinin kendisi buna iyi bir örnektir: anlamı belirsiz bilgileri belirtmesi ama o an seçili bir öntanımlı ileti kataloğuna da bağımlı olmaması gerektiğinden, **gettext** işlevlerini dahili olarak kullanır.

```
char *dgettext(const char *alanadı,                                    işlev
               const char *msgid)
```

dgettext işlevi **gettext** işlevi gibi davranır. Farklı olarak ilk argümanında çevirinin aranacağı ileti kataloğu belirtilir. Eğer *alanadı* ile bir boş gösterici belirtilmişse, katalog olarak öntanımlı katalog kullanıldığından **dgettext** işlevi tamamen **gettext** işlevi gibi davranır.

gettext işlevinde olduğu gibi geçmişten gelen bir özellik olarak dönüş türü **char *** 'dır ve dönen dizgenin asla değiştirilmemesi gerekir.

```
char *dcgettext(const char *alanadı,                                   işlev
                const char *msgid,
                int         kategori)
```

dcgettext işlevi **dgettext** işlevinin argümanlarına ek olarak üçüncü bir argüman alır. Bu argüman *kategori* argümanıdır ve ileti kataloğunun yerelini belirtmek için kullanılır. Yani, yerel kategorisi ve alanadı birlikte, kullanılacak kataloğun yerini tespit ederler (aşağıya bakınız).

dgettext işlevi **dcgettext** işlevi kullanılarak şöyle ifade edilebilir:

```
dgettext (domain, string)
```

yerine

```
dcgettext (domain, string, LC_MESSAGES)
```

yazılırsa sonuç aynı olur.

Bu, üçüncü parametre için umulan değerin hangisi olduğunu da gösterir. **locale.h** başlık dosyasında bulunan kategorilerden biri kullanılabilir. Normalde kullanılabilecek değerler: **LC_CTYPE**, **LC_COLLATE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC** ve **LC_TIME**. **LC_ALL** değerinin kullanılmadığına dikkat edin. Bu isimle ortam değişkenlerinin değeri arasında bir ilişki yoktur.

dcgettext işlevi, **gettext** işlevlerine sahip diğer sistemlerle uyumluluk adına gerçeklenmiştir. Aslında *kategori* parametresinde **LC_MESSAGES** dışında bir değeri kullanmanın gerekli olduğu bir durum yoktur. Bunun dışında bir seçim sadece sinir bozucu olabilir.

gettext işlevinde olduğu gibi geçmişten gelen bir özellik olarak dönüş türü **char *** 'dır ve dönen dizgenin asla değiştirilmemesi gerekir.

Yukarıdaki üç işlevi bir yazılımda kullanırken sıklıkla karşılaşılan durum *msgid* argümanının bir dizge sabit olmasıdır. Yani bu duruma göre bir eniyileme yapmak gerekir. Bunun hakkında biraz düşününce, yeni bir ileti kataloğunun yüklenmesi mümkün olmadıkça bir iletinin çevirisinin değiştirilemeyeceği şeklinde gerçekleşmesi gerekir. Bu eniyileme **gettext**, **dgettext** ve **dcgettext** işlevleriyle gerçeklenmiştir.

2.1.2. **gettext** kataloğunun yeri

Belirtilen bir iletinin çevirisini elde eden işlevler olağanüstü basit bir arayüze sahiptir. Fakat yazılımın kullanıcısına istediği çeviriyi seçme fırsatı vermek ve yazılımcıya da katalog dosyalarının aranacağı yerler bakımından bir hakimiyet sağlamak gerekir. Tüm bunları denetim altına alacak mekanizma yani kod oldukça karmaşıktır ama kullanımı kolaydır.

catgets işlevleri tarafından da uygulanan iki temel görevin yerine getirilmesi gerekir:

1. İleti kataloglarının yeri. Hepsi aynı pakete karşılık gelen farklı dillerde bir miktar dosya vardır. Genellikle, bunlar dosya sistemi üzerinde belli bir dizin altında bulunurlar.

   Kurulmuş çok sayıda keyfi paket olabilir ve bunlar dosyalarını bulmak için farklı kuralları izleyebilir.

2. Kullanıcının isteklerine uyarlanmış çeviri dosyalarının aranması paket tarafından belirtilen konuma göreli olmalıdır. Yani, kullanıcı paketi seçerek her dil ile ilgili dosyayı konumlayabilmelidir.

Bu, **gettext** belirtimi tarafından gerek duyulan işlevsellik olduğu kadar **catgets** işlevlerinin de yapabildiği bir şeydir. Ancak, çözümlenmesi gereken bazı sorunlar vardır:

- Kullanılacak dil farklı yollarla belirtilebilir. Bunun için genel bir standart kabul edilmiş değildir ve kullanıcı yazılımdan ne anlatmak istediğini anlamasını bekler. Örneğin, Almanca çeviriyi seçmek için kullanıcı **de**, **german** ya da **deutsch** yazabilir ve yazılımın daima buna bir tepki vermesini bekler.

- Bazan kullanıcının belirtimi çok ayrıntılı olabilir. Eğer kullanıcı kendini, Almanya'da, Almanca konuşan ve ISO 8859–1 karakter kodlamasını kullanan biri olarak tarif etmek için `de_DE.ISO-8859-1` belirtimini kullanırsa bu belirtimle tam olarak eşleşen bir ileti kataloğu bulamayacaktır. Ama **de** ile eşleşen bir katalog olabilir. Ayrıca, makinada kullanılan karakter kümesi daima ISO 8859–1 ise, ileti kataloğunun bunu kullanmaması için bir sebep yoktur. (Buna *ileti kalıtımı* diyoruz.)

- Eğer istenen dilde bir katalog yoksa, son çare yazılımcının dili olmamalıdır. Kullanıcı iletileri daha iyi anlayacağı başka diller ve hatta bunlara bir öncelik sırası belirtebilmelidir.

Yapılandırma eylemlerini iki parçaya ayırıyoruz: biri yazılımcı tarafından uygulanan, diğeri kullanıcı tarafından uygulanan. Kullanıcının eylemlerine temel teşkil edeceğinden burada önce yazılımcının kullanabileceği işlevlerden bahsedeceğiz.

Önceki bölümde açıklanan işlevlerde iletilerin bir *alan adı*na göre seçilebildiğini görmüştük. Bu, ayrı bir alan kullanan her yazılım parçası için eşsiz olması gereken basit bir dizgedir. Bu isim sayesinde aynı anda çok sayıda keyfi alana sahip bir yazılım kullanmak mümkün olur. Örneğin, C kütüphanesini kullanan bir yazılım **foo** alan adını kullanırken, GNU C kütüphanesinin kendisi **libc** alan adını kullanır. Burada önemli olan aynı anda sadece bir alanın etkin olduğudur. Bu aşağıdaki işlevle denetlenir.

`char *textdomain(const char *alanadı)` işlev

textdomain işlevi ilerde **gettext** çağrılarında öntanımlı alan olarak *alanadı* ile belirtilen alanı etkinleştirir. **dgettext** ve **dcgettext** çağrılarında *alanadı* parametresi ile bir boş gösterici belirtilmedikçe bu alanın etkin olmayacağını unutmayın.

İlk **textdomain** çağrısından önceki öntanımlı alan adı **messages**'dir. Bu, **gettext** arayüzünün belirtiminde belirtilen isimdir. Herhangi bir başka isimden çok daha iyidir. Sorunlarla karşılaşmak istenmiyorsa, hiçbir yazılım bu ismi kullanmamalıdır.

İşlev etkinleştirilen öntanımlı alan adı ile döner. Eğer işlev dönüş değeri için yer ayıramamışsa **NULL** döndürür ve *errno* genel değişkenine **ENOMEM** değerini atar. İşlevin dönüş türü **char *** olduğundan dönen dizge değiştirilmemelidir. Dönüş değeri için yer ayırma işlemi işlev tarafından dahili olarak yapılır.

Eğer *alanadı* bir boş gösterici ise hiçbir alan adı etkinleştirilmez. Sadece geçerli öntanımlı alan adı döner.

Eğer *alanadı* ile bir boş dizge belirtilmişse, öntanımlı alan adı ilk değer olan **messages** yapılır. **messages** alanının asla kullanılmaması gerektiğinden dönüş değeri bu değerle sorgulanabilir.

```
char *bindtextdomain(const char *alanadı,                                    işlev
                     const char *dizinadı)
```

bindtextdomain işlevi farklı diller için *alanadı* alanının ileti kataloglarını içeren dizini belirtmekte kullanılır. Doğru olması bakımından, bu dizinin dizin hiyerarşisi içinde olması beklenir. Aşağıda buna daha ayrıntılı değinilmiştir.

Yazılımla gelen çevirilerin **/foo/bar** gibi bir dizin hiyerarşisine yerleştirilmiş olması yazılımcı açısından önemlidir. Yazılım bundan sonra bu dizinle bir **bindtextdomain** çağrısı yaparak alanı bağlar. Böylece katalogların bulunması kesin olarak sağlanır. Düzgün çalışan bir yazılım kullanıcının bir ortam değişkeni belirtmesine ihtiyaç duymaz.

bindtextdomain çağrısı, eğer önceki bağlanan alanların üzerine yazılmaması için farklı *alanadı* argümanları gerekliyse, defalarca yapılabilir.

Eğer yazılım bir **chdir** çağrısı ile çalışma dizinini değiştirdiği bir sırada **bindtextdomain** çağrısının yapılabilmesi için *dizinadı* argümanının mutlak bir dosyayolu olması önem kazanır. Asi takdirde argümanda belirtilen dizin bu sırada değişebilir.

Eğer *dizinadı* parametresi bir boş gösterci ise, **bindtextdomain** *alanadı* alanı için o an seçili dizin ile döner.

bindtextdomain işlevi normalde seçilen dizinin ismini içeren bir dizgeye gösterici ile döner. Dizge işlev tarafından dahili olarak ayrıldığından yazılımcı bu dizgenin içeriğini değiştirmemelidir. Eğer sistem **bindtextdomain** çalışırken bir bellek çıktısı (core dosyası) çıktılarsa, işlev **NULL** ile döner ve *errno* genel değişkenine ilgili hata durumu atanır.

2.1.3. Gelişkin gettext işlevleri

Şimdiye kadar bahsedilen **gettext** işlevleriyle ilgili tüm mevcut yaklaşımlarda (ve **catgets** işlevlerinin de tamamında) tamamen gözardı edilmiş gerçek dünyaya özgü bir sorun vardır. Burada çoğul biçimlerin nasıl elde edildiğinden bahsedeceğiz. (Ç.N: Türkçe için sorun değil ama diğer diller için başlı başına bir sorun olduğunu birazdan farkedeceksiniz.)

Kimsenin uluslararasılaştırma ile fazla ilgilenmediği zamanlarda (hatta, sonrasında da) Unix kaynak koduna baktığınızda şuna benzer kodlar görürdünüz[3]:

```
printf ("%d file%s deleted", n, n == 1 ? "" : "s");
```

Kodu uluslararsılaştırmaya çalışan kişlerden gelen ilk tepkilerden sonra ya böyle formülasyonlardan vazgeçildi ya da şöyle dizgeler kullanılmaya başlandı:

```
if (n == 1)
  printf ("%d file deleted", n);
else
  printf ("%d files deleted", n);
```

Ancak bu da soruna çözüm olmadı. Miktarlı nesne isimlerine sadece ‘s’ gibi tek bir ek alan dillere çözüm olurken, tamamına bir çözüm sağlayamadı. Çünkü çoğul biçimlerin elde edilmesi her dil ailesinde farklıydı. Bunların arasında (hatta dil ailesindeki diller içinde bile) farklı olabilen iki şey vardır:

- Çoğul biçimlerin kurgulanışı farklıdır. Bu bazı düzensizlikler içeren dillerle ilgili bir sorundur. Örneğin, Almanca’da bu çok belirgindir. İngilizce ve Almanca aynı dil ailesine (Germen) mensup oldukları halde çoğul isim biçiminin (‘s’ ekleme) kurgulanışında farklılıklar vardır.

- Çoğul biçimlerin sayısı farklıdır. Bir tekil, bir de çoğul biçim bulunan Romen ve Germen dillerine aşina olanlar için bu biraz şaşırtıcıdır (Türkçe'den başka dil bilmeyene daha da şaşırtıcı, çünkü Türkçe'de böyle bir olgu *hiç* yok; 1 elma, 5 armut deriz; 1 elma, 5 armutlar demeyiz).

  Ancak, diğer dil ailelerine bir biçimden çok sayıda biçime kadar değişik biçimler sözkonusudur. Bu konuda daha ayrıntılı bilgi ayrı bir bölümde verilmiştir.

Bunun sonucu olarak, uygulama yazarları kodlarında bu sorunu çözümlemekle uğraşmak istemezler. Bu sadece kendi dillerinde yerelleştirme yapanların sorunu gibi görünse de **gettext**'in genişletilmiş arayüzü buna bir çözüm içerir.

Bu ek işlevler bir anahtar dizge yerine iki anahtar dizge ve bir sayısal argüman alırlar. Sayısal argümanı ve ilk dizgeyi bir anahtar olarak kullanarak, gerçeklenim, çevirmen tarafından belirlenen doğru çoğul biçim kurallarını seçer. Normal **gettext** davranışına benzer şekilde, bir ileti kataloğu bulunamadığı zaman bu iki dizge dönüş değerini oluştururlar. Germen dili kuralları gereğince, ilk dizgenin tekil biçimde, ikinci dizgenin çoğul biçimde olduğu varsayılır.

Bunun sonucu olarak, dil katalogları olmayan yazılımlar iletileri Germen dili kullanılıyormuş gibi basarlar. Bu bir sınırlama olarak görülebilir ancak, GNU C Kütüphanesi (keza GNU **gettext**) GNU paketinin bir parçası olarak yazılmıştır ve GNU projelerinin kodlama standartları yazılımın İngilizce yazılmasını gerektirdiğinden, kaçınılmaz olarak bu böyle olacaktır.

```
char *ngettext(const char        *msgid1,                                  işlev
               const char        *msgid2,
               unsigned long int n)
```

ngettext işlevi ileti kataloğunun bulunması bakımından **gettext** işlevi gibidir. Ancak, iki ek argüman alır. *msgid1* ile belirtilen dizge tekil biçimli ileti üretir. Ayrıca katalogdaki iletiyi aramak için anahtar olarak kullanılır. *msgid2* ile belirtilen dizge ise çoğul biçimli girdi/girdiler üretir. *n* parametresi çoğul biçimlerin sayısını belirler. Bir ileti kataloğu bulunamamışsa, işlev, **n == 1** ise *msgid1* ile, aksi takdirde *msgid2* ile döner.

İşlevin kullanımına bir örnek:

```
printf (ngettext ("%d file removed", "%d files removed", n), n);
```

n değerinin **printf** işlevinde de kullanıldığına dikkat edin. Sadece **ngettext** işlevinin argümanı olması yeterli değildir.

```
char *dngettext(const char        *alanadı,                                 işlev
                const char        *msgid1,
                const char        *msgid2,
                unsigned long int n)
```

dngettext işlevi ileti kataloğunun seçilmesi bakımından **dgettext** işlevi gibidir. Ancak, iki ek argüman alır. Tekil/Çoğul dizgelerin elde edilmesi bakımından da **ngettext** işlevi gibidir.

```
char *dcngettext(const char        *alanadı,                                 işlev
                 const char        *msgid1,
                 const char        *msgid2,
                 unsigned long int n,
                 int               kategori)
```

dcngettext işlevi ileti kataloğunun seçilmesi bakımından **dcgettext** işlevi gibidir. Ancak, iki ek argüman alır. Tekil/Çoğul dizgelerin elde edilmesi bakımından da **ngettext** işlevi gibidir.

2.1.4. Çoğul Biçimler Sorunu

Bir önceki bölümde sorun ile ilgili bir açıklama yaplmıştı. Şimdi burada sorunun nasıl çözümlendiğini açıklayacağız. Dilbilimciler olmaksızın, desteklenen her dil için çoğul biçimlerin neye göre şekillendiği, kaç çoğul biçim olduğu gibi bilgileri elde etmek mümkün olmazdı.

Bu bakımdan, çoğul biçimlerin seçim kurallarını belirlemeleri çevirmenlere bırakılarak çözüme ulaşıldı. Formül her dil için değişiklik gösterdiğinden, bilgiyi kodun içine yerleştirerek bu mümkün oldu (yeni dillerin kullanımını engellememek için hala bazı genişletmeler gerekiyor). Bu konu GNU **gettext** kılavuzunda daha ayrıntılı bulunabilir. Burada daha az bilgi vermekle yetineceğiz.

Çoğul biçimin seçimi hakkındaki bilgi ileti kataloğunun başlık kısmında (Başlık, boş bir **msgid** dizgesidir) şuna benzer bir girdi olarak bulunur:

```
Plural-Forms: nplurals=2; plural=n == 1 ? 0 : 1;
```

nplurals değeri bu dilde kaç çoğul biçin olduğunu belirten bir tamsayı olmalıdır. **plural** ise C dilinin sözdizimine uygun bir ifade olmalıdır. Ancak bu ifadede, negatif sayılar bulunmamalı, sayılar onluk tabanda olmalı ve değişken olarak sadece **n** bulunmalıdır. Bu ifade **ngettext**, **dngettext** veya işlevlerinden biri ile değerlendirilecektir. Bu işlevlere aktarılan **n** değeri bu ifadedede yerine konarak değerlendirilir. Sonuç, **nplurals** değerinden küçük, eşit veya büyük olmalıdır.

Bu noktada bilinen kurallar aşağıda dil ailelerine göre sınıflandırılarak listelenmiştir. Fakat, listeden de görüleceği gibi kurallar bir dil ailesinin tamamına genelleştirilememektedir.[4]

Sadece bir biçim:
Bazı dillerde tekil/çoğul biçim farkı yoktur. Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=1; plural=0;
```

Bu özellikteki diller:

Uralca/Fin–Uygur Dil Ailesi
Macarca

Asya Dilleri
Japonca, Korece

Altay/Türkçe Dil Ailesi
Türkçe[5]

Biri tekil biçim olmak üzere iki biçim
Yazılımların çoğu İngilizce olduğundan bu biçimi kullanır. Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=2; plural=n != 1;
```

(Not: Burada sıfır ya da bir ile sonuçlanan mantıksal C ifadesi kullanılmıştır.)

Bu özellikteki diller:

Germen Dil Ailesi
Danca, Felemenkçe, İngilizce, Almanca, Norveççe, İsveççe

Uralca/Fin–Uygur Dil Ailesi
Estonca, Fince

Helen Dil Ailesi

Yunanca

Batı Samî/Kenan Dil Ailesi
İbranice

İtalik/Latin Dil Ailesi
İtalyanca, Portekizce, İspanyolca

Hint–Arî Dil Ailesi
Esperanto

Biri 0 ve 1 için tekil biçim olmak üzere 2 biçim
Dil ailesindeki olağandışı durum. Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=2; plural=n>1;
```

Bu özellikteki diller:

İtalik/Latin Dil Ailesi
Fransızca, Brezilya Portekizcesi

Sıfır için özel bir durum ile üç biçim
Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=3; plural=n%10==1 && n%100!=11 ? 0 : n != 0 ? 1 : 2;
```

Bu özellikteki diller:

Baltık Dil Ailesi
Latviya dili

1 ve 2 için özel bir durum ile üç biçim
Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=3; plural=n==1 ? 0 : n==2 ? 1 : 2;
```

Bu özellikteki diller:

Keltçe Dil Ailesi
İrlanda Dili

1[2–9] ile biten sayılar için özel bir durum ile üç biçim
Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=3; \
    plural=n%10==1 && n%100!=11 ? 0 : \
           n%10>=2 && (n%100<10 || n%100>=20) ? 1 : 2;
```

Bu özellikteki diller:

Baltık Dil Ailesi
Litvanya dili

1[1–4] ile bitenler hariç; 1, 2, 3, 4 ile biten sayılar için özel bir durum ile üç biçim
Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=3; \
    plural=n%100/10==1 ? 2 : n%10==1 ? 0 : (n+9)%103 ? 2 : 1;
```

Bu özellikteki diller:

İslav Dil Ailesi
Hırvatça, Çekçe, Rusça, Ukrayna dili

1, 2, 3, 4 için özel bir durum ile üç biçim
Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=3; \
     plural=(n==1) ? 1 : (n>=2 && n<=4) ? 2 : 0;
```

Bu özellikteki diller:

İslav Dil Ailesi
Slovakça

1 için ve 2, 3 veya 4 ile biten bazı sayılar için özel bir durum ile üç biçim
Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=3; \
    plural=n==1 ? 0 : \
           n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 : 2;
```

Bu özellikteki diller:

İslav Dil Ailesi
Lehçe

1 için ve 02, 03 veya 04 ile biten tüm sayılar için özel bir durum ile dört biçim
Böyle bir dil için ileti kataloğunun başlığında şu satır bulunur:

```
Plural-Forms: nplurals=4; \
    plural=n%100==1 ? 0 : n%100==2 ? 1 : n%100==3 || n%100==4 ? 2 : 3;
```

Bu özellikteki diller:

İslav Dil Ailesi
Slovence

2.1.5. `gettext`'te karakter kümesi dönüşümü

gettext sadece bir ileti kataloğunda çeviri aramaz, ayrıca çeviriyi çıktı karakter kümesine anında dönüştürür. Kullanıcı, ileti kataloğunda kullanılan karakter kümesinden farklı bir karakter kümesi kullanıyorsa, bu faydalı bir özelliktir.

Çıktı karakter kümesi öntanımlı olarak, geçerli yerelin **LC_CTYPE** kategorisine bağımlı **nl_langinfo (CODESET)** değeridir. Ancak, bir yazılımın iletileri yerelden bağımsız bir kodlama (UTF–8) ile saklanıyorsa, **gettext** ailesindeki işlevler çevirinin kodlamasını **bind_textdomain_codeset** işlevini kullanarak bu kodlamaya dönüştürür.

gettext işlevinin *msgid* argümanında verilen dizgenin karakter dönüşümüne konu olmadığını unutmayın. Eğer **gettext**i *msgid*'nin karşılığı olan çeviriyi bulamazsa *msgid* dizgesini dönüştürmeksizin döndürür. Bu bakımdan tüm *msgid* dizgelerinin US–ASCII dizgeler olması tercih edilmelidir.

```
char *bind_textdomain_codeset(const char *alanadı,                    işlev
                              const char *karakter_kümesi)
```

bind_textdomain_codeset işlevi *alanadı* ileti kataloglarının çıktı karakter kümesini belirtmek için kullanılabilir. *karakter_kümesi* argümanı ya **iconv_open** işlevinde kullanılabilen geçerli bir karakter kümesi dizgesine bir gösterici ya da bir boş gösterici olmalıdır.

karakter_kümesi olarak boş gösterici belirtilmişse, işlev *alanadı* isimli alanın o an seçilmiş karakter kümesi ile döner. **NULL** dönmüşse, henüz seçilmiş bir karakter kümesi yok demektir.

bind_textdomain_codeset işlevi defalarca kullanılabilir. Aynı *alanadı* ile yapılan her çağrı bir önceki kodlamayı değiştirir.

bind_textdomain_codeset işlevi seçilen karakter kümesininin ismini içeren bir dizgeye bir gösterici döndürür. Dizge işlev tarafından dahili olarak ayrıldığından yazılımcı tarafından değiştirilmemelidir. Eğer sistem **bind_textdomain_codeset** çalışırken bir bellek çıktısı (core dosyası) çıktılarsa, işlev **NULL** ile döner ve *errno* genel değişkenine ilgili hata durumu atanır.

2.1.6. GUI Yazılımlarının Sorunları

gettext işlevlerinin büyük sorunlara yol açtığı kullanım yerlerinden biri görsel arayüzlü (GUI) yazılımlardır. Burada sorun çevilecek iletilerin çoğunun çok kısa olması gerekliliğinden kaynaklanır. Bunlara genellikle menülerde rastlanır. Bu dizgeler tam bir cümle içeremedikleri gibi bir dilde aynı olan dizgelerin diğer dillerde farklı olabilmesi sorun oluşturur. Bu tek sözcüklük dizgelerde iyice belirginleşir.

Çoğu kimse, **gettext** yaklaşımı yerine bu sorunun görülmediği **catgets** işlevlerinin kullanılması gerektiğini söyler. Ancak, bu sorunları **gettext** işlevleri kullanıldığında aşmanın çok basit ve oldukça güçlü bir yöntemi vardır.

Görsel arayüzlü bir yazılımın şöyle bir menüsü olduğunu varsayalım:

```
+------------+------------+---------------------------------------+
| File       | Printer    |                                       |
+------------+------------+---------------------------------------+
| Open     | | Select   |
| New      | | Open     |
+----------+ | Connect  |
             +----------+
```

Kodun bazı yerlerinde **gettext** ailesinden işlevlerde **File**, **Printer**, **Open**, **New**, **Select** ve **Connect** dizgelerinin çevirilerine erişildiğini varsayalım. Ancak dikkat ederseniz iki yerde işleve aktarılan dizge **Open** olacaktır. Yukarıda açıklanan açmazdan dolayı bu dizgenin çevirileri aynı olmayabilecektir.

Bu sorunun çözümlerinden biri belirsizliği ortadan kaldırmak için dizgeyi yapay olarak uzatmak ama çevirmenden sadece asıl dizgeyi çevirmesini istemektir. Fakat bir çeviri bulunamadığında yazılım ne yapacak? Uzatılan dizge basılamaz. O halde işlevin biraz değiştirilmiş bir sürümünü kullanmamız gerekir.

Dizgenin uzatılmasında tektip bir yönteme ihtiyaç vardır. Örneğin, yukarıdaki örnek için uzatılmış dizgeler şöyle seçilebilir:

```
Menu|File
Menu|Printer
Menu|File|Open
Menu|File|New
Menu|Printer|Select
Menu|Printer|Open
Menu|Printer|Connect
```

Böylece dizgelerin hepsi farklı oldu. Eğer **gettext** işlevini aşağıdaki işlevle kullanırsak herşey istendiği gibi olacaktır:

```
char *
sgettext (const char *msgid)
{
  char *msgval = gettext (msgid);
  if (msgval == msgid)
    msgval = strrchr (msgid, '|') + 1;
  return msgval;
}
```

Bu küçük işlevin tek yaptığı bir çevirinin bulunmadığı zaman uzatılmış dizgeden özgün dizgeyi elde etmektir. Dönen değer girdi değeri olduğundan bu işlem doğrudan bir gösterici karşılaştırması olarak yapıldığında daha verimli olur. Bir çevirinin bulunmaması halinde girdi dizgesinin `|` karakterini içereceğini bildiğimize göre dizge içinde bu karakterin son görüldüğü yeri bulup bu noktaya bir gösterici döndürmek yeterli olacaktır.

Eğer uzatılmış dizgeleri tutarlı şekilde kullanır ve **`gettext`** çağrılarını **`sgettext`** çağrıları ile değiştirirsek (bunu sadece görsel yazılımda gerektiği yerde yapmak yeterlidir), uluslararasılaştırılabilen bir yazılım üretebiliriz.

GNU C gibi ileri düzey derleyicilerle **`sgettext`** işlevi bir satıriçi işlev veya bunun gibi bir makro olarak gerçeklenebilir:

```
#define sgettext(msgid) \
  ({ const char *__msgid = (msgid);             \
     char *__msgstr = gettext (__msgid);        \
     if (__msgval == __msgid)                   \
       __msgval = strrchr (__msgid, '|') + 1;   \
     __msgval; })
```

Diğer **`gettext`** işlevleri de (**`dgettext`**, **`dcgettext`** ve **`ngettext`**) benzer şekilde bir sarmalayıcı işlevle kullanılabilir.

Şimdi bir soru gelecek akıllara: GNU C kütüphanesinde neden böyle işlevler yok? Bu sorunun iki şıklı bir yanıtı var:

- Yazılması kolaydır, bu bakımdan kullanıldıkları projenin içinde yazılabilirler. Bu bir yanıt değil derseniz, ikinci şık ile birlikte değerlendirildiğinde birşey ifade eder:
- C kütüphanesinde heryerde çalışabilecek bir sürümün bulunmasını sağlayacak bir yöntem yok. Sorun, uzatilmiş dizgede asıl dizgenin başına eklenen dizgeleri arasındaki karakterin seçiminden kaynaklanıyor. Yukarıdaki örnekte kullanılan `|` karakteri bu tür bağlamlarda sıkça kullanılan bir ayraç karakteri olarak iyi bir seçimdir ama ileti dizgelerinde de sık kullanılan bir karakterdir.

  Eğer bu ayraç karakteri ileti dizgesinde de kullanılmışsa ne olacak? Ya da ayraç karakteri kodun derlendiği makinada yoksa ne olacak (örneğini ISO C için `|` karakteri gerekli değildir; bu nedenle ISO C yazılım geliştirme ortamında ayrıca bir `iso646.h` dosyası vardır).

Sadece bir açıklama daha kaldı. Yukarıdaki sarmalayıcı işlev uzatılmış dizgenin tamamının çevrilmeyeceği esasına dayandırılmıştır. Mantık bunu gerektirdiği için böyle yapılmıştır. Çeviri, arama için bir anahtar olarak kullanılmadığından neyi nasıl içerdiğinin bir önemi yoktur, hem gereksiz bellek harcamanın da alemi yok.

2.1.7. `gettext` kullanan yazılımların kullanımı

Geçtiğimiz bölümde yazılımcının yazılımının iletilerini uluslararasılaştırmak için ne yapabileceği açıklandı. Ancak, sonuçta görmek istediği iletiyi kullanıcı seçecektir. Kullanıcı onları anlayabilmelidir.

POSIX yerel modeli, kullanılan yereli seçmede **`LC_COLLATE`**, **`LC_CTYPE`**, **`LC_MESSAGES`**, **`LC_MONETARY`**, **`LC_NUMERIC`** ve **`LC_TIME`** ortam değişkenlerini kullanır. Evvelce açıklandığı gibi **`gettext`** ayrıca bunları da kullanır.

Bunun nasıl olduğunu anlayabilmek için bir ileti kataloğunun yerini bulmakta kullanılan dosya isminin hangi bileşenlerden oluştuğuna bakalım:

dizin_ismi/*yerel_dil*/LC_*kategori*/*alan_ismi*.mo

dizin_ismi için öntanımlı değer sisteme özeldir. C kütüphanesi yapılandırılırken `--prefix` seçeneği ile belirtilen dizine göre hesaplanır. `--prefix` değeri normalde **/usr** ya da **/** dizinidir. *dizin_ismi* ise genellikle:

`/usr/share/locale`

dizinidir. İleti kataloglarını içeren `.mo` dosyaları sistemden bağımsız dosyalar olduklarından **/usr/share** öneki tercih edilmiştir. **bindtextdomain** işlevinin ikinci parametresinde sadece *dizin_ismi* ile belirtilen parça verilir. İşlev dosya yolunun geri kalanını aşağıda açıklandığı gibi oluşturur.

kategori yerel kategorisinin ismidir. **gettext** ve **dgettext** için bu daima **LC_MESSAGES**'dır. **dcgettext** işlevinde ise bu değer üçüncü argümanda belirtilir. Ancak evvelce da açıklandığı gibi **LC_MESSAGES** kategorsinden başka bir kategori belirtmekten kaçınılmalıdır.

yerel_dil elemanı kategoriyi içeren dildir. Bu değer **setlocale** işlevindeki gibi kullanıcının dil seçimine bağlı olarak belirlenir. Bazı ortam değişkenlerine belli bir öncelik sırasına göre bakarak bu değer belirlenir. Bu sıra şöyledir:

```
LANGUAGE
LC_ALL
LC_xxx
LANG
```

Çok bildik görünüyor. Bu sıralama **LANGUAGE** ortam değişkeni dışında **setlocale** işlevinin kullandığı sıralama ile aynıdır. Peki ama neden **LANGUAGE** değişkenine bakıyoruz?

Bunun sebebi bu değişkenlerin değerlerinden elde edilecek sözdiziminin **setlocale** işlevinden farklı olmasıdır. Eğer **LC_ALL** değişkenine aşağıdaki genişletilmiş sözdizimine uygun bir değer atarsak, **setlocale** işlevi bu değeri asla kullanamayacaktır. Bir ek değişkenle bu sorun aşıldığı gibi, bu değişken sayesinde kullanıcının ileti dilini yerel ayarlarından bağımsız olarak belirleyebilme imkanı vardır.

LC_xxx değişkenlerinin değerleri sadece bir yerele ait belirtimi içerebilirken **LANGUAGE** değişkeninin değerinde çok sayıda dil iki nokta üstüstelerle ayrılarak belirtilebilir. Böylece kullanıcı tercih ettiği dilleri bu değişken sayesinde bir öncelik sırasıyla belirtebilir.

Dosya ismini oluşturan son parça olan *alan_ismi* ya **textdomain** işlevi ile etkinleştirilen ya da **dgettext** veya **dcgettext** işlevinde ilk parametre olarak blirtilen yazılım kodunun alan ismidir; başka bir deyişle yazılımcının pakete verdiği isimdir. Örneğin GNU C kütüphanesi için alan ismi **libc**'dir.

Yazılımcının bu işlemi nasıl yapacağını gösteren bir örnek kod parçası:

```
{
  setlocale (LC_ALL, "");
  textdomain ("test-package");
  bindtextdomain ("test-package", "/usr/local/share/locale");
  puts (gettext ("Hello, world!"));
}
```

Yazılım ilk çalıştırıldığında öntanımlı alan **messages**, öntanımlı yerel "C"dir. **setlocale** çağrısı yereli kullanıcının atadığı ortam değişkenlerine bağlı olarak değiştirir. Yukarıda bahsedildiği gibi İleti kataloglarının yerini bulmak için **gettext** işlevleri **LC_MESSAGES** kategorisini, karakter kümesi dönüşümü için **LC_CTYPE** kategorisini kullanıyordu. **setlocale** çağrısı ile bunlar kullanıcının seçimlerine göre belirlenmiş oluyor. **textdomain** çağrısı ile **test-package** alan isimli ileti kataloğunun kullanılacağını, **bindtextdomain** çağrısı ile de bu kataloğun bulunacağı yeri **/usr/local/share/locale** olarak belirtmiş oluyoruz.

Kullanıcı **LANGUAGE** ortam değişkenine **tr** değerini atarsa **gettext** işlevi

`/usr/local/share/locale/tr/LC_MESSAGES/test-package.mo`

iletiği kataloğunu kullanacaktır.

Bu örnekte **LANGUAGE** ortam değişkenine **tr** değerini atadığımızı varsaydık. Eğer kullanıcı bunun yerine değeri `tr_TR.UTF-8` olan **LC_ALL** ortam değişkenini kullanmak isteseydi ne olacaktı? Yukarıda bu gibi durumlara pek sık rastlanmadığından bahsetmiştik. Örneğin, kullanıcının tercih ettiği dil olarak son çare resmi dil değilse kullanıcı kendi lehçesini tercih edebilir.

gettext bu gibi duramlara da hazırlıklıdır. İşlevler bu değişkenin değerinin biçimini tanır. Değeri parçalarına ayırarak içinden kendi kullanacağı değeri alır. Bu işlem şüphesiz bir önkabule dayanır. Ortam değişkeninin değerini oluşturan biçim az ya da çok standarttır. X/Open belirtimi şöyledir:

dil[_ülke[.karakterkümesi]][@değiştirici]

Yerel ismi şu sıraya göre ayıklanır:

1. **karakterkümesi**
2. **normalleştirilmiş karakterkümesi**
3. **ülke**
4. **değiştirici**

dil alanı bilinen sebeplerle asla kaldırılmayacaktır.

Burada yeni olan **normalleştirilmiş karakterkümesi** girdisidir. Çoğu kişi standart haline gelmiş isimleri yanlış hatırlar ve örneğin ISO–8859–9 yerine iso8859–9, iso_8859–9, 88599, 8859–9 gibi değerler yazabilirler. **normalleştirilmiş karakterkümesi** değeri şu kurallar izlenerek oluşturulur:

1. Sayılar ve harfler dışında tüm karakterler kaldırılır.
2. Harflerin tamamı küçük harfe dönüştürülür.
3. Rakamlar **"iso"** dizgesi ile öncelenmemişse öncelenir.

Örneğimizdeki isim **iso88599** olarak normalleştirilir. Böylece kullanıcıya yerel ismini belirtirken daha fazla serbestlik sağlanmış olur.

Ancak bu kadar genişletilmiş işlevsellik bile aynı dili belirtirken kullanılabilen farklı isimler için bir çözüm sağlamaz (örneğin, **tr** ve **turkish**). Hem yerel gerçeklemesi hem de **gettext** bu gibi durumlarda yardımseverdir.

/usr/share/locale/locale.alias dosyası (C kütüphanesini derlerken **/usr** yerine ne belirttiyseniz, **/usr** yerine onu yazın) dil isimleriyle yerel isimlerini eşleştiren bir liste içerir. Sistem yöneticisi bu listeye yeni girdiler eklemekte özgürdür. Ortam değişkenleri ile seçilmiş isim bu listedekilerle karşılaştırılır. Bir eşleşme bulunursa ikinci sütundaki değer kullanılır.

2.2. gettext için Yardımcı Uygulamalar

GNU C kütüphanesi **gettext** işlevleri için yazılımın kaynak kodundan ileti kataloğunu oluşturacak yazılımları içermez. GNU projesinin bir parçası olarak GNU gettext paketi geliştiricinin ihtiyaç duyabileceği herşeyi içerir. Bu paketle sağlanmış araçlar, **catgets** işlevlerinin ileti kataloğunu elde eden **gencat** uygulamasından daha fazla yetenekli olmayı gerektirir.

gencat uygulamasına eşdeğer olarak **msgfmt** uygulaması gösterilebilir. Metin biçimindeki ileti kataloglarından **gettext** işlevlerinin kullanabileceği ikilik biçimdeki ileti kataloglarını elde etmekte kulanılır. Ama sırf bu değil, pakette daha pek çok uygulama vardır.

xgettext uygulaması bir kaynak dosyasından çevrilebilir dizgeleri toplamakta kullanılır. Yani yazılımcı geliştirme sırasında ileti kataloğunu elle oluşturmak zorunda değildir. Bu uygulama girdinin daha iyi anlaşılabilmesini ve çıktının özelleştirilebilmesini sağlayan seçenekler içerir.

Paket, sadece yazılımın geliştiricisinin değil çevirmenlerin de kullanabileceği araçları içerir. GNU **gettext** Kılavuzunda bunlar hakkında ayrıntılı bilgi bulabilirsiniz.

IX. Arama ve Sıralama

İçindekiler

Bu kısımda keyfi nesne dizilerinde arama ve sıralama için kullanılan işlevlere yer verilmiştir. Bu işlevlere dizi üyelerinin sayısı ve dizideki nesnelerin boyutları ile birlikte uygun bir karşılaştırma işlevi argüman olarak verilir.

1. Karşılaştırma İşlevinin Tanımlanması

Sıralı dizi kütüphanesi işlevlerinin kullanılması sırasında dizi elemanlarının nasıl karşılaştırılacağını belirtmelisiniz.

Bunu yapmak için, dizideki iki elemanları karşılaştırıcak bir karşılaştırma işlevi sağlamalısınız. Kütüphane bu işlevi çağırırken karşılaştırılacak dizi elemanlarını işleve argüman olarak belirtecektir. Karşılaştıma işleviniz **strcmp** (*Dizi/Dizge Karşılaştırması* (sayfa: 104)) işlevinin yaptığı gibi bir değer döndürmelidir: ilk argüman ikincisinden küçükse negatif, büyükse pozitif değer döner.

Aşağıdaki örnekte, **double** türünden sayılardan oluşan bir dizi ile çalışan bir karşılaştırma işlevi vardır:

```
int
compare_doubles (const void *a, const void *b)
{
  const double *da = (const double *) a;
  const double *db = (const double *) b;

  return (*da > *db) - (*da < *db);
}
```

Karşılaştırma işlevlerinin veri türü olan isim `stdlib.h` başlık dosyasında tanımlanmıştır. Bu veri türü bir GNU oluşumudur.

```
int comparison_fn_t (const void *, const void *);
```

2. Dizi Arama İşlevleri

Genel olarak bir dizi içindeki belli bir elemanın aranması, hemen hemen tüm elemenlara bakılmasını gerektirir. GNU C kütüphanesi düzgün doğrusal arama yapan işlevleri içerir. Aşağıdaki iki işlevin prototipleri `search.h` başlık dosyasında bulunabilir.

```
void *lfind(const void       anahtar,                                  işlev
            void            *başlangıç,
            size_t          *üye–sayısı,
            size_t           boyut,
            comparison_fn_t  karş–işlevi)
```

Bu işlev, *başlangıç*'dan itibaren *boyut* baytlık ***üye–sayısı** elemanlı bir dizide *anahtar* tarafından gösterilen bir eşleşmeyi içeren elemanı arar. *karş–işlevi* ile gösterilen işlev iki elemanın karşılaştırılması için kullanılır.

Dönen değer, *başlangıç* da başlayan dizideki bulunmuşsa eşleşen eleman için bir gösterici ile döner, eşleşen eleman yoksa **NULL** döner.

Bu işlevin ortalama çalışma süresi ***üye–sayısı** /2 dir. Bu işlev, genellikle sadece bir diziye eklenen ya da diziden silinen elemanları almak için kullanılır, bu durumda arama yapmadan önce dizinin sıralanması pek kullanışlı olmayabilir.

```
void *lsearch(const void       anahtar,                                    işlev
              void            *başlangıç,
              size_t          *üye–sayısı,
              size_t           boyut,
              comparison_fn_t  karş–işlevi)
```

Bu işlev **lfind** işlevine benzer. Verilen dizide bir elemanı arar ve bulduğunda bu elemanla döner. Farklı olan, bir eşleşme bulunamazsa, *anahtar* ile gösterilen nesneyi (*boyut* baytlık olarak) dizinin sonuna ekler ve bu eklemeyi belirtmek için ***üye–sayısı** değerini bir arttırır.

İşlevi çağıran açısından bunun anlamı, dizinin elemanı içerdiğinden emin olamıyorsanız, *başlangıç* da başlayan dizi için ayrılan bellekte en azından *boyut* baytlık kullanılmamış alanın bulunmasının gerekliliğidir. **lsearch** işlevi çağrılırken daima bellekte yeterli yerin bulunması gerektiğinden, dizinin aranan elemanı içerdiğinden eminseniz **lfind** işlevini kullanmak daha iyidir.

Anahtarla eşleşen bir elemanı sıralanmış bir dizide aramak için **bsearch** işlevini kullanın. Bu işlevin prototipi `stdlib.h` başlık dosyasındadır.

```
void *bsearch(const void       anahtar,                                    işlev
              const void      *dizi,
              size_t           üye–sayısı,
              size_t           boyut,
              comparison_fn_t  karş–işlevi)
```

Bu işlev, *boyut* baytlık *üye–sayısı* eleman içeren önceden sıralanmış *dizi* dizisinde *anahtar*'a eşdeğerde bir nesneyi arar.

karş–işlevi, karşılaştırma yapmak için kullanılır. Bu işlev iki argümanla çağrılabilmeli ve birinci argümanın ikinci argümandan küçük, büyük ya da ona eşit olması durumlarına bağlı olarak sırasıyla sıfırdan küçük, büyük bir değerle ya da sıfır ile dönmelidir. *dizi* dizisinin elamanları bu karşılaştırma işlevi ile ilgili olarak artan sırada sıralanmış olmalıdır.

Eşleşme bulunmuşsa dönen değer eşleşen dizi elemanına bir göstericidir, aksi takdirde boş gösterici döner. Dizi birden fazla eşleşme içeriyorsa, dönen değerin hangi eleman olacağı belirlenmemiştir.

İşlev ikilik arama (binary search) algoritması kullanarak gerçeklendiğinden işlevin ismi bu algortimanın isminden türetilmiştir.

3. Dizi Sıralama İşlevi

Vereceğiniz bie karşılaştırma işlevi kullanılarak bir diziyi sıralamak isterseniz, **qsort** işlevi bu iş için biçilmiş kaftandır. Bu işlevin prototipi `stdlib.h` başlık dosyasındadır.

```
void qsort(void            *dizi,                                    işlev
           size_t           üye–sayısı,
           size_t           boyut,
           comparison_fn_t  karş–işlevi)
```

Bu işlev, *boyut* baytlık *üye–sayısı* eleman içeren *dizi* dizisini sıralar.

karş–işlevi, karşılaştırma yapmak için kullanılır. Bu işlev iki argümanla çağrılabilmeli ve birinci argümanın ikinci argümandan küçük, büyük ya da ona eşit olması durumlarına bağlı olarak sırasıyla sıfırdan küçük, büyük bir değerle ya da sıfır ile dönmelidir.

Uyarı

İki nesnenin karşılaştırması eşitlikle sonuçlanıyorsa, sıralama sonrası hangisinin önce olacağı kestirilemez. Bu durumda sıralamanın kararsız olduğundan söz edilebilir. Bu sadece karşılaştırma elemanların belli bir bölümünde yapıldığında bir fark oluşturur. İki elemanın aynı sıralama anahtarı olması başka bakımlardan bir fark oluşturabilir.

Kararlı sıralama istiyorsanız, karşılaştırma işlevini yazarken bu sonucun elde edilmesini sağlayabilirsiniz. İki eleman arasında başka kriterlere göre ayrım yapılabiliyorsa buna göre ek bir karşılaştırma yaparsınız ya da adreslerini karşılaştırırsınız. Ancak bu işlemler sıralama algoritmasının verimini düşüreceğinden çok gerekli olmadıkça yapmamanız önerilir.

Aşağıda **double** türünden sayılardan oluşan bir dizi ile çalışan bir sıralama örneği vardır. Kullanılan karşılaştırma işlevi *Karşılaştırma İşlevinin Tanımlanması* (sayfa: 203) bölümündeki örnekte tanımlanmıştı:

```
{
  double *dizi;
  int boyut;
  …
  qsort (dizi, boyut, sizeof (double), compare_doubles);
}
```

İşlev çabuk sıralama (quick sort) algoritması kullanarak gerçeklendiğinden işlevin ismi bu algortimanın isminden türetilmiştir.

Bu kütüphanedeki **qsort** gerçeklemesi sadece dizinin kapladığı bellek bölgesi ile sınırlı kalmayabilir ve diziyi saklamak için fazladan bellek kullanabilir.

4. Arama ve Sıralama Örneği

Bu örnekte, bir yapılar dizisi ile **qsort** ve **bsearch** işlevlerinin kullanımı gösterilmiştir. Dizideki nesneler **name** alanlarına göre **strcmp** işlevi kullanılarak sıralanmakta ve nesnelere isimleri ile bakılmaktadır.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Sıralanacak gösteri karakterlerine bir dizi tanımlayalım. */

struct critter
{
  const char *name;
  const char *species;
};
```

```
struct critter muppets[] =
{
  {"Kermit", "frog"},
  {"Piggy", "pig"},
  {"Gonzo", "whatever"},
  {"Fozzie", "bear"},
  {"Sam", "eagle"},
  {"Robin", "frog"},
  {"Animal", "animal"},
  {"Camilla", "chicken"},
  {"Sweetums", "monster"},
  {"Dr. Strangepork", "pig"},
  {"Link Hogthrob", "pig"},
  {"Zoot", "human"},
  {"Dr. Bunsen Honeydew", "human"},
  {"Beaker", "human"},
  {"Swedish Chef", "human"}
};

int count = sizeof (muppets) / sizeof (struct critter);


/* Sıralama ve arama için kullanılacak karşılaştırma işlevi. */

int
critter_cmp (const struct critter *c1, const struct critter *c2)
{
  return strcmp (c1->name, c2->name);
}


/* Bir karakter hakkında bilgi basalım. */

void
print_critter (const struct critter *c)
{
  printf ("%s, the %s\n", c->name, c->species);
}

/* Sıralanmış dizi üzerinde arama yapalım. */

void
find_critter (const char *name)
{
  struct critter target, *result;
  target.name = name;
  result = bsearch (&target, muppets, count, sizeof (struct critter),
                    critter_cmp);
  if (result)
    print_critter (result);
  else
    printf ("Couldn't find %s.\n", name);
}

/* Main işlevi. */
```

```
int
main (void)
{
  int i;

  for (i = 0; i < count; i++)
    print_critter (&muppets[i]);
  printf ("\n");

  qsort (muppets, count, sizeof (struct critter), critter_cmp);

  for (i = 0; i < count; i++)
    print_critter (&muppets[i]);
  printf ("\n");

  find_critter ("Kermit");
  find_critter ("Gonzo");
  find_critter ("Janice");

  return 0;
}
```

Yazılımın çıktısı:

```
Kermit, the frog
Piggy, the pig
Gonzo, the whatever
Fozzie, the bear
Sam, the eagle
Robin, the frog
Animal, the animal
Camilla, the chicken
Sweetums, the monster
Dr. Strangepork, the pig
Link Hogthrob, the pig
Zoot, the human
Dr. Bunsen Honeydew, the human
Beaker, the human
Swedish Chef, the human

Animal, the animal
Beaker, the human
Camilla, the chicken
Dr. Bunsen Honeydew, the human
Dr. Strangepork, the pig
Fozzie, the bear
Gonzo, the whatever
Kermit, the frog
Link Hogthrob, the pig
Piggy, the pig
Robin, the frog
Sam, the eagle
Swedish Chef, the human
Sweetums, the monster
Zoot, the human

Kermit, the frog
```

```
Gonzo, the whatever
Couldn't find Janice.
```

5. İsim–Değer Çiftleri ile Arama İşlevi

Bu kısma gelinceye kadar sıralı ve sırasız dizilerde arama yapılmasından bahsedildi. Bilgileri aramadan önce düzenlemek için başka yöntemler de vardır. Bilgi girme, silme ve arama maliyetleri farklıdır. Olası bir gerçekleme de isim–değer çiftleri tablosu (hashing table)kullanmaktır. Bu kısımda bahsedilecek işlevler `search.h` başlık dosyasında bildirilmiştir.

`int` **`hcreate`**`(size_t` *sayı*`)` işlev

`hcreate` işlevi en az *sayı* eleman içeren bir isim–değer çifti tablosu oluşturur.Bu tabloyu daha sonra genişletmek mümkün olmadığından eleman sayısı akıllıca seçilmelidir. Bu işlevi gerçeklemekte kullanılan yöntemler tablodaki eleman sayısının kullanılması olası en büyük elemen sayısından daha büyük olarak belirlenmesini gerektirir. %80 den fazlası dolu olan isim–değer çifti tabloları çalışmak için yetersiz olur. Yöntem tarafından garanti edilen sabit erişim süresi bir kaç çakışma mevcut olduğunda aşılabilir. Bu konuda daha fazla bilgi isterseniz Knuth'un "The Art of Computer Programming, Part 3: Searching and Sorting" adlı eserine bakınız.

Bu işlevin en zayıf tarafı bir yazılım için en çok bir tablonun olabilmesidir. Tablo yazılımcının denetimi dışında yerel bellek bölgesinde oluşturulur. GNU C kütüphanesi, bu arayüze benzeyen ve çok sayıda tablonun tutulmasını mümkün kılan evresel (reentrant) bir arayüz ile çalışan ek işlevlere de sahiptir.

Bir yazılım içinde birden fazla isim–değer çifti tablosu kullanımı, biçimsel tablonun bir **`hdestroy`** çağrısı ile kaldırılmasından sonra mümkün olur.

Tablo başarıyla oluşturulmuşsa işlev sıfırdan farklı bir değerle döner. Aksi takdirde ya zaten kullanılan bir tablo vardır ya da yazılım yetersiz bellekle çalışıyordur ki, bu durumda işlev sıfır değeriyle döner.

`void` **`hdestroy`**`(void)` işlev

`hdestroy` işlevi daha önceki bir **`hcreate`** çağrısı ile ayrılan özkaynakları serbest bırakmak için kullanılabilir. Bu işlev çağrıldıktan sonra tekrar bir **`hcreate`** çağrısı yapabilmek ve farklı boyutta bir tablo oluşturmak mümkün olur.

Önemli

`hdestroy` çağrısı sırasında tablo içindeki elemanlar bu işlev tarafından serbest bırakılmaz. Bu, dizgeleri serbest bırakacak yazılım kodunun sorumluluğudur. Tabloda mevcut tüm elemanlarla tek tek uğraşacak bir işlev olmadığından bellekteki elemanların tümünün serbest bırakılması mümkün değildir. Tablo ve tablo içindeki tüm elemanların serbest bırakılması önemliyse, yazılımcı tablo elemanlarının bir listesini tutmalı ve **`hdestroy`** çağrısından önce bu listeyi kullanarak tüm elemanların verilerini serbest bırakmalıdır. Bu çok tatsız bir mekanizmadır ve isim–değer çifti tablolarının bu türünün yazılım tarafından bir kere oluşturulup, yazılım sonlandırılıncaya kadar kullanılacağını gösterir.

Arama işleminin yapılacağı isim–değer çiftleri aşağıdaki veri türü ile oluşturulur:

`struct` **`ENTRY`** veri türü

Bu yapının her iki elemanı da boş karakter sonlandırmalı dizge göstericisidir. Bu durum **`hsearch`** işlevlerinin işlevselliğinin sınırlarını belirler. Sadece boş karakter ile sonlandırılabilen verilerle çalışılabilir, genel ikilik verilerle çalışabilmek mümkün değildir.

`char *`**`key`**
Tablo içindeki elemanı ifade eden ya da arama anahtarı olarak kullanılacak olan boş karakter sonlandırmalı dizgenin göstericisidir (çiftin isim parçası).

`char *`**`data`**
Çiftin değer parçasını oluşturan boş karakter sonlandırmalı dizgenin göstericisidir. İşlevler sadece bir mevcut girdiyi aramak için kullanılırsa, bu eleman kullanılmadığından tanımsız kalabilir.

```
ENTRY *hsearch(ENTRY  girdi,                                                  işlev
               ACTION eylem)
```

Bir isim değer çifti tablosunda arama yapmak için **`hcreate`** işleviyle oluşturulan tabloda arama yapmak için **`hsearch`** işlevi kullanılmalıdır. Bu işlev ya (*eylem* değeri olarak **`FIND`** verimişse) bir elemanı aramakta ya da tabloya yeni bir isim elemanı girmekte kullanılır. Girdiler değiştirilemez.

Yapının isim elemanını oluşturan **`key`** elemanı **`ENTRY`** türünde bir nesnenin göstericisidir. Bir tablo üzerindeki bir girdiyi konumlamak için yapının **`key`** elemanı kullanılır.

`key` ile eşleşen bir girdinin varlığı durumunda *eylem* yoksayılır. Bulunan girdi döner. Bir eşleşme bulunamazsa ve *eylem* değeri **`FIND`** ise işlev boş gösterici ile döner. Bir eşleşme bulunamazsa ve *eylem* değeri **`ENTER`** ise *girdi* ile ilklendirilerek tabloya yeni bir girdi eklenir ve işlev eklenen girdinin göstericisi ile döner.

Şimdiye dek bahsedilen işlevler geneldir ve yazılımda bir defada en çok bir isim–değer çifti tablosunun varolabildiği durumda kullanılabilirler. Bunun aksine durumlarda bir GNU oluşumu olarak aşağıdaki işlevler kullanılabilir. Bu işlevler **`struct hsearch_data`** türünde nesnelerden oluşan bir isim–değer çifti tablosu ile çalışır. Bu veri türü şeffaf değildir, yani üyeleri doğrudan değişirilemez.

```
int hcreate_r(size_t               sayı,                                      işlev
              struct hsearch_data *tablo)
```

`hcreate_r` işlevi en az *sayı* eleman içeren *tablo* isimli isim–değer çiftleri tablosunu ilklendirir. Bu işlev, yazılımcı tarafından denetlenebilir bir tablo oluşturmak dışında **`hcreate`** işlevi gibidir.

Bu işlev bir defada birden fazla isim–değer çiftleri tablosu oluşturulabilmesini mümkün kılar. **`struct hsearch_data`** nesnesi için gereken bellek özdevimli ayrılabilir ancak bu işlevi çağırmadan önce sıfırla doldurularak ilklendirilmelidir.

İşlem başarılıysa işlev sıfırdan farklı bir değerle döner. Sıfır değeri dönmüşse ya birşeyler yanlış gitmiştir ya da yazılım yetersiz bellekle çalışıyordur.

```
void hdestroy_r(struct hsearch_data *tablo)                                   işlev
```

`hdestroy_r` işlevi **`hcreate_r`** işlevi ile oluşturulan *tablo* tablosu tarafından kullanılan tüm kaynakları serbest bırakır. Tablonun içindeki elemanların serbest bırakılması bakımından **`hdestroy`** gibidir.

```
int hsearch_r(ENTRY               girdi,                                      işlev
              ACTION              eylem,
              ENTRY             **dönüşdeğeri,
              struct hsearch_data *tablo)
```

`hsearch_r` işlevi **`hsearch`** işlevine eşdeğerdir. İlk iki argüman aynıdır. Ancak tek bir genel tablo yerine **`hcreate_r`** işlevi ile ilklendirilen ve *tablo* ile gösterilen bir tablo ile çalışır.

`hcreate` işlevinden diğer bir farkı da tabloda bulunan değeri işlevin dönüş değeri olarak değil, *dönüşdeğeri* parametresi tarafından gösterilen bir gösterici değişkeni içinde döndürmesidir. İşlevin geri dönüş değeri işlev başarılı ise sıfırdan farklı, değilse sıfırdır. *errno* genel değişkeni başarısızlığın sebebini gösterir:

ENOMEM
: Tablo doludur, **hsearch_r** işlevi *eylem* değeri olarak **ENTER** belirtilerek, bilinmeyen bir *girdi* ile çağrılmıştır.

ESRCH
: *eylem* parametresi olarak **FIND** belirtilmiş ve tabloda belirtilen *girdi* bulunamamıştır.

6. Ağaç Arama İşlevi

Verimli bir arama yapmak için verileri düzenlemenin bir yolu da ağaçları kullanmaktır. **tsearch** işlev ailesi büyük boyutlardaki verileri eleman sayısının logaritmasından daha kısa bir sürede bulmayı garantileyen hoş bir arayüz sağlar. GNU C kütüphanesinin gerçeklemesi bu sınırı, basit ikilik ağaç gerçeklemelerinin veri girdilerinin sebep olduğu sorunlarda bile aşılmamasını garanti eder.

Bu bölümde açıklanan işlevlerin tümü System V ve X/Open belitimlerinde açıklanmıştır ve dolayısıyla taşınabilirdir.

hsearch işlevlerinin aksine **tsearch** işlevleri sadece boş karakter sonlandırmalı dizgelerle değil her türlü veri ile kullanılabilir.

tsearch işlevleri ayrıca veri yapılarının ilklendirilmesini gerektiren işlevleri içermemek gibi bir ayrıcalığa sahiptir. **NULL** ile ilklendirilmiş **void *** türünde bir göstergeç geçerli bir ağaçtır ve genişletilebilir ya da aranabilirdir. Bu işlevlerin prototipleri `search.h` başlık dosyasındadır.

```
void *tsearch(const void       anahtar,                                  işlev
              void           **kök,
              comparison_fn_t  karş–işlevi)
```

tsearch işlevi **kök* ile gösterilen ağaç içinde *anahtar* ile eşleşen elemanı arar. *karş–işlevi* ile belirtilen işlev iki elemanın karşılaştırılmasında kullanılır. *karş–işlevi* parametresiyle belirtilen işlevlerin özellikleri için *Karşılaştırma İşlevinin Tanımlanması* (sayfa: 203) bölümüne bakınız.

Ağaç içinde *anahtar* ile eşleşen bir girdi bulunamazsa değer ağaca eklenir. **tsearch** işlevi *anahtar* ile gösterilen nesnenin bir kopyasını oluşturmaz (boyutu bilinmediğinden). Bunu yerine ağacın yapısı içinde olduğunu belirtmek için ona bir başvuru ekler.

Ağacın kök düğümünü bazan değiştirmek gerekebileceğinden ağaç, göstericisinin göstericisi olarak belirtilmiştir. Bu nedenle çağrıdan sonra *kök* tarafından gösterilen değişkenin aynı değerde olmayacağı varsayılmalıdır. Bu da **tsearch** işlevinin aynı ağaç için peşpeşe çağrılamayacağı anlamına gelir. Ancak işlevi peşpeşe farklı ağaçlarla çağırmak bir sorun oluşturmaz.

Dönen değer ağaçta eşleşen elemanın göstericisidir. Yeni bir eleman oluşturulmuşsa dönen değer yeni oluşturulan elemanın göstericisidir. Bir girdi oluşturulmuş ama bellek yetmemişse boş gösterici döner.

```
void *tfind(const void       anahtar,                                   işlev
            void *const     *kök,
            comparison_fn_t  karş–işlevi)
```

tfind işlevi **tsearch** işlevine benzer. *anahtar* ile eşleşen elemana bir gösterici ile döner. Eşleşen bir eleman bulunamazsa yeni bir eleman girmez ve boş gösterici ile döner. (*kök* parametresinin bir sabit göstericiyi gösterdiğine dikkat edin).

tsearch işlevlerinin bir özelliği de **hsearch** işlevlerinin aksine elemanların silinebilmesidir.

```
void *tdelete(const void       anahtar,                                    işlev
              void           **kök,
              comparison_fn_t karş–işlevi)
```

anahtar ile eşleşen belli bir elemanı ağaçtan kaldırmak için **tdelete** işlevi kullanılabilir. Eleman silindikten sonra silinen düğümün ata düğümüne bir gösterici döner. Ağaçta silinecek bir eleman bulunamazsa işlev boş gösterici ile döner. İşlev ağacın kökünü silerse dönen değer **NULL** olmayan anlamsız bir değer olabilir.

```
void tdestroy(void        *kök,                                            işlev
              __free_fn_t serb–işl)
```

Arama ağacını tamamen silmek isterseniz **tdestroy** işlevini kullanabilirsiniz. **tsearch** işlevi tarafından oluşturulan ve *kök* ile kökü belirtilen bir ağaca ayrılan tüm kaynakları serbest bırakır.

Ağacın her düğümü için *serb–işl* çağrılır. Veri göstericisi argüman olarak işleve aktarılır. Böyle bir işlem gerekli değilse *serb–işl* hiçbir şey yapmayan bir işlevi göstermelidir. İşlev ne olursa olsun çağrılır.

Bu işlev bir GNU oluşumudur. System V veya X/Open belirtimlerinde yoktur.

Ağaç veri yapısını oluşturan ve yokeden işlevlere ek olarak, ağacın her elemanına uygulanan bir işlev daha vardır. İşlev aşağıdaki türde olmalıdır:

```
void __action_fn_t (const void *düğüm, VISIT değer, int seviye);
```

düğüm, düğümün veri değeridir (**tsearch** işlevine verilen *key* argümanı). *seviye* düğümün ağaçtaki derinliğine karşı düşen sayısal bir değerdir. Kök düğümün derinliği 0 dır, sonraki düğüm 1, sonraki 2 diye gider. `VISIT` bir sıralı sayı sabit türüdür (enumeration type).

```
VISIT                                                                      veri türü
```

VISIT değeri düğümün ağaçtaki durumunu ve işlevin nasıl çağrıldığını belirtir. Düğümün durumu ya sonuncu ya da dahili düğümdür. Her sonuncu düğüm için işlev yalnız ve yalnız bir kere çağrılır. Her dahili düğüm için ise üç kere çağrılır: ilk çocuk işlenmeden önce, ilk çocuk işlendikten sonra ve her iki çocuk işlendikten sonra. Böylece ağacın enine üç yöntemini kullanmak (hatta hepsini birlikte) mümkün olur.

`preorder`
: Düğüm, bir dahili düğümdür ve işlev ilk çocuk düğüm işlenmeden önce çağrılır.

`postorder`
: Düğüm, bir dahili düğümdür ve işlev ilk çocuk düğüm işlendikten sonra çağrılır.

`endorder`
: Düğüm, bir dahili düğümdür ve işlev her iki çocuk düğüm işlendikten sonra çağrılır.

`leaf`
: Düğüm sonuncudur.

```
void twalk(const void   *kök,                                              işlev
           __action_fn_t eylem)
```

kök ile gösterilen ağacın her düğümü için *eylem* parametresi ile belirtilen işlevi çağırır. Sonuncu düğümler için işlev *değer* argümanında **leaf** belirtilerek sadece bir kere çağrılır. İç düğümler için işlev *değer* argümanında ilgili değer belirtilerek üç kere çağrılır. *eylem* işlevinin *seviye* argümanı ağacın kökünden çocuklara inildikçe artan değerler alır. Kökün seviyesi 0 dır.

twalk işlevinin *eylem* parametresi için kullanılan işlevler ağaç verisini değiştirmemelidir çünkü aynı ağaç üzerinde aynı anda birden fazla evrede **twalk** çalıştırılabilir. Ayrıca aynı anda paralel olarak **tfind** de çağrılabilir. Ağaçta değişiklik yapan işlevler kullanılmamalıdır, aksi takdirde bir davranış tanımlanmamıştır.

X. Şablon Eşleme

İçindekiler

GNU C kütüphanesi iki çeşit kalıp için şablon eşleme oluşumu içerir: düzenli ifadeler ve dosya isimleri için özel kalıplar. Kütüphane ayrıca, kabuğun yaptığı değişken yorumlama, komut kullanımı ve metnin sözcüklere ayrılması için oluşumlar da içerir.

Kabuk yorumları ile ilgili daha ayrıntılı bilgi için GNU Bash Başvuru Kılavuzu[B241] kitabına bakabilirsiniz.

Düzenli ifadeler ile ilgili daha ayrıntılı bilgi için Regex Kütüphanesi Başvuru Kılavuzu[B242] kitabına bakabilirsiniz.

1. Dosya İsmi Kalıpları

Bu bölümde belli bir dizgenin bir kalıp ile nasıl eşleştildiği anlatılacaktır. Sonuç bir evet ya da hayır yanıtıdır: dizge bir kalıpla ya eşleşir ya da eşleşmez. Bu bölümde bahsedilen sembollerin hepsi `fnmatch.h` başlık dosyasında bildirilmiştir.

```
int fnmatch(const char *kalıp,                                    işlev
            const char *dizge,
            int         seçenekler)
```

Bu işlev *dizge* dizgesi ile *kalıp* kalıbının eşleşip eşleşmediğine bakar. Eğer eşleşiyorlarsa `0` ile döner, aksi takdirde sıfırdan farklı **`FNM_NOMATCH`** değeri ile döner. *kalıp* ve *dizge* argümanlarının her ikisi de dizgedir.

seçenekler ile eşleşme ile ilgili ayrıntıları değiştiren seçenek bitleri belirtilir. Belirtilebilecek seçeneklerin listesi için aşağıya bakınız.

GNU C Kütüphanesinde, **`fnmatch`** işlevinin hata deneyimi yoktur. Daima eşleşme olup olmadığına ilişkin bir yanıt döndürür. Ancak, işlevin diğer gerçeklenimleri bazan hata raporlayabilir. Bu durumda dönen değerleri **`FNM_NOMATCH`** değerinden farklı olacaktır.

seçenekler argümanında belirtilebilecek seçenekler:

`FNM_FILE_NAME`
: **/** karakteri dosya isimlerinde özel olarak ele alınır. Bu seçenek etkinse, *kalıp*, *dizge* içindeki **/** karakteri ile eşleşemez. Eşleştirmenin tek yolu *kalıp* dizgesinin **/** karakterini içermesidir.

`FNM_PATHNAME`
: Bu POSIX.2 ile gelen bir **FNM_FILE_NAME** takma ismidir. Biz dosya ismi yerine dosya yolu demediğimizden bu ismin kullanımını önermiyoruz.

`FNM_PERIOD`
: **.** karakteri dosya isimlerinin başında ise özel olarak ele alınır. Bu seçenek etkinse ve **.** karakteri *dizge* dizgesinin ilk karakteri ise *kalıp* ile eşleşemez.

  FNM_PERIOD ve **FNM_FILE_NAME** birlikte belirtilmişse, **.** karakterinin *dizge* dizgesinin başında olmasının yanında **/** karakterini izlediği durumda da eşleşme seğlanır. (Kabuk **FNM_PERIOD** ve **FNM_FILE_NAME** seçeneklerini dosya isimleriyle eşleşme ararken birlikte kullanır.)

`FNM_NOESCAPE`
: **** karakteri kalıplarda özel olarak ele alınmaz. Normalde, **** kendinden sonraki karakteri önceler; bu seçenek etkinse bu özellik kapatılır, böylece sadece kendisiyle eşleşebilir. Önceleme etkin olduğunda, **\?** kalıbı sadece **?** karakteri ile eşleşir, çünkü kalıp içindeki soru işareti sıradan bir karakter gibi işlem görür.

  Eğer **FNM_NOESCAPE** seçeneği etkinse, **** bir sıradan karakterdir.

`FNM_LEADING_DIR`
: *dizge* içinde **/** karakterini izleyen karakterler yoksayır; başka bir deyişle *kalıp* dizgesi *dizge* dizisinin başındaki bir dizin ismimi diye bakılır.

  Bu seçenek etkinse, **foo*** ya da **foobar** bir kalıp olarak **foobar/frobozz** dizgesi ile eşleşir.

`FNM_CASEFOLD`
: *dizge* ile *kalıp* karşılaştırılırken harf büyüklüğüne bakılmaz.

`FNM_EXTMATCH`
: Normal kalıplardan başka **ksh**'da bahsedilen ek kalıplarda tanınır. Aşağıdaki listedeki *kalıp–listesi*, **|** karakterinin ayraç olarak kullanıldığı bir kalıp listesidir.

  `?(`*kalıp-listesi*`)`
  : *kalıp–listesi* ile belirtilen kalıplar girdi dizgesiyle hiç eşleşmeyebilir ya da bir kere eşleşebilir.

  `*(`*kalıp-listesi*`)`
  : *kalıp–listesi* ile belirtilen kalıplar girdi dizgesiyle hiç eşleşmeyebilir ya da defalarca eşleşebilir.

  `+(`*kalıp-listesi*`)`
  : *kalıp–listesi* ile belirtilen kalıplar girdi dizgesiyle ya en azından bir kere ya da defalarca eşleşebilir.

  `@(`*kalıp-listesi*`)`
  : *kalıp–listesi* ile belirtilen kalıplar girdi dizgesiyle sadece bir kere eşleşebilir.

  `!(`*kalıp-listesi*`)`
  : *kalıp–listesi* ile belirtilen kalıplar girdi dizgesiyle hiç eşleşmiyorsa eşleşme sağlanmış demektir.

2. Genelleme

Kalıpların ana örnek kullanımı bir dizindeki dosyalarda eşleşme aramak ve eşleşenleri listelemektir. Bu işleme ***genelleme*** denir.

Bu işlemi bir dizindeki dosya isimlerini tek tek **fnmatch** ile sınayarak da yapabilirdiniz ama yavaş olurdu (alt dizinleri de kendiniz okumak zorunda kalacağınız için).

Kütüphane özellikle bu işi yapan bir işlev içerir: **glob**.

glob ve bu kısımdaki diğer semboller `glob.h` başlık dosyasında bildirilmiştir.

2.1. `glob` çağrısı

Genelleme işleminin sonucu bir dosya isimleri göstericileri dizisidir. Bu diziyi döndürmek için **glob** işlevi aslında bir yapı olan özel bir veri türü, **glob_t** kullanır. **glob** işlevine yapının adresini aktarırsanız, işlev, sonucu yapının alanlarına yazar.

glob_t veri türü

Bu veri türü bir dizge göstericileri dizisine bir gösterici tutar. Daha açıkçası, bu dizinin adresini ve boyutunu kaydeder. GNU gerçeklenimi standartdışı olarak biraz daha fazla alan içerir.

`gl_pathc`
: Gösterici dizisindeki eleman sayısı; **GLOB_DOOFFS** seçeneği kullanılmışsa baştaki boş girdiler hariç (aşağıdaki `gl_offs`'a bakınız).

`gl_pathv`
: Gösterici dizisinin adresi; bu alan **char **** türündedir.

`gl_offs`
: **gl_pathv** alanındaki adresten itibaren ilk gerçek elemanın konumu. Diğer alanların tersine bu alan **glob** işlevi için daima bir girdidir (diğer alanları işlev doldurur).

  Eğer konuma sıfırdan farklı bir değer belirtirseniz, gösterici dizisinin başlangıcından itibaren bazı elemanlar boş kalacaktır. (**glob** işlevi oraları boş göstericilerle doldurur.)

  gl_offs alanı sadece **GLOB_DOOFFS** seçeneği etkinse anlamlıdır. Aksi takdirde bu alanın ne içerdiğine bakılmaksızın konumun sıfır olduğu varsayılır, yani ilk eleman gösterici dizisinin başına konur.

`gl_closedir`
: **closedir** işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **void (*) (void *)** türündedir.

  Bu alan bir GNU oluşumudur.

`gl_readdir`
: Bir dizinin içeriğini okumakta kullanılan **readdir** işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **struct dirent *(*) (void *)** türündedir.

  Bu alan bir GNU oluşumudur.

`gl_opendir`
: **opendir** işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **void *(*) (const char *)** türündedir.

Bu alan bir GNU oluşumudur.

`gl_stat`
Dosya sistemindeki bir nesne hakkında bilgi döndüren **stat** işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **int (*) (const char *, struct stat *)** türündedir.

Bu alan bir GNU oluşumudur.

`gl_lstat`
Dosya sistemindeki bir nesne hakkında bilgi döndüren, ancak sembolik bağları izlemeyen **lstat** işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **int (*) (const char *, struct stat *)** türündedir.

Bu alan bir GNU oluşumudur.

glob64 işlevinde kullanmak üzere `glob.h` başlık dosyası bu türe çok benzer bir başka tanım içerir. **glob64_t** türü **glob_t** türünden sadece **gl_readdir**, **gl_stat** ve **gl_lstat** elemanlarının farklı türde olması ile farklıdır.

glob64_t veri türü

Bu veri türü bir dizge göstercileri dizisine bir gösterici tutar. Daha açıkçası, dizinin adresini ve boyutunu kaydeder. GNU gerçeklenimi standartdışı olarak biraz daha fazla alan içerir.

`gl_pathc`
Gösterici dizisindeki eleman sayısı; **GLOB_DOOFFS** seçeneği kullanılmışsa baştaki boş girdiler hariç (aşağıdaki gl_offs'a bakınız).

`gl_pathv`
Gösterici dizisinin adresi; bu alan **char **** türündedir.

`gl_offs`
gl_pathv alanındaki adresten itibaren ilk gerçek elemanın konumu. Diğer alanların tersine bu alan **glob** işlevi için daima bir girdidir (diğer alanları işlev doldurur).

Eğer konuma sıfırdan farklı bir değer belirtirseniz, gösterici dizisinin başlangıcından itibaren bazı elemanlar boş kalacaktır. (**glob** işlevi oraları boş göstericilerle doldurur.)

gl_offs alanı sadece **GLOB_DOOFFS** seçeneği etkinse anlamlıdır. Aksi takdirde bu alanın ne içerdiğine bakılmaksızın konumun sıfır olduğu varsayılır, yani ilk eleman gösterici dizisinin başına konur.

`gl_closedir`
closedir işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **void (*) (void *)** türündedir.

Bu alan bir GNU oluşumudur.

`gl_readdir`
readdir64 işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **struct dirent64 *(*) (void *)** türündedir.

Bu alan bir GNU oluşumudur.

`gl_opendir`

opendir işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **void *(*) (const char *)** türündedir.

Bu alan bir GNU oluşumudur.

gl_stat
: Dosya sistemindeki bir nesne hakkında bilgi döndüren **stat64** işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **int (*) (const char *, struct stat64 *)** türündedir.

  Bu alan bir GNU oluşumudur.

gl_lstat
: Dosya sistemindeki bir nesne hakkında bilgi döndüren, ancak sembolik bağları izlemeyen **lstat64** işlevinin başka bir gerçekleniminin adresi. **GLOB_ALTDIRFUNC** seçeneği etkinse kullanılır. Bu alan **int (*) (const char *, struct stat64 *)** türündedir.

  Bu alan bir GNU oluşumudur.

```
int glob(const char *kalıp,                                              işlev
         int          seçenekler,
         int        (*hata–işlevi) (const char *dosyaismi, int hatakodu),
         glob_t      *dizi–gst)
```

glob işlevi *kalıp* kalıbını kullanarak geçerli dizin içinde genelleme yapar. Sonuçları yeni ayırdığı gösterici dizisine yerleştirir ve bu dizinin adresi ile boyutunu **dizi–gst* içine koyar. *seçenekler* argümanı seçenek bitlerinin bir birleşimidir; bu seçenekler hakkında ayrıntılı bilgiyi *Genelleme Seçenekleri* (sayfa: 217) bölümünde bulabilirsiniz.

Genellemenin sonucu dosya isimleri dizisidir. **glob** işlevi her sonuç dizgesi için bir dizge ve bu dizgelerin adreslerini saklamak için **char **** türünde bir gösterici dizisi ayırır. Dizinin son elemanı bir boş göstericidir. Bu gösterici dizisine *dizge göstericileri dizisi* denir.

Bu diziyi döndürmek için **glob** adresini ve uzunluğunu (sonlandırıcı boş gösterici dışında dizge gösterici sayısı) **dizi–gst* içinde saklar.

Normalde, **glob** işlevi dosya isimlerini döndürmeden önce alfabetik sıraya sokar. Bunu **GLOB_NOSORT** seçeneğini kullanarak yaptırmayabilir ve sonucu daha çabuk alabilirsiniz. Genellikle, sıralamayı **glob**'a yaptırmak daha iyidir. Eğer dosyaları alfabetik sırayla görüntülerseniz, kullanıcı işlemin hızının sizin yazılımınızdan kaynaklandığını sanacaktır.

glob başarılı olursa 0 ile döner. Aksi takdirde şu hata kodlarından biri ile döner:

GLOB_ABORTED
: Bir dizini açarken bir hata oluştu ya **GLOB_ERR** seçeneğini kullanmışsınız ya da *hata–işlevi*'nin sıfırdan farklı bir değer döndürmesini sağlamışsınız. **GLOB_ERR** seçeneği ve *hata–işlevi* ile ilgili açıklamalar için *Genelleme Seçenekleri* (sayfa: 217) bölümüne bakınız.

GLOB_NOMATCH
: Kalıp mevcut dosyaların hiçbiriyle eşleşmedi. **GLOB_NOCHECK** seçeneğini belirtirseniz bu hatayı asla almazsınız, çünkü bu seçenek **glob**'a en azından bir dosya eşleşmiş *gibi* davranmasını söyler.

GLOB_NOSPACE
: Sonuçlar için ayrılacak bellek yok.

Bir hata sırasında, **glob** o ana kadar bulduğu tüm sonuçlara ilişkin bilgiyi **dizi–gst* ile döndürür.

Önemli

glob işlevi LFS arayüzleri olmaksızın elde edilemeyen dizin ya da dosyalar saptarsa başarısız olmaz. **glob** bu işlevlerin dahili olarak kullanılacağı kabulüyle gerçeklenmiştir. Bu en azından Unix standardında yapılmış bir kabullerden biridir. GNU oluşumu kullanıcıya kendi dizin elde etme yöntemini sağlayarak ve **stat** işlevleriyle bunu biraz daha karmaşıklaştırır. Eğer bunlar kullanıcı tanımlı işlevler olsalardı büyük bir dosyaya ya da dizine rastlandığında **glob** başarısız *olabilecekti*.

```
int glob64(const char *kalıp,                                          işlev
           int          seçenekler,
           int        (*hata–işlevi) (const char *dosyaismi, int hatakodu),
           glob64_t    *dizi–gst)
```

glob64 işlevi büyük dosya arayüzünün parçası olarak eklenmiştir, ancak kullanılan arayüz orjinal LFS arayüzü değildir. Bunun sebebi basittir: gereksizdir. **glob64** işlevi için gerekli olan herşey GNU oluşumunun kullanıcıya kendi dizin elde etme yöntemini sağlaması ve **stat** işlevleriyle eklenmiştir. **struct dirent** ve **struct stat** veri türleri **_FILE_OFFSET_BITS** seçimine bağlı olduğundan **readdir** ve **stat** işlevleri bu seçime bağlı olarak çalışır.

Bu farkların dışında **glob64** tamamen **glob** gibi çalışır.

Bu işlev bir GNU oluşumudur.

2.2. Genelleme Seçenekleri

Bu bölümde **glob** işlevinin *seçenekler* argümanında belirtilebilecek seçenekler açıklanmıştır. Seçtiğiniz seçenekleri C'nin bit seviyesi VEYA işleci olan **|** işlecini kullanarak birlikte belirtebilirsiniz.

GLOB_APPEND
Önceki bir **glob** çağrısı ile üretilen dizge göstericileri dizisine bu genişletmeden dizgeler eklenir. İşlem bir dizgeye aralarında boşluklar bırakarak yeni sözcükler eklenmesi gibidir.

Ekleme işlemi sırasında, **glob** çağrısından önce dizge göstericileri dizisine ilişkin bilgileri içeren yapının içeriğinde değişiklik yapmamalısınız. Eğer ilk çağrıda **GLOB_DOOFFS** seçeneğini kullanmışsanız, ekleme amacıyla yaptığınız çağrıda da bu seçeneği kullanmalısınız.

Yapının **gl_pathv** üyesindeki gösterici ikinci **glob** çağrısından sonra artık geçerli olmayacaktır, çünkü **glob** işlevi diziyi yeniden ayırır. Böylece her **glob** çağrısından sonra **glob_t** yapısının **gl_pathv** üyesinden doğru gösterici elde edilir; çağrılar arasında gösterici asla saklanmamalıdır.

GLOB_DOOFFS
Gösterici dizisinin başlangıcında boş göstericiler bırakır. **gl_offs** alanında bunun sayısı belirtilir.

GLOB_ERR
Eğer *kalıp*'ın eşleştirilmesi sırasında okunması gereken dizinlerin okunmasında zorluk varsa, bir hata raporlanır ve hemen çıkılır. Okunmak istenen dizinin erişim izinlerinin yetersizliği bu zorluklardan biridir.

glob çağrısında hataları elde edebilen bir *hata–işlevi* belirterek bu hatalar üzerinde daha fazla denetim sağlayabilirsiniz. *hata–işlevi* olarak bir boş gösterici belirtmemişseniz, **glob** bir dizini okuyamadığında hemen çıkmaz, iki argümanla şuna benzer biçimde *hata–işlevi* işlevini çağırır:

```
(*hata–işlevi) (dosyaismi, hatakodu)
```

dosyaismi argümanı **glob** işlevinin açamadığı ya da okuyamadığı dizinin ismidir, *hatakodu* ise **glob** tarafından raporlanan **errno** değeridir.

Hata eylemci işlev sıfırdan farklı bir değerle dönerse **glob** hemen döner, aksi takdirde işleme devam eder.

GLOB_MARK
: Kalıp dizin ismiyle eşleşirse, dizin ismi döndürülürken ismine **/** eklenir.

GLOB_NOCHECK
: Eğer kalıp hiçbir dosya ismiyle eşleşmezse, işlev sanki bir eşleşme bulunmuş gibi döner. (Normalde, kalıp hiçbirşeyle eşleşmemişse **glob** eşleşme bulunmadığını belirtecek şekilde döner.)

GLOB_NOSORT
: Dosya isimleri alfabetik olarak sıralanmaz; okundukları sırada döndürülürler. (Pratikte sıralama, dosyaların dizine giriş sırasına bağlıdır.) Sıralama yapılmak istenmemesinin tek sebebi zaman kazanmak olabilir.

GLOB_NOESCAPE
: Kalıpta **** karakteri özel olarak değil kendisi olarak ele alınır. Normalde, **** başka bir karakteri öncelemek için kullanılır. Bu seçenek bu şekilde yorumlanmasını engeller. Önceleme etkinse, **\?** kalıbı sadece **?** dizgesiyle eşleşir, Çünkü bu biçimde belirtilen soru işareti kalıpta sıradan bir karakter olarak işlem görür.

  GLOB_NOESCAPE seçeneği etkinse,**** bir sıradan karakterdir.

  glob işlevi **fnmatch** işlevini defalarca çağırarak çalışır. **GLOB_NOESCAPE** seçeneğini **fnmatch** çağrılarında **FNM_NOESCAPE** seçeneğini etkinleştirmek için kullanır.

2.3. Diğer Genelleme Seçenekleri

Önceki bölümde açıklanan seçeneklerin yanında **glob** işlevinin GNU gerçeklemesinde geçerli başka seçenekler de vardır. Bu seçenekler `glob.h` başlık dosyasında tanımlanmıştır. Bu seçeneklerin bazıları günümüzdeki kabuk gerçeklenimlerinde kullanılan işlevselliği sağlar.

GLOB_PERIOD
: **.** (nokta) karakteri özel karakter olarak ele alınır. Bu durumda özel kalıp karakteri olarak eşleştirilmez. *Dosya İsmi Kalıpları* (sayfa: 212) bölümündeki **FNM_PERIOD** seçeneğine de bakınız.

GLOB_MAGCHAR
: **GLOB_MAGCHAR** seçeneği **glob** işlevinin *seçenekler* parametresinde kullanmak için değildir. **glob** işlevi bu seçeneği, eğer kalıp, özel kalıp karakterleri içeriyorsa sonucun bu yolla elde edileceğini belirtmek için **glob_t** yapısının *gl_flags* üyesinde kullanır.

GLOB_ALTDIRFUNC
: Bu seçenek etkinse, **glob** gerçeklemesi dosya sistemine erişmek için kütüphanedeki işlevleri değil, kullanıcı tarafından *dizi–gst* ile gösterilen yapı içinde belirtilen işlevleri kullanır. Dizinlere erişim için kullanılan işlevler *Dizinlere Erişim* (sayfa: 353) ve *Bir Dosyanın Özniteliklerinin Okunması* (sayfa: 374) bölümlerinde açıklanmıştır.

GLOB_BRACE
: Bu seçenek etkinse kaşlı ayraçlar özel olarak ele alınır. Bu durumda kaşlı ayraçların doğru gruplanması gerekir. Yani kaşlı ayraçlar çiftler halinde olmalıdır. İç içe gruplamalar yapılabilir. Böylece bir gruplama bir diğerini tanımlamakta kullanılabilir. Bir kaşlı ayraç ifadesi, başka bir kaşlı ayraç ifadesinin içinde tanımlanmışsa onun dışına çıkmaması gerektiğine dikkat etmelisiniz.

Bir kaşlı ayraç grubundaki dizgeler, virgüllerle ayrılarak ayrı ifadeler haline getirilebilir. Bu durumda virgüllerin bu amaçla kullanıldıkları ve dizgelerin virgül içermedikleri varsayılır. Virgül kullanılarak ayrılan ifadeler aynı seviyede olmalıdır. Alt ifadeler kaşlı ayraç içine alınmışsa bunların içindeki virgüller eşleştirilmez. Kaşlı ayraçlı alt gruplarda daha derin seviyeleri elde etmede kullanılır. Bir kullanım örneği:

```
glob ("{foo/{,bar,biz},baz}", GLOB_BRACE, NULL, &result)
```

Eğer hataları gözönüne almazsak, şuna eşdeğerdir:

```
glob ("foo/", GLOB_BRACE, NULL, result)
glob ("foo/bar", GLOB_BRACE|GLOB_APPEND, NULL, &result)
glob ("foo/biz", GLOB_BRACE|GLOB_APPEND, NULL, &result)
glob ("baz", GLOB_BRACE|GLOB_APPEND, NULL, &result)
```

GLOB_NOMAGIC
: Eğer kalıp herhangi bir özel kalıp karakteri içermiyorsa (doğrudan dosya ismi verilmişse), bu isimde bir dosya yoksa bile bu kalıp dizgesi döndürülür.

GLOB_TILDE
: Bu seçenek etkinse ve kalıp yaklaşık işareti (`~`) ile başlıyorsa, bu karakter özel olarak ele alınır. Bu durumda yaklaşık işaretinin bir kullanıcının ev dizinini gösterdiği varsayılır.

  Eğer `~` kalıptaki tek karakterse ya da onu bir **/** (bölü çizgisi) izliyorsa, bu kalıp sürecin sahibinin ev dizini ile eşleştirilir. Bilgi, **getlogin** ve **getpwnam** işlevleri kullanılarak sistem veritabanlarından okunur. Örneğin, ev dizini `/home/bart` olan **bart** kullanıcısı için çağrı şöyle olurdu:

```
glob ("~/bin/*", GLOB_TILDE, NULL, &result)
```

  Bu çağrıdan `/home/bart/bin` döner. Burada başka bir kullanıcının ev dizini de belirtilebilirdi. Bunun için yaklaşık işaretinden sonra kullanıcının ismini belirtmek yeterlidir. **homer**'in `bin` dizinini almak istersek:

```
glob ("~homer/bin/*", GLOB_TILDE, NULL, &result)
```

  Eğer kullanıcı ismi geçersizse ya da ev dizini bir nedenle saptanamamışsa kalıba dokunulmaksızın kalıp dizgesi sonuç olarak döndürülür. Son örnekte, **homer** isminde bir kullanıcı yoksa işlev `~homer` dizinini aramaz ve sonuç olarak **"~homer/bin/*"** dizgesini döndürür.

  Bu işlevsellik C kabuklarında **nonomatch** seçeneğinin etkin olduğu duruma eşdeğerdir.

GLOB_TILDE_CHECK
: Bu seçenek belirtilmişse **glob** işlevi **GLOB_TILDE** seçeneği verilmiş gibi davranır. Tek fark, belirtilen kullanıcı yoksa ya da bir ev dizini saptanamamışsa kalıp dizgesi döndürülmez ve işlem bir hata ile sonuçlanır.

  Bu işlevsellik C kabuklarında **nonomatch** seçeneğinin etkin olmadığı duruma eşdeğerdir.

GLOB_ONLYDIR
: Bu seçenek belirtilmişse, bu, çağrıcının sadece kalıpla eşleşen dizinle ilgilendiğine dair bir *ipucu* olarak değerlendirilir. Eğer dosya hakkında saptanan bilgi onun bir dizin olmadığını belirtiyorsa bunlar reddedilir ama bunların türünü saptamak için ek bir çalışma yapılmaz. Yani çağrıcı hala bir süzme çalışması yapabilir.

  Bu işlevsellik sadece GNU **glob** gerçeklemesi ile kullanılabilir. Aslında dahili kullanım için düşünülmüşse de kullanıcı için de yararlı olabileceğinden burada belgelenmiştir.

glob çağrısı çoğu durumda döndürdüğü sonucu saklamak için özkaynak ayırır. **glob** işlevi hep aynı **glob_t** nesnesi ile çağrılsa bile her çağrıda özkaynaklar önce serbest bırakılıp sonra tekrar ayrıldığından bir kaçak oluşmaz. Fakat bu işlem her çağrıda hep aynı süre içinde olmaz.

`void` **globfree**`(glob_t *`*dizi–gst*`)` işlev

globfree işlevi önceki **glob** çağrılarında *dizi–gst* ile gösterilen nesne için ayrılan yeri serbest bırakır. Bu **glob_t** türündeki nesne artık kullanılmayacaksa bu işlev çağrılarak bu alan serbest bırakılmalıdır.

`void` **globfree64**`(glob64_t *`*dizi–gst*`)` işlev

Bu işlev **globfree** işlevinin benzeridir, farklı olarak **glob64** işlevi ile ayrılan bir **glob64_t** nesnesini serbest bırakmak için kullanılır.

3. Düzenli İfade Eşleştirme

GNU C kütüphanesi düzenli ifade eşleştirmesi için iki arayüz içerir. Biri standart POSIX.2 arayüzü diğeri ise yıllardır GNU sistemlerinde kullanılan arayüzdür.

Her iki arayüz de `regex.h` başlık dosyasında bildirilmiştir. Eğer **_POSIX_C_SOURCE** makrosunu tanımlarsanız, sadece POSIX.2 işlevleri, yapıları ve sabitleri bildirilir.

Düzenli ifadeler ile ilgili daha ayrıntılı bilgi için Regex Kütüphanesi Başvuru Kılavuzu[B248] kitabına bakabilirsiniz.

3.1. POSIX Düzenli İfadelerinin Derlenmesi

Bir düzenli ifadeyi eşleştirme amacıyla kulanmadan önce onu *derlemeniz* gerekir. Bu tam anlamıyla bir derleme işlemi değildir. Derleme sonucunda bir takım makina kodları değil, özel bir veri yapısı üretilir. Fakat derlemenin amacı aynı kalır, amaç daha hızlı işlem yapmaktır. (Derlenmiş düzenli ifadelerin nasıl eşleştirildiği *Derlenmiş POSIX Düzenli İfadelerinin Eşleştirilmesi* (sayfa: 222) bölümünde anlatılmıştır.)

Derlenmiş düzenli ifadeler için özel bir veri türü vardır:

regex_t veri türü

Bir derlenmiş düzenli ifadeyi barındıran nesnenin türüdür. Aslında bir yapıdır. Sadece bir üyesinden bahsedeceğiz:

`re_nsub`
: Derlenmiş düzenli ifade içindeki parantezli alt ifadelerin sayısını belirtir.

Yapının başka alanları da vardır, sadece kütüphane içindeki işlevler tarafından kullanıldığından onlardan burada bahsetmeyeceğiz.

Bir **regex_t** nesnesini oluşturduktan sonra bir düzenli ifadeyi bir **regcomp** çağrısı ile derlemelisiniz.

```
int regcomp(regex_t *restrict     şablon,                  işlev
            const char *restrict  düzenli_ifade,
            int                   derleme–seçenekleri)
```

regcomp işlevi, bir dizgeyle eşleştirmek üzere **regexec** işlevinde kullanmak için bir düzenli ifadeyi bir veri yapısına derler. Derlenmiş düzenli ifadenin biçimi verimli bir eşleştirmeyi mümkün kılacak şekilde tasarlanmıştır. Derlenen düzenli ifadeyi işlev **şablon* nesnesine yerleştirir.

regex_t türünde bir nesne ayırıp bunun adresini **regcomp** işlevine aktarmalısınız.

derleme–seçenekleri argümanı düzenli ifadelerin sözdizimi ve anlamsal bütünlüğünü denetleyen bazı seçenekleri belirtmek için kullanılır. Bkz. *POSIX Düzenli İfade Seçenekleri* (sayfa: 222).

REG_NOSUB seçeneği belirtilmişse, **regcomp** işlevi derlenen düzenli ifadeye alt ifadelerin nasıl eşleştirileceği bilgisini yerleştirmez. Bu durumda, **regexec** çağrısında *eşleşenler* ve *eşleşen–sayısı* argümanlarına **0** değerini aktarmalısınız.

Eğer **REG_NOSUB** seçeneğini belirtmezseniz, derlenen düzenli ifade alt ifadelerin nasıl eşleştirileceği bilgisini de içerir. Ayrıca, **regcomp** işlevi *şablon*'un kaç alt ifade içerdiğini *şablon*`->re_nsub` üyesine yazar. Bu bilgiyi alt ifade eşleştirmesinde tutulacak bilgi için ayrılacak dizinin uzunluğuna karar vermek için kullanabilirsiniz.

regcomp işlevi düzenli ifadeyi başarıyla derleyebilmişse **0** ile döner; aksi takdirde, sıfırdan farklı bir hata kodu ile döner (aşağıya bakınız). Dönen hata kodundan hata dizgesini üretmek için **regerror** işlevini kullanabilirsiniz; bkz. *POSIX Şablonunun Temizlenmesi* (sayfa: 224).

regcomp işlevinin döndürdüğü sıfırdan farklı değerler:

`REG_BADBR`
: Düzenli ifade içinde geçersiz bir **\{…\}** yapısı vardır. Geçerli bir **\{…\}** yapısı ya tek bir sayı ya da virgülle ayrılmış iki sayı içermelidir.

`REG_BADPAT`
: Düzenli ifade içinde bir sözdizimi hatası var.

`REG_BADRPT`
: **?** veya ***** gibi yineleme işleçleri yanlış yerde (bu işleçlerden önce üzerinde işlem yapılacak bir alt ifade olmalıdır).

`REG_ECOLLATE`
: Düzenli ifade geçersiz bir karakter karşılaştırma elemanı içeriyor (geçerli yerelde tanımlı olanlardan biri değil). Bkz. *Yerellerin Etkilediği Eylemlerin Sınıflandırılması* (sayfa: 165).

`REG_ECTYPE`
: Düzenli ifade geçersiz bir karakter sınıfı ismi içeriyor.

`REG_EESCAPE`
: Düzenli ifade **** ile bitiyor.

`REG_ESUBREG`
: Geçersiz sayıda *****rakam* yapısı var.

`REG_EBRACK`
: Düzenli ifade gruplanmamış köşeli ayraçlar içeriyor.

`REG_EPAREN`
: Düzenli ifade gruplanmamış parantezler içeriyor ya da **\(** ve **\)** arasında olması gereken bir temel düzenli ifadenin sarmalayıcılarından biri eksik.

`REG_EBRACE`
: Düzenli ifadede **\{** ve **\}** sarmalayıcılardan biri eksik.

`REG_ERANGE`
: Bir aralık ifadesindeki sonlandırıcılardan biri geçersiz.

`REG_ESPACE`

regcomp için bellek yetersiz.

3.2. POSIX Düzenli İfade Seçenekleri

Bunlar bir düzenli ifadeyi **regcomp** ile derlerken *derleme–seçenekleri* argümanında belirtilebilecek bit değerli seçeneklerdir.

REG_EXTENDED
: Şablon temel bir düzenli ifadenin değil, gelişmiş bir düzenli ifadenin karşılığı olarak oluşturulur.

REG_ICASE
: Harfler eşleştirilirken harf büyüklüğüne bakılmaz.

REG_NOSUB
: *eşleşenler* dizisinin içeriği için birşey yapılmaz.

REG_NEWLINE
: *dizge* içindeki satırsonu karakterleri dizgeyi satırlara bölmek için kullanılır. Böylece bir satırsonu karakterinden önceki bir **$** karakteri ve satırsonu karakterinden sonraki bir ^ karakteri eşleştirilebilir. Ayrıca, **.** (nokta) veya **[^…]** ifadesinin bir satırsonu karakteri ile eşleşmesine izin verilmez.

  Aksi takdirde satırsonu karakteri sıradan bir karakter olarak işlem görür.

3.3. Derlenmiş POSIX Düzenli İfadelerinin Eşleştirilmesi

Bir düzenli ifadeyi *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220) bölümünde açıklandığı gibi derledikten sonra **regexec** işlevinde dizgelerle karşılaştırarak eşleşmeleri arayabilirsiniz. Bir düzenli ifade demirleme işleçlerini (^ veya **$**) içermedikçe bir dizge içinde bulunan her eşleşme başarılı sayılır.

```
int regexec(const regex_t *restrict şablon,                    işlev
            const char *restrict    dizge,
            size_t                  eşleşen–sayısı,
            regmatch_t              eşleşenler[restrict],
            int                     icra–seçenekleri)
```

Bu işlev derlenmiş düzenli ifadeyi içeren **şablon* ile *dizge*'yi eşlemeye çalışır.

regexec işlevi düzenli ifade eşleştirilebilmişse **0** ile aksi takdirde sıfırdan farklı bir değerle döner. Sıfırdan farklı dönüş değerlerinin listesi aşağıdadır. Sıfırdan farklı değerler için hata iletilerini üretmek için **regerror** işlevini kullanabilirsiniz; bkz. *POSIX Şablonunun Temizlenmesi* (sayfa: 224).

icra–seçenekleri argümanı ile bit değerli seçeneklerden birini veya bir kaçını belirtebilirsiniz.

Düzenli ifade ya da alt düzenli ifadelerle eşleşen *dizge* parçaları hakkında bilgi edinmek için *eşleşenler* ve *eşleşen–sayısı* argümanlarını kullanabilirsiniz. Aksi takdirde *eşleşen–sayısı* için **0**, *eşleşenler* için **NULL** değeri belirtmelisiniz. Bkz. *Alt İfadelerle Eşleşmeler* (sayfa: 223).

Eşleşme aramak için kullanılan derlenmiş düzenli ifadeyi hangi yerel için derlemişseniz, arama işlemini aynı yereli kullanarak yapmalısınız.

regexec işlevinin *icra–seçenekleri* argümanında kullanılabilecek seçenekler:

REG_NOTBOL
: Belirtilen dizgenin başlangıcı bir satırın başlangıcı sayılmaz; daha genel olarak, dizgeden önce bir metin bulunduğu kabulü yapılmaz.

REG_NOTEOL
: Belirtilen dizgenin sonu bir satırın sonu sayılmaz; daha genel olarak, dizgeden sonra bir metin bulunduğu kabulü yapılmaz.

regexec işlevinden dönebilecek sıfırdan farklı değerler:

REG_NOMATCH
: Şablon dizge ile eşleşmedi. Bu aslında bir hata değildir.

REG_ESPACE
: **regexec** için bellek yetersiz.

3.4. Alt İfadelerle Eşleşmeler

regexec işlevi *şablon* içindeki parantezli alt–ifadeleri eşleştirmek amacıyla kullanıldığına eşleşen *dizge* parçalarını kaydeder. Sonuçları **regmatch_t** türündeki yapı dizgesi ile döndürür. Dizinin ilk elemanı (indisi **0** olan eleman) düzenli ifadenin tamamıyla eşleşen dizge parçasını içerir. Dizinin diğer elemanlarının herbirine tek bir parantezli alt–ifade ile eşleşen dizge parçasının başlangıcı ve sonu kaydedilir.

regmatch_t veri türü

regexec işlevine aktarılan *eşleşenler* dizisinin veri türüdür. İki yapı üyesi içerir:

rm_so
: Alt dizgenin *dizge* içindeki başlangıç konumu. Bu değeri *dizge*'ye ekleyerek alt dizgenin adresini bulabilirsiniz.

rm_eo
: Alt dizgenin *dizge* içindeki bitiş konumu.

regoff_t veri türü

regoff_t bir işaretli tamsayı tür için bir takma isimdir. **regmatch_t** alanları **regoff_t** türündedir.

regmatch_t elemanları konumsal alt–ifadelere karşılıktır; ilk elemanda (indisi **1** olan eleman) eşleşen ilk alt–ifade, ikincisinde ikinci alt–ifade,... kayıtlıdır. alt–ifadelerin sırası ifadede bulundukları sıraya göredir.

regexec işlevine *eşleşenler* dizisinde kaç eşleşme saklanacağını *eşleşen–sayısı* ile belirtebilirsiniz. Eğer asıl ifade *eşleşen–sayısı* ile belirtilenden fazla alt–ifade içeriyorsa, kalan alt ifadelerin konumları hakkında bilgi edinemezsiniz. Fakat bu şablonun belli bir dizge ile eşleşip eşleşmemesini etkilemez.

regexec işlevinin eşleşen alt–ifadenin yeri hakkında bilgi döndürmesini istemiyorsanız ya *eşleşen–sayısı* ile **0** değerini aktarın ya da düzenli ifadeyi derlerken **regcomp** işlevini **REG_NOSUB** seçeneği ile kullanın.

3.5. Alt–ifade Eşlemesindeki Sorunlar

Bazan bir alt–ifade hiçbir karakter içermeyen bir altdizge ile eşleşebilir. Bu durum örneğin **fum** dizgesi **f\(o*\)** ifadesiyle eşleştiğinde ortaya çıkar. (Aslında sadece **f** eşleşir.) Bu durumda, konumların her ikisi de bulunan boş altdizgeleri gösterir. Bu örnekte her iki konum da **1**'dir.

Bazan bir düzenli ifade, alt–ifadeler hiç kullanılmadan eşleşir; örneğin, **ba\(na\)*** ifadesi **ba** dizgesiyle eşleşirken parantezli alt ifade kullanılmaz. Bu durumda bu alt–ifadenin elemanındaki her iki alan da **-1** içerir.

Bazan düzenli ifadenin tamamının eşleştirilmesinde belli bir alt–ifade defalarca eşleşebilir; örneğin, **ba\(na\)*** ifadesi **bananana** dizgesi ile eşleştirilirken parantezli alt–ifade üç kere eşleştirilmiştir. Bu durumda **regexec**

çağrısı alt–ifade ile eşleşen son dizge parçasının konumlarını kaydeder; **bananana** için bu komumlar **6** ve **8**'dir.

Fakat son eşleşme daima seçilenlerden biri olmaz. Eşleşme için en elverişli olanın öncelikli olanlardan biri olduğu söylenebilir. Yani, bir alt–ifade bir diğerinin içinde görünüyorsa sonuçlar son eşleşen dış alt–ifadedeki eşleşmeyi ifade eden içteki alt–ifade için raporlanır. Örneğin, **\(ba\(na\)*s \)*** ifadesinin **"bananas bas "** dizgesi ile eşleştiğini varsayalım. Son defasında içteki ifade aslında ilk sözcüğün sonuna doğru eşlenir. Ancak ikinci sözcüğün tekrar eşleşeceği varsayıldığında eşleme başarısız olur. **regexec**, "na" alt–ifadesinin kullanılmadığını raporlar.

Bu kuralın uygulandığı başka bir örnek:

```
\(ba\(na\)*s \|nefer\(ti\)* \)*
```

Bu ifade **bananas nefertiti** ile eşleşir. İlk sözcükte "na" alt–ifadesi eşleşir, ama ikinci sözcükte bu alt–ifade değil, onun alternatifi olan alt–ifade eşleşir. Bir kere daha, dış alt–ifadenin ikinci yinelemesi birincinin üzerine yazar ve bu ikinci yinelemede "na" alt ifadesi kullanılmaz. Bu durumda, **regexec**, "na" alt–ifadesinin kullanılmadığını raporlar.

3.6. POSIX Şablonunun Temizlenmesi

Bir derlenmiş düzenli ifadeyi artık kullanmayacaksanız, bir **regfree** çağrısı ile ona ayrılan belleği serbest bırakabilirsiniz.

```
void regfree(regex_t *şablon)                                    işlev
```

regfree çağrısı **şablon* ile gösterilen derlenmiş düzenli ifadeye ayrılan belleği serbest bırakır. Bu alan, bu kılavuzda üyelerinin hepsini açıklamadığımız **regex_t** yapısının çeşitli iç alanlarını içerir.

regfree işlevi **şablon* nesnesinin kendisini serbest bırakmaz.

Başka bir düzenli ifadeyi **regex_t** yapısında derlemeden önce bu yapı için ayrlan alanı **regfree** ile serbest bırakmalısınız.

regcomp veya **regexec** işlevi bir hara raporladığında, bunu bir hata dizgesine dönüştürmek için **regerror** işlevini kullanabilirsiniz.

```
size_t regerror(int                    hatakodu,                 işlev
                const regex_t *restrict şablon,
                char *restrict          tampon,
                size_t                  uzunluk)
```

Bu işlev *hatakodu* ile belirtilen hata kodu için bir hata iletisi üretir ve bunu *tampon* adresinden başlayan *uzunluk* baytlık dizgeye yerleştirir. *şablon* argümanı ile belirtilen derlenmiş düzenli ifade hatanın oluştuğu **regcomp** veya **regexec** işlevinde kullanılmış olan nesne olmalıdır. *şablon* argümanında değer olarak **NULL**'da belirtebilirsiniz ve anlamlı bir hata iletisi alabilirsiniz, ama alacağınız hata iletisi ayrıntılı olmayacaktır.

Hata iletisi belirttiğiniz *uzunluk* bayta sığmazsa, **regerror** hata iletisinin sığdığı kadarını yerleştirir. İşlev, ister tam dizgeyi ister kırpılmış dizgeyi döndürsün, daima dizgeyi boş karakterle sonlandırır.

İşlevin dönüş değeri hata iletisinin tamamının uzunluğudur. Bu değer *uzunluk*'tan küçükse dizge kırpılmamış demektir, bu dizgeyi kullanabilirsiniz, aksi takdirde işlevi daha büyük bir tamponla yeniden çağırmalısınız.

Bu örnekte kullanılan **regerror** çağrısında hata iletisi için gereken alan özdevimli olarak ayrılmaktadır:

```
char *get_regerror (int errcode, regex_t *compiled)
{
  size_t length = regerror (errcode, compiled, NULL, 0);
  char *buffer = xmalloc (length);
  (void) regerror (errcode, compiled, buffer, length);
  return buffer;
}
```

4. Kabuk Usulü Sözcük Yorumlama

Sözcük yorumlama, kabuğun yaptığı gibi değişkenleri, komutları ve dosya ismi kalıplarını sözcüklerine ayırma ve bunların yerine bazı ikameler yapma işlemlerini ifade eder.

Örneğin, **`ls -l foo.c`** yazarsanız bu dizgeyi kabuk üç sözcüğe ayırır: **`ls`**, **`-l`** ve **`foo.c`**. Bu sözcük yorumlamanın en temel işlemidir.

`ls *.c` yazarsanız, bu dizge çok sayıda sözcüğe ayrışır, çünkü **`*.c`** sözcüğü çok sayıda dosya ismiyle değiştirilecektir. Bu işleme ***dosyaismi yorumlaması*** denir ve sözcük yorumlamanın bir parçasıdır.

`echo $PATH` kullanırsanız, dosya yolunuz basılır. Burada sözcük yorumlamanın başka bir parçası olan ***değişken ikamesi***'nden yararlanılır.

Sıradan yazılımlar kabuğun uyguladığı gibi sözcük yorumlaması uygulamak için bir kütüphane işlevi olan **`wordexp`** işlevini kullanabilirler.

4.1. Sözcük Yorumlama Katmanları

Bir sözcük dizilimine sözcük yorumlaması uygulandığında, sırasıyla aşağıdaki dönüşümler yapılır:

1. ***Yaklaşık yorumlaması***: **`~foo`** dizgesinin **`foo`** kullanıcısının ev dizini ismi ile değiştirilmesi.
2. Bundan sonra, aynı adımda soldan sağa 3 adımda şu dönüşümler uygulanır:
   - ***Değişken ikamesi***: **`$foo`** gibi başvurular için ortam değişkenleri ikame edilir.
   - ***Komut İkamesi***: **`` `cat foo` ``** gibi bir dizge ya da eşdeğeri olan **`$(cat foo)`** dizgesi kullanıldığında kabuk içteki dizgeyi komut olarak çalıştırır ve bu dizgeyi komutun çıktısı ile değiştirir.
   - ***Aritmetik yorumlama***: **`$(($x-1))`** gibi bir dizge, ifadenin sonucu ile değiştirilir.
3. ***Sözcüklere ayırma***: Dizgenin sözcüklerine ayrılması.
4. ***Dosyaismi yorumlaması***: **`*.c`** gibi bir sözcük, uzantısı `.c` olan dosyaların listesi ile değiştirilir. Dosyaismi yorumlaması sözcüğün tamamına bir defada uygulanır ve sözcük kendisiyle eşleşen sıfır ya da daha fazla dosya ismiyle değiştirilir.
5. ***Tırnak kaldırma***: Dizgeyi sarmalayan tırnakları kaldırılması; yukarıdaki yorumlamalar uygulandıktan sonra yukarıdaki yorumlamaların sonucu olmayan ve öncelenmemiş tüm **`\`**, **`'`** ve **`"`** karakterleri kaldırılır.

Bu dönüşümler hakkında daha ayrıntılı bilgiyi GNU Bash Başvuru Kılavuzu'nun Kabuk Yorumları[B256] kısmında bulabilirsiniz.

4.2. `wordexp` çağrısı

Sözcük yorumlaması ile ilgili tüm işlevler, sabitler ve veri türleri `wordexp.h` başlık dosyasında bulunur.

Sözcük yorumlaması sonuç olarak bir dizge göstericileri dizisi üretir. Bu dizgeyi döndürmek için **wordexp** işlevi bir yapı olan özel bir veri türü, **wordexp_t** kullanır. **wordexp** işlevine bu yapının adresini aktarırsanız, işlev sonuç hakkında bilgiyi, bu yapının elemanlarına kaydederek döndürür.

wordexp_t veri türü

Bu veri türü, bir sözcük göstericileri dizisine bir gösterici tutar. Daha açık ifade etmek gerekirse, sözcük göstericileri dizisinin adresini ve eleman sayısının kaydını tutar.

`we_wordc`
: Gösterici dizisinin eleman sayısı.

`we_wordv`
: Gösterici dizisinin adresi. Bu alan **char **** türündedir.

`we_offs`
: Gösterici dizisindeki ilk gerçek elemanın konumu veya ilk gerçek elemana erişmek için **we_wordv** alanındaki adrese eklenecek sayı. Diğer alanların aksine bu alan işlev için bir girdidir; işlev diğer alanları çıktıyı döndürmek için kullanır.

  Konum olarak sıfırdan farklı bir değer belirtirseniz, dizinin bu elemanından önceki elemanları boş kalır (işlev, bu konumları boş göstericilerle doldurur).

  we_offs alanı sadece **WRDE_DOOFFS** seçeneği etkinse anlamlıdır. Aksi takdirde bu alanın içerdiği değere bakılmaksızın gösterici dizisinin ilk elemanının sıfırıncı eleman olduğu varsayılır.

```
int wordexp(const char *sözcükler,                                  işlev
            wordexp_t  *gst–dizisi,
            int         derleme–seçenekleri)
```

İşlev, *sözcükler* dizgesine sözcük yorumlaması uygular ve sonucu ayırdığı bir gösterici dizisinde saklayarak bu dizinin adresini ve eleman sayısını **gst–dizisi* içinde döndürür. *derleme–seçenekleri* argümanını bit değerli seçenekleri belirtmek için kullanabilirsiniz; bu seçenekler için *Sözcük Yorumlama Seçenekleri* (sayfa: 227) bölümüne bakınız.

sözcükler dizgesinde **|** **;** karakterlerini tersbölü karakteri ile öncelemeden kullanmamalısınız. Aksi takdirde **WRDE_BADCHAR** hata kodunu alırsınız. Bir sözcük yorumlama oluşumunu sarmalamak dışında parantez ve kaşlı ayraç kullanmayın. Eğer tırnak karakteri olarak **"** karakterini kullanıyorsanız, bir ikincisi ile kapatmalısınız.

Sözcük yorumlamasının sonucu bir sözcük dizisidir. **wordexp** işlevi sonuçlanan her sözcük için bir dizge ve bu dizgelerin göstericilerinden oluşan **char **** türünde bir dizi ayırır. Gösterici dizisinin son elemanı bir boş göstericidir. Bu diziye *dizge göstericileri dizisi* denir.

wordexp dizinin adresini ve eleman sayısını (sonlandırıcı boş gösterici hariç) *gst–dizisi* ile gösterilen yapı içinde döndürür.

wordexp başarılı olduğu takdirde 0 ile aksi takdirde şu hata kodlarından biri ile döner:

`WRDE_BADCHAR`
: *sözcükler* girdi dizgesi **|** gibi öncelenmemiş geçersiz karakter içeriyor.

`WRDE_BADVAL`
: Girdi dizgesi tanımlanmamış bir kabuk değişkenini ifade ediyor ve bu tür ifadeleri yasaklayan **WRDE_UNDEF** seçeneğini belirtmişsiniz.

WRDE_CMDSUB
: Girdi dizgesi komut ikamesi içeriyor ve siz bunların kullanımını **WRDE_NOCMD** seçeneğini kullanarak yasaklamışsınız.

WRDE_NOSPACE
: Sonucu yerleştirmek için bellek ayrılamıyor. Bu durumda işlev yer ayırabildiği kadar sonuçla döner.

WRDE_SYNTAX
: Girdi dizgesinde bir sözdizimi hatası var. Örneğin, tırnak karakterinin eşi yoksa bu bir sözdizimi hatasıdır.

`void` **wordfree**`(wordexp_t *`*gst–dizisi*`)` işlev

**gst–dizisi* yapısında adresi belirtilen gösterici dizisini serbest bırakır. İşlev **gst–dizisi*'nin kendisini serbest bırakmaz.

4.3. Sözcük Yorumlama Seçenekleri

Bu bölümde **wordexp** işlevinin *derleme–seçenekleri* argümanında belirtebileceğiniz seçenekler açıklanacaktır. Seçtiğiniz seçenekleri bir C işleci olan **|** ile birleştirerek kullanabilirsiniz.

WRDE_APPEND
: Önceki bir **wordexp** çağrısı ile üretilmiş sözcük göstericileri dizisine bu yorumlamadan sözcükler eklenir. Bu yolla bir dizgenin sonuna aralarında boşluk bırakarak sözcük ekler gibi diziye sözcük ekleyebilirsiniz.

  Ekleme işlemi sırasında, **wordexp** çağrıları arasında sözcük göstericileri dizisinde değişiklik yapmamalısınız. Ayrıca ilk **wordexp** çağrısında **WRDE_DOOFFS** seçeneğini belirtmişseniz, ekleme işlemi için yaptığınız çağrıda da bu seçeneği belirtmelisiniz.

WRDE_DOOFFS
: Sözcük gösterici disinin başlangıcındaki elemanların boş kalmasını sağlar. Kaç elemanın boş kalacağı **we_offs** üyesinde belirtilir. Boş elemanlar boş gösterici içerir.

WRDE_NOCMD
: Komut ikamesi yapılmaz; eğer girdi komut ikamesini gerektiriyorsa, bir hata raporlanır.

WRDE_REUSE
: Önceki bir **wordexp** çağrısı ile üretilmiş sözcük göstericileri dizisini yeniden kullanılabilir yapar. Bu seçeneğin kullanıldığı bir **wordexp** çağrısında yeni bir gösterici dizisi ayrılmaz, mevcut olan (gerekirse genişletilerek) kullanılır.

  Bu işlem sırasında dizi başka bir bellek bölgesine taşınabileceğinden eski göstericiyi saklayıp bu çağrıdan sonra onu kullanmamamalısınız. Her çağrıdan sonra göstericiyi **we_pathv** üyesinden öğrenebilirsiniz.

WRDE_SHOWERR
: Komut ikamesinde çalıştırılan komutlardan dönen hata iletileri gösterilir. Yani, bu komutların sürecin standart hata akımını kullanması sağlanır. Öntanımlı olarak, **wordexp** bu komutlara hata iletilerinin gösterilmediği bir standart hata akımı atar.

WRDE_UNDEF
: Eğer girdi tanımsız bir kabuk değişkeni içeriyorsa bir hata raporlanır.

4.4. `wordexp` Örneği

Bu örnekte **`wordexp`** işlevi muhtelif dizgeleri yorumlayıp sonuçlarını birer kabuk komutu olarak kullanmaktadır. Ayrıca, **`WRDE_APPEND`** seçeneği ile **`wordfree`** işlevinin kullanımı da gösterilmiştir.

```
int
yorumla_ve_calistir (const char *komut, const char **secenekler)
{
  wordexp_t sonuc;
  pid_t pid
  int durum, i;

  /* Çalıştırılacak komut için dizgeyi yorumlayalım.  */
  switch (wordexp (komut, &sonuc, 0))
    {
    case 0:                     /* Başarılı.  */
      break;
    case WRDE_NOSPACE:
      /* WRDE_NOSPACE hatası varsa, sonuç kısmende olsa
         ayrılmıştır. Onu serbest bırakalım.  */
      wordfree (&sonuc);
    default:                    /* Bazı başka hatalar.  */
      return -1;
    }

  /* Argümanları elde etmek için dizgeyi yorumlayalım.  */
  for (i = 0; secenekler[i] != NULL; i++)
    {
      if (wordexp (secenekler[i], &sonuc, WRDE_APPEND))
        {
          wordfree (&sonuc);
          return -1;
        }
    }

  pid = fork ();
  if (pid == 0)
    {
      /* Burada bir alt süreç oluşturup komutu çalıştırıyoruz. */
      execv (sonuc.we_wordv[0], sonuc.we_wordv);
      exit (EXIT_FAILURE);
    }
  else if (pid < 0)
    /* Çatallama başarısız oldu, durumu raporlayalım.  */
    durum = -1;
  else
    /* Bu üst süreç, alt sürecin bitmesini bekleyecek.  */
    if (waitpid (pid, &status, 0) != pid)
      durum = -1;

  wordfree (&sonuc);
  return durum;
}
```

4.5. Yaklaşık (~) Yorumlaması Hakkında

Kendi ev dizininizi belirtmek için bir dosya isminin başında kullandığınız ~ işareti kabuk sözdiziminin standart

parçalarından biridir. Bir *kullanıcı*'nın ev dizinini belirtmek için ise ~*kullanıcı* yazabilirsiniz.

Yaklaşık (~) yorumlaması, ~ işaretini ev dizini ismine dönüştürme işlemidir.

Yaklaşık (~) yorumlaması ~ karakteri ile başlayan, ilk boşluk ya da bölü çizgisi karakterine kadar tüm karakterlere uygulanır. ~ karakteri sadece bir sözcüğün başındaysa anlamlıdır; tek başına olsa bile yorumlama yapılır (Kabukta **echo ~** yazın).

Tek başına ~ işareti için HOME ortam değişkeninin değeri kullanılır. ~ karakterinden sonra gelen sözcük kullanıcı veritabanında **getpwname** ile aranır ve veritabanında kayıtlı ev dizini kullanılır. Bu durumda, ~ karakterinden sonra kendi kullanıcı isminizi yazarsanız ve HOME ortam değişkenindeki değer, veritabanında kayıtlı değerden farklıysa, tek başına ~ kullanarak yapılan yorumlamadan farklı bir sonuç dönebilir.

4.6. Değişken İkamesi Hakkında

Kabuk sözdiziminin bir parçası da bir kabuk değişkeninin değerini bir komutta kullanırken **$***değişken* şeklinde kullanmaktır. Buna ***değişken ikamesi*** denir ve yapılan sözcük yorumlamanın bir parçasıdır.

Bir ikame yapmak için bir değişken iki yöntemle kullanılabilir:

${*değişken***}**
: Eğer değişken ismi kaşlı ayraçlar içine alınırsa, değişken ismini bittiği yerin belirsizliği tamamen ortadan kalkar. Böylece değişken ismine sözcük eklenebilir. Örneğin, `${foo}` değişkeninin değeri `traktör` ise `${foo}ler` ifadesinin sonucu `traktörler` olur.

$*değişken*
: Değişken ismi kaşlı ayraçlar içine alınmazsa, değişken ismi **$** ile başlar ve alfanümerik karakterler ve altçizgi karakterinden oluşabilir. İsimden sonra gelen herhangi bir noktalama işareti değişken ismini sonlandırır. Örneğin, `$foo-pulluk` ifadesinin sonucu `traktör-pulluk` olur.

Kaşlı ayraçlar kullanıldığında değeri değiştirmek için çeşitli oluşumları kullanmak ya da bazı sınamalar yapmak mümkün olur.

${*değişken***:-***öntanımlı***}**
: *değişken*'e değer atanmamışsa ya da tanımsızsa değer olarak *öntanımlı* kullanılır, aksi takdirde *değişken*'in değeri kullanılır.

${*değişken***:=***öntanımlı***}**
: *değişken*'e değer atanmamışsa ya da tanımsızsa değer olarak *öntanımlı* kullanılır ve *değişken*'e *öntanımlı* değeri atanır.

${*değişken***:?***ileti***}**
: *değişken*'e değer atanmışsa ya da tanımlıysa, değeri kullanılır. Aksi takdirde, sözcük ikamesinin başarısız olduğu kabulüyle standart hata akımına bir hata iletisi olarak *ileti* basılır.

${*değişken***:+***sözcük***}**
: Sadece, *değişken*'e değer atanmışsa ya da tanımlıysa, değeri kullanılır. Aksi takdirde sonuç hiçbir şey olur.

${#*değişken***}**
: Bu ifadenin değeri, değişkenin değerindeki karakter sayısıdır. Sonuç onluk tabandadır. Örneğin, `${#foo}` ifadesinin sonucu 7'dir, çünkü `traktör` 7 karakter uzunluktadır.

Aşağıdaki değişken ikamelerinde, değişkene bir değer ikame etmeden önce değeri kısmen kaldırılır. *önek* ve *sonek* basitçe birer dizge olmayabilir; dosyaismi kalıpları kullanılabilir. Ancak bu bağlamda bu kalıplar dosya isimleri ile değil, değişkenin değerinin parçaları ile eşleştirilirler.

${*değişken***%%***sonek***}**
değişken'in değeri, değerin sonunda *sonek* kalıbı ile eşleşen kısmı iptal edilerek kullanılır.

sonek ile eşleşen çok sayıda parça varsa en uzun eşleşme iptal edilir.

Örneğin, `${foo%%r*}` ifadesinin sonucu `t` olur, çünkü **`r*`** kalıbı ile değerin sonunda eşleşen en uzun parça `raktör`'dür.

${*değişken***%***sonek***}**
değişken'in değeri, değerin sonunda *sonek* kalıbı ile eşleşen kısmı iptal edilerek kullanılır.

sonek ile eşleşen çok sayıda parça varsa en kısa eşleşme iptal edilir.

Örneğin, `${foo%r*}` ifadesinin sonucu `traktö` olur, çünkü **`r*`** kalıbı ile değerin sonunda eşleşen en kısa parça `r`'dir.

${*değişken***##***önek***}**
değişken'in değeri, değerin başlangıcında *önek* kalıbı ile eşleşen kısmı iptal edilerek kullanılır.

önek ile eşleşen çok sayıda parça varsa en uzun eşleşme iptal edilir.

Örneğin, `${foo##*t}` ifadesinin sonucu `ör` olur, çünkü **`*t`** kalıbı ile değerin başlangıcında eşleşen en uzun parça `trakt`'tır.

${*değişken***#***önek***}**
değişken'in değeri, değerin başlangıcında *önek* kalıbı ile eşleşen kısmı iptal edilerek kullanılır.

önek ile eşleşen çok sayıda parça varsa en kısa eşleşme iptal edilir.

Örneğin, `${foo#*t}` ifadesinin sonucu `raktör` olur, çünkü **`*t`** kalıbı ile değerin başlangıcında eşleşen en kısa parça `t`'dir.

XI. Girdi/Çiktı İşlemlerine Giriş

İçindekiler

Çoğu yazılımlar ya girdi (veri okuma) ya çıktı (veri yazma) işlemi yapar ya da her ikisinide. GNU C kütüphanesi bu girdi/çıktı işlevlerini o kadar geniş bir yelpazede içerir ki, hangi işlevi kullanmak gerektiğine karar vermek işin en zor tarafını oluşturur.

Bu kısımda girdi ve çıktı ile ilgili kavramlara ve terminolojiye bir giriş yapacağız. GNU G/Ç oluşumları ile ilgili diğer kısımlar şunlardır:

- *Akımlar Üzerinde Giriş/Çıkış* (sayfa: 236) kısmı, akımlar üzerindeki işlemler ile biçimli girdi ve çıktı işlemlerini kapsar.
- *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305) kısmı, dosya tanımlayıcılar üzerinden temel G/Ç ve denetim işlevlerini içerir.
- *Dosya Sistemi Arayüzü* (sayfa: 351) kısmı, dizin ve dosyaların erişim kipleri ve dosya iyeliği gibi öznitelikleri değiştirmekle ilgili işlevleri içerir.
- *Borular ve FIFOlar* (sayfa: 393) kısmı, süreçler arası temel iletişim oluşumları hakkında bilgiler içerir.
- *Soketler* (sayfa: 398) kısmı, süreçler arası iletişimin ağ desteğini de içeren daha karmaşık oluşumlarını kapsar.
- *Düşük Seviyeli Uçbirim Arayüzü* (sayfa: 442) kısmında uçbirimler ve diğer seri aygıtlar üzerinde G/Ç işlemlerinin nasıl yapıldığı anlatılmıştır.

1. Girdi/Çıktı Kavramları

Bir dosyayı okumak ya da bir dosyaya yazmak isterseniz, önce dosyaya bir iletişim veya bir bağlantı kanalı oluşturmalısınız. Bu işleme ***dosyanın açılması*** denir. Bir dosyayı okumak veya yazmak için açabileceğiniz gibi her iki işlem için de açabilirsiniz.

Bir açık dosyaya bağlantı için ya bir akım ya da bir dosya tanımlayıcı kullanılır. Bunu dosya okuma ve yazma işlevlerine bir parametre olarak aktararak işleve iletişim için bunu kullanmasını belirtirsiniz. Bir işlev hem akımlarla hem de dosya tanımlayıcılarla çalışamaz, bunlar farklı işlemlerdir.

Bir dosya üzerinde okuma veya yazma işlemini bitirdiğinizde dosyayı kapatarak bağlantıyı sonlandırmalısınız. Bir dosya tanımlayıcı veya akımı kapattıktan sonra dosya üzerinde artık bir okuma veya yazma işlemi yapılamaz.

1.1. Akımlar ve Dosya Tanımlayıcılar

Bir dosya üzerinde giriş ve çıkış işemleri yapmak isterseniz, yazılımınız ile dosya arasında bağlantıyı temsil eden şu iki temel mekanizmadan birini kullanmalısınız: dosya tanımlayıcılar ve akımlar. Dosya tanımlayıcılar `int` türünden nesneler olduğu halde akımlar `FILE *` türünden nesnelerdir.

Dosya tanımlayıcılar girdi ve çıktı işlemleri için daha ilkel ve düşük seviyeli bir arayüzdür. Bir dosya tanıtıcısı veya bir akım normal bir dosya ile bağlantı kurmaktan başka, başka bir süreç ile iletişim kurmak için bir aygıt (örn uçbirim), bir soket veya bir boruhattını da temsil edebilir. Ancak, belli bir aygıta özel işlemleri denetlemek istersseniz bir dosya tanımlayıcı kullanmak zorundasınız, akımları bu tür bir işlem için kullanabilmenizi sağlayacak bir oluşum yoktur. Ayrıca, örneğin, bloklanamayan (ya da kutuplu) girdiler (*Dosya Durum Seçenekleri* (sayfa: 341)) gibi özel kiplerde girdi ve çıktı işlemlerine ihtiyaç duyarsanız yine dosya tanıtıcılarını kullanmak zorundasınız.

Akımlar, dosya tanımlayıcı oluşumlarına göre daha üst seviyede bir arayüzdür. Akımlar, bütün dosya çeşitleri için üç farklı tamponlama tarzı (*Akım Tamponlama* (sayfa: 292)) seçilebilmesi dışında hemen hemen aynıdır.

Akımları kullanamanın en büyük getirisi, dosya tanıtıcılar için sağlanan işlevlerden daha güçlü ve daha zengin girdi ve çıktı işlemi türününün (denetim işlemlerinin aksine) uygulanabilmesidir. Dosya tanımlayıcılar, karakter bloklarını aktarmak için basit işlevler içerirken akım arayüzü, daha güçlü biçimli girdi ve çıktı işlevleri (**`printf`** ve **`scanf`**) yanında karakter ve satır yönlenimli girdi ve çıktı işlevlerini içerir.

Akımlar dosya tanımlayıcılar üzerine kurulduklarından bir akımdaki dosya tanımlayıcıyı çıkarıp düşük seviyeli işlemleri dosya tanımlayıcılar üzerinden uygulayabilirsiniz. Ayrıca, dahili olarak bir bağlantıyı dosya tanımlayıcı olarak açabilir ve bir akımı bu dosya tanımlayıcı ile ilişkilendirebilirsiniz.

Genelde, sadece dosya tanımlayıcılarla yapılabilen işlemleri yapmak dışında akımları kullanmayı tercih etmelisiniz. Başlangıç seviyesinde bir yazılımcı iseniz ve hangi işevi kullanacağınıza karar veremiyorsanız, biçimli girdi ve çıktı (*Biçimli Girdi* (sayfa: 277) ve *Biçimli Çıktı* (sayfa: 255)) işlevlerine yoğunlaşmanızı öneririz.

Yazılımınızın GNU dışındaki sistemlere taşınabilirliği ile ilgileniyorsanız, akımlar kadar taşınabilir olmayan dosya tanımlayıcı işlevlerinden uzak durmalısınız. ISO C çalışan her sistemde akımlara destek olduğunu umabilirsiniz, ama GNU dışı sistemler dosya tanımlayıcıların tamamını desteklemeyebilir veya dosya tanımlayıcılarla çalışan GNU işlevlerinin bir alt kümesini destekliyor olabilir. GNU kütüphanesindeki dosya tanımlayıcı işlevlerin çoğu yine de POSIX.1 standardındadır.

1.2. Dosyada Konumlama

Bir dosya üzerinde okunacak ya da yazılacak bir baytın yerini belirten bir açık dosya özelliği de ***dosya konumu***dur. GNU sisteminde ve tüm POSIX sistemlerinde dosya konumu, dosya başlangıcından itibaren ilgili baytın bulunduğu yeri belirten bir tamsayıdır.

Dosya konumu, dosya ilk açıldığında dosyanın başlangıcına ayarlanır ve her okuma veya yazma işleminde ilerletilir. Yani bir dosyaya erişim normalde sıralıdır.

Sıradan dosyalar için dosyanın herhangi bir yerine yazmak ya da okumak mümkündür. Bazı dosya çeşitlerinde de bu mümkündür. Bunun mümkün olduğu dosyalara bazan ***rasgele erişimli*** dosyalar denir. Dosya konumunu akımlarda **`fseek`** işlevini kullanarak (*Dosyalarda Konumlama* (sayfa: 288)), dosya tanımlayıcılarda **`lseek`** işlevini kullanarak (*Girdi ve Çıktı İlkelleri* (sayfa: 308)) değiştirebilirsiniz. Sıralı erişimi desteklemeyen dosyalar üzerinde dosya konumunu değiştirmeye çalışırsanız **`ESPIPE`** hatasını alırsınız.

Ekleme erişimli açılmış akım ya da tanımlayıcılar özellikle çıktılamaya konudur: bu tür dosyalara çıktılama *daima* dosya *sonuna* dosya konumuna bakılmaksızın, sıralı ekleme olarak yapılır. Aynı dosya üzerinde okuma yapıldığında dosya konumu hala kullanılabilir durumda kalır, çıktılama bunu etkilemez.

Bir dosya üzerinde birden fazla uygulama çalıştığında ne olacağını düşünüyor olabilirsiniz. Her uygulama kendi çalışma bölgesinde çalışır, her uygulamanın kendi dosya göstericisi olduğundan diğer uygulamanın yaptıklarından etkilenmez.

Fiili olarak, her dosya açılışında ayrı bir dosya konumu oluşturulur. Dolayısıyla, aynı yazılım içinde bir dosyayı iki kere açarsanız, iki akım veya dosya tanımlayıcı için birbirinden bağımsız dosya konumlayıcınız olur.

Aksine, bir tanımlayıcı açar ve bu tanımlayıcıyı kopyalayarak bir tanımlayıcı daha elde ederseniz, iki dosya konumlayıcı daima aynı dosya konumunu gösterir, birinin değiştirilmesi diğerinde de aynı etkiyi oluşturur.

2. Dosya İsimleri

Dosyaya bir bağlantı açma sırasında veya bir dosyayı silmek gibi işlemlerde dosyayı belirtmek için bazı yöntemlere ihtiyacınız olur. Hemen hemen tüm dosyalar hatta teyp sürücüleri ve uçbirimler bile dizgelerden oluşan isimlere sahiptir. Bu dizgelere ***dosya isimleri*** denir. Bir dosyayı açmak ve üzerinde işlem yapmak istediğinizde bu ismi belirtirsiniz.

Bu kısımda dosya isimleri için teamüller ile işletim sisteminin onlarla nasıl çalıştığı üzerinde duracağız.

2.1. Dizinler

Dosya isimlerinin sözdizimini anlayabilmek için öncelikle dosya sisteminin dizin hiyerarşisini nasıl düzenlediğini anlamamız gerekir.

Bir ***dizin***, diğer dosyalarla ilişkili bilgiler içeren bir dosyadır; bu ilişkilere ***bağ*** ya da ***dizin girdisi*** denir. Bazan "bir dizindeki dosyalar"dan bahsedilir ama gerçekte dizinler dosyalar için bilgiler içerir, dosyaların kendilerini içermez.

Bir dizin girdisi olarak bir dosya ismine ***dosya ismi bileşeni*** denir. Genelde, bir dosya ismi bölü çizgileri (**/**) ile ayrılmış çok sayıda dizge bileşenden oluşur. Bir dosya ismindeki bileşenler onun dizini ile birlikte ismini belirtir.

POSIX standardı gibi bazı belgelerde bizim dosya ismi dediğimiz şeye *dosyayolu* ve dosya ismi elemanları dediğimiz şeylere de *dosyaismi* ya da *dosyayolu bileşenleri* denir. Biz bu terminolojiyi kullanmayacağız, çünkü "yol" (path) denilen şey tamamen farklı bir şeydir (dosyanın aranacağı dizinlerin listesidir) ve "dosyayolu" dendiğinde kullanıcıların kafasının karıştığını düşünüyoruz. GNU belgelerinde daima "dosya ismi" ve "dosya ismi bileşenleri" (bazan da sadece bileşen) terimlerini kullanacağız. **PATH_MAX** gibi bazı makrolar POSIX terminolojisini kullanır. Bu makrolar POSIX standardında tanımlı olduklarından isimlerini değiştiremiyoruz.

Dizinler üzerinde yapılan işlemlerle ilgili daha ayrıntılı bilgiyi *Dosya Sistemi Arayüzü* (sayfa: 351) bölümünde bulabilirsiniz.

2.2. Dosya İsmi Çözümlemesi

Bir dosya ismi bölü çizgisi (**/**) karakterleri ile ayrılmış dosya ismi bileşenlerinden oluşur. GNU C kütüphanesinin desteklendiği sistemlerde çok sayıda peşpeşe verilen **/** karakterleri tek bir **/** karakterine eşdeğerdir.

Bir dosya isminin hangi dosyaya ait olduğunun saptanması işlemi ***dosya ismi çözümlemesi*** adını alır. Bu işlem dosya ismi üzerinde soldan sağa ilerleyerek bileşenleri saptamak ve önceki bileşen tarafından isimlendirilen dizindeki ardışık bileşenlere tekrar aynı işlemi uygulamak şeklindedir. Şüphesiz, her dosya ismini oluşturan dizinler normal dosya isimleri yerine dizinler olarak mevcut olmalı ve ilgili izinler sürecin erişimi için yeterli olmalıdır; aksi takdirde, dosya ismi çözümlemesi başarısız olur.

Bir dosya ismi bir **/** karakteri ile başlıyorsa, dosya ismindeki ilk bileşen sürecin ***kök dizini***dir (genellikle sistem üzerindeki tüm süreçler aynı kök dizini paylaşır). Böyle, kök dizin ile başlayan bir dosya ismi ***mutlak dosya ismi*** adını alır.

Aksi takdirde, dosya ismindeki ilk bileşen *çalışma dizini* (sayfa: 351) olur. Bu tür dosya isimlerine ***göreli dosya ismi*** denir.

. (nokta) ve **..** (nokta nokta) bileşenleri özel anlama sahiptir. Her dizin bu dosya ismi bileşenlerini içerir. Bu bileşenlerden **.** (nokta) dizinin kendisini, **..** (nokta nokta) ise ***üst dizin***i yani bu dizini içeren dizine bir bağ belirtir. Özel bir durum olarak kök dizindeki **..** (nokta nokta) kök dizinin üst dizini olmadığından kendisini belirtir; bu durumda **/..** ile **/** eşdeğerdir.

Bazı dosya ismi örnekleri:

```
/a
```

Kök dizindeki `a` isimli dosya.

`/a/b`
Kök dizindeki `a` isimli dizindeki `b` isimli dosya.

`a`
İçinde olduğumuz dizindeki `a` isimli dosya.

`/a/./b`
`/a/b` ile aynı.

`./a`
İçinde olduğumuz dizindeki `a` isimli dosya.

`../a`
İçinde olduğumuz dizinin üst dizinindeki `a` isimli dosya.

Bir dosya ismi isteğe bağlı olarak bir **`/`** ile bitebilir. Kök dizini için dosya ismi olarak **`/`** belirtebilirsiniz, ancak boş dizge bir dosya ismi olarak anlamlı değildir. İçinde olduğunuz dizini belirtmek için dosya ismi olarak **`.`** veya **`./`** kullanın.

GNU sisteminde başka bazı sistemlerin aksine dosya isimleri sözdiziminin parçası olarak dosya türleri (veya dosya uzantısı) ve dosya sürümleri için yerleşik bir destek yoktur. Bir çok uygulama dosya isimleri için bu tür kabullerde bulunurlar, örneğin C kaynak dosyaları için genellikle **`.c`** soneki kullanılır, ancak dosya sisteminin kendisinde bu çeşit kabuller bulunmaz.

2.3. Dosya İsmi Hataları

Dosya isimlerini argüman olarak alan işlevler, dosya ismi sözdizimi ile veya dosya isminin bulunması ile ilgili sorunlar olduğunda genellikle bunları **`errno`** hata durumu ile saptarlar. Bu kılavuzda bu tür hatalardan ***olağan dosya ismi hataları*** olarak sözedilecektir.

`EACCES`
Süreç, dosya isminin bir dizin bileşeni için arama izinlerine sahip değil.

`ENAMETOOLONG`
Bu hata ya bir dosya isminin uzunluğunun **`PATH_MAX`** değerinden büyük olduğunda ya da tek başına dosya ismi **`NAME_MAX`** değerinden daha uzun olduğunda kullanılır. Bkz. *Dosya Sistemi Kapasite Sınırları* (sayfa: 795).

GNU sisteminde dosya isimlerinin uzunlukları için zorlayıcı bir sınır yoktur, ancak bazı dosya sistemlerinde bir bileşenin uzunluğu ile ilgili belli bir sınır olabilir.

`ENOENT`
Bu hata dosya ismindeki bir dizin bileşeninin bulunamaması durumunu ya da bileşen bir *sembolik bağ* (sayfa: 365) olduğunda hedef dosyanın mevcut olmadığı durumu bildirir.

`ENOTDIR`
Bir dosya ismindeki bir dizin bileşeni dosya ismi olarak mevcut ama bir dizin değil.

`ELOOP`
Dosya ismi aranırken çok fazla sembolik bağ çözümlendi. Sistem, olası döngüleri saptamak için ilkel bir yöntem olarak bir dosya isminde çözümlenebilecek sembolik bağ sayısına bir sınırlama getirir. Bkz. *Sembolik Bağlar* (sayfa: 365).

2.4. Dosya İsimlerinin Taşınabilirliği

Dosya isimlerinin sözdizimi ile ilgili GNU sisteminde ve diğer POSIX sistemlerinde normalde kullanılan kurallardan *Dosya İsimleri* (sayfa: 232) bölümünde bahsedilmişti. Ancak, diğer işletim sistemleri başka kabullerde bulunabilirler.

Dosya isimlerinin taşınabilirliğinin neden önemli olduğunu belirleyen iki durum vardır:

- Yazılımınızda dosya isimleri sözdizimi ile ilgili bazı önkabuller yapar veya gömülü sabit dosya ismi dizgeleri kullanırsanız, farklı sözdizimi kuralları kullanılan sistemlerde dosya isimlerinin elde edilmesi zorlaşır.
- Yazılımınızın başka sistemlerde çalışması konusuyla ilgilenmiyor olsanız bile, farklı isimlendirme kuralları kullanılan dosyalara erişim yine de mümkün olur. Örneğin, bir ağ üzerinden farklı isimleme kuralları olan bir işletim sisteminde bulunan bir dosyaya erişmek, hatta başka işletim sistemlerinde kullanılan biçimlemelerin kullanıldığı disklerde okuma ve yazma yapmak zorunda kalabilirsiniz.

ISO C standardında dosya ismi sözdizimi hakkında çok az bilgi varır, sadece dosya isimlerinin dizgeler olduğundan bahseder. Dosya isimlerinin uzunlukları ve dosya isimlerinde kullanılabilen karakterler ile ilgili değişen kısıtlamalara ek olarak farkı işletim sistemlerinde örneğin, dizinlerin yapıları ve dosya türleri ile uzantıları gibi kavramlar için farklı kabuller ve sözdizim kuralları uygulanır. Örneğin dosya sürümleri gibi bazı kavramlar için bazı sistemlerde destek varken bazılarında yoktur.

POSIX.1 standardı, dosya ismi bileşeni dizgeleri ve dosya isimlerinin uzunlukları ile dosya isimlerinde kullanılan karakterlerle ilgili dosya ismi sözdizimine ek kısıtlamalar getiren gerçeklemelere izin verir. Yine de, GNU sistemi bu sınırlamalara ihtiyaç duymaz; dosya isimlerinde boş karakter hariç her karakter kullanılabilir ve dosya ismi dizgeleri ile ilgili hiçbir sınırlama da yoktur.

XII. Akımlar Üzerinde Giriş/Çıkış

İçindekiler

Bu oylumda akımları oluşturmak ve üzerlerinde giriş ve çıkış işlemleri yapmak için kullanılan işlevler anlatılmıştır. *Girdi/Çıktı İşlemlerine Giriş* (sayfa: 231) oylumunda değinildiği gibi, bir akım bir dosya, aygıt ya da sürece bir iletişim kanalı sağlayan yüksek seviyeli ve tamamen soyut bir kavramdır.

1. Akımlar (Streams)

Tarihsel sebeplerle, bir akımı ifade eden C veri yapısının türü "stream" değil **FILE**'dır. Kütüphane işlevlerinin çoğu **FILE*** türü nesnelerle ilgilendiğinden bazan ***dosya göstericisi*** deyimi "akım" anlamında kullanılmıştır. Bu, birçok C kitabında terminoloji bakımından talihsiz sayılabilecek karışıklıklara yol açmıştır. Bu kılavuzda "dosya" ve "akım" terimleri teknik duyarlılıkla ve dikkatle kullanılmıştır. **FILE** türü `stdio.h` başlık dosyasında bildirilmiştir.

FILE veri türü

Akım nesneleri için kullanılan veri türüdür. Bir **FILE** türünden nesne, ilişkilendirildiği dosyayla yapılan bağlantı hakkında dosya konum belirteci ve tamponlama bilgisi gibi şeyleri içeren dahili durum bilgisinin tamamını tutar. Her akım ayrıca **ferror** ve **feof** işlevleri ile sınanabilen hata ve dosyasonu durum belirteçlerine de sahiptir. Bkz. *Dosya Sonu ve Hatalar* (sayfa: 286).

FILE türünden nesneler dahili olarak giriş/çıkış kütüphanesi işlevleri tarafından ayrılır ve yönetilir. **FILE** türünden kendi nesnenizi oluşturmaya çalışmayın. Bunu yaparsanız bu nesnelerin kendileri ile değil sadece göstericileri (**FILE *** değerleri) ile çalışabilirsiniz. Bu nedenle kütüphaneyi kullanmalısınız.

2. Standart Akımlar

Yazılımınızda, **main** işlevi çağrıldığı anda üç tane önceden tanımlanmış ve kullanıma hazır akıma sahip olur. Bunlar süreç için oluşturulmuş "standart" giriş ve çıkış kanallarıdır.

Bu akımlar `stdio.h` başlık dosyasında bildirilmiştir.

`FILE *`**stdin** değişken

Yazılım için normal giriş kaynağı olan ***standart girdi*** akımıdır.

`FILE *`**stdout** değişken

Yazılımın normal çıktısı için kullanılan ***standart çıktı*** akımıdır.

`FILE *`**stderr** değişken

Yazılım tarafından hata iletileri ve tanı amaçlı kullanılan ***standart hata*** akımıdır.

GNU sisteminde kabuk tarafından sağlanan boruhattı ve yönlendirme oluşumlarını kullanarak bu akımlara uygun süreçleri ve dosyaları belirtebilirsiniz. (Kabuklarda bu oluşumları gerçekleştirmekte kullanılan ilkeller *Dosya Sistemi Arayüzü* (sayfa: 351) bölümünde açıklanmıştır.) Diğer işleim sistemlerinin çoğu benzer mekanizmaları sağlar ancak kullanımı ile ilgili ayrıntılar değişiklik gösterir.

GNU C kütüphanesinde **stdin**, **stdout** ve **stderr** normal değişkenlerdir ve diğer değişkenler gibi onlara da değer atanabilir. Örneğin, bir dosyayı standart çıktıya yönlendirmek isterseniz şöyle yapmalısınız:

```
fclose (stdout);
stdout = fopen ("standart-cikti-dosyasi", "w");
```

Bilgi

Diğer sistemlerde **stdin**, **stdout** ve **stderr** normal yoldan birşeyler atayamayacağınız makrolardır. Ancak **freopen** işlevini onların kapatılması ve yeniden açılması etkilerini elde etmek için kullanabilirsiniz. Bkz. *Akımların Açılması* (sayfa: 238).

Üç akım; **stdin**, **stdout** ve **stderr** sürecin başlangıcında yönlenimsiz değildir (Bkz. *Akımlar ve Uluslararasılaştırma* (sayfa: 244)).

3. Akımların Açılması

Bir dosyanın **fopen** işlevi ile açılması bir yeni akım ve bu akım ile dosya arasında bir bağlantı oluşturur. Bu bir yeni dosya oluşturulmasına sebep olabilir.

Bu kısımda açıklanan herşey `stdio.h` başlık dosyasında bildirilmiştir.

```
FILE *fopen(const char *dosyaismi,                                    işlev
            const char *açıştürü)
```

dosyaismi dosyasına G/Ç için bir akım açar ve bu akıma bir gösterici ile döner.

açıştürü argümanı dosyanın nasıl açılacağını ve sonuçlanan akımın özniteliklerini belirtmekte kullanılan bir dizgedir. Aşağıdaki dizgelerden biri ile başlamalıdır:

`r`
: Mevcut bir dosyayı salt–okunur olarak açar.

w
: Dosyayı sadece yazmak için açar. Dosya zaten varsa içeriği silinir. Aksi takdirde yeni bir dosya oluşturulur.

a
: Bir dosyayı sadece sonuna yazmak için açar. Dosya zaten varsa, içeriği değiştirilmez ve akıma yapılan çıktı dosyanın sonuna eklenir. Aksi takdirde bir yeni, boş dosya oluşturulur.

r+
: Bir dosyayı oku–yaz olarak açar. Dosyanın içeriği değiştirilmez ve konumlayıcı dosyanın başına yerleştirilir.

w+
: Bir dosyayı oku–yaz olarak açar. Dosya zaten varsa içeriği silinir. Aksi takdirde yeni bir dosya oluşturulur.

a+

Dosyayı hem okumak hemde eklemek için açar ya da oluşturur. Dosya zaten varsa içeriği değiştirilmez. Aksi takdirde yeni bir dosya oluşturulur. Dosya konumlayıcı okumak için dosyanın başına konumlansa da çıktı daima dosyanın sonuna eklenir.

Gördüğünüz gibi, **+** işareti bir akımın hem girdi hem de çıktı yapabilmesini ister. ISO standardına göre böyle bir akımı kullanırken **fflush** çağrısı (Bkz. *Akım Tamponlama* (sayfa: 292)) yapmalı ya da okumadan yazmaya veya yazmadan okumaya geçerken **fseek** (Bkz. *Dosyalarda Konumlama* (sayfa: 288)) gibi bir dosya konumlama işlevine çağrı yapmalısınız. Aksi takdirde, dahili tamponlar olması gerektiği gibi boşaltılamaz. GNU C kütüphanesi için böyle bir sınırlama yoktur; bir akım üzerinde okuma ve yazma işlemlerini istediğiniz sırada yapabilirsiniz.

Bunlardan sonra çağrı için bayrakları belirtmek için ek karakterler kullanılabilir. Daima önce kipi (**r**, **w+**, vs.) yerleştirin; bu tüm sistemler tarafından anlaşılacağı garantili olan tek parçadır.

GNU C kütüphanesi *açıştürü* argümanında kullanmak için bir ek karakter tanımlar: **x** karakteri yeni bir dosya oluşturmaya zorlar — eğer *dosyaismi* diye bir dosya varsa, **fopen** onu açmak yerine başarısız olur. **x** kullanırsanız bir mevcut dosyanın bilmeden üzerine yazmamayı garantilemiş olursunuz. Bu, **open** işlevinin O_EXCL seçeneğine eşdeğerdir (Bkz. *Dosyaların Açılması ve Kapatılması* (sayfa: 306)).

açıştürü argümanındaki **b** karakteri bir standart anlama sahiptir; bir metin akımı yerine bir ikilik akım isteğinde bulunur. Fakat POSIX sistemlerinde (GNU dahil) bu bir fark oluşturmaz. **+** ve **b** birlikte belirtilirse, herhangi bir sırada yazılabilirler. Bkz. *İkilik ve Metin Akımları* (sayfa: 287).

açıştürü argümanı **,ccs=***DİZGE* içeriyorsa, buradaki *DİZGE* bir kodlu karakter kümesinin ismi olarak alınır ve **fopen** akımı geniş–yönlenimli olarak imler, yani *DİZGE* karakter kümesi ile ilgili dönüşümlere uygun dönüşüm işlevleri yerleştirilir. Başka bir akım dahili olarak yönlenimsiz açılır ve yönlenime ilk dosya işleminde karar verilir. Eğer ilk işlem bir geniş karakter işlemiyse, akım sadece geniş yönlenimli olarak imlenmez ayrıca yerelin kodlu karakter kümesine dönüşüm işlevleri yüklenir. Bu artık bu noktadan sonra yerelin seçildiği **LC_CTYPE** kategorisi değiştirilse bile değişmeyecektir.

açıştürü argümanında başka karakterler varsa bunlar yoksayılır. Bunlar diğer sistemler için anlamlı olabilir.

Açma işlemi başarısız olursa **fopen** bir boş gösterici ile döner.

Kaynak dosyalar bir 32 bitlik makinada **_FILE_OFFSET_BITS == 64** ile derlendiğinde LFS arayüzü eski arayüzle şeffaf olarak değiştirildiğinden bu işlev için **fopen64** tanımı kullanılır.

Aynı dosya için aynı anda açılmış çok sayıda akıma (ya da dosya tanıtıcı) sahip olabilirsiniz. Sadece girdi yapacaksanız bu doğru dürüst çalışır ama çıktı akımları da varsa dikkatli olmalısınız. Bkz. *Akımlarla Tanıtıcıları Karıştırmanın Tehlikeleri* (sayfa: 316). Bu durum, akımlar tek süreç tarafından da açılsa, çok sayıda süreç tarafından da açılsa hemen hemen aynıdır. Aynı anda erişimden kaçınmak için dosya kilitleme oluşumları yararlı olabilir. Bkz. *Dosya Kilitleri* (sayfa: 346).

```
FILE *fopen64(const char *dosyaismi,                                    işlev
              const char *açıştürü)
```

Bu işlev **fopen** işlevine benzer ama, akıma **open64** işlevi kullanılarak açılan dosya için bir gösterici döner. Diğer yandan bu akım 32 bitlik makinalar üzerinde 2^{31} bayttan büyük dosyalar üzerinde bile çalışır.

Bilgi

Dönen gösterici yine **FILE *** türünde olacaktır. Çünkü LFS arayüzü için özel bir **FILE** türü yoktur.

Kaynak dosyaları bir 32 bitlik makina üzerinde **_FILE_OFFSET_BITS == 64** ile derlenirse bu işlev **fopen** ismi ile kullanılabilir yani eski arayüz ile şeffaf olarak yer değiştirilir.

```
int FOPEN_MAX                                                    makro
```

Bu makronun değeri gerçeklemenin garanti edebildiği aynı anda açılabilecek en büyük akım sayısına karşılık olan bir sabit ifadesidir. Bu değerden daha fazla sayıda akım açılabilir ama garantili değildir. Bu sabitin değeri üç standart akımı da (**stdin**, **stdout** ve **stderr**) içererek en azı sekizdir.

POSIX.1 sistemlerde bu değer **OPEN_MAX** parametresi tarafından belirlenir. Bkz. *Genel Sınırlar* (sayfa: 784).

BSD ve GNU'da **RLIMIT_NOFILE** özkaynak sınırı tarafından belirlenir. Bkz. *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

```
FILE *freopen(const char *dosyaismi,                             işlev
              const char *açıştürü,
              FILE       *akım)
```

Bu işlev, **fclose** ve **fopen** işlevlerinin bir birleşimi gibidir. Önce *akım* ile gösterilen akımı kapatır, kapatırken oluşan hatalar yoksayılır. (Hataların yoksayılmasından dolayı, akıma bir çıktılama yaptıysanız bir çıktı akımı üzerinde **freopen** kullanmamalısınız.) Sonra da *dosyaismi* isimli dosyayı **fopen** işlevindeki türlerden biri olarak *açıştürü* kipinde açar ve aynı *akım* nesnesi ile ilişkilendirir.

İşlem başarısız olursa bir boş gösterici döner; aksi takdirde **freopen** işlevi *akım* ile döner.

freopen işlevi geleneksel olarak **stdin** gibi standart akımlara kendi seçtiğiniz bir dosyayla bağlanmak için kullanılır. Bu, bir standart akıma belirli amaçlar için doğrudan yazarak kullanmak için yararlıdır. GNU C kütüphanesinde standart akımları basitçe kapatabilir ve yeni birini **fopen** ile açabilirsiniz. Ancak bazı sistemler bu tarz çalışmayı desteklemez, bu nedenle **freopen** kullanımı daha taşınabilir olacaktır.

Kaynak dosyalar bir 32 bitlik makinada **_FILE_OFFSET_BITS == 64** ile derlendiğinde LFS arayüzü eski arayüzle şeffaf olarak değiştirildiğinden bu işlev için **freopen64** tanımı kullanılır.

```
FILE *freopen64(const char *dosyaismi,                           işlev
                const char *açıştürü,
                FILE       *akım)
```

Bu işlev **freopen** işlevine benzer. Tek fark, bu akım 32 bitlik makinalar üzerinde 2^{31} bayttan büyük dosyalar üzerinde bile çalışır. *akım* tarafından gösterilen akımın kipi bu işlev için önemli olmadığından, akımın **fopen64** veya **freopen64** kullanarak açılmış olmaması gerekir.

Kaynak dosyaları bir 32 bitlik makina üzerinde **_FILE_OFFSET_BITS == 64** ile derlenirse bu işlev **freopen** ismi ile kullanılabilir yani eski arayüz ile şeffaf olarak yer değiştirilir.

Bazı durumlarda, verilen akımın okuma ve yazma için kullanılabilir olup olmadığını bilmek kullanışlıdır. Bu bilgi normalde kullanılabilir değildir ve ayrıca hatırlanması gerekir. Solaris, akım tanımlayıcıdan bu bilgiyi alacak bir kaç işlev içerir ve bu işlevler GNU C kütüphanesinde de vardır.

```
int __freadable(FILE *akım)                                      işlev
```

__freadable işlevi *akım* ın okuma iznine sahip olup olmadığını saptar. Okuma izni varsa sıfırdan farklı bir değer ile döner. Sadece yazma izni olan akımlarda işlev sıfır ile döner.

Bu işlev `stdio_ext.h` başlık dosyasında bildirilmiştir.

```
int __fwritable(FILE *akım)                                      işlev
```

__fwritable işlevi *akım* ın yazma iznine sahip olup olmadığını saptar. Yazma izni varsa sıfırdan farklı bir değer ile döner. Sadece okuma izni olan akımlarda işlev sıfır ile döner.

Bu işlev `stdio_ext.h` başlık dosyasında bildirilmiştir.

Biraz daha farklı çeşitte sorunlar için iki işlev daha vardır. Bunlar biraz daha hassas bilgiler sağlar.

```
int __freading(FILE *akım)                                                    işlev
```

__freading işlevi *akım* 'dan son olarak okuma işlemi yapılmışsa ya da akım salt okunur olarak açılmışsa sıfırdan farklı bir değerle döner, aksi takdirde sıfır ile döner. Bir akımın okuma ve yazma için açılıp açılmadığının saptanması işlemi son yazma sırasında kullanılmışsa içerik, tampon ve diğer başka şeyler arasında sonuçların yazılmasına izin verilir.

Bu işlev `stdio_ext.h` başlık dosyasında bildirilmiştir.

```
int __fwriting(FILE *akım)                                                    işlev
```

__fwriting işlevi *akım* 'a son olarak yazma işlemi yapılmışsa ya da akım salt yazılır olarak açılmışsa sıfırdan farklı bir değerle döner, aksi takdirde sıfır ile döner.

Bu işlev `stdio_ext.h` başlık dosyasında bildirilmiştir.

4. Akımların Kapatılması

Bir akım **fclose** ile kapatıldığında akım ile dosya arasındaki bağlantı kesilir. Bir akım kapatıldıktan sonra artık onun üzerinde bir işlem yapılamaz.

```
int fclose(FILE *akım)                                                        işlev
```

Bu işlev *akım* 'ın kapatılmasına ve dosya ile bağlantısının kesilmesine sebep olur. Tamponlanmış çıktılar yazılır, tamponlanmış girdiler ise iptal edilir. Dosya başarıyla kapatılırsa işlev sıfırla döner, aksi takdirde **EOF** ile döner.

Bir çıkış akımını kapatmak için **fclose** çağrısı yapıldığında hataların saptanması önem kazanır, çünkü gerçek hayatta hergün aldığınız hatalar bu sırada oluşabilir. Örneğin, **fclose** tampondaki çıktıyı yazarken dsik doluverirse bir hata alabilirsiniz. Hatta tamponun boş olduğunu bildiğiniz durumda bile eğer NFS kullanıyorsanız dosya kapatılırken hata alabilirsiniz.

Bu işlev `stdio.h` başlık dosyasında bildirilmiştir.

GNU C kütüphanesinde tüm akımları kapatmak için ayrı bir işlev vardır.

```
int fcloseall(void)                                                           işlev
```

Bu işlev sürecin açık olan tüm akımlarını kapatır ve bunlarla bağlantılı dosyalara olan bağlantılar kesilir. Tüm tamponlanmış veri yazılır, tamponlanmış girdiler ise iptal edilir. İşlev, tüm dosyalar hatasız kapatılırsa sıfır ile, hata oluşursa **EOF** ile döner.

Bu işlev çok özel durumlarda örneğin bir hata oluşup sürecin sonlandırılması gerektiğinde kullanılmalıdır. Normalde her akım sorunlarının tanımlanabilmesi için ayrı ayrı kapatılmalıdır. Ayrıca bu işlev *standart akımları* (sayfa: 237) da kapatacağından sorunlar çıkarır.

Bu işlev `stdio.h` başlık dosyasında bildirilmiştir.

Yazılımınız **main** işlevinden normal çıkış yaparsa ya da bir **exit** (sayfa: 681) çağrısı ile çıkarsa tüm açık akımlar özdevinimli olarak düzgünce kapatılır. Yazılımınız başka bir şekilde (örneğin bir **abort** (sayfa: 683) çağrısı ile ya da bir *ölümcül sinyal* (sayfa: 601) ile) sonlandırılırsa açık akımlar düzgünce kapatılamayabilir. Tamponlanmış çıktılar dosyalara yazılamayabilir ve dosyalar eksik ya da boş kalabilir. Akımların tamponlanması ile ilgili daha ayrıntılı bilgi *Akım Tamponlama* (sayfa: 292) bölümünde bulunabilir.

5. Akımlar ve Evreler

Akımlar çok evreli yazılımlarda tek evreli yazılımlardaki gibi kullanılır. Ancak yazılımcı olası karışıklıkların farkında olmalıdır. Birçok akım işlevinin tasarımı ve gerçeklenmesi çok evreli yazılım geliştirmeyle ilgili ek gereksinimlerden oldukça etkileneceğinden yazılımın birinin evreleri asla kullanamayabileceğininin bilinmesi ayrıca önemlidir.

POSIX standardı öntanımlı olarak akım işlemlerinin atomik olmasını gerektirir. Örneğin iki akım işleminin iki evre halinde aynı anda bir akıma uygulanması işlemler sırayla yapılıyormuş gibi yapılmasına sebep olacaktır. Okuma ve yazma sırasında uygulanan tampon işlemleri aynı akımı kullanan diğerlerinden korunur. Bu, her akımın yapılacak bir çalışma öncesi (dolaylı olarak) elde edilen bir dahili kilitleme nesnesine sahip olması ile sağlanır.

Ancak bunun yeterli olmadığı ya da bunu istenmediği durumlar da vardır. Eğer yazılım birden fazla akım işlevi çağrısının atomik olarak yapılmasını gerektiriyorsa dolaylı kilitleme mekanizması yetersiz kalır. Bir yazılımın üretmek istediği bir satırı çeşitli işlev çağrılarıyla oluşturması buna bir örnek olarak verilebilir. Atomlara ayırma işi tüm işlev çağrıları üzerinde değil, işlevlerin kendileri tarafından kendi işlemlerini atomlarına ayırarak yapılmalıdır. Bunun olabilmesi için de akım kilitleme işleminin yazılım kodunda yapılması gerekir.

`void` **`flockfile`**`(FILE *`*akım*`)` işlev

> **`flockfile`** işlevi *akım* ile ilişkili dahili kilitleme nesnesi elde etmekte kullanılır. İşlev, başka hiçbir evrenin doğrudan **`flockfile`**/**`ftrylockfile`** üzerinden veya dolaylı olarak bir akım işlevi çağrısı üzerinden akımı kilitleyememesini sağlar. Kilit elde edilinceye kadar evre engellenecektir. **`funlockfile`** işlevine yapılacak bir doğrudan çağrı kilit nesnesini serbest bırakacaktır.

`int` **`ftrylockfile`**`(FILE *`*akım*`)` işlev

> **`ftrylockfile`** işlevi **`flockfile`** işlevi gibi *akım* ile ilişkili dahili kilitleme nesnesi elde etmeye çalışır. **`flockfile`** işevinin aksine bu işlev kilit kullanılabilir değilse engelleme yapmaz. Kilit başarıyla elde edilirse işlev sıfırla döner, aksi takdirde akımı başka bir evre kilitlemiştir.

`void` **`funlockfile`**`(FILE *`*akım*`)` işlev

> **`funlockfile`** işlevi *akım* 'ın kilitleme nesnesini serbest bırakır. Akım bir **`flockfile`** çağrısı ya da başarılı bir **`ftrylockfile`** çağrısı tarafından önceden kilitlenmiş olmalıdır. Akım işlemleri tarafından uygulanan dolaylı kilitlemeler sayılmaz. **`funlockfile`** işlevi bir hata durumu döndürmez ve o anki evre tarafından kilitlenmemiş bir akım için yapılan bir çağrının davranışı tanımsızdır.

Yukarıdaki işlevlerin nasıl kullanılacağını gösteren ve çok evreli yazılımlarda bile kullanılabilecek aşağıdaki örnekte bir çıktı satırı atomik olarak üretilmektedir (evet, aynı iş tek bir **`fprintf`** çağrısı ile yapılabilirdi ama bazan bu mümkün olmaz):

```
FILE *fp;
{
   …
   flockfile (fp);
   fputs ("This is test number ", fp);
   fprintf (fp, "%d\n", test);
   funlockfile (fp)
}
```

Doğrudan kilitleme olmaksızın **`fputs`** çağrısı döndükten sonra ve **`fprintf`** işlevinin `number` sözcüğünden sonra numarayı basmasından önce başka bir evrenin *fp* akımını kullanması mümkün olabilir.

Bu açıklamalardan sonra akımlardaki kilitleme nesnelerinin basit karşılıklı red nesneleri (mutexes – mutual exclusion objects) olmadığı anlaşılmış olmalıdır. Aynı akımın aynı evre içinde iki kere kilitlenmesi mümkün

olduğundan kilitleme nesneleri, ardışık karşılıklı red nesnelerine eşdeğer olmalıdır. Bu karşılıklı red nesneleri sahibinin izini takibeder ve kendi sayısınca kilit elde eder. Akımdaki kilitleme nesnelerini tamamen serbest bırakmak için aynı evreler tarafından aynı sayıda **funlockfile** çağrısı gereklidir. Örneğin:

```
void
foo (FILE *fp)
{
  ftrylockfile (fp);
  fputs ("in foo\n", fp);
  /* Bu çok yanlış!!!  */
  funlockfile (fp);
}
```

Burada önemli olan, **funlockfile** işlevinin *sadece* **ftrylockfile** işlevinin akımı başarıyla kilitlediği anlaşıldığı takdirde kullanılabileceğidir. Örnekte **ftrylockfile** işlevinin sonucu gözardı edildiğinden yapılan işlem yanlıştır. **flockfile** kullanılsaydı bu yanlış olmayacaktı. Yukarıdaki gibi bir kodun sonucunda ya **funlockfile** o anki evre tarafından kilitlenmeyen bir akımı serbest bırakmaya çalışacak ya da akımı vakitsiz olarak serbest bırakacaktır. Kod aşağıdaki gibi olmalıydı:

```
void
foo (FILE *fp)
{
  if (ftrylockfile (fp) == 0)
    {
      fputs ("in foo\n", fp);
      funlockfile (fp);
    }
}
```

Kilitlemenin niçin gerekli olduğu konusunu hallettiğimize göre artık kilitlemenin ne zaman istenmediği ve bu durumda ne yapılabileceği üzerinde duralım. Kilitleme işlemleri (doğrudan ya da dolaylı) bedavaya gelmez. Bir kilit alınmasa bile maliyeti sıfır değildir. Uygulanan işlemler çok işlemcili ortamlardaki güvenli bellek işlemlerini gerektirir. Böyle sistemlerdeki çok sayıda yerel önbellekle bu oldukça maliyetlidir. Yani, en iyisi çok gerekli değilse (bir akımın iki veya daha fazla evre tarafından kullanılamayacağı bağlamlarda) kilitlemeden tamamen kaçınmaktır. Bu çoğu zaman uygulama kodu için ugulanabilir; çok sayıda bağlam içinde kullanılabilen kütüphane kodu için bir bağlam kilitleme kullanımı ve tutuculuk anlamında öntanımlı olacaktır.

Kilitlemeden kaçınmada iki temel mekanizma vardır. İlki akım işlemlerinin **_unlocked** sonekli biçimlerini kullanmaktır. POSIX standardı bunların pek azını tanımlar ve GNU kütüphanesi birkaç tane daha ekler. İşlevlerin bu biçimleri isimlerine **_unlocked** soneki getirilen işlevlerle benzer davranışı gösterir; tek farkla, akımları kilitlemezler. Bu işlevler potansiyel olarak çok daha hızlı olduğundan daha çok tercih edilirler. Bu sadece kilitleme işlemlerinden kendilerini korumalarından dolayı değildir. Daha önemli olarak, **putc** ve **getc** işlevleri çok basittir ve geleneksel olarak (evrelere girişten önce) tampon boş değilse çok hızlı olan makrolar halinde gerçeklenmişlerdir. Kilitleme gereksinimlerinin eklenmesiyle bu işlevlerin kodu çok büyüdüğünden artık makrolar halinde gerçeklenememektedirler. Ancak bu makrolar aynı işlevsellikle yeni isimler altında (**putc_unlocked** ve **getc_unlocked**) hala kullanılabilmektedir. Hızlarındaki bu dev farktan dolayı **_unlocked** sonekli işlevlerin kullanılması kilitleme gerekli olduğu durumlarda bile tercih edilmesine sebep olmaktadır. Kilitleme ile birlikte kullanıma bir örnek:

```
void
foo (FILE *fp, char *buf)
{
  flockfile (fp);
  while (*buf != '/')
```

```
    putc_unlocked (*buf++, fp);
  funlockfile (fp);
}
```

Bu örnekte **putc** işlevi kullanılsaydı ve doğrudan kilitleme olmasaydı, **putc** işlevi döngü sonlanana kadar her çağrıda bir olmak üzere kilidi defalarca elde edecekti. Yukarıdaki örnek **putc_unlocked** makrosu akım tamponunun kilitleme olmaksızın doğrudan değiştirilmesi anlamında kullanılarak yazılmıştır.

Kilitlemeden kaçınmak için ikinci yol Solaris'de bulunan ve GNU C kütüphanesinde de kullanılabilir olan standart dışı bir işlevi kullanmaktır.

```
int __fsetlocking(FILE *akım,                                    işlev
                  int   tür)
```

__fsetlocking işlevi, akım işlemlerinin *akım* 'ın kilitleme nesnesini dolaylı elde edip etmeyeceğini seçmekte kullanılır. Öntanımlı olarak kilit dolaylı elde edilir ancak bu işlev kullanılarak kilidin alınması iptal edilebilir ya da tekrar yerleştirilebilir. *tür* parametresi olarak kullanılabilecek üç değer vardır:

FSETLOCKING_INTERNAL
: *akım* öntanımlı dahili kilitlemeyi hemen kullanmaya başlayacaktır. **_unlocked** sonekli biçim hariç her akım işlemi akımı dolaylı olarak kilitleyecektir.

FSETLOCKING_BYCALLER
: **__fsetlocking** işlevi döndükten sonra akım kilitleme için kullanıcı sorumlu olur. Durum **FSETLOCKING_INTERNAL** ile öntanımlı duruma döndürülünceye kadar bunu dolaylı olarak yapacak bir akım işlemi yoktur.

FSETLOCKING_QUERY
: **__fsetlocking** işlevi sadece akımın o anki kilitleme durumunu sorgular. Dönen değer duruma bağlı olarak ya **FSETLOCKING_INTERNAL** ya da **FSETLOCKING_BYCALLER** olacaktır.

__fsetlocking işlevinin dönüş değeri akımın çağrı öncesi durumunu belirtmek üzere ya **FSETLOCKING_INTERNAL** ya da **FSETLOCKING_BYCALLER** olacaktır.

İşlev ve *tür* parametresinin değerleri `stdio_ext.h` başlık dosyasında bildirilmiştir.

Bu işlev özellikle yazılım kodu **_unlocked** işlevleri hakkında yeterli bilgiye sahip olunmadan yazılmışsa (ya da yazılımcı onları çok delice kullanmışsa) yararlıdır.

6. Akımlar ve Uluslararasılaştırma

ISO C90 geniş karakter kümeleriyle çalışmak için **wchar_t** isimli yeni bir veri türü tanımladı. **wchar_t** dizgelerini doğrudan doğruya çıktılayacak bir olasılıktı eksik olan. Biri onları çok baytlı dizgelere **mbstowcs** kullanarak çevirdi (hala bir **mbsrtowcs** yok) ve sonra normal akım işlevlerini kullandı. Bu yapılabilir olurken anlamsız olmayan dönüşümler uyguladığından ve yazılımın karmaşıklığını ve boyutlarını fazlaca arttırdığından çok hantal oldu.

Unix standardı daha erken olarak (XPG4.2'de sanırım) **printf** ve **scanf** işlevleri için iki ek biçim belirteci tanımladı. Tek geniş karakterin okunması ve basılması **%C** belirteci ile geniş karakterli dizgelerinki ise **%S** belirteci kullanılarak mümkün hale getirildi. Bu belirteçler geniş karakter türü kullanmak dışında tıpkı **%c** ve **%s** belirteçleri gibi çalışıyordu. Ancak geniş karakterler ve dizgeler kullanılmadan önce çok baytlı dizgelere ya da tersine çevriliyordu.

Bu bir başlangıçtı ama hala yeterince iyi değildi. **printf** ve **scanf** kullanımı hep tercih edilen birşey değildi. Daha küçük ve daha hızlı diğer işlevler geniş karakterlerle çalışamıyordu. İkincisi, **printf** ve **scanf** işlevleri

için geniş karakterleri oluşturulmasını sağlayacak bir biçim dizgesine sahip değildi. Biçim dizgesi temel olmayan karakterlere sahipse sonuçta biçim dizgesi üretilmiş gibi olurdu.

ISO C 90'ın 1. düzeltmesinde sorunu çözümlemek için yeni bir işlevler kümesi eklendi. Akım işlevlerinin çoğu bir karakter veya dizge yerine bir geniş karakter veya bir geniş karakterli dizge alır hale getirildi. Yeni işlevler aynı akımlar (**stdout** gibi) üzerinde işlem yapmaktadır. Bu normal ve geniş karakterler G/Ç işlemleri için ayrı akım işlevleri kullanılan C++ çalışma anı kütüphanesindeki modelden farklıdır.

Aynı akımın hem normal hem de geniş karakterlerle kullanılabilmesi bir sınırlama ile mümkün olur: Bir akım ya normal karakterleri kullanır ya da geniş karakterleri, ikisi birden olmaz. Bir kere karar verildikten sonra dönüşü yoktur. Sadece **freopen** ya da **freopen64** işlevleri yönlenimi sıfırlayabilir. Yönlenime üç yolla karar verilebilir:

- Normak karakter işlevlerinden biri kullanılmışsa (**fread** ve **fwrite** işlevleri dahil) akım "geniş yönlenimli değildir" olarak imlenir.
- Geniş karakter işlevlerinden biri kullanılmışsa akım "geniş yönlenimli" olarak imlenir.
- Yönlenimi belirlemek için **fwide** işlevi kullanılır.

Geniş ve geniş olmayan işlemlerin aynı akım üzerinde asla karıştırılmaması gerekliliği önemlidir. Bunun için tasarlanmış bir tanılama yolu yoktur. Uygulama basitçe tuhaflaşır hatta daha basitçe çökebilir. **fwide** işlevi bundan kaçınmanıza için yardımcı olabilir.

```
int fwide(FILE *akım,                                                    işlev
          int   kip)
```

fwide işlevi *akım* yönleniminin belirlenmesinde ve sorgulanmasında kullanılabilir. *kip* parametresi bir pozitif değer belirtiyorsa akım geniş yönlenimli, negatif bir değer belirtiyorsa dar yönlenimlidir. **fwide** ile önceki yönlenimi değiştirmek mümkün değildir. *akım* zaten yönlenimli ise çağrı hiçbir şey yapmaz.

kip sıfırsa, o anki yönlenim durumu sorgulanır ve hiçbir değişiklik yapılmaz.

fwide işlevi yönlenimin durumuna bağlı olarak, dar, hiçbiri veya geniş yönlenimli olmasına göre sırayla bir negatif değer, sıfır veya bir pozitif değerle döner.

Bu işlev ISO C09'ın 1. düzeltmesinde tanımlanmış ve `wchar.h` başlık dosyasında bildirilmiştir.

Genel olarak bir akımın yönlenimini mümkün olduğu kadar erken belirlemek daha iyidir. Bu özellikle **stdin**, **stdout** ve **stderr** standart akımlarında ortaya çıkacak bazı sürprizlerden sizi koruyabilir. Bu akımlardan birini kullanan bazı kütüphane işlevleri bazı durumlarda akımın yönlenimini uygulamanın kalanında, sonlanma ve hata üretilmesi dışında farklı bir yolla kullanır. Akımların yönleniminin yanlış kullanımında hata üretilmediğini unutmayın. Bir akımın oluşturulduktan sonra yönlenimsiz bırakılması normalde, sadece akımları farklı bağlamlarda kullanmak üzere oluşturan kütüphane işlevleri için gereklidir.

Akımların kullanıldığı ve bu akımarın farklı bağlamlarda kullanılabildiği bir kodu yazarken bir akımı kullanmadan önce akımın yönlenimini (kütüphane arayüzünün kuralları belli bir yönlenimi talep etmedikçe) sorgulamak önemlidir. Aşağıdaki küçük işlev bunu örneklemektedir:

```
void
print_f (FILE *fp)
{
  if (fwide (fp, 0) > 0)
    /* Dönen pozitif değer geniş yönlenimi gösterir.  */
    fputwc (L'f', fp);
  else
    fputc ('f', fp);
}
```

Burada **print_f** işlevi, akım önceden yönlenimsiz olsa da akımın yönlenimine karar vermektedir (yukarıdaki bilgileri iyi özümlediyseniz olumsuz birşey olmayacağını görürsünüz).

wchar_t değerleri için kullanılan karakter kodlaması belirtilmemiştir ve yazılımcı onun hakkında herhangi bir kabulde bulunmamalıdır. **wchar_t** değerlerinin G/Ç işlemleri için bunun anlamı, bu değerlerin akıma doğrudan doğruya yazılmasının imkansızlığıdır. Bu ISO C yerel modelinde izlenen yol da değildir. Baytların alttaki ortamdan okunması ve oraya yazılması yerine yapılacak olan önce **wchar_t** için gerçekleme tarafından seçilen dahili yerele dönüşümdür. Harici karakter kodlaması o anki yerelin `LC_CTYPE` kategorisi tarafından ya da **fopen**, **fopen64**, **freopen** veya **freopen64** işlevlerine verilen kip belirtiminin parçası olan **ccs** değeri tarafından Dönüşümün ne zaman ve nasıl olacağı belirsizdir ve kullanıcıya görünür değildir.

Bir akım yönlenimsiz durumda oluşturulduğunda bu noktada onunla ilişkili bir dönüşüm yapılmaz. Kullanılacak dönüşüm akım yönlenimlendiği sırada **LC_CTYPE** kategorisi tarafından saptanmış olacaktır. Dikkatli olunmazsa, yerel çalışma anında değiştirilirse sürprizli durumlarla karşılaşabilirsiniz. Bu da, akımın yönleniminin mümkün olduğunca erken belirlenmesinin önemini gösteren iyi bir sebeptir. Şüphesiz bu işlemi bir **fwide** çağrısı ile yapmalısınız.

7. Karakterlerin ve Satırların Basit Çıktılanması

Bu kısımda karakter ve satır yönlenimli çıktılama işlemleri için kullanılan işlevler açıklanmıştır.

Bu kısımdaki dar yönlenimli akım işlevleri `stdio.h` başlık dosyasında, geniş yönlenimli akım işlevleri ise `wchar.h` başlık dosyasında bildirilmiştir.

```
int fputc(int    c,                                                    işlev
          FILE *akım)
```

fputc işlevi *c* karakterini **unsigned char** türüne çevirir ve onu *akım* 'a yazar. Bir hata oluşursa `EOF` döner, aksi takdirde *c* karakteri döner.

```
wint_t fputwc(wchar_t wc,                                              işlev
              FILE    *akım)
```

fputwc işlevi *wc* geniş karakterini *akım* 'a yazar. Bir yazma hatası oluşursa `WEOF` döner, aksi takdirde *wc* karakteri döner.

```
int fputc_unlocked(int    c,                                           işlev
                   FILE *akım)
```

fputc_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **fputc** işlevine eşdeğerdir.

```
wint_t fputwc_unlocked(wint_t wc,                                      işlev
                       FILE   *akım)
```

fputwc_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **fputwc** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

```
int putc(int    c,                                                     işlev
         FILE *akım)
```

Birçok sistemin daha hızlı olması için onu bir makro olarak gerçekleştirmesi dışında tamamen **fputc** gibidir. Makrolar için genel kurala bir istisna olarak *akım* argümanını bir kereden fazla değerlendirebilmesi ona önem kazandırır. Genellikle **putc** işlevi tek bir karakterin yazılması için kullanılacak en iyi işlevdir.

```
wint_t putwc(wchar_t wc,                                                    işlev
             FILE    *akım)
```

Birçok sistemin daha hızlı olması için onu bir makro olarak gerçekleştirmesi dışında tamamen **fputwc** gibidir. Makrolar için genel kurala bir istisna olarak *akım* argümanını bir kereden fazla değerlendirebilmesi ona önem kazandırır. Genellikle **putwc** işlevi tek bir geniş karakterin yazılması için kullanılacak en iyi işlevdir.

```
int putc_unlocked(int    c,                                                 işlev
                  FILE *akım)
```

putc_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **putc** işlevine eşdeğerdir.

```
wint_t putwc_unlocked(wchar_t wc,                                           işlev
                      FILE    *akım)
```

putwc_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **putwc** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

```
int putchar(int c)                                                          işlev
```

putchar işlevi **putc** işlevinin *akım* argümanına değer olarak **stdout** belirtilerek çağrılması ile eşdeğerdir. Yani bu işlev bir karakteri doğrudan standart çıktıya basar.

```
wint_t putwchar(wchar_t wc)                                                 işlev
```

putwchar işlevi **putwc** işlevinin *akım* argümanına değer olarak **stdout** belirtilerek çağrılması ile eşdeğerdir. Yani bu işlev bir geniş karakteri doğrudan standart çıktıya basar.

```
int putchar_unlocked(int c)                                                 işlev
```

putchar_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **putchar** işlevine eşdeğerdir.

```
wint_t putwchar_unlocked(wchar_t wc)                                        işlev
```

putwchar_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **putwchar** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

```
int fputs(const char *s,                                                    işlev
          FILE       *akım)
```

fputs işlevi *s* dizgesini *akım* 'a yazar. Dizgeyi sonlandıran boş karakter yazılmadığı gibi sonuna bir satırsonu karakteri de eklemez. Sadece dizge içindeki karakterleri çıktılar.

Bir yazma hatası oluşursa işlev `EOF` ile döner, aksi takdirde negatif olmayan br değerle döner. Örneğin:

```
fputs ("Burada ", stdout);
fputs ("hava ", stdout);
fputs ("serin\n", stdout);
```

kodu `Burada hava serin` dizgesini sonuna bir satırsonu karakteri ekleyerek çıktılar.

```
int fputws(const wchar_t *ws,                                               işlev
           FILE          *akım)
```

fputws işlevi *s* geniş karakterli dizgesini *akım* 'a yazar. Dizgeyi sonlandıran boş karakter yazılmadığı gibi sonuna bir satırsonu karakteri de eklemez. Sadece dizge içindeki karakterleri çıktılar.

Bir yazma hatası oluşursa işlev `WEOF` ile döner, aksi takdirde negatif olmayan br değerle döner.

```
int fputs_unlocked(const char *s,                                    işlev
                   FILE       *akım)
```

fputs_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **fputs** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

```
int fputws_unlocked(const wchar_t *ws,                               işlev
                    FILE          *akım)
```

fputws_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **fputws** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

```
int puts(const char *s)                                              işlev
```

puts işlevi *s* dizgesini sonuna bir satırsonu karakteri ekleyerek standart çıktıya basar. Dizgeyi sonlandıran boş karakter çıktılanmaz. (**fputs** işlevi bu işlevden farklı olarak dizgenin sonuna satırsonu karakteri eklemez.)

puts nasit iletileri basmak için oldukça kullanışlı bir işlevdir. Örneğin:

```
puts ("Bu bir ileti.");
```

deyimi sonuna bir satırsonu karakteri ekleyerek `Bu bir ileti.` metnini çıktılar.

```
int putw(int    sözcük,                                              işlev
         FILE *akım)
```

Bu işlev *sözcük* sözcüğünü (**int** türündedir) *akım* 'a yazar. SVID ile uyumluluk için vardır, ancak bunun yerine **fwrite** kullanmanız önerilir. Bkz. *Blok Girişi ve Çıkışı* (sayfa: 254).

8. Karakter Girdilerinin Alınması

Bu kısımda girdi olarak karakter alan işlevler açıklanmıştır. Bu kısımdaki dar yönlenimli akım işlevleri `stdio.h` başlık dosyasında, geniş yönlenimli akım işlevleri ise `wchar.h` başlık dosyasında bildirilmiştir.

Bu işlevler bir karakter girdisinde, dar yönlenimli ise bir **int** türünden değerle, geniş yönlenimli ise bir **wint_t** türünden bir değerle ya da **EOF**/**WEOF** özel değeri (genellikle –1) ile döner.

Dar yönlenimli akım işlevleri için, işlev sonucunu bir karakter olarak kullanmayı tasarlıyorsanız bile, bu işlevlerin sonucunun **char** türü yerine **int** türünden bir değişkene atanması önemlidir. **EOF** değerinin **char** türünden bir değişkene atanması onun bir karakterlik boyuta indirilmesine sebep olur ki bu durumda artık geçerli bir karakterden (`(char) -1`) farkı kalmaz. Öyleyse, **getc** ve arkadaşlarını daima **int** türünden bir değişkene atayarak çağırmalı ve dönen değerin **EOF** olup olmadığına bakmalısınız. Dönen değer **EOF** değilse artık onu bir **char** türünden değişkene bilgi kaybı olmadan atayabilirsiniz.

```
int fgetc(FILE *akım)                                                işlev
```

Bu işlev *akım* 'daki sonraki karakteri bir **unsigned char** olarak okur ve değerini **int** türüne dönüştürerek döndürür. Bir dosya sonu durumunda ya da bir hata oluştuğunda hata yerine **EOF** ile döner.

```
wint_t fgetwc(FILE *akım)                                            işlev
```

Bu işlev *akım* 'daki sonraki geniş karakteri okur ve değerini döndürür. Bir dosya sonu durumunda ya da bir hata oluştuğunda hata yerine **WEOF** ile döner.

`int` **`fgetc_unlocked`**`(FILE *`*akım*`)` işlev

`fgetc_unlocked` işlevi akımı dolaylı olarak kilitlememesi dışında **`fgetc`** işlevine eşdeğerdir.

`wint_t` **`fgetwc_unlocked`**`(FILE *`*akım*`)` işlev

`fgetwc_unlocked` işlevi akımı dolaylı olarak kilitlememesi dışında **`fgetwc`** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

`int` **`getc`**`(FILE *`*akım*`)` işlev

Birçok sistemin daha hızlı olması için onu bir makro olarak gerçekleştirmesi dışında tamamen **`fgetc`** gibidir. Makrolar için genel kurala bir istisna olarak *akım* argümanını bir kereden fazla değerlendirebilmesi ona önem kazandırır. Genellikle **`getc`** işlevi tek bir karakterin okunması için kullanılacak en iyi işlevdir.

`wint_t` **`getwc`**`(FILE *`*akım*`)` işlev

Birçok sistemin daha hızlı olması için onu bir makro olarak gerçekleştirmesi dışında tamamen **`fgetwc`** gibidir. Makrolar için genel kurala bir istisna olarak *akım* argümanını bir kereden fazla değerlendirebilmesi ona önem kazandırır. Genellikle **`getwc`** işlevi tek bir geniş karakterin okunması için kullanılacak en iyi işlevdir.

`int` **`getc_unlocked`**`(FILE *`*akım*`)` işlev

`getc_unlocked` işlevi akımı dolaylı olarak kilitlememesi dışında **`getc`** işlevine eşdeğerdir.

`wint_t` **`getwc_unlocked`**`(FILE *`*akım*`)` işlev

`gewtc_unlocked` işlevi akımı dolaylı olarak kilitlememesi dışında **`getwc`** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

`int` **`getchar`**`(void)` işlev

`getchar` işlevi **`getc`** işlevinin *akım* argümanına değer olarak **`stdin`** belirtilerek çağrılması ile eşdeğerdir. Yani bu işlev bir karakteri doğrudan standart girdiden okur.

`wint_t` **`getwchar`**`(void)` işlev

`getwchar` işlevi **`getwc`** işlevinin *akım* argümanına değer olarak **`stdin`** belirtilerek çağrılması ile eşdeğerdir. Yani bu işlev bir geniş karakteri doğrudan standart girdiden okur.

`int` **`getchar_unlocked`**`(void)` işlev

`getchar_unlocked` işlevi akımı dolaylı olarak kilitlememesi dışında **`getchar`** işlevine eşdeğerdir.

`wint_t` **`getwchar_unlocked`**`(void)` işlev

`getwchar_unlocked` işlevi akımı dolaylı olarak kilitlememesi dışında **`getwchar`** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

Aşağıda **`fgetc`** kullanarak girdi alan bir işlev örneği vardır. İşlev **`fgetc (stdin)`** yerine **`getc`** ya da **`getchar`** kullanarak da çalışırdı. Kod ayrıca geniş yönlenimli akım işlevleri için de aynı şekilde çalışırdı.

```
int
evet_mi_hayir_mi (const char *soru)
{
  fputs (soru, stdout);
```

```
  while (1)
    {
      int c, cevap;
      /* Bir boşluk yazarak cevabı sorudan ayıralım. */
      fputc (' ', stdout);
      /* Satırdaki ilk karakteri okuyalım.
         Bu cevap karakteri olmalı ama olmayabilir. */
      c = tolower (fgetc (stdin));
      answer = c;
      /* Satırın kalanını iptal edelim. */
      while (c != '\n' && c != EOF)
        c = fgetc (stdin);
      /* Yanıt geçerliyse uygun dönüşü yapalım. */
      if (cevap == 'y')
        return 1;
      if (cevap == 'n')
        return 0;
      /* Cevap geçersiz: geçerli cevabı almaya çalışalım. */
      fputs ("Cevabınız e ya da h olmalı: ", stdout);
    }
}
```

`int` **getw**`(FILE *`*akım*`)` işlev

Bu işlev bir sözcüğü (**int** türünde) *akım* 'dan okur. SVID ile uyumluluk için vardır, ancak bunun yerine **fread** kullanmanız önerilir. Bkz. *Blok Girişi ve Çıkışı* (sayfa: 254). **getc** işlevinin tersine herhangi bir **int** türünden değer geçerli bir sonuç olmalıydı. İşlev bir hata oluştuğunda ya da dosya sonu saptandığında **EOF** ile döner, ancak –1 değerini bir geçerli sözcükten ayırmak için bir yol yoktur.

9. Satır Yönlenimli Girdi

Birçok yazılım girdiyi satır temelinde yorumladığından bir akımdan bir metin satırını okuyacak işlevlerin de bulunması yararlıdır.

Standart C bunu yapacak işlevlere sahiptir ama onlar çok güvenilir değildir: boş karakterler ve hatta uzun satırlar (**gets** için) onları şaşırtabilir. Bu nedenle GNU kütüphanesi satırları güvenilir bir şekilde kolayca okumak için standart olmayan **getline** işlevini içerir.

Diğer bir GNU oluşumu olan **getdelim** işlevi, **getline** işlevini genelleştirir. İşlev belirtilmiş bir ayraç karakteriyle ayrılmış kayıtlardan bir kaydı okur.

Bu işlevlerin tümü `stdio.h` başlık dosyasında bildirilmiştir.

`ssize_t` **getline**`(char **`*satır–gstr*`,`
`size_t *`*n*`,`
`FILE *`*akım*`)` işlev

Bu işlev *akım* 'daki bir tam satırı okur, metni satırsonu karakteri ve bir sonlandırıcı boş karakterle bir tampona yazar ve bu tamponun adresini ***satır–gstr** içinde saklar.

getline çağrısından önce, ***n** bayt uzunluğundaki **malloc** ile ayrılmış bir tamponun adresi ***satır–gstr** içine yerleştirilmelidir. Bu tampon yeterli uzunluktaysa, **getline** okuduğu satırı bu tampona yazar. Aksi takdirde **getline** işlevi **realloc** kullanarak tamponu büyütür ve yeni tampon adresini ***satır–gstr** argümanı ile ve arttırdığı uzunluğu ***n** argümanı ile geri döndürür. Bkz. *Özgür Bellek Ayırma* (sayfa: 50).

***satır–gstr** olarak bir boş gösterici ve ***n** için 0 değerini vererek çağrı yaparsanız, **getline** işlevi **malloc** işlevini kullanarak tamponu sizin için ayırır.

Her iki durumda, **getline** döndüğünde, ***satır–gstr** satırın metnini gösteren bir **char *** dır.

getline başarılı olduğunda okunan karakterlerin sayısı (satırsonu karakteri ve bir boş karakter dahil) ile döner. Bu değer satırın parçası olan boş karakterleri sonlandırıcı olarak eklenen boş karakterden ayırabilmenizi sağlar.

Bu işlev bir GNU oluşumudur ve bir akımdan satırları okumak için önerilen bir yoldur. Aynı amaçlı diğer standart işlevler güvenilir değildir.

Bir hata oluşursa ya da herhangi bir bayt okunmadan dosya sonuna gelinirse **getline** işlevi **-1** ile döner.

```
ssize_t getdelim(char  **satır–gstr,                                    işlev
                 size_t *n,
                 int     ayraç,
                 FILE   *akım)
```

Bu işlev okumayı sonlandıran karakterin belirtilebilmesi dışında **getline** gibidir. *ayraç* argümanı ile ayraç karakteri belirtilir. İşlev bu karakteri görene kadar ya da bu karakter yoksa dosya sonunu görene kadar metni okur.

satır–gstr içinde saklanan metin ayraç karakterini ve sonlandırıcı olarak bir boş karakter içerir. **getline** gibi **getdelim** işlevi de gerekirse *satır–gstr* tamponunu büyütür.

getline bir **getdelim** işlevi ile aşağıdaki gibi gerçeklenebilir:

```
ssize_t
getline (char **lineptr, size_t *n, FILE *stream)
{
  return getdelim (lineptr, n, '\n', stream);
}
```

```
char *fgets(char *s,                                                    işlev
            int   kar–sayısı,
            FILE *akım)
```

fgets işlevi *akım* 'dan bir satırı satırsonu karakterine kadar bu karakterde dahil olmak üzere okur ve *s* dizgesine sonuna bir boş karakter ekleyerek yerleştirir. *s* dizgesindeki alanı *kar–sayısı* karakterlik olarak belirtebilirsiniz, ancak okunan karakterlerin sayısı *kar–sayısı* – 1 olacaktır. Kalan son yer, dizgenin sonunu gösteren boş karakter için kullanılacaktır.

fgets çağrısı yaptığınızda sistem zaten dosyanın sonundaysa *s* dizisinin içeriği değiştirilmez ve bir boş gösterici döner. Bir okuma hatası olduğunda da boş gösterici döner. Aksi takdirde dönen değer *s* göstericisidir.

Uyarı

Girdi verisi bir boş karakter içeriyorsa bunu belirtemezsiniz. Bu nedenle, sadece verinin bir boş karakter içermediğini biliyorsanız **fgets** işlevini kullanın, yoksa kullanmayın. Kullanıcı tarafından düzenlenmiş dosyaları okumak için **fgets** kullanmayın, çünkü kullanıcı bir boş karakter girmişse ya onu olması gerektiği gibi okumanız ya da bir hata iletisi basmanız gerekir. Daha iyisi, **fgets** yerine **getline** kullanmanızı öneririz.

```
wchar_t *fgetws(wchar_t *ws,                                    işlev
                int      kar–sayısı,
                FILE     *akım)
```

fgetws işlevi *akım* 'dan geniş karakterleri satırsonu karakterine kadar bu karakterde dahil olmak üzere okur ve *ws* dizgesine sonuna bir boş karakter ekleyerek yerleştirir. *ws* dizgesindeki alanı *kar–sayısı* karakterlik olarak belirtebilirsiniz, ancak okunan geniş karakterlerin sayısı *kar–sayısı* – 1 olacaktır. Kalan son yer, dizgenin sonunu gösteren boş karakter için kullanılacaktır.

fgetws çağrısı yaptığınızda sistem zaten dosyanın sonundaysa *ws* dizisinin içeriği değiştirilmez ve bir boş gösterici döner. Bir okuma hatası olduğunda da boş gösterici döner. Aksi takdirde dönen değer *ws* göstericisidir.

Uyarı

Girdi verisi bir boş karakter içeriyorsa bunu belirtemezsiniz. Bu nedenle, sadece verinin bir boş karakter içermediğini biliyorsanız **fgetws** işlevini kullanın, yoksa kullanmayın. Kullanıcı tarafından düzenlenmiş dosyaları okumak için **fgetws** kullanmayın, çünkü kullanıcı bir boş karakter girmişse ya onu olması gerektiği gibi okumanız ya da bir hata iletisi basmanız gerekir.

```
char *fgets_unlocked(char *s,                                   işlev
                     int    kar–sayısı,
                     FILE *akım)
```

fgets_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **fgets** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

```
wchar_t *fgetws_unlocked(wchar_t *ws,                           işlev
                         int      kar–sayısı,
                         FILE     *akım)
```

fgetws_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **fgetws** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

```
char *gets(char *s)                                             önerilmeyen işlev
```

gets işlevi standart girdiden bir satırı satırsonu karakterine kadar okur ve *s* dizgesine yerleştirir. Satırsonu karakteri iptal edilir. (İşlev bu davranışı ile satırsonu karakterini de dizgeye yerleştiren **fgets** işlevinden faklıdır.) Bir okuma hatası olduğunda ya da dosya sonu saptandığında boş gösterici döner. Aksi takdirde dönen değer *s* göstericisidir.

Uyarı

gets işlevi **çok tehlikelidir**. Çünkü *s* dizgesinin taşmasına karşı bir korumaya sahip değildir. GNU kütüphanesi işlevi sadece uyumluluk adına içerir. Bunun yerine daima ya **fgets** ya da **getline** işlevini kullanmalısınız. Bunu hatırlatmak için ilintileyici (GNU **ld** kullanıyorsanız) **gets** kullanırsanız sizi uyaracaktır.

10. Okunmamış Yapmak

Çözümleyici yazılımlarda akımdaki sonraki karakterin ne olduğuna bakmak ama onu akımdan silmeksizin okumak çoğunlukla kullanışlıdır. Buna girdiye "önceden bakış" denir, çünkü yazılımınız daha sonra okuyacağı girdinin anlık bir görüntüsünü alır.

Akım G/Ç işlemlerini kullanarak, girdiyi önce akımdan okuyup sonra da okunmamış yaparak (buna girdiyi "geri itme" de denir) girdiye öncesinden bakabilirsiniz. Bir karakterin okunmamış yapılması onu akımdan tekrar okunabilmesi için kullanılabilir yapar. Sonraki bir **fgetc** çağrısı ya da diğer bir girdi işlevi ile onu akımdan okuyabilirsiniz.

10.1. Okunmamış Yapmak Ne Demek

Okunmamış yapma işlemi burada resimsel örneklerle açıklanmaya çalışılmıştır. Bir dosyayı okuyan bir akımınız olduğunu ve dosyanın altı karakter içerdiğini varsayalım. Bu harfler `foobar` olsun. İlk üç karakteri okumuş olalım. Durum bunun gibi olacaktır:

```
f  o  o  b  a  r
         ^
```

Yani sonraki girdi karakteri **b** olacaktır.

byi okumak yerine **o** harfini okunmamış yapalım. Durum şöyle olur:

```
f  o  o  b  a  r
      |
      o--
      ^
```

Böylece sonraki girdi karakterleri **o** ve **b** olacaktır.

o harfi yerine **9** karakterini de okunmamış yapabiliriz. Bu durumu şöyle gösterebiliriz:

```
f  o  o  b  a  r
      |
      9--
      ^
```

Burada sonraki girdi karakterleri **9** ve **b** olacaktır.

10.2. Okunmamış Nasıl Yapılır

Bir karakteri okumamış yapan işlev **ungetc**'dir, çünkü **getc** işlevinin tersini yapar.

```
int ungetc(int    c,                                                    işlev
           FILE *akım)
```

> **ungetc** işlevi, *akım* akımına sıradaki karakterden önce okunacak ilk girdi olarak *c* karakterini yerleştirir.
>
> *c* karakteri olarak **EOF** verilirse işlev hiçbir işlem yapmaz ve **EOF** ile döner. Bu özelliğini kullanrak, *c* karakteri olarak **getc** işlevinin dönüş değerini vererek **getc** den dönen değer üzerinde hata denetimi yapmanız gerekmez.
>
> Akıma gönderdiğiniz karakterin son okuduğunuz karakter olması gerekmez. Hatta **ungetc** kullanmak için akımdan son karakteri okumanız da gerekmez. Ancak bir akımdan hiçbir okuma yapmadan bir karakteri akıma geri itmenin de bir anlamı yoktur. GNU C kütüphanesi ikilik kipte açılmış dosyalar için de bu oluşum için destek verir, diğer sistemlerde bu yoktur.

GNU C kütüphanesi akıma sadece tek karakterin geri itilmesini destekler. Hiçbir okuma yapmadan işlevi peşpeşe iki defa kullanamazsınız. Diğer sistemlerde çok sayıda karakteri akıma geri itebilir ve akıma ittiğiniz karakterleri ters sırada okuyabilirsiniz (Son ittiğiniz karakteri ilk olarak okursunuz).

Karakterlerin akıma geri itilmesi dosyada bir değişiklik yapmaz, sadece dahili tampon etkilenir. Bir dosya konumlama işlevi (**fseek**, **fseeko** ve **rewind** işlevlerinden biri; bkz. *Dosyalarda Konumlama* (sayfa: 288)) çağrılırsa geri itilmiş olarak bekleyen karakterler iptal edilir.

Okunmamış karakter olarak bir akıma itilen karakter dosya sonuna denk gelirse, akımın dosyasonu belirteci temizlenir, çünkü artık akımda okunacak bir karakter vardır. Karakteri okuduktan sonra tekrar dosyasonu saptanacaktır.

```
wint_t ungetwc(wint_t wc,                                                    işlev
               FILE  *akım)
```

ungetwc işlevi **ungetc** işlevi ile bir geniş karakterle çalışması dışında tamamen aynıdır.

Aşağıdaki örnekte **getc** ve **ungetc** işlevleri akımdaki boşluk karakterlerini ayıklamakta kullanılmıştır. İşlev rastladığı boşluk olmayan her karakteri akıma geri iter. Böylece akım daha sonra okunduğunda bu karakterler okunmamış olarak yeniden okunabilir.

```
#include <stdio.h>
#include <ctype.h>

void
skip_whitespace (FILE *stream)
{
  int c;
  do
    /* EOF için bir sınama yapmaya gerek yoktur,
       çünkü ungetc EOF'u yoksayar.  */
    c = getc (stream);
  while (isspace (c));
  ungetc (c, stream);
}
```

11. Blok Girişi ve Çıkışı

Bu kısımda veri blokları üzerinde giriş ve çıkış işlemlerinin nasıl yapıldığı anlatılmıştır. Bu işlevleri kullarak ikilik verilerin giriş ve çıkış işlemlerini yapabileceğiniz gibi satırlar ve karakterler yerine sabit uzunluktaki metinler üzerinde de giriş ve çıkış işlemleri yapabilirsiniz.

İkilik dosyalar, genellikle, bir çalışan yazılımın içeriği ile aynı biçimdeki veri bloklarının okunması ve yazılmasında kullanılır. Başka bir deyişle, istenen bellek blokları (sadece karakter ya da dizge nesneleri değil) bir ikilik dosyaya kaydedilir ve aynı yazılım tarafından anlamlı olarak tekrar okunur.

Verinin ikilik biçimde saklanması biçimli G/Ç işlemlerini kullanmaktan çok daha verimlidir. Ayrıca gerçek sayıların iklik biçimde okunması dönüşüm sırasında olası hassasiyet kayıplarının önüne geçer. Diğer taraftan, ikilik dosyalar birçok standart dosya araçları (örn., metin düzenleyiciler) ile düzenlenemez ve dilin farklı gerçeklemeleri ya da farklı bilgisayarlar arasında taşınabilir değildir.

Bu işlevler `stdio.h` başlık dosyasında bildirilmiştir.

```
size_t fread(void  *veri,                                                    işlev
             size_t boyut,
             size_t miktar,
             FILE  *akım)
```

Bu işlev, *boyut* uzunluktaki *miktar* sayıda nesneyi *akım* akımından okur ve *veri* dizisine yazar. Bir hata oluşur veya dosya sonuna ulaşılırsa, okunan nesnelerin sayısı olan ve belirtilen *miktar* dan daha küçük bir sayı ile döner. *boyut* ya da *miktar* sıfır değeriyle verilmişse işlev hiçbir işlem yapmaz ve sıfır ile döner.

İşlev bir nesnenin ortasında dosya sonu saptarsa tamamı okunabilen nesnelerin sayısı ile döner ve kısmen okunan nesneyi iptal eder. Bu nedenle gerçek dosya sonu akımda kalır.

```
size_t fread_unlocked(void  *veri,                                    işlev
                      size_t boyut,
                      size_t miktar,
                      FILE  *akım)
```

fread_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **fread** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

```
size_t fwrite(const void *veri,                                       işlev
              size_t      boyut,
              size_t      miktar,
              FILE       *akım)
```

Bu işlev, *boyut* uzunluktaki *miktar* sayıda nesneyi *veri* dizisinden okur ve *akım* akımına yazar. İşlev başarılı olursa *miktar* ile döner. Farklı bir değer dönmüşse bu, alan yetersiz gibi bazı hataları belirtir.

```
size_t fwrite_unlocked(const void *veri,                              işlev
                       size_t      boyut,
                       size_t      miktar,
                       FILE       *akım)
```

fwrite_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **fwrite** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

12. Biçimli Çıktı

Bu kısımda açıklanan işlevler (**printf** ve ilgili işlevler) biçimli çıktılama için kullanışlı bir yol sağlar. **printf** işlevi kalan argümanların değerlerinin nasıl biçimlendirileceğini belirten bir ***biçim dizgesi*** veya bir ***şablon dizgesi*** ile çağrılır.

Yazılımınız satır ya da karakter yönlenimli işlemler uygulayan bir süzme aracı değilse, bu kısımda açıklanan **printf** veya diğer bir benzer işlevle veriyi biçimli olarak kolayca çıktılayabilirsiniz. Bu işlevler özellikle hata iletilerini, tabloları ve benzerlerini basmak için kullanışlıdır.

12.1. Biçimli Çıktılamanın Temelleri

printf işlevi istenen sayıda argümanı basmak için kullanılabilir. İşlevi çağırırken belirteceğiniz şablon dizge içinde sadece ek argümanların sayısına ilişkin bilgi değil onların türleri ve nasıl biçimlenerek çıktılanacağına ilişkin bilgileri de verebilirsiniz.

Şablon dizge içindeki bir özellik içermeyen karakterler oldukları gibi akıma çıktılanırken şablon dizge içindeki bir **%** karakteri ile vurgulanan ***dönüşüm belirtimleri*** altbileşen argümanların biçimlenerek akıma çıktılanmasını sağlar. Örneğin,

```
int pct = 37;
char filename[] = "foo.txt";
printf ("`%s' dosyasının %%%d'lik kısmı işlendi.\nLütfen sabırlı olun.\n",
        filename, pct);
```

kodu şöyle bir çıktı üretir:

```
'foo.txt' dosyasının %37'lik kısmı işlendi.
Lütfen sabırlı olun.
```

Bu örnekteki **%d** biçimleyeceği ikinci artbileşen argümanın **int** türünden olduğunu ve bir onluk tamsayı olarak basılacağını belirtir. **%s** ise ilk artbileşen argümanın bir dizge olduğunu belirtir. **%d** belirtimini önceleyen **%%** karakterleri **%** işareti olarak yorumlanır.

Bir tamsayı argümanın bir işaretsiz sekizlik, onluk ya da onaltılık tabanda bir sayı olarak (sırasıyla **%o**, **%u**, **%x**) ya da bir karakter **%c**) olarak yorumlanmasını sağlanabilir.

Bir gerçek sayı **%f** ile ondalık gösterimde **%e** ile üstel gösterimde basılabilir. **%g** dönüşümü ile gerçek sayıya, sayının büyüklüğüne bağlı olarak **%f** veya **%g** dönüşümü uygulanır.

Biçimlemeyi **%** ile dönüşüm karakteri arasına ***değiştiriciler*** yerleştirerek daha hassas olarak ayarlayabilirsiniz. Bunlar bazan normal dönüşüm davranışını oldukça değiştirebilir. Örneğin, çoğu dönüşüm belirtimleri bir en küçük alan genişliği ile alanın sağa mı yoksa sola mı yanaştırılarak yazılacağını belirleyen imlerin kulanılmasına izin verir.

Bu tür değiştirici ve imler dönüşüm türüne bağlı olarak farklı yorumlanabilir. Bunlar size başta oldukça karmaşık görünebilir, endişelenmeyin; bir değiştirici ya da im kullanmadan da serbest biçimde gayet anlamlı çıktılar üretebilirsiniz. Bu değiştirici ve imler özellikle ve çoğunlukla tablo benzeri çıktılarda daha hoş görünümler elde etmek amacıyla kullanılır.

12.2. Çıktı Dönüşüm Sözdizimi

Bu bölümde bir **printf** şablon dizgesindeki dönüşüm belirtimlerinin tam sözdizimi ayrıntılarıyla açıklanmıştır.

Şablon dizgesindeki bir dönüşüm belirtiminin parçası olmayan karakterler oldukları gibi akıma çıktılanır. Şablon dizgesinde çok baytlı karakter dizileri de kullanılabiir (Bkz. *Karakter Kümeleriyle Çalışma* (sayfa: 126)).

Bir **printf** şablon dizgesindeki dönüşüm belirtimlerinin genel sözdizimi:

% [*konum–num* $] *imler genişlik* [. *hassasiyet*] *tür dönüşüm*

veya

% [*konum–num* $] *imler genişlik* . * [*param–no* $] *türdönüşüm*

Örneğin, **%-10.8ld** dönüşüm belirtecinde, **-** imi, **10** alan genişliğini, **8** hassasiyeti, **l** veri türünü, **d** ise dönüşüm tarzını belirtir. (Bu dönüşüm belirteci artbileşen argümanının en az 10 karakter genişlikteki bir alana onluk tabanda 8 basamaklı bir **long int** türündeki değerinin sola yanaşık olarak basılacağını belirtir.)

Çıktı dönüşüm belirtimlerinde **%** işaretini izleyen belirteçler sırayla aşağıda açıklanmıştır:

- İsteğe bağlı bir konumlama parametresi. Normalde **printf** şablon dizgesini izleyen argümanları verildikleri sıraya göre basar. Ancak bazı durumlarda (örneğin çevirilerde) bu sıra uygun düşmez ve bunların sırasını değiştirmek gerekir. Bu sıra değişikliğini belirtmek için ayrı bir parametre kullanılır.

  Sözdizimindeki *konum–num* parçası 1 den çağrıdaki artbileşen argümanlarının sayısına kadar bir aralıkta bir tamsayı olmalıdır. Bazı gerçeklemeler belirtilebilecek artbileşen argümanlarının sayısına bir üst sınır koyar. Bu sınır değer aşağıdaki sabit ile belirlenir.

NL_ARGMAX	makro

NL_ARGMAX sabitinin değeri bir **printf** çağrısında belirtilebilecek artbileşen argümanların sayısının üst sınırıdır. Çalışma anında etkin olan bu değer **sysconf**'u **_SC_NL_ARGMAX** parametresi ile kullanarak alınabilir. Bkz. *Sysconf Tanımı* (sayfa: 787).

Bazı sistemler oldukça düşük değerlere sahiptir, örneğin System V için bu değer 9'dur. GNU C kütüphanesinde böyle bir sınır yoktur.

Şablon dizgesindeki dönüşüm belirtimlerinden birinde bu parametre gerekliyse hepsinde olmalıdır. Aksi takdirde, işlevin davranışı tanımlanmamıştır.

- Dönüşüm belirtiminin normal davranışını değiştiren sıfır veya daha fazla ***im karakteri***.
- ***En küçük alan genişliği***ni belirten isteğe bağlı bir onluk tamsayı. Normal dönüşüm bundan daha az karakter üretiyorsa eksik karakterlerin yeri boşluklarla doldurulur. Bu bir *en küçük* değerdir; normal dönüşüm bundan daha fazla karakter üretiyorsa, değer alan genişliği ile sınırlanmaz. Normalde çıktı alan içinde sağa yanaştırılır.

  Alan genişliğini ***** ile de belirtebilirsiniz. Bu durumda argünman listesindeki sonraki argüman alan genişliği olarak kullanılır. Değer **int** türünden olmalıdır. Değer negatifse **-** işareti im karakteri olarak, mutlak değer ise alan genişliği olarak kullanılır.
- Sayısal dönüşümlerde basamak sayısını belirtmek için isteğe bağlı bir ***hassasiyet*** değeri. Hassasiyet belirtilmişse, bir noktadan (**.**) sonra gelen bir onluk tamsayıdır (verilmezse öntanımlı değeri sıfırdır).

  Hassasiyeti ***** ile de belirtebilirsiniz. Bu durumda argünman listesindeki sonraki argüman hassasiyet olarak kullanılır. Değer **int** türünden olmalıdır. Negatifse yoksayılır. Hem alan genişliğini hem de hassasiyeti ***** ile belirtirseniz, alan genişliği argümanı hassaiyet argümanından önce olmalıdır. Diğer C kütüphanesi sürümleri bu sözdizimini tanıyamayabilir.
- İsteğe bağlı bir ***tür değiştirme karakteri***. Artbileşen argümanın türünden farklı bir tür belirtmek için kullanılır. (Örneğin tamsayı dönüşümleri **int** türünde varsayılır, ama **h**, **l** veya **L** harfi ile başka bir tür belirtebilirsiniz.)
- Uygulanacak dönüşümü belirten bir karakter (***dönüşüm belirteci***).

İzin verilen seçenekler ve onların yorumlanışı farklı dönüşüm belirteçleri için aynı değildir. Bir seçeneğin kullanıldığı bir dönüşümdeki yorumlanışı ile ilgili bilgileri, o dönüşümün açıklamalarında bulabilirsiniz.

-Wformat seçeneği ile GNU C derleyicisi **printf** ve ilgili işlevleri denetler. Biçim dizgesine bakarak doğru sayı ve türde argüman belirtilip belirtilmediğini denetler. Yazdığınız bir **printf** tarzı biçim dizgesini denetlemek için GNU C sözdizimini derleyiciye söyleyecek bir işlev de vardır. (GCC info'sundaki "İşlev Özniteliklerinin Bildirilmesi" [Declaring Attributes of Functions] bölümüne bakınız.)

12.3. Çıktı Dönüşüm Belirteçlerinin Listesi

Aşağıda tüm farklı dönüşümler özetlenmiştir:

`%d, %i`
: Bir tamsayıyı işaretli onluk sayı olarak basar. Ayrıntılar için *Tamsayı Dönüşümleri* (sayfa: 258) bölümüne bakınız. **%d** ve **%i** çıktı için eşanlamlıdır ama **scanf** ile girdi için kullanıldıklarında farklıdırlar (Bkz. *Girdi Dönüşüm Belirteçlerinin Listesi* (sayfa: 278)).

`%o`
: Bir tamsayıyı bir işaretsiz sekizlik sayı olarak basar. Ayrıntılar için *Tamsayı Dönüşümleri* (sayfa: 258) bölümüne bakınız.

`%u`
: Bir tamsayıyı bir işaretsiz onluk sayı olarak basar. Ayrıntılar için *Tamsayı Dönüşümleri* (sayfa: 258) bölümüne bakınız.

`%x, %X`
: Bir tamsayıyı bir işaretsiz onaltılık sayı olarak basar. **`%x`** küçük harfleri, **`%X`** büyük harfleri kullanır. Ayrıntılar için *Tamsayı Dönüşümleri* (sayfa: 258) bölümüne bakınız.

`%f`
: Bir gerçek sayıyı ondalık gösterimle basar. Ayrıntılar için *Gerçek Sayı Dönüşümleri* (sayfa: 260) bölümüne bakınız.

`%e, %E`
: Bir gerçek sayıyı üstel gösterimle basar. **`%e`** küçük harf, **`%E`** büyük harf kullanır. Ayrıntılar için *Gerçek Sayı Dönüşümleri* (sayfa: 260) bölümüne bakınız.

`%g, %G`
: Bir gerçek sayıyı büyüklüğüne bağlı olarak ya ondalık ya da üstel gösterimle basar. **`%g`** küçük harf, **`%G`** büyük harf kullanır. Ayrıntılar için *Gerçek Sayı Dönüşümleri* (sayfa: 260) bölümüne bakınız.

`%a, %A`
: Bir gerçek sayıyı 2 tabanındaki üstel kısmını onluk sayı, kesir kısmını onaltılık sayı olarak basar. **`%a`** küçük harf, **`%A`** büyük harf kullanır. Ayrıntılar için *Gerçek Sayı Dönüşümleri* (sayfa: 260) bölümüne bakınız.

`%c`
: Tek karakter basar. Ayrıntılar için *Diğer Çıktı Dönüşümleri* (sayfa: 262) bölümüne bakınız.

`%C`
: Unix standardına destek amacıyla vardır ve **`%lc`** yerine kullanılmak içindir.

`%s`
: Bir dizge basar. Ayrıntılar için *Diğer Çıktı Dönüşümleri* (sayfa: 262) bölümüne bakınız.

`%S`
: Unix standardına destek amacıyla vardır ve **`%ls`** yerine kullanılmak içindir.

`%p`
: Bir gösterici değeri basar. Ayrıntılar için *Diğer Çıktı Dönüşümleri* (sayfa: 262) bölümüne bakınız.

`%n`
: O ana kadar basılan karakterlerin sayısını alır. Bu belirteç bir çıktı üretmez. Ayrıntılar için *Diğer Çıktı Dönüşümleri* (sayfa: 262) bölümüne bakınız.

`%m`
: **`errno`** değerine karşılık olan hata iletisini basar. Bu bir GNU oluşumudur. Ayrıntılar için *Diğer Çıktı Dönüşümleri* (sayfa: 262) bölümüne bakınız.

`%%`
: Sadece **`%`** işareti basar. Ayrıntılar için *Diğer Çıktı Dönüşümleri* (sayfa: 262) bölümüne bakınız.

Bir dönüşüm belirtiminin sözdizimi geçersizse beklenmeyen şeyler olabilir, yani bunu yapmamaya çalışın. Şablon dizgesinde belirtilenden daha az sayıda artbileşen argüman varsa ya da artbileşen argümanların türleri ile dönüşüm belirtimleri uyumsuzsa sonucun ne olacağı belli olmaz. Şablon dizgesinde belirtilenden daha çok sayıda artbileşen argüman varsa fazla argümanlar basitçe yoksayılır; bu bazan kullanışlıdır.

12.4. Tamsayı Dönüşümleri

Bu bölümde **%d**, **%i**, **%o**, **%u**, **%x** ve **%X** dönüşüm belirteçleri ile kullanılan seçenekler açıklanmıştır. Bu dönüşümler tamsayıları çeşitli biçimlerde basmak için kullanılır.

%d ve **%i** belirteçlerinin herikisi de bir işaretsiz ondalık sayı olarak; **%o**, **%u** ve **%x** belirteçleri sırasıyla işaretsiz sekizlik, onluk ve onaltılık sayı olarak basılır. **%X** belirteci **%x** belirteci ile aynı olmakla beraber, **abcdef** karakterleri yerine **ABCDEF** karakterlerini kullanır.

Aşağıdaki im karakterleri anlamlıdır:

\-
: Alan içinde sonucu sola yanaştırır (normalde sağa yanaştırılır).

\+
: İşaretli **%d** ve **%i** belirteçleri için değer pozitifse artı işareti basar.

(boşluk karakteri)
: İşaretli **%d** ve **%i** belirteçleri için sonuç bir artı ya da eksi işareti içermiyorsa işaret yerine bir boşluk basar. **+** im karakteri belirtilmişse sonuç mutlaka bir işaret içerir. Hem **+** hem de boşluk karakteri belirtilmişse boşluk yoksayılır.

\#
: **%o** belirteci için sayının hassasiyeti arttırılmış gibi **0** ile öncelenmesini sağlar. **%x** veya **%X** için sırasıyla **0x** veya **0X** ile öncelenmesini sağlar. **%d**, **%i** veya **%u** için kullanışlı hiçbir şey yapmaz. Bu imin kullanılması **strtoul** (*Tamsayıların Çözümlenmesi* (sayfa: 528)) tarafından ve **scanf** işlevinin **%i** dönüşümü (*Sayısal Girdi Dönüşümleri* (sayfa: 280)) ile çözümlenen çıktıyı üretir.

\'
: **LC_NUMERIC** kategorisi için yerel tarafından belirtildiği gibi sayıyı gruplara ayırır. Bu im bir GNU oluşumudur. Bkz. *Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169).

0
: Alanda boş kalan yerleri boşluk yerine sıfırlarla doldurur. Sıfırlar işaretten ya da taban belirtecinden sonra konur. Bu imle birlikte **-** imi ya da hassasiyet belirtilmişse bu im yoksayılır.

Bir hassasiyet belirtilmişse, görüntülenecek hane sayısıdır ve gerekliyse baştaki sıfırlar üretilir. Hassasiyet belirtilmişse sayı olduğu gibi basılır. Açıkça belirterek sıfır hassasiyetle bir sıfır değeri dönüştürmek isterseniz hiçbir karakter üretilmez.

Bir tür değiştirici olmaksızın, karşı düşen argüman bir **int** (işaretli **%i** ve **%d** dönüşümleri için) ya da **unsigned int** (işaretsiz **%o**, **%u**, **%x** ve **%X** dönüşümleri için) olarak ele alınır. **printf** ve arkadaşları değişkin işlevler olduklarından **char** ve **short** argümanlar öntanımlı argüman terfileri tarafından özdevinimli olarak **int** türüne dönüştürülür. Diğer tamsayı türleri için aşağıdaki tür değiştiricileri kullanabilirsiniz:

hh
: Argümanın duruma göre **signed char** veya **unsigned char** olarak yorumlanacağını belirtir. Öntanımlı argüman terfileri çerçevesinde **char** türünden bir değer **int** veya **unsigned int** türüne terfi ettirilir. Ancak **hh** tür değiştiricisi onu tekrar **char** türüne çevirir.

  Bu değiştirici ISO C99'da tanımlanmıştır.

h
: Argümanın duruma göre **short int** veya **unsigned short int** olarak yorumlanacağını belirtir. Öntanımlı argüman terfileri çerçevesinde **short** türünden bir değer **int** veya **unsigned int** türüne terfi ettirilir. Ancak **h** tür değiştiricisi onu tekrar **short** türüne çevirir.

`j`
: Argümanın duruma göre **`intmax_t`** veya **`uintmax_t`** olarak yorumlanacağını belirtir.

  Bu değiştirici ISO C99'da tanımlanmıştır.

`l`
: Argümanın duruma göre **`long int`** veya **`unsigned long int`** olarak yorumlanacağını belirtir. İki **`l`** karakteri biraz aşağıda açıklanan **`L`** değiştiricisine benzer.

  Bir **`%c`** veya **`%s`** belirteci ile birlikte kullanılmışsa argüman bir geniş karakter veya bir geniş karakter dizgesi olarak yorumlanır. **`l`** değiştiricisinin bu kullanımı ISO C90 1. düzeltmesinde tanımlanmıştır.

`L, ll, q`
: Argümanın **`long long int`** türünde yorumlanacağını belirtirler. (Bu tür GNU C derleyicisi tarafından desteklenen bir oluşumdur. Fazla uzun tamsayıları desteklemeyen sistemlerde bu değiştirici **`long int`** olarak değerlendirilir.)

  `q` değiştiricisi 4.4 BSD de bulunur. Bir **`long long int`** bazan "quad" **`int`** olarak da isimlendirilir.

`t`
: Argümanın **`ptrdiff_t`** türünde yorumlanacağını belirtir.

  Bu değiştirici ISO C99'da tanımlanmıştır.

`z, Z`
: Argümanın **`size_t`** türünde yorumlanacağını belirtirler.

  `z` değiştiricisi ISO C99'da tanımlanmıştır. **`Z`** ise bu eklemeden önceki bir GNU oluşumudur ve yeni kodlarda kullanılmamalıdır.

Aşağıda bir şablon dizgesi örneği vardır:

```
"|%5d|%-5d|%+5d|%+-5d|% 5d|%05d|%5.0d|%5.2d|%d|\n"
```

`%d` için farklı seçeneklerin kullanıldığı bu dizge aşağıdaki sonuçları verir:

```
|    0|0    |   +0|+0   |    0|00000|     |   00|0|
|    1|1    |   +1|+1   |    1|00001|    1|   01|1|
|   -1|-1   |   -1|-1   |   -1|-0001|   -1|  -01|-1|
|100000|100000|+100000|+100000| 100000|100000|100000|100000|100000|
```

Son satırda en küçük alan genişliğine sığmayan bir sayı belirtilmiş olduğuna dikkat ediniz.

Aşağıda ise işaretsiz tamsayılar çeşitli biçim seçenekleri ile kullanılmıştır:

```
"|%5u|%5o|%5x|%5X|%#5o|%#5x|%#5X|%#10.8x|\n"
```

```
|    0|    0|    0|    0|    0|    0|    0|  00000000|
|    1|    1|    1|    1|   01|  0x1|  0X1|0x00000001|
|100000|303240|186a0|186A0|0303240|0x186a0|0X186A0|0x000186a0|
```

12.5. Gerçek Sayı Dönüşümleri

Bu bölümde gerçek sayıların dönüşüm belirtimlerine yer verilmiştir. Bunlar **`%f`**, **`%e`**, **`%E`**, **`%g`** ve **`%G`** dönüşümleridir.

`%f` dönüşümü argümanını ondalık gösterimle aşağıdaki biçimde çıktılar:

[**-**] *ddd* **.** *ddd*

Buradaki ondalık noktayı izleyen basamakların sayısını hassasiyet parametresi ile belirtebilirsiniz.

%e dönüşümü argümanını üstel gösterimle aşağıdaki biçimde çıktılar:

[**-**] *d* **.** *ddd* **e** [**+**|**-**] *dd*

Burada da ondalık noktayı izleyen basamakların sayısını hassasiyet parametresi ile belirtebilirsiniz. Üs daima son iki hanede bulunur. **%E** dönüşümü de aynı biçimi üretir tek farkla **e** yerine **E** harfi kullanılır.

Üs –4 den küçükse ya da hassasiyete eşit veya daha büyükse **%g** ve **%G** dönüşümleri argümanı **%e** veya **%E** (sırasıyla) ile eşdeğerdir. Aksi takdirde, **%f** ile eşdeğerdir. **0** hassasiyet **1** olarak değerlendirilir. Sonucun ondalık kısmının sağındaki sıfırlar kaldırılır ve nokta sadece sağında bir sayı varsa gösterilir.

%a ve **%A** dönüşümleri gerçek sayıyı metin biçiminde çıktılar. Bu çıktı farklı yazılım ve/veya makinalar arasında metin verisi olarak aktarılabilir. Sayılar aşağıdaki biçimde gösterilir:

[**-**] **0x** *h* **.** *hhh* **p** [**+**|**-**] *dd*

Ondalık noktanın solunda sadece bir rakam bulunur. Sayı normalleştirilmemişse bu rakam **0**'dır. Aksi takdirde gerçeklemede kaç bit ayrıldığına bağlı olarak herhangi bir rakam olabilir. Ondalık noktanın sağındaki basamakların sayısı hassasiyet için belirtilen değerle sınırlıdır. Hassasiyet olarak sıfır verilmişse sayının en doğru gösterilebileceği yeterlikteki (veya **FLT_RADIX** (sayfa: 824) değeri 2'nin kuvvetlerindeki bir değer değilse yanyana iki değerin iki ayrı değer olarak seçilebileceği yeterlikte) basamak sayısı kullanılır. **%a** dönüşümü için onaltılık sayı, **0x** ve **p** ile gösterilen önek ve üstel değer için küçük harfler kullanılır. **%A** dönüşümünde bunlar için büyük harfler kullanılır. Üstel kısım iki tabanına göre onluk bir sayı olarak en az bir rakamla ifade edilir, ancak değer gerektiriyorsa daha fazla rakam da kullanılabilir.

Basılan değer sonsuzluk ya da sayı olmayan bir değerse ve dönüşüm belirteçleri **%a**, **%e**, **%f** veya **%g** ise çıktı [**-**]**inf** ya da**nan**'dır; dönüşüm belirteçleri **%A**, **%E** veya **%G** ise çıktı [**-**]**INF** veya **NAN**'dır.

Davranışı değiştirmek için kullanılan imler şunlardır:

-
: Alan içinde sonucu sola yanaştırır (normalde sağa yanaştırılır).

+
: Sonuç daima bir artı ya da eksi işareti içerir.

(boşluk karakteri)
: Sonuç bir artı ya da eksi işareti içermiyorsa işaret yerine bir boşluk basar. **+** im karakteri belirtilmişse sonuç mutlaka bir işaret içerir. Hem **+** hem de boşluk karakteri belirtilmişse boşluk yoksayılır.

#
: Sonuç ondalık noktanın sağında hiçbir sayı olmasa bile ondalık nokta gösterilir. **%g** ve **%G** dönüşümleri için noktanın sağı, konmaması belirtilmemişse, sıfırlarla doldurulur.

'
: **LC_NUMERIC** kategorisi için yerel tarafından belirtildiği gibi tamsayı kısmı gruplara ayırır. Bu im bir GNU oluşumudur. Bkz. *Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169).

0
: Alanda boş kalan yerleri boşluk yerine sıfırlarla doldurur. Sıfırlar işaretten sonra konur. Bu imle birlikte **-** imi belirtilmişse bu im yoksayılır.

Bir hassasiyet belirtilmişse, **%f**, **%e** ve **%E** dönüşümleri için ondalık noktadan sonraki basamakların sayısıdır. Bu dönüşümler için öntanımlı hassasiyet 6'dır. Hassasiyet olarak **0** değeri belirtilmişse ondalık nokta basılmaz.

%g ve **%G** dönüşümleri için hassasiyet kıymetli hanelerin sayısıdır. Kıymetli haneler noktanın solundaki tek hane ile noktanın sağındakilerin tamamıdır. Hassasiyet **0** ise ya da belirtilmemişse **1** belirtilmiş varsayılır. Basılacak değer belirtilen hane sayısı ile doğru olarak gösterilemeyecekse gösterilebilecek en yakın sayıya yuvarlanır.

Tür değiştirici belirtilmeksizin tüm gerçek sayı dönüşüm belirteçleri argümanı **double** türünde yorumlar. (Öntanımlı argüman terfileri çerçevesinde **float** türünden argümanlar özdevinimli olarak **double** türüne dönüştürülür.) Desteklenen tür değiştirici:

L
: Argümanın **long double** türünde yorumlanacağını belirtir.

Aşağıdaki örneklerde gerçek sayı dönüşümleri kullanılarak sayıların nasıl basılacağı gösterilmiştir:

```
"|%13.4a|%13.4f|%13.4e|%13.4g|\n"
```

Şablon digesi için çıktılar:

```
|  0x0.0000p+0|       0.0000|   0.0000e+00|            0|
|  0x1.0000p-1|       0.5000|   5.0000e-01|          0.5|
|  0x1.0000p+0|       1.0000|   1.0000e+00|            1|
| -0x1.0000p+0|      -1.0000|  -1.0000e+00|           -1|
|  0x1.9000p+6|     100.0000|   1.0000e+02|          100|
|  0x1.f400p+9|    1000.0000|   1.0000e+03|         1000|
| 0x1.3880p+13|   10000.0000|   1.0000e+04|        1e+04|
| 0x1.81c8p+13|   12345.0000|   1.2345e+04|    1.234e+04|
| 0x1.86a0p+16|  100000.0000|   1.0000e+05|        1e+05|
| 0x1.e240p+16|  123456.0000|   1.2346e+05|    1.235e+05|
```

%g dönüşümünde sağdaki sıfırların kaldırıldığına dikkat edin.

12.6. Diğer Çıktı Dönüşümleri

Bu bölümde **printf** için çeşitli dönüşümlerden bahsedilmiştir.

%c dönüşümü tek bir karakter basar. **l** değiştiricisinin olmadığı durumda **int** türündeki argüman önce **unsigned char** türüne dönüştürülür. Bundan sonra, bir geniş yönlenimli akım işlevi kullanılmışsa karakter karşı düşen geniş karaktere dönüştürülür. **-** imi kullanılmış, başka bir im tanımlanmamış, tür ve hassasiyet belirtilmemişse karakter alanda sola yanaştırılır. Örneğin:

```
printf ("%c%c%c%c%c", 'h', 'e', 'l', 'l', 'o');
```

deyimi **hello** basar.

Bir **l** değiştiricisi varsa argümanın **wint_t** türünde olacağı umulur. Bir çokbaytlı işlev içinde kullanılmışsa geniş karakter çıktıya eklenmeden önce bir çok baytlı karaktere dönüştürülür. Bu durumda birden fazla bayt içeren çıktı üretilebilir.

%s dönüşümü bir dizge basar. Bir **l** değiştiricisi yoksa karşı düşen argüman **char *** (ya da **const char ***) türünde olmalıdır. Bir geniş yönlenimli akım işlevi kullanılmışsa, dizge bir geniş karakter dizgesine dönüştürülür. Yazılacak karakterlerin sayısı hassasiyet parametresi ile belirtilebilir; aksi takdirde, sonlandıran boş karaktere kadar boş karakter hariç dizge akıma çıktılanır. **-** imi kullanılmış ama başka bir im tanımlanmamış, tür ve hassasiyet belirtilmemişse dizge alanda sola yanaştırılır. Örneğin:

```
printf ("%3s%-6s", "no", "where");
```

deyimi **nowhere** basar.

Bir **l** değiştiricisi varsa argümanın **wchar_t** (ya da **const wchar_t ***) türünde olacağı umulur.

%s dönüşümüne istemeyerek argüman olarak bir boş gösterici aktarırsanız, GNU kütüphanesi onu **(null)** olarak basar. Bunun çökmekten daha iyi olduğunu düşünüyoruz. Ancak bir boş göstericiyi bile bile aktarmak iyi bir uygulama olmayacaktır.

%m dönüşümü **errno** hata kodunun karşılığı olan dizgeyi basar. Bkz. *Hata İletileri* (sayfa: 41). Burada:

```
fprintf (stderr, "`%s' açılamıyor: %m\n", filename);
```

deyimi ile

```
fprintf (stderr, "`%s' açılamıyor: %s\n", filename, strerror (errno));
```

deyimi eşdeğerdir. **%m** dönüşümü bir GNU oluşumudur.

%p dönüşümü bir gösterici değeri basar. Karşı düşen argüman **void *** türünde olmalıdır. Uygulamada herhangi bir türden gösterici kullanabilirsiniz.

GNU sisteminde boş olmayan göstericiler bir **%#x** dönüşümü kullanılmış gibi işaretsiz tamsayılar olarak basılır. Boş göstericiler **(nil)** olarak basılır. (Diğer sistemlerde göstericiler farklı basılabilir.) Örneğin:

```
printf ("%p", "testing");
```

deyimi **0x**'den sonra **"testing"** dizge sabitinin adresini bir onaltılık sayı olarak basar; **testing** sözcüğünü basmaz.

- imi kullanılmış ama başka bir im tanımlanmamış, tür ve hassasiyet belirtilmemişse gösterici değeri alanda sola yanaştırılır.

%n belirteci diğer dönüşüm belirteçlerine benzemez. Bir **int** türüne gösterici olan bir argüman kullanır, ama birşey çıktılamaz, onun yerine o çağrı ile çıktılanan karakterlerin sayısını kaydeder. **h** ve **l** tür değiştiricileri kullanılarak (birim, hassasiyet veya alan genişliği belirtilemez) argümanın türü **int *** yerine **short int *** veya **long int *** türünde belirtilebilir. Örneğin:

```
int nchar;
printf ("%d %s%n\n", 3, "bears", &nchar);
```

kod parçası

```
3 bears
```

basar ve **nchar**'a 7 değerini atar, çünkü basılan karakterlerin sayısı 7'dir.

%% belirteci bir **%** işareti basar. Bu belirteç bir argüman kullanmaz ve im, alan genişliği, hassasiyet veya tür değiştirici belirtilemez.

12.7. Biçimli Çıktı İşlevleri

Bu bölümde **printf** ve ilgili işlevlerin nasıl çağrıldığı anlatılmıştır. Bu işlevlerin prototipleri `stdio.h` başlık dosyasındadır. Bu işlevlerin değişken sayıda argüman almalarından dolayı, onları kullanmadan önce prototipleri ile bildirmelisiniz. Şüphesiz bunu yapmanın en kolay yolu onları içeren `stdio.h` başlık dosyasının kaynak dosyasında içerilmesidir.

```
int printf(const char *şablon, ...)                                    işlev
```

printf işlevi *şablon* biçim dizgesinin denetimi altında isteğe bağlı argümanları standart çıktıya basar ve basılan karakterlerin sayısı ile döner; bir hata oluşursa bir negatif değerle döner.

```
int wprintf(const wchar_t *şablon, …)                                    işlev
```

wprintf işlevi *şablon* geniş biçim dizgesinin denetimi altında isteğe bağlı argümanları standart çıktıya basar ve basılan geniş karakterlerin sayısı ile döner; bir hata oluşursa bir negatif değerle döner.

```
int fprintf(FILE        *akım,                                           işlev
            const char *şablon,
            …)
```

Bu işlev **printf** gibidir, tek farkla, çıktıyı standart çıktıya değil *akım* 'a yazar.

```
int fwprintf(FILE           *akım,                                       işlev
             const wchar_t *şablon,
             …)
```

Bu işlev **wprintf** gibidir, tek farkla, çıktıyı standart çıktıya değil *akım* 'a yazar.

```
int sprintf(char        *s,                                              işlev
            const char *şablon,
            …)
```

Bu işlev **printf** gibidir, tek farkla, çıktıyı standart çıktıya değil *s* dizisine yazar. Dizge bir boş karakterle sonlandırılır.

sprintf işlevi sonlandırıcı boş karakter hariç *s* dizisine yazılan karakterlerin sayısı ile döner.

Bu işlevin davranışı, birbirini kapsayan nesneler arasında kopyalama yapılırsa, örneğin, **%s** belirtecinin argümanı olarak *s* dizgesi verilmişse, tanımsızdır. Bkz. *Kopyalama ve Birleştirme* (sayfa: 94).

Uyarı

sprintf işlevi *tehlikeli* olabilir. Potansiyel olarak *s* dizgesi için ayrılan yer biçim dizgesinin ürettiği dizge için yetersiz olabilir. Alan genişliği parametresinin en küçük alan genişliğini belirttiğini ve bazan bu genişliğin aşılabildiğini hatırlayın.

Bu sorundan kaçımak için aşağıda açıklanacak olan **snprintf** veya **asprintf** işlevlerini kullanabilirsiniz.

```
int swprintf(wchar_t        *s,                                          işlev
             size_t          boyut,
             const wchar_t *şablon,
             …)
```

Bu işlev **printf** gibidir, tek farkla, çıktıyı standart çıktıya değil *ws* geniş karakter dizisine yazar. Dizge bir boş geniş karakterle sonlandırılır. *boyut* argümanı ile üretilecek en büyük karakter sayısı belirtilir. Sonlandırıcı boş karakter de bu sınırın içinde olduğundan *ws* dizgesi için en az *boyut* geniş karakterlik yer ayrılmalıdır.

swprintf işlevi sonlandırıcı boş geniş karakter hariç *ws* geniş dizisine yazılan karakterlerin sayısı ile döner. Çıktı belirtilen tampona sığmazsa bir negatif sayı ile döner. Bu davranışı ile işlev **snprintf** işlevinden farklıdır.

Bilgi

Benzeri olan dar yönlenimli akım işlevleri daha az parametre alır. **swprintf** işlevi aslında **snprintf** işlevinin karşılığıdır. **sprintf** işlevi tehlikeli olabildiğinden, ISO C komitesi aynı yanlışı tekrar yapmaktan kaçınmalı ve **sprintf** işlevinin tam karşılığı olan bir işlevi tanımlamama kararı almalıdır.

```
int snprintf(char        *s,                                                    işlev
             size_t       boyut,
             const char *şablon,
             …)
```

Bu işlev **sprintf** gibidir, tek farkla, *boyut* argümanı ile üretilecek en büyük karakter sayısı belirtilir. Sonlandırıcı boş karakter de bu sınırın içinde olduğundan *s* dizgesi için en az *boyut* karakterlik yer ayrılmalıdır.

snprintf işlevi sonlandırıcı boş geniş karakter hariç *s* dizisine yazılan karakterlerin sayısı ile döner. Bu değer *boyut* 'a eşit ya da ondan büyükse sonucun tüm karakterleri *s* dizisine yazılamamış demektir. Bu durumda daha büyük çıktı dizisi ile çağrıyı yinelemelisiniz. Aşağıda bunun yapıldığı bir örnek vardır:

```
/*  isim isminde ve deger değerinde
    bir değişkenin değerini açıklayan bir ileti oluşturalım. */
char *
ileti_yap (char *isim, char *deger)
{
  /* Nasılsa 100 karakterlikten daha fazla yere ihtiyaç olmaz. */
  int boyut = 100;
  char *tampon = (char *) xmalloc (boyut);
  int nkar;
  if (tampon == NULL)
    return NULL;

  /* Ayrılan alana iletiyi yazmaya çalışalım. */
  nkar = snprintf (tampon, boyut, "%s değişkeninin değeri %s dir",
                   isim, deger);
  if (nkar >= boyut)
    {
      /* Artık ne kadar yere ihtiyaç olduğunu
         bildiğimize göre tamponu yeniden ayıralım. */
      tampon = (char *) xrealloc (tampon, nkar + 1);
      boyut = nkar + 1;
      buffer = (char *) xrealloc (tampon, boyut);

      if (tampon != NULL)
        /* Tekrar deniyoruz. */
        snprintf (tampon, boyut, "%s değişkeninin değeri %s dir",
                  isim, deger);
    }
  /* Son çağrı çalıştığından dizge ile dönüyoruz. */
  return tampon;
}
```

Uygulamada, **asprintf** işlevini kullanmak çok daha kolaydır.

Dikkat

GNU C kütüphanesinin 2.1 öncesi sürümlerinde dönüş değeri sonlandırıcı boş karakter hariç saklanan karakterlerin sayısıydı. *s* dizisinde yeterli yer yoksa **−1** dönüyordu. Bu, ISO C99 standardına uyum sırasında değiştirildi.

12.8. Biçimli Çıktıyı Özdevimli Ayırma

Bu bölümdeki işlevler biçimli çıktı oluşturur ve sonucu özdevimli ayrılan belleğe yazar.

```
int asprintf(char       **gstr,                                          işlev
             const char *şablon,
             …)
```

Bu işlev, çıktıyı sizin ayırdığınız bir tampona değil, özdevimli ayrılmış (**malloc** ile; Bkz. *Özgür Bellek Ayırma* (sayfa: 50)) bir dizgeye yazaması dışında **sprintf** işlevi gibidir. *gstr* argümanı, bir **char *** nesnesinin adresi olmalıdır. Başarılı bir **asprintf** çağrısı bu konumda yeni ayrılmış bir dizgeye gösterici saklayacaktır.

Dönen değer ayrılan dizgenin karakter sayısıdır; dizge için yer ayrılamadığı durumlarda ve hata durumunda sıfırdan küçük bir değerle döner.

Aşağıdaki örnekte, **snprintf** örneğindeki sonucun **asprintf** kullanarak nasıl elde edildiği gösterilmektedir:

```
/*  isim isminde ve deger değerinde
    bir değişkenin değerini açıklayan bir ileti oluşturalım. */
char *
ileti_yap (char *isim, char *deger)
{
  char *sonuc;
  if (asprintf (&sonuc, "%s değişkeninin değeri %s dir", isim, deger) < 0)
    return NULL;
  return sonuc;
}
```

```
int obstack_printf(struct obstack *yığınak,                              işlev
                   const char     *şablon,
                   …)
```

Bu işlev, alan ayırmak için *yığınak* yığınağını kullanması dışında **asprintf** işlevinin benzeridir. Yığınaklarla ilgili bilgi için *Yığınaklar (Obstacks)* (sayfa: 64) bölümüne bakınız.

Karakterler nesnenin sonuna yazılır. Onları almak için nesneyi **obstack_finish** ile bitirmelisiniz. Bkz. *Büyüyen Nesneler* (sayfa: 69).

12.9. Değişkin Çıktı İşlevleri

Yerleşik biçimli çıktı işlevleri olarak **printf** oluşumunu kullanarak kendi değişkin **printf** benzeri işlevlerinizi tanımlayabilmeniz için **vprintf** işlevi ve arkadaşları oluşturulmuştur.

Bu tür işlevleri tanımlamanın en doğal yolu, "**printf** işlevini çağır ve bu şablonla birlikte ilk 5 argümandan sonra benim argümanlarımı aktar" diyecek bir dil yapısı kullanmak olurdu. Fakat C'de bunu yapmanın bir yolu yoktur. Ayrıca üçüncü düzey C dilinde işlevinizin kaç argüman aldığını söyleyecek bir yol olmadığından, bir yol üretmek de zor olurdu.

Bu yöntemle mümkün olmadığından "ilk 5 argümandan sonra benim argümanlarımı aktar" diyebileceğiniz bir **va_list**'i aktaracağınız bir seri işlev, **vprintf** serisini ürettik.

Gerçek bir işlev yerine bir makro tanımlamak yeterli olduğundan, GNU C derleyicisi bunu makrolarla çok daha kolay yapacak bir yol sağlar. Örneğin:

```
#define myprintf(a, b, c, d, e, rest...) \
        printf (mytemplate , ## rest)
```

Değişken sayıda argümanlı makrolar hakkında ayrıntılı bilgi edinmek için GCC info'sundaki "Macros with Variable Numbers of Arguments" (Değişken sayıda argümanlı makrolar) bölümüne bakınız. Bu yol sadece makrolara ayrılmıştır, gerçek işlevlere uygulanmaz.

vprintf veya bu bölümde listelenmiş diğer işlevleri çağırmadan önce değişken sayıdaki argümanı ilklendirmek için bir **va_start** çağrısı (*Değişkin İşlevler* (sayfa: 814)) yapmalısınız. Sonra da kendinizin elde etmek istediğiniz argümanları almak için **va_arg** çağrıları yapmalısınız.

va_list göstericinize seçtiğiniz ilk argümanı gösterdikten sonra bir **vprintf** çağrısı yapmaya hazırsınız demektir. İşlevinizle aktardığınız bu argüman ve artbileşen argümanlar ile belirttiğiniz şablon dizgesi **vprintf** işlevi tarafından kullanılacaktır.

Diğer sistemlerde **vprintf** çağrısından sonra **va_list** göstericisi geçersiz duruma gelebilir ve **vprintf** çağrısından sonra **va_arg** işlevini kullanamayabilirsiniz. Bu durumda bir **va_end** çağrısından sonra **va_start** ile göstericiyi yeniden almalısınız. **vprintf** işlevi işlevinizin argüman listesini ortadan kaldırmaz.

GNU C böyle sınırlamalara sahip değildir. **va_list** göstericisini **vprintf** çağrısına aktardıktan sonra bu göstericiyi kullanarak argümanları almaya devam edebilirsiniz ve **va_end** çağrısı da hiçbir işlem yapmaz. (**vprintf** çağrısından sonra yaptığınız **va_arg** çağrıları ile alınan argümanlar **vprintf** çağrısındaki argümanlarla aynıdır.)

Bu işlevlerin prototipleri `stdio.h` başlık dosyasında bildirilmiştir.

```
int vprintf(const char *şablon,                                    işlev
            va_list     arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **printf** ile aynıdır.

```
int vwprintf(const wchar_t *şablon,                                işlev
             va_list        arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **wprintf** ile aynıdır.

```
int vfprintf(FILE        *akım,                                    işlev
             const char *şablon,
             va_list     arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **vprintf** ile aynıdır.

```
int vfwprintf(FILE           *akım,                                işlev
              const wchar_t *şablon,
              va_list        arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **vwprintf** ile aynıdır.

```
int vsprintf(char        *s,                                       işlev
             const char *şablon,
             va_list     arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **sprintf** ile aynıdır.

```
int vswprintf(wchar_t       *s,                                    işlev
              size_t         boyut,
              const wchar_t *şablon,
              va_list        arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **swrintf** ile aynıdır.

```
int vsnprintf(char       *s,                                     işlev
              size_t      boyut,
              const char *şablon,
              va_list     arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **snprintf** ile aynıdır.

```
int vasprintf(char      **gstr,                                   işlev
              const char *şablon,
              va_list     arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **vasprintf** ile aynıdır.

```
int obstack_vprintf(struct obstack *yığınak,                      işlev
                    const char     *şablon,
                    va_list         arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine *arglist–gstr* argüman listesi göstericisinden alması dışında **obstack_vprintf** ile aynıdır.

Aşağıda **vfprintf** kullanılan bir örnek vardır. Örnekteki işlev yazılımın ismi ile önceleyerek standart hataya bir ileti basmaktadır. (**program_invocation_short_name** değişkeninin açıklamaları için *Hata İletileri* (sayfa: 41) bölümüne bakınız.)

```
#include <stdio.h>
#include <stdarg.h>

void
eprintf (const char *sablon, ...)
{
  va_list ap;
  extern char *program_invocation_short_name;

  fprintf (stderr, "%s: ", program_invocation_short_name);
  va_start (ap, sablon);
  vfprintf (stderr, sablon, ap);
  va_end (ap);
}
```

eprintf işlevi şöyle çağrılabilir:

```
eprintf ("`%s' diye bir dosya yok\n", filename);
```

GNU C'de **printf** tarzı biçim dizgesi kullanan bir işlevi derleyiciye bildirebileceğiniz özel bir yapı vardır. Bu yapı kullanıldığında işlevin her çağrısı için kullanılan argümanların türleri ve sayısı denetlenir ve biçim dizgesiyle eşleşmeyenlar için sizi uyarır. Örneğin, bu yapıyı **eprintf** için şöyle kullanabilirsiniz:

```
void eprintf (const char *sablon, ...)
        __attribute__ ((format (printf, 1, 2)));
```

Bu kod parçası derleyiciye **eprintf** işlevinin ilk argümanının **printf** biçim dizgesi olarak, biçim dizgesini oluşturan diğer argümanların başlangıcının ikinci argüman olarak kullanıldığını söyler. İşlev özniteliklerinin bildirilmesi ile ilgili ayrıntılı bilgi edinmek için GCC kılavuzunun (info) "Declaring Attributes of Functions" (İşlevlerin Özniteliklerinin Bildirilmesi") bölümüne bakınız.

12.10. Bir Şablon Dizgesinin Çözümlenmesi

parse_printf_format işlevini kullanarak bir şablon dizgesindeki dönüşüm belireteçlerinin karşılığı olan argümanların sayısı ve türü hakkında bilgi alabilirsiniz. Bu işlevi kullanarak kullanıcının uygulamadan geçersiz argümanlar girerek bir çökmeye sebep olmasından kaçınmak için **printf**'e bir arayüz oluşturabilirsiniz.

Bu bölümde açıklanan tüm işlevler `printf.h` başlık dosyasında bildirilmiştir.

```
size_t parse_printf_format(const char *şablon,                    işlev
                           size_t      n,
                           int        *argtürleri)
```

Bu işlev *şablon* biçim dizgesi tarafından kullanılacak argümanların sayısı ve türleri hakkında bilgi ile döner. Bu bilgi her elemanı bir argüman için olmak üzere *argtürleri* dizisine kaydedilir. Bu bilgi aşağıda listelenen çeşitli **PA_** makroları kullanılarak kodlanır.

n argümanı ile *argtürleri* dizisindeki eleman sayısı belirtilir. Bu sayı, **parse_printf_format** işlevinin yazmayı deneyeceği en çok eleman sayısıdır.

parse_printf_format işlevi *şablon* dizgesinin gerektirdiği argüman sayısı ile döner. Bu sayı *n* ile belirtilenden büyükse dönen bilgi sadece ilk *n* argüman içindir. Tüm argümanlar için bilgi almak isterseniz, daha büyük bir dizi ayırıp işlevi tekrar çağırmayı denemelisiniz.

Argüman türleri temel türlerin ve tür değiştirici parametrelerinin bir birleşimi olarak kodlanır.

```
int PA_FLAG_MASK                                                  makro
```

Bu makro tür değiştirici seçenek bitleri için bir bit maskesidir. Bir argümanın seçenek bitlerini çıkarmak için **(argturleri[i] PA_FLAG_MASK)** ifadesini, temel tür kodunu çıkarmak için **(argturleri[i] ~PA_FLAG_MASK)** ifadesini yazabilirsiniz.

Tamsayı değerler olarak temel türleri ifade eden sembolik sabitler:

`PA_INT`
: **int** temel türünü belirtir.

`PA_CHAR`
: **char** türüne dönüşen **int** temel türünü belirtir.

`PA_STRING`
: Bir boş karakter sonlandırmalı dizge gösteren **char *** temel türünü belirtir.

`PA_POINTER`
: Herhangi bir gösterici olarak **void *** temel türünü belirtir.

`PA_FLOAT`
: **float** temel türünü belirtir.

`PA_DOUBLE`

double temel türünü belirtir.

PA_LAST
: Kendi yazılımınız için **PA_LAST**'ın artan değerleri olarak ek temel türler tanımlayabilirsiniz. Örneğin, **foo** ve **bar** veri türlerinin kodlamasını kendi özelleştirilmiş **printf** dönüşümleri ile şöyle tanımlamalısınız:

```
#define PA_FOO  PA_LAST
#define PA_BAR  (PA_LAST + 1)
```

Aşağıdaki bir temel türü değiştiren seçenek bitleri listelenmiştir. Bunları temel türlerle VEYA'lanarak birleştirilebilir.

PA_FLAG_PTR
: Bu bit 1 ise kodlanan türün bir değere değil temel türe bir gösterici olduğunu belirtir. Örneğin, **PA_INT|PA_FLAG_PTR** ifadesi **int *** türü içindir.

PA_FLAG_SHORT
: Bu bit 1 ise temel türün **short** ile değiştirildiğini belirtir. (Bu **h** tür değiştiricisine karşılıktır.)

PA_FLAG_LONG
: Bu bit 1 ise temel türün **long** ile değiştirildiğini belirtir. (Bu **l** tür değiştiricisine karşılıktır.)

PA_FLAG_LONG_LONG
: Bu bit 1 ise temel türün **long long** ile değiştirildiğini belirtir. (Bu **L** tür değiştiricisine karşılıktır.)

PA_FLAG_LONG_DOUBLE
: Bu **PA_FLAG_LONG_LONG** ile eşanlamlıdır ve teamülen **long double** türü belirten **PA_DOUBLE** temel türü ile kullanılır.

12.11. Bir Şablon Dizgesinin Çözümlenmesi Örneği

Bu örnekte bir biçim dizgesi için gereken argüman türleri çözümlenmektedir. Bu örneğe özel olarak argüman türlerini **NUMBER**, **CHAR**, **STRING** ve **STRUCTURE** isimleri ile sınıflandırdık (başka türlerde var tabii ama bu sadece bir örnek).

```
/* Belirtilen narg adet nesne şanlon dizgesi için
   geçerli mi, değil mi?
   Geçerliyse 1,
   değilse 0 dönsün ve bir hata iletisi bassın. */

int
validate_args (char *sablon, int nargs, OBJECT *args)
{
  int *argturleri;
  int ngereken;

  /* Argümanlar hakkında bilgi alalım.
     Her dönüşüm belirtimi en az iki karakterlik olmalı,
     o halde dizgenin yarı uzunluğundan daha fazla sayıda
     belirtim olamaz. */

  argturleri = (int *) alloca (strlen (sablon) / 2 * sizeof (int));
  ngereken = parse_printf_format (sablon, nargs, argturleri);
```

```
  /* Argüman sayısına bakalım.  */
  if (ngereken > nargs)
  {
    error ("argüman sayısı çok az (en az %d argüman gerekiyor)", ngereken);
    return 0;
  }

  /* Her argüman için istenen C türüne bakalım
     ve verilen nesne uygun mu görelim. */
  for (i = 0; i < ngereken; i++)
  {
    int istenen;

    if (argturleri[i] & PA_FLAG_PTR)
      istenen = STRUCTURE;
    else
      switch (argturleri[i] & ~PA_FLAG_MASK)
      {
        case PA_INT:
        case PA_FLOAT:
        case PA_DOUBLE:
          istenen = NUMBER;
          break;
        case PA_CHAR:
          istenen = CHAR;
          break;
        case PA_STRING:
          istenen = STRING;
          break;
        case PA_POINTER:
          istenen = STRUCTURE;
          break;
      }
    if (TYPE (args[i]) != istenen)
    {
      error ("%d. argümanın türü uygun değil", i);
      return 0;
    }
  }
  return 1;
}
```

13. `printf` İşlevinin Özelleştirilmesi

GNU C kütüphanesi uygulamanızın önemli veri yapılarını basacak becerikli yöntemleri **`printf`**'e öğretecek özelleştirilmiş dönüşüm belirteçleri tanımlamanızı mümkün kılar.

Bunu yapmanın yolu **`register_printf_function`** işlevini kullanarak dönüşümleri kaydetmektir; Bkz. *Yeni Dönüşümlerin Kaydı* (sayfa: 272). Bu işleve aktaracağınız argümanlardan biri asıl çıktılamayı yapacak olan işleve bir gösterici olacaktır; bu işlevin nasıl yazılacağı *Kotarıcı İşlevin Tanımlanması* (sayfa: 274) bölümünde anlatılmıştır.

Ayrıca tanımladığınız dönüşüm belirteçleri tarafından kullanılacak argümanların sayısı ve türü hakkında bilgi veren bir işlev de yazabilirsiniz; bu konu için *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269) bölümüne bakınız.

Bu kısımda sözü edilen oluşumlar `printf.h` başlık dosyasında tanımlanmıştır.

Taşınabilirlik Bilgisi:

printf şablon dizgesi sözdiziminin geliştirilebilirlik özelliği bir GNU oluşumudur. ISO C standardında buna benzer birşey yoktur.

13.1. Yeni Dönüşümlerin Kaydı

register_printf_function işlevi printf.h başlık dosyasında bildirilmiştir ve yeni tanımladığınız bir dönüşüm belirtecini kaydetmek için kullanılır.

```
int register_printf_function(int                      belirteç,                  işlev
                             printf_function          kotarıcı–işlev,
                             printf_arginfo_function  argtürleri–işlevi)
```

Bu işlev *belirteç* dönüşüm belirteç karakterini tanımlar. Eğer *belirteç* **'Y'** ise dönüşüm belirteci **%Y** olacaktır. **%s** gibi yerleşik dönüşüm belirteçlerini de tanımlayabilirsiniz ama **#** gibi im karakterlerini **l** gibi tür değiştiricileri dönüşüm karakteri olarak tanımlayamazsınız. İşlevi bu karakterlerden biri ile çağırırsanız hiçbir etkisi olmaz. Küçük harfleri dönüşüm karakteri olarak tanımlamasanız iyi olur, çünkü ISO C standardı, standardın gelecekteki iyileştirmelerinde başka küçük harflerinde standarda dahil edebileceği konusunda uyarıyor.

kotarıcı–işlev, bir şablon dizgesinde tanımladığınız karakterlerlerden birine rastlarsa **printf** ve arkadaşları tarafından çağrılacak işlevdir. İşleve argüman olarak atanan böyle bir işlevin nasıl tanımlanacağı *Kotarıcı İşlevin Tanımlanması* (sayfa: 274) bölümünde açıklanmıştır. Bu argüman bir boş gösterici belirtirseniz, *belirteç* için atanan kotarıcı işlevi kaldırılacaktır.

argtürleri–işlevi ise, bu dönüşüm belirteci bir şablon dizgesinde kullanıldığında **parse_printf_format** işlevi tarafından çağrılır. Bu konu hakkında daha ayrıntılı bilgi almak için *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269) bölümüne bakınız.

Dikkat

GNU C kütüphanesinin 2.0 öncesi sürümlerinde *argtürleri–işlevi* işlevinin **parse_printf_format** işlevi çağrılmadıkça tanımlanması gerekmiyordu. Bu şimdi değişti. Artık, dönüşüm belirteci şablon dizgesinde kullanılmışsa her **printf** çağrısında bu işlev çağrılmaktadır.

İşlev başarı durumunda **0** ile döner. Hata oluşması halinde (*belirteç* aralık dışındaysa) **-1** ile döner.

Standart dönüşüm belirteçlerini de yeniden tanımlayabilirseniz de, karışıklığa yol açma olasılığından dolayı iyi bir fikir olmayacaktır. Başkaları tarafından yazılış kütüphane işlevleri bunu yaparsanız hata verebilir.

13.2. Dönüşüm Belirteci Seçenekleri

Bir dönüşüm belirteci olarak **%A** tanımladınız diyelim, bir şablon örneğin, **%+23A** veya **%-#A** içeriyorsa ne olacak? Dönüşüm belirteçleri ile birlikte verilen seçenekleri tanımlayacağınız bir yapı olabilir.

Hem *kotarıcı–işlev* hem de *argtürleri–işlevi* bir dönüşüm belirtecinde görülen seçenekler hakkında bilgi içeren bir **struct printf_info** yapısına gösterici alır. Bu veri türü printf.h başlık dosyasında bildirilmiştir.

```
struct printf_info                                                          veri türü
```

Bu yapı bir şablon dizgesindeki dönüşüm belirteçleri için kullanılan argüman türleri ve sayıları hakkında bilgi veren ve bu belirteçleri çıktılayan işlevlere bilgi aktarmak için kullanılır. Yapı üyeleri şunlardır:

`int` **prec**
: Belirtilen hassasiyettir. Hassasiyet belirtilmemişse değeri **-1** dir. Hassasiyet ***** olarak verilmişse, kotarıcı işleve aktarılan **printf_info** yapısı argüman listesinden alınan değeri içerir. Ancak argüman türleri ve sayısı için bilgi veren işleve aktarılan yapı, değer bilinmediğinden bir **INT_MIN** değeri içerir.

`int` **width**
: Belirtilen en küçük alan genişliğidir. **0** değeri bir genişlik belirtilmediği anlamına gelir. Alan genişliği ***** olarak verilmişse, kotarıcı işleve aktarılan **printf_info** yapısı argüman listesinden alınan değeri içerir. Ancak argüman türleri ve sayısı için bilgi veren işleve aktarılan yapı, değer bilinmediğinden bir **INT_MIN** değeri içerir.

`wchar_t` **spec**
: Belirtilen dönüşüm belirteci karakteridir. Bunun yapı içinde bulunmasının sebebi çok sayıda karaktere aynı kotarıcı işlevi kaydedebilmenizi sağlamaktır, ancak bu olmasa bile kotarıcı işlevi çağırdığınızda onları ayıracak bir yol vardır.

`unsigned int` **is_long_double**
: **L**, **ll** veya **q** tür değiştiricileri belirtilmişse değeri mantıksal değer olarak doğrudur. Gerçek sayı dönüşümlerinde bu **long double** iken tamsayı dönüşümlerinde bir **long long int** dir.

`unsigned int` **is_char**
: **hh** tür değiştiricisi belirtilmişse değeri mantıksal değer olarak doğrudur.

`unsigned int` **is_short**
: **h** tür değiştiricisi belirtilmişse değeri mantıksal değer olarak doğrudur.

`unsigned int` **is_long**
: **l** tür değiştiricisi belirtilmişse değeri mantıksal değer olarak doğrudur.

`unsigned int` **alt**
: **#** imi belirtilmişse değeri mantıksal değer olarak doğrudur.

`unsigned int` **space**
: **(boşluk karakteri)** imi belirtilmişse değeri mantıksal değer olarak doğrudur.

`unsigned int` **left**
: **-** imi belirtilmişse değeri mantıksal değer olarak doğrudur.

`unsigned int` **showsign**
: **+** imi belirtilmişse değeri mantıksal değer olarak doğrudur.

`unsigned int` **group**
: **'** imi belirtilmişse değeri mantıksal değer olarak doğrudur.

`unsigned int` **extra**
: Bu üyenin değerini bağlama özel bir anlamı vardır. Yapı **printf** işlevi tarafından kullanılmışsa değeri **0** dır. Kullanıcı tarafından tanımlanmış bir işlev tarafından kullanıldığında ise herhangi bir değer içerebilir.

`unsigned int` **wide**
: Akım geniş yönlenimli ise bu üyenin değeri **1** dir.

`wchar_t` **pad**

Çıktıda en küçük alan genişliğinde boş kalan yerlere yerleştirilecek karakterdir. Boş kalan yerler sıfırlarla doldurulursa bu üyenin değeri `'0'` dır, aksi takdirde `' '` tur.

13.3. Kotarıcı İşlevin Tanımlanması

Şimdi `register_printf_function` işlevine aktarılacak kotarıcı işlev ile argüman bilgileri işlevinin nasıl tanımlanacağına bakalım.

Uyumluluk Bilgisi:

Arayüz GNU libc 2.0 sürümünde değiştirilmiştir. Bundan önce üçüncü argümanın türü **`va_list *`** idi.

Kotarıcı işlevi bu prototipin benzeri olarak bildirebilirsiniz.:

```
int işlev–ismi (FILE                        *akım,
                const struct printf_info *bilgi,
                const void *const        *args)
```

akım argümanı, kotarıcı işlevin çıktıyı yazacağı akımdır.

bilgi argümanı, şablon dizgesindeki belirteç ile birlikte belirtilen çeşitli seçenekler hakkında bilgi içeren bir yapının göstericisidir. Bu veri yapısını kotarıcı işlevin içinde değiştirmemelisiniz. Bu veri yapısının açıklamaları için *Dönüşüm Belirteci Seçenekleri* (sayfa: 272) böümüne bakınız.

args argüman sayısının atanacağı göstericidir. Argüman sayısı, yazılımcı tarafından tanımlanan argüman bilgileri işlevi çağrılarak elde edilir.

Sizin kotarıcı işleviniz de tıpkı **`printf`** işlevi gibi değer döndürmelidir: Ya çıktılanan karakterlerin sayısı ile dönmeli ya da hata durumunda hatayı ifade eden negatif bir değer ile dönmelidir.

`printf_function` veri türü

Bu, kotarıcı işlevin türüdür.

Yazılımınızda **`parse_printf_format`** işlevini kullanacaksanız, **`register_printf_function`** ile tanımlayacağınız her dönüşüm belirteci için *argtürleri–işlevi* argümanı ile aktaracağınız işlevi de tanımlamalısınız.

Bu işlevleri buna benzer bir prototiple bildirebilirsiniz:

```
int işlev–ismi (const struct printf_info *bilgi,
                size_t                    n,
                int                      *argtürleri)
```

İşlevden dönen değer, dönüşümü umulan argümanların sayısı olmalıdır. İşlev ayrıca bu argümanların herbiri için tür bilgisini içeren en çok *n* elemanlı *argtürleri* dizisini de doldurmalıdır. Bu bilgi **`PA_`** makroları kullanılarak kodlanır. (Bu, **`parse_printf_format`** işlevinin kullandığı çağrı ile aynı yapıdaysa uyarılacasınız.)

`printf_arginfo_function` veri türü

Dönüşüm belirteçleri tarafından kullanılan argümanların türleri, seçenekleri ve sayıları hakkında bilgi döndüren işlevin türüdür.

13.4. `printf` Genişletme Örneği

Burada bir `printf` kotarıcı işlevinin nasıl tanımlandığı örneklenmiştir. Bu yazılım **`Kitap`** isimli bir veri yapısı ve **`Kitap *`** argümanları hakkında bilgi basacak **`%K`** dönüşüm belirtecini tanımlar. **`%K`** dönüşüm belirteci en küçük alan genişliği ile sola yanaştırma imini desteklemekte diğer herşeyi yoksaymaktadır.

```
#include <stdio.h>
#include <stdlib.h>
#include <printf.h>

typedef struct
{
  char *isim;
}
Kitap;

int
kitap_bas (FILE *akim,
              const struct printf_info *bilgi,
              const void *const *args)
{
  const Kitap *k;
  char *tampon;
  int uzunluk;

  /* Bir dizgeye çıktılanacak biçim. */
  k = *((const Kitap **) (args[0]));
  uzunluk = asprintf (&tampon, "<Kitap %p: %s>", k, k->isim);
  if (uzunluk == -1)
    return -1;

  /* En küçük alan genişliğini doldurup akıma basalım. */
  uzunluk = fprintf (akim, "%*s",
                (info->left ? -info->width : info->width),
                tampon);

  /* Ortalığı temizle ve dön. */
  free (tampon);
  return uzunluk;
}


int
kitap_bas_argbilgi (const struct printf_info *bilgi, size_t n,
                      int *argturleri)
{
  /* Daima tek bir argüman alıyoruz ve bu yapı için bir gösterici
     oluyor.. */
  if (n > 0)
    argturleri[0] = PA_POINTER;
  return 1;
}


int
main (void)
{
  /* Basılacak Kitabı oluşturalım. */
  Kitap kitabim;
  kitabim.isim = "Kitabım";

  /* Kitap için kotarıcı işlevi kaydedelim. */
```

```
  register_printf_function ('K', kitap_bas, kitap_bas_argbilgi);

  /* Şimdi kitabimi basalım. */
  printf ("|%K|\n", &kitabim);
  printf ("|%35K|\n", &kitabim);
  printf ("|%-35K|\n", &kitabim);

  return 0;
}
```

Yazılımın çıktısı şöyle olacaktır:

```
|<Kitap 0xffeffb7c: Kitabım>|
|        <Kitap 0xffeffb7c: Kitabım>|
|<Kitap 0xffeffb7c: Kitabım>        |
```

13.5. Yerleşik Kotarıcı İşlevler

GNU libc ayrıca **printf** kotarıcı oluşumunun somut ve kullanışlı bir uygulamasını içerir. Gerçek sayıları özel bir yolla basmak üzere iki işlev vardır.

```
int printf_size(FILE                        *akım,                    işlev
                const struct printf_info    *bilgi,
                const void *const           *args)
```

Verilen bir gerçek sayıyı **%f** dönüşüm belirtecini kullanarak özel bir şekilde basar. Sayıyı 1000 den daha küçük bir sayı olarak ifade eden bir birim karakteri kullanılır. Bu birim karakteri bir bölene karşılıktır. Mümkün iki bölen vardır; biri 1000'in kuvvetleri diğeri 1024'ün kuvvetleri. Birim, küçük harf ise 1024'ün kuvvetleri, büyük harf ise 1000'in kuvvetleri kullanılır.

Birim bayt, kilobayt, megabayt, gigabayt, vs. karşılığı bir karakterdir. Tamamı tablo olarak:

Birim harfi	Çarpan	Birim ismi	Birim harfi	Çarpan
''	1		''	1
k	2^{10} (1024)	kilo	K	10^3 (1000)
m	2^{20}	mega	M	10^6
g	2^{30}	giga	G	10^9
t	2^{40}	tera	T	10^{12}
p	2^{50}	peta	P	10^{15}
e	2^{60}	egza	E	10^{18}
z	2^{70}	zeta	Z	10^{21}
y	2^{80}	yotta	Y	10^{24}

Öntanımlı hassasiyet 3 tür, örneğin 1024 sayısı için dönüşüm belirtimi küçük birim harfi ile **%.3fk** olarak yazılır ve bunun çıktısı **1.000k** olur.

register_printf_function işlevinin gereksinimlerinden dolayı argümanlar hakkında bilgi döndüren bir işlev daha üretmeliyiz.

```
int printf_size_info(const struct printf_info *akım,                  işlev
                     size_t                    n,
                     int                      *argtürleri)
```

Bu işlev şablon dizgesindeki belirteçlere karşılık olacak argümanların türleri hakkında bilgiyi *argtürleri* içinde döndürür. Bu işlev için tek argüman vardır.

Bu iki işlevi kullanırken aşağıdakine benzer bir çağrı ile kaydedilmelidir:

```
register_printf_function ('B', printf_size, printf_size_info);
```

Burada **'B'** belirteç karakteri bütük harf olduğundan sayıları 1000'in kuvveti olarak basacak bir işlevi kaydetmiş olduk. Buna ek olarak **'b'** harfini de kullanmak için bir çağrı daha yapılmalıdır:

```
register_printf_function ('b', printf_size, printf_size_info);
```

Böylece 1024'ün kuvvetini de basabileceğiz. Burada dikkat ettiyseniz iki işlevde dönüşüm belirteçleri farklıdır. **printf_size** işlevi sadece büyük ve küçük harfler arasındaki farkı bilir.

'B' ve **'b'** kullanımı hiç de tesadüf değildir. Hatta bunları biçim belireci olarak kullanan bazı sistemler de olduğundan bu harfleri kullanmanız önerilir.

14. Biçimli Girdi

Bu kısımda açıklanan işlevler (**scanf** ve ilgili işlevler) biçimli çıktı oluşumlarına benzer olarak biçimli girdi için kullanılır. Bu işlevler bir ***biçim dizgesi*** veya bir ***şablon dizgesi***nin denetimi altında değer okumak için bir mekanizma sağlar.

14.1. Biçimli Girdi Okumanın Temelleri

scanf çağrıları argümanları bir şablon dizgenin denetimi altında okumasından dolayı yüzeysel olarak **printf** çağrılarına benzer. Şablon dizgedeki dönüşüm belirtimlerinin sözdizimleri **printf** işlevininkilere çok benzese de, şablonun yorumlanması sabit alanlı biçimlemeden ziyade daha bir serbest biçimli ve basit kalıp eşleştirme yönündedir. Örneğin, çoğu **scanf** dönüşümü dosyadaki boş alanları (boşluk, sekme, satırsonu) atlar ve çıktı dönüşümlerindekinin aksine sayısal girdi dönüşümleri için hassasiyet diye bir kavrama sahip değildir. Ekseriyetle, şablondaki boşluk olmayan karakterlerin girdi akımındaki karakterlerle eşleşeceği umulur, ancak bir eşleşmenin bulunamaması akım üzerindeki bir girdi hatası ile karıştırılmaz.

scanf ile **printf** arasındaki diğer bir farklı alan, **scanf** işlevinin isteğe bağlı argümanlarının doğrudan değer olarak değil göstericiler sağlayarak alındığını unutmamanız gerektiğidir; okunan değerler göstericilerin gösterdiği nesnelerde saklanır. Deneyimli yazılımcılar bile bazan bunu unutur, eğer yazılımınız **scanf** ile ilgili olarak tuhaf hatalar veriyorsa bu özelliği için çifte kontrol yapmalısınız.

Bir ***eşleşme hatası*** oluştuğunda, **scanf** işlevi ilk eşleşmeyen karakteri akımdan okunacak sonraki karakter olarak bırakarak hemen döner. Normalde **scanf** işlevinden dönen değer atanmış değerlerin sayısıdır. Bu değere bakarak bir okunmamış karakter varsa eşleşme hatasının oluştuğu yeri bulabilirsiniz.

scanf işlevi genellikle, tabloların içeriklerini okumak gibi şeyler için kullanılır. Örneğin, buradaki işlev bir **double** dizisini ilklendirmek için **scanf** işlevini kullanmaktadır:

```
void
diziyioku (double *dizi, int n)
{
  int i;
  for (i = 0; i < n; i++)
    if (scanf (" %lf", &(dizi[i])) != 1)
      invalid_input_error ();
}
```

Biçimli girdi işlevleri, biçimli çıktı işlevleri kadar sık kullanılmazlar. Kısmen, onları düzgün olarak kullanmak biraz dikkat gerektirdiğindendir. Diğer bir sebep, bir eşleşme hatasından kurtulmanın zorluğudur.

Bir tek başına, sabit bir kalıpla eşleşmeyecek bir girdiyi okumaya çalışacaksanız, **scanf** kullanmaktansa, bir sözel tarayıcı üretmede Flex ya da bir çözümleyici üretmede Bison gibi bir araç kullanmak sizin için daha iyi olabilir. Bu araçlar hakkında bilgi almak için Bison.info ve Flex.info'ya bakabilirsiniz.

14.2. Girdi Dönüşüm Sözdizimi

Bir **scanf** sablon dizgesi, sıradan çok baytlı karakterler arasına serpiştirilmiş **%** ile başlayan dönüşüm belirtimleri içeren bir dizgedir.

Şablondaki boşluk karakterleri (**isspace** işlevinin tanıdıkları, bkz. *Karakterlerin Sınıflandırılması* (sayfa: 82)) girdi akımından boşluk karakterlerini okutur ve bunlar iptal edilir. Eşleşmesi istenen boşluk karakterleri ile okunacak boşluk karakterlerinin aynı karakterler olması gerekmez. Örneğin şablona'**,** ' yazarsanız bir virgül ve virgülün önünde ve/veya ardında isteğe bağlı boşluk karakterleri ile eşleşir.

Dönüşüm belirtimlerinin parçası olmayan tüm karakterler girdidekilerle aynen eşleşmelidir; bu eşleşme olmazsa bir eşleşme hatası oluşur.

Bir **scanf** şablon dizgesindeki dönüşüm belirtimlerinin genel şekli:

```
% imler genişlik tür dönüşüm
```

Ayrıntıya girersek, dönüşüm belirtimi bir **%** işaretini izleyen aşağıdaki parçalardan oluşur:

- İsteğe bağlı ***** *im karakteri*, belirtim için okunan metni yoksaymasını söyler. **scanf** bu imi kullanan bir belirtim bulduğunda,dönüşüm belirtiminin kalanı tarafından yönlendirildiği girdiyi okur, ama bu girdiyi iptal eder, bir gösterici kullanılmaz ve başarılı atamalar sayacı arttırılmaz.
- İsteğe bağlı **a** *im karakteri* (sadece dizge dönüşümlerinde geçerli) dizgeyi saklamak için yeterli uzulukta bir tampon ayrılmasını söyler. (Bu im bir GNU oluşumudur.) Bkz. *Dizge Dönüşümlerinde Özdevimli Ayırma* (sayfa: 283).
- İsteğe bağlı bir *en büyük alan genişliği*. Bir onluk tamsayıdır. En büyük genişlik aşıldığında ya da bir eşleşmeyen karaktere rastlandığında hangisi önce oluşursa girdi akımından karakterlerin okunması durdurulur. Çoğu dönüşümler okunan boşluk karakterlerini (bunlar açıkça belgelendirilmiş değildir) iptal eder ve bu iptal edilen karakterler en büyük alan genişliğinden sayılmaz. Dizge girdi dönüşümleri girdinin sonuna bir boş karakter ekler, bu karakter de en büyük alan genişliğine dahil edilmez.
- İsteğe bağlı bir *tür değiştirici karakter*. Örneğin, bir **int** türünden gösterici argümanı belirten **%d** dönüşüm belirteci ile **l** tür değiştiricisini tamsayı dönüşümü için belirterek göstericini türünü **long int** olarak değiştirebilirsiniz.
- Uygulanacak dönüşümü belirten bir karakter.

İzin verilen seçenekler ve onların yorumlanışı farklı dönüşüm belirteçleri hep aynı değildir. Bir seçeneğin kullanıldığı bir dönüşümdeki yorumlanışı ile ilgili bilgileri, o dönüşümün açıklamalarında bulabilirsiniz.

-Wformat seçeneği ile GNU C derleyicisi **scanf** ve ilgili işlevleri denetler. Biçim dizgesine bakarak doğru sayı ve türde argüman belirtilip belirtilmediğini denetler. Yazdığınız bir **scanf** tarzı biçim dizgesini denetlemek için GNU C sözdizimini derleyiciye söyleyecek bir sözdizimi de vardır. (GCC info'sundaki "İşlev Özniteliklerinin Bildirilmesi" [Declaring Attributes of Functions] bölümüne bakınız.)

14.3. Girdi Dönüşüm Belirteçlerinin Listesi

Aşağıda tüm farklı dönüşümler özetlenmiştir:

`%d`

Seçmeli olarak onluk tabanda yazılmış bir işaretli tamsayı ile eşleşir. Bkz. *Sayısal Girdi Dönüşümleri* (sayfa: 280).

`%i`

Bir tamsayı sabit için C dilinde tanımlı herhangi bir biçimdeki bir işaretli tamsayı ile seçmeli olarak eşleşir. Bkz. *Sayısal Girdi Dönüşümleri* (sayfa: 280).

`%o`

Sekizlik tabanda yazılmış bir işaretli tamsayı ile eşleşir. Bkz. *Sayısal Girdi Dönüşümleri* (sayfa: 280).

`%u`

Onluk tabanda yazılmış bir işaretsiz tamsayı ile eşleşir. Bkz. *Sayısal Girdi Dönüşümleri* (sayfa: 280).

`%x, %X`

Onaltılık tabanda yazılmış bir işaretsiz tamsayı ile eşleşir. Bkz. *Sayısal Girdi Dönüşümleri* (sayfa: 280).

`%e, %f, %g, %E, %G`

Bir gerçek sayı ile eşleşir. Bkz. *Sayısal Girdi Dönüşümleri* (sayfa: 280).

`%s`

Boşluk içermeyen bir dizge ile eşleşir. Bkz. *Dizgeler için Girdi Dönüşümleri* (sayfa: 281). **`l`** tür değiştiricisinin varlığı dizgenin geniş karakterli ya da çok baytlı dizge olarak ele alınmasını sağlar. **`%s`** bir geniş karakter işlevinde kullanılmışsa dizge çoklu **`wcrtomb`** çağrılarıyla çok baytlı dizgeye dönüştürülür. Bu, tamponda, okunan her karakter için **`MB_CUR_MAX`** baytlık yer ayrılması gerektiği anlamına gelir. **`%ls`** bir çok baytlı işlevde kullanıldığında ise sonuç yazılımcının sağladığı tampona yazılmadan önce çoklu **`mbrtowc`** çağrılarıyla geniş karakterlere dönüştürülür.

`%S`

Unix standardı ile uyumluluk için **`%ls`** yerine kullanılmak üzere desteklenmiştir.

`%[`

Belirtilmiş bir kümeye ait olan karakterlerin bir dizgesi ile eşleşir. Bkz. *Dizgeler için Girdi Dönüşümleri* (sayfa: 281). **`l`** tür değiştiricisinin varlığı dizgenin geniş karakterli ya da çok baytlı dizge olarak ele alınmasını sağlar. **`%[`** bir geniş karakter işlevinde kullanılmışsa dizge çoklu **`wcrtomb`** çağrılarıyla çok baytlı dizgeye dönüştürülür. Bu, tamponda, okunan her karakter için **`MB_CUR_MAX`** baytlık yer ayrılması gerektiği anlamına gelir. **`%l[`** bir çok baytlı işlevde kullanıldığında ise sonuç yazılımcının sağladığı tampona yazılmadan önce çoklu **`mbrtowc`** çağrılarıyla geniş karakterlere dönüştürülür.

`%c`

Bir ya da daha fazla karakterden oluşan bir dizge ile eşleşir. Okunacak karakterlerin sayısı dönüşüm belirtiminde belirtilmiş olan en büyük karakter genişliğindedir. Bkz. *Dizgeler için Girdi Dönüşümleri* (sayfa: 281).

`%c` bir geniş yönlenimli akım işlevinde kullanılırsa okunan bir geniş karakter karşılığı olan çokbaytlı karaktere dönüştürüldükten sonra saklanır. Bu dönüşüm birden fazla karakter üretebilir, bundan dolayı tamponda her karakter için **`MB_CUR_MAX`** baytlık yer sağlanmalıdır. **`%lc`** bir çok baytlı işlevde kullanılırsa, bir çokbaytlı dizi (bayt değil) olarak ele alınır ve sonuç **`mbrtowc`** çağrılarıyla dönüştürülür.

`%C`

Unix standardı ile uyumluluk için **`%lc`** yerine kullanılmak üzere desteklenmiştir.

`%p`

`printf` için **`%p`** çıktı dönüşüm belirteci tarafından kullanılan biçimle aynı gerçekleme tanımlı biçimde bir gösterici değeri ile eşleşir. Bkz. *Diğer Girdi Dönüşümleri* (sayfa: 283).

`%n`

Bu belirteç herhangi bir karakter okumaz; çağrı tarafından o ana kadar okunan karakterlerin sayısını kaydeder. Bkz. *Diğer Girdi Dönüşümleri* (sayfa: 283).

`%%`

Girdi akımındaki bir `%` karakteri ile eşleşir. Karşılık olarak bir arüman belirtilmez. Bkz. *Diğer Girdi Dönüşümleri* (sayfa: 283).

Bir dönüşüm belirtiminin sözdizimi geçersizse, davranış tanımlanmamıştır. yani bunu yapmamaya çalışın. Şablon dizgesinde belirtilenden daha az sayıda artbileşen argüman varsa ya da artbileşen argümanların türleri ile dönüşüm belirtimleri uyumsuzsa sonucun ne olacağı belli olmaz. Şablon dizgesinde belirtilenden daha çok sayıda artbileşen argüman varsa fazla argümanlar basitçe yoksayılır.

14.4. Sayısal Girdi Dönüşümleri

Bu bölümde, sayısal değerlerin okunması sırasındaki **`scanf`** dönüşümleri açıklanmıştır.

`%d` dönüşümü seçmeli olarak onluk tabanda bir işaretli tamsayı ile eşleşir. Sözdizimi **`strtol`** işlevinin *taban* argümanının **`10`** değerli çağrısıyla aynı şekilde tanınır. (Bkz. *Tamsayıların Çözümlenmesi* (sayfa: 528).)

`%i` dönüşümü bir tamsayı sabit için C dilinde tanımlı herhangi bir biçimdeki bir işaretli tamsayı ile seçmeli olarak eşleşir. Sözdizimi **`strtol`** işlevinin *taban* argümanının **`0`** değerli çağrısıyla aynı şekilde tanınır. (Bkz. *Tamsayıların Çözümlenmesi* (sayfa: 528).) (Bu sözdizimindeki tamsayıları **`printf`** işlevini **`#`** im karakteri ve **`%x`**, **`%o`** veya **`%d`** dönüşümlerinden biri birlikte kullanarak çıktlayabilirsiniz. Bkz. *Tamsayı Dönüşümleri* (sayfa: 258).)

Örneğin, **`10`**, **`0xa`**, **`012`** dizgelerinin her birini **`%i`** dönüşümü altında birer tamsayı olarak okumalısınız. Bu dizgelerin her biri onluk tabanda **`10`** sayısıdır.

`%o`, **`%u`** ve **`%x`** dönüşümleri sırayla sekizlik, onluk ve onaltılık tabandaki işaretsiz tamsayılarla eşleşir. Sözdizimi **`strtol`** işlevinin *taban* argümanının sırasıyla **`8`**, **`10`** veya **`16`** değerli çağrısıyla aynı şekilde tanınır. (Bkz. *Tamsayıların Çözümlenmesi* (sayfa: 528).)

`%X` ve **`%x`** dönüşümleri eşanlamlıdır. Her ikisi de rakam olarak büyük ya da küçük harfleri tanır.

Öntanımlı tür, **`%d`** ve **`%i`** dönüşümleri için **`int *`**, diğer tamsayı dönüşümleri için **`unsigned int *`** dir. Tamsayıların türleri için aşağıdaki tür değiştiricileri kullanabilirsiniz:

`hh`

Argümanın **`signed char *`** veya **`unsigned char *`** türünde olduğunu belirtir.

Bu değiştirici ISO C99'da tanımlanmıştır.

`h`

Argümanın **`short int *`** veya **`unsigned short int *`** türünde olduğunu belirtir.

`j`

Argümanın **`intmax_t *`** veya **`uintmax_t *`** türünde olduğunu belirtir.

Bu değiştirici ISO C99'da tanımlanmıştır.

`l`

Argümanın **`long int *`** veya **`unsigned long int *`** türünde olduğunu belirtir. İki **`l`** karakteri aşağıda açıklanan **`L`** karakteri gibidir.

Bu karakter **`%c`** veya **`%s`** dönüşümü ile birlikte kullanıldığında argümanın sırasıyla bir geniş karakter veya bir geniş karakterli dizgeye bir gösterici olduğunu belirtir. **`l`** değiştiricisinin bu kullanımı ISO C90 1. düzeltmesinde tanımlanmıştır.

`ll, L, q`
: Argümanın **`long long int *`** veya **`unsigned long long int *`** türünde olduğunu belirtir. (Bu tür GNU C derleyicisi tarafından desteklenen bir oluşumdur. Fazla uzun tamsayıları desteklemeyen sistemlerde bu değiştirici **`long int`** olarak değerlendirilir.)

  `q` değiştiricisi 4.4 BSD de bulunur. Bir **`long long int`** bazan "quad" **`int`** olarak da isimlendirilir.

`t`
: Argümanın **`ptrdiff_t *`** türünde olduğunu belirtir.

  Bu değiştirici ISO C99'da tanımlanmıştır.

`z`
: Argümanın **`size_t *`** türünde olduğunu belirtir.

  Bu değiştirici ISO C99'da tanımlanmıştır.

`%e`, **`%f`**, **`%g`**, **`%E`** ve **`%G`** girdi dönüşümlerinin tümü birbirinin yerine kullanılabilir. Bunların tümü **`strtod`** işlevindeki sözdizimi ile aynı sözdiziminde, seçimli olarak bir gerçek sayı ile ile eşleşir (Bkz. *Gerçek Sayıların Çözümlenmesi* (sayfa: 533)).

Gerçek sayı girdi dönüşümleri için öntanımlı argüman türü **`float *`** dır. (Çıktı dönüşümlerinde öntanımlı tür **`double`** dır ve öntanımlı argüman terfileri çerçevesinde bir **`float`** argüman **`double`** türe terfi ettirilir. Ama girdi dönüşümlerinde bu terfi uygulanmaz.) Gerçek sayıların türleri için aşağıdaki tür değiştiricileri kullanabilirsiniz:

`l`
: Argümanın **`double *`** türünde olduğunu belirtir.

`L`
: Argümanın **`long double *`** türünde olduğunu belirtir.

Yukarıdaki sayı çözümleme biçimlerinin tümü için isteğe bağlı bir ek **`'`** imi vardır. Bu im kullanıldığında **`scanf`** işlevi girdi dizgesinin o an geçerli yerelin sayı gruplama kurallarına uygun olacağını umar (Bkz. *Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169)).

`"C"` veya **`"POSIX"`** yeereli geçerli yerelse bir fark olmaz. Fakat diğer yerellerde bu dizge yerele özgü biçime uygun verilmiş olmalıdır. Aksi takdirde doğru biçimli en uzun önek işlenir.

14.5. Dizgeler için Girdi Dönüşümleri

Bu bölümde dizge ve karakterlerin okunması için **`%s`**, **`%S`**, **`%[`**, **`%c`** ve **`%C`** girdi dönüşüm belirteçleri açıklanmıştır.

Bu dönüşümlerden girdilerin alınması için iki seçeneğiniz vardır:

- Onu saklamak için bir tampon sağlayın. Bu öntanımlıdır. Bu tampona **`char *`** veya **`wchar_t *`** türünde bir argüman sağlayın (ikincisi için **`l`** değiştiricisi olmalı).

  Uyarı

  Sağlam bir yazılım için girdi (sonlandırıcı boş karakter dahil), sağladığınız tamponun boyunu aşmamalıdır. Genelde, bunu yapmanın tek yolu, en büyük alan genişliğini tampon boyunun bir eksiği olarak vermektir. *Bir tampon oluşturuyorsanız taşmalardan kaçınmak için uzunluğunu daima en büyük alan genişliğine eşit uzunlukta seçmelisiniz.*

- Ne kadar büyüklükte bir tampon gerektiğini **a** im karakterini belirterek **scanf** işlevine sorun. Bu bir GNU oluşumudur. Tampon adresi için **char **** türünde bir argüman belirtmelisiniz. Bkz. *Dizge Dönüşümlerinde Özdevimli Ayırma* (sayfa: 283).

%c dönüşümü en basitidir: daima sabit sayıda karakterle eşleşir. En büyük alan genişliği kaç karakter okunacağını söyler; en büyük alan genişliğini belirtmezseniz öntanımlı olarak 1 değeri kullanılır. Bu dönüşüm okuduğu metnin sonuna bir sonlandırıcı boş karakter eklemez. Ayrıca metnin içindeki boşluk karakterlerini atlar. Özellikle sonraki n karakteri okur, bu kadar karakter bulamazsa başarısız olur. **%c** dönüşümü daima sabit uzunlukta okuma yaptığından taşmadan kaçınmak için tamponu yeterli uzunlukta yapmalısınız.

%lc veya **%C** dönüşümleri akımın, harici bayt akımından açıldığı sırada saptanan dönüşüm kullanılarak çevrilen geniş karakterlerin saklanmasını sağlar. Ortamdan okunan baytların sayısı **MB_CUR_LEN** * n ile sınırlıdır ve çıktı dizgesinde saklanan en çok n geniş karakter alınır.

%s dönüşümü boşluk karakterleri olmayan karakterlerden oluşan bir dizge ile eşleşir. Dahili boşlukları atlar ve iptal eder, fakat biraz okuma yaptıktan sonra çok fazla boşluk karakterine rastlarsa durur. Okuduğu metnin sonuna bir boş karakter ekler.

Örneğin, okunacak girdi,

```
hello, world
```

ise, **%10c** dönüşümü **" hello, wo"** üretir. Aynı girdi için **%10s** dönüşümü kullanılırsa, **"hello,"** üretilir.

Uyarı

%s için bir alan genişliği belirtirseniz, okunan karakter sayısı boşluk karakterine rastlanan yer ile sınırlıdır. Bu hemen hemen kaçınılmaz olarak geçersiz bir girdinin yazılımınızın çökmesine sebep olacağı anlamına gelir ki bu bir yazılım hatasıdır.

%ls ve **%S** dönüşümleri **%s** gibi kotarılır, bir farkla, dış bayt dizisi, kendi karakter kodlaması ile geniş karakterlere akım ile ilişkili dönüşüm kullanılarak çevrilir. Belirteç ile birlikte genişlik belirtilirse, bunlar geniş karakterleri öçtüğünden, akımdan kaç bayt okunacağı doğrudan doğruya saptanmaz. Fakat bir üst sınır, genişlik değeri ile **MB_CUR_MAX** çarpılarak hesaplanabilir.

Belli kriterlere göre sizin seçiminize bağlı olarak karakterlerin okunmasını sağlamak isterseniz **%[** dönüşümünü kullanın. **[** ve **]** ayraçlarının arasını düzenli ifadelerdeki sözdizimini kullanarak yazabilirsiniz. Özel durumlar olarak:

- **]** karakteri ifadenin ilk karakteri olarak belirtilebilir.
- Bir gömülü **–** karakteri (ifadenin ilk veya son karakteri olamaz) bir karakter aralığını belirtmek için kullanılabilir.
- Bir **^** karakteri **[** ayracından sonra kullanılırsa, girdi karakterleri burada listelenen karakterlerin dışındakilerdir.

%[dönüşümü dahili boşluk karakterlerini atlamaz.

Aşağıda bazı **%[** dönüşüm örnekleri ve anlamları vardır:

`%25[1234567890]`
: 25 haneye kadar bir sayı ile eşleşir.

`%25[][]`
: 25 köşeli ayraca kadar bir dizge ile eşleşir.

`%25[^ \f\n\r\t\v]`
: Hiçbir boşluk karakteri içermeyen 25 karaktere kadar bir dizge ile eşleşir. Bu **%s** den tamamen farklıdır, çünkü girdi, boşluk karakterlerinden biri ile başlarsa bu **%[** dönüşümü bir eşleşme hatası bildirir ama **%s** dönüşümü onu basitçe iptal eder.

`%25[a-z]`
: 25 karaktere kadar küçük harflerle eşleşir.

%c ve **%s** gibi **%[** belirteci de **l** tür değiştiricisi varsa geniş karakterleri üretebilir. Yukarıda bu konu ile ilgili bahsedilen herşey **%l[** için de geçerlidir.

Bir hatırlatma daha

Bir en büyük genişlik belirtmezseniz ya da **a** imini kullanmazsanız **%s** ve **%[** belirteçleri *tehlikelidir*, çünkü girdi çok uzun olursa tampon taşar. Tamponun ne kadar uzunlukta olduğunun önemi yoktur, bir kullanıcı onu da taşıracak bir girdi yapabilir. İyi geliştirilmiş bir yazılım geçersiz bir girdiyi kapsamlı bir hata iletisi ile bildirir, çökerek değil.

14.6. Dizge Dönüşümlerinde Özdevimli Ayırma

Bir GNU oluşumu biçimli girdi olarak bir uzunluk sınırı olmaksızın bir dizgeyi güvenli olarak okur. Bu özelliği kullandığınızda bir tampon ayırmanız gerekmez. **scanf** veriyi tutacak kadar büyüklükte tamponu kendi ayırır ve size onun adresini verir. Bu özelliği kullanmak için bir im karakteri olarak **a** karakterini **%as** olarak ya da **%a[0-9a-z]** olarak yazın.

Girdinin saklanacağı tamponun adresini tutacak gösterici argümanını **char **** türünde sağlamalısınız. **scanf** işlevi bir tampon ayırır ve onun adresini bu argümanın gösterdiği yere kaydeder. Tampona ihtiyacınız kalmadığında **free** ile serbest bırakmalısınız.

Bu örnekte **%[…]** belirteci **a** imi ile birlikte kullanılarak *değişken= değer* çifti halinde bir "değişken ataması" okunmaktadır.

```
{
  char *degisken, *deger;

  if (2 > scanf ("%a[a-zA-Z0-9] = %a[^\n]\n",
                 &degisken, &deger))
  {
    invalid_input_error ();
    return 0;
  }

  …
}
```

14.7. Diğer Girdi Dönüşümleri

Bu bölümde çeşitli girdi dönüşümlerine yer verilmiştir.

%p belrteci bir gösterici değeri okumakta kullanılır. **printf** (*Diğer Çıktı Dönüşümleri* (sayfa: 262)) için kullanılan **%p** çıktı dönüşüm belirteci ile aynı sözdizimini tanır; şöyleki, **%x** belirtecinin yaptığı gibi bir onaltılık sayı kabul eder. Karşılığı olan argüman **void **** türünde olmalıdır; yani göstericide bir yerin adresi saklanır.

Eğer değer okunduğu yazılımın icrası sırasında özgün olarak yazılmamışsa sonuçlanan gösterici değerinin geçerli olacağı garanti edilmez.

%n belirteci işlevin çağrısı sırasında o ana kadar okunan karakterlerin sayısını üretir. Belirtece kaşılık olan argüman **int *** türünde olmalıdır. Bu dönüşüm belirteci, **printf** için kullanılan **%n** ile aynı şekilde çalışır. *Diğer Çıktı Dönüşümleri* (sayfa: 262) bölümündeki örneğe bakınız.

%n dönüşümü sadece başarılı eşleşmeler veya bastırılmış atamalı dönüşümleri saptamak için bir mekanizmadır. **%n**'den önce bir eşleşme hatası oluşursa, **scanf**, **%n**'i işlemeden döndüğünden, argümanına bir değer atanmaz. **scanf** işlevini çağırmadan önce argüman yuvasına **-1** değerini yerleştirirseniz, çağrıdan sonra da bu değer hala duruyorsa, bu, **%n** işlenmeden önce bir hata oluştuğunu gösterir.

Son olarak, **%%** belirteci akımdaki bir **%** işareti ile eşleşir, bu belirteç için bir argüman kullanılmaz. Bu belirteç ile birlikte bir im, alan genişliği ya da tür değiştirici belirtilmesine izin verilmez.

14.8. Biçimli Girdi İşlevleri

Bu bölümde biçimli girdi uygulamak için kullanılan işlevler açıklanmıştır. Bu işlevlerin prototipleri `stdio.h` başlık dosyasında bildirilmiştir.

```
int scanf(const char *şablon, …)                                    işlev
```

scanf işlevi *şablon* biçim dizgesinin denetimi altında biçimli girdiyi standart girdiden okur. İsteğe bağlı argümanlar sonuçlanan değerlerin alındığı yerlere göstericilerdir.

Dönen değer normalde başarılı atamaların sayısıdır. Herhangi bir eşleşme bulunmadan önce bir dosya sonu durumu saptanırsa, şablondaki tek karakterler ve boşluk karakterleri içerilerek **EOF** ile döner.

```
int wscanf(const wchar_t *şablon, …)                                 işlev
```

wscanf işlevi *şablon* biçim dizgesinin denetimi altında biçimli girdiyi standart girdiden okur. İsteğe bağlı argümanlar sonuçlanan değerlerin alındığı yerlere göstericilerdir.

Dönen değer normalde başarılı atamaların sayısıdır. Herhangi bir eşleşme bulunmadan önce bir dosya sonu durumu saptanırsa, şablondaki tek karakterler ve boşluk karakterleri içerilerek **WEOF** ile döner.

```
int fscanf(FILE        *akım,                                        işlev
           const char *şablon,
           …)
```

Bu işlev, girdinin standart girdi yerine *akım* akımından okunması dışında **scanf** gibidir.

```
int fwscanf(FILE           *akım,                                    işlev
            const wchar_t *şablon,
            …)
```

Bu işlev, girdinin standart girdi yerine *akım* akımından okunması dışında **wscanf** gibidir.

```
int sscanf(const char *s,                                            işlev
           const char *şablon,
           …)
```

Bu işlev, girdinin standart girdi yerine boş karakter sonlandırmalı *s* dizgesinden alınması dışında **scanf** gibidir. Dizge sonunun aşılması bir dosya sonu durumu olarak ele alınır.

Bu işlevin davranışı, birbirini kapsayan nesneler arasında kopyalama yapılırsa, örneğin, **%s**, **%S** veya **%[** belirtecinin denetimi altında okunacak bir dizgeyi alacak argüman olarak *s* dizgesi verilmişse, tanımsızdır.

```
int swscanf(const wchar_t *ws,                                       işlev
            const char    *şablon,
            …)
```

Bu işlev, girdinin standart girdi yerine boş karakter sonlandırmalı *s* dizgesinden alınması dışında **wscanf** gibidir. Dizge sonunun aşılması bir dosya sonu durumu olarak ele alınır.

Bu işlevin davranışı, birbirini kapsayan nesneler arasında kopyalama yapılırsa, örneğin, **%s**, **%S** veya **%[** belirtecinin denetimi altında okunacak bir dizgeyi alacak argüman olarak *ws* dizgesi verilmişse, tanımsızdır.

14.9. Değişkin Girdi İşlevleri

Yerleşik biçimli çıktı işlevleri olarak **scanf** oluşumunu kullanarak kendi değişkin **scanf** benzeri işlevlerinizi tanımlayabilmeniz için **vscanf** işlevi ve arkadaşları oluşturulmuştur. Bu işlevler **vprintf** serisi biçimli çıktı işlevlerini andırır. Nasıl kulanıldıkları ile ilgili önemli bilgileri *Değişkin Çıktı İşlevleri* (sayfa: 266) bölümünde bulabilirsiniz.

Taşınabilirlik Bilgisi

Bu bölümdeki işlevler ISO C99'da tanımlanmıştı ve daha önce GNU oluşumları olarak vardı.

```
int vscanf(const char *şablon,                                          işlev
           va_list     arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine **va_list** türündeki *arglist–gstr* argüman listesi göstericisinden alması dışında **scanf** ile aynıdır *Değişkin İşlevler* (sayfa: 814)).

```
int vwscanf(const wchar_t *şablon,                                      işlev
            va_list        arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine **va_list** türündeki *arglist–gstr* argüman listesi göstericisinden alması dışında **wscanf** ile aynıdır *Değişkin İşlevler* (sayfa: 814)).

```
int vfscanf(FILE       *akım,                                           işlev
            const char *şablon,
            va_list     arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine **va_list** türündeki *arglist–gstr* argüman listesi göstericisinden alması dışında **fscanf** ile aynıdır.

```
int vfwscanf(FILE          *akım,                                       işlev
             const wchar_t *şablon,
             va_list        arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine **va_list** türündeki *arglist–gstr* argüman listesi göstericisinden alması dışında **fwscanf** ile aynıdır.

```
int vsscanf(const char *s,                                              işlev
            const char *şablon,
            va_list     arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine **va_list** türündeki *arglist–gstr* argüman listesi göstericisinden alması dışında **sscanf** ile aynıdır.

```
int vswscanf(const wchar_t *s,                                          işlev
             const wchar_t *şablon,
             va_list        arglist–gstr)
```

Bu işlev değişken sayıdaki argümanlarını doğrudan almak yerine **va_list** türündeki *arglist–gstr* argüman listesi göstericisinden alması dışında **swscanf** ile aynıdır.

GNU C'de **scanf** tarzı biçim dizgesi kullanan bir işlevi derleyiciye bildirebileceğiniz özel bir yapı vardır. Bu yapı kullanıldığında işlevin her çağrısı için kullanılan argümanların türleri ve sayısı denetlenir ve biçim dizgesiyle eşleşmeyenlar için sizi uyarır. İşlev özniteliklerinin bildirilmesi ile ilgili ayrıntılı bilgi edinmek için GCC kılavuzunun (info) "Declaring Attributes of Functions" (İşlevlerin Özniteliklerinin Bildirilmesi") bölümüne bakınız.

15. Dosya Sonu ve Hatalar

Bu kısımda açıklanan işlevlerin bir çoğu işlemin başarısızlıkla tamamlandığını belirten **EOF** makrosunun değerini döndürürler. **EOF** hem dosya sonunu hem de bir takım hataların olduğunu belirttiğinden, dosya sonu için **feof**, hatalar için de **ferror** işlevleri sağlanmıştır. Bunları kullanarak dosya sonu ile ilgili hataları ayrı ayrı elde edebilirsiniz. Bu işlevler akım nesnesinin dahili durumunun bir parçası olan göstergelere bakarlar, bu göstergeler akım üzerindeki önceki G/Ç işlemleri tarafından oluşturulan durumu gösterirler.

`int` **EOF** makro

Bu makro dosya sonu veya bazı hata durumlarını gösteren ve dar yönlenimli akım işlevleri tarafından döndürülen bir tamsayı değerdir. Diğer kütüphanelerde değeri herhangi bir negatif değer olabilirse de GNU kütüphanesinde değeri **-1** dir.

Bu sembol `stdio.h` başlık dosyasında bildirilmiştir.

`int` **WEOF** makro

Bu makro dosya sonu veya bazı hata durumlarını gösteren ve geniş yönlenimli akım işlevleri tarafından döndürülen bir tamsayı değerdir. Diğer kütüphanelerde değeri herhangi bir negatif değer olabilirse de GNU kütüphanesinde değeri **-1** dir.

Bu sembol `wchar.h` başlık dosyasında bildirilmiştir.

`int` **feof**`(FILE *`*akım*`)` işlev

feof işlevi, *akım* akımında sadece ve sadece dosya sonu göstergesi etkin ise sıfırdan farklı bir değerle döner.

Bu sembol `stdio.h` başlık dosyasında bildirilmiştir.

`int` **feof_unlocked**`(FILE *`*akım*`)` işlev

feof_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **feof** işlevi ile aynıdır.

Bu işlev bir GNU oluşumudur.

Bu sembol `stdio.h` başlık dosyasında bildirilmiştir.

`int` **ferror**`(FILE *`*akım*`)` işlev

ferror işlevi, *akım* akımında sadece ve sadece hata göstergesi etkin ise sıfırdan farklı bir değerle döner. Hata göstergesi, akımda bir önceki işlem sırasında bir hatanın oluştuğunu gösterir.

Bu sembol `stdio.h` başlık dosyasında bildirilmiştir.

`int` **ferror_unlocked**`(FILE *`*akım*`)` işlev

ferror_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **ferror** işlevi ile aynıdır.

Bu işlev bir GNU oluşumudur.

Bu sembol `stdio.h` başlık dosyasında bildirilmiştir.

Akım ile ilşkili hata göstergesi ayarlarına ek olarak, akımlar üzerinde işlem yapan işlevler, dosya tanıtıcılar üzerinde düşük seviyeli işlemler yapan eşdeğerleri ile aynı şekilde **errno** değerini ayarlarlar. Örneğin, **fputc**, **printf** ve **fflush** gibi bir akıma çıktılama yapan tüm işlevler *yazma* üzerine gerçeklendiklerinden bunlar için *yazma* ile ilgili **errno** hataları anlamlı olmaktadır. Dosya tanıtıcı seviyesi G/Ç işlemleri ile ilgili daha fazla bilgi edinmek için *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305) kısmına bakınız.

16. Hatalardan Kurtulma

Bir hatayı ya da dosya sonu durumunu **clearerr** işlevi ile doğrudan temizleyebilirsiniz.

`void` **clearerr**`(FILE *`*akım*`)` işlev

Bu işlev *akım* akımı ile ilgili dosya sonu ve hata göstergelerini temizler.

Ayrıca dosya konumlama işlevleri de (*Dosyalarda Konumlama* (sayfa: 288)) akım ile ilgili dosya sonu göstergesini temizler.

`void` **clearerr_unlocked**`(FILE *`*akım*`)` işlev

clearerr_unlocked işlevi akımı dolaylı olarak kilitlememesi dışında **clearerr** işlevi ile aynıdır.

Bu işlev bir GNU oluşumudur.

Bilgi

Hata göstergesinin temizlenmesi ve başarısız akım işlevinin yinelenmesi *doğru olmaz*. Başarısız bir yama işleminden sonra, tamponda olup da dosyaya gönderilmesi gereken verinin bir kısmı iptal olabilir. Bir kere yinelense bile verinin kaybına ya da tekrarına sebep olabilir.

Başarısız bir okuma ise ikinci deneme için dosya göstericisini ilgisiz bir konumda bırakabilir. Her iki durumda da işlemi yinelemeden önce bilinen konuma ilerlemeniz gerekir.

Hataların çoğu kurtarılabilir değildir; ikinci bir deneme daima aynı şekilde başarısız olur. En iyisi karmaşık hata kurtarma kodları yazmak yerine, hatayı kullanıcıya raporlayarak işlemi kesmektir.

Bir önemli hata durumu da **EINTR** dir (*Sinyallerle Kesilen İlkeller* (sayfa: 626)). Çoğu akım G/Ç gerçeklemesi onu az çok sakıncalı sıradan bir hata olarak ele alır. Tüm sinyalleri **SA_RESTART** bayrağıyla kurarak bu rahatsız edici durumdan kaçınabilirsiniz.

Benzer sebeplerle, bir akımın dosya tanıtıcısının bloklamayan G/Ç olarak ayarlanması genellikle tavsiye edilmez.

17. İkilik ve Metin Akımları

GNU sistemleri ve diğer POSIX uyumlu işletim sistemleri tüm dosyaları karakterlerin tektip dizisi olarak tanır. Diğer yandan, başka bazı sistemler metin içeren dosyalarla ikilik veri içeren dosyalar arasında ayrım yapar; ISO C giriş ve çıkış oluşumları da bu ayrıma göredir. Bu kısımda böyle sistemler arasında taşınabilir yazılımların nasıl geliştirileceğinden bahsedilmiştir.

Bir akımı açarken ya bir ***metin akımı*** ya da bir ***ikilik akım*** belirtebilirsiniz. Bir iklik akımı, **fopen** işlevinin *açıştürü* argümanına **b** değiştiricisini belirterek açabilirsiniz; bkz. *Akımların Açılması* (sayfa: 238). Bu seçenek olmaksızın bir dosya bir metin akımı olarak açılır.

Metin akımları ile ikilik akımlar çeşitli bakımlardan ayrılır:

- Veri, bir ikilik akımdan basitçe karakterlerin uzun bir serisi olarak, bir metin akımından ise satırsonu (**'\n'**) karakterleri ile sonlandırılmış satırlara bölünerek okunur. Bazı sistemler, 254 karakterden (satırsonu karakteri dahil) uzun satırlar içeren metin akımlarında başarısız olabilir.

- Bazı sistemlerde, metin dosyaları sadece basılabilen karakterleri, yatay sekme karekterlerini ve satırsonu karakterlerini içerebilir ve diğer karakterler desteklenmeyebilir. Buna karşın, ikilik akımlar her türlü karakteri içerebilir.
- Bir metin akımı içinde bir satırsonu karakteri ile öncelenmiş boşluk karakterleri, dosya tekrar okunduğunda görünmeyebilir.
- Daha genel olarak, bir metin akımından okunan veya metin akımına yazılan karakterler arasında birebir eşleşme gerekli olmayabilir.

Bir ikilik akım, bir metin akımına göre daha tahmin edilebilir ve daha yetenekli olduğuna göre metin akımlarının ne amaçla sunulduğunu düşünebilirsiniz. Niçin basitçe sadece ikili akımlar kullanılmaz? Yanıtı, bu işletim sistemlerinde, metin ve ikilik akımların farklı dosya biçimlerini kullanması ve diğer metin yönlenimli uygulamalarla birlikte çalışırken sıradan bir metin dosyasının okumanın ve ona yazmanın tek yolunun bir metin akımı kullanmak olmasıdır.

GNU kütüphanesinde ve tüm POSIX sistemlerde, iklik akımlar ile metin akımları arasında bir fark yoktur.Bir akımı açtığınızda ikilik bair akım isteyip istemediğinize bakılmaksızın aynı çeşit akım alırsınız. Bu akım, metin akımlarının sahip olduğu bazı kısıtlamalar olmaksızın her türlü dosya içeriği ile kullanılabilir.

18. Dosyalarda Konumlama

Bir akım üzerinde ***dosya konumlama***, akımın dosyanın neresinde okuma veya yazma yaptığı ile ilgilidir. Akım üzerinde G/Ç, dosya üzerinde dosya konumlamayı ilerletir. GNU sisteminde, dosya konumu bir tamsayı ile ifade edilir ve dosyanın başlangıcından itibaren bayt sayısını gösterir. Bkz. *Dosyada Konumlama* (sayfa: 232).

Sıradan bir disk dosyasında yapılan G/Ç işlemlerinde, dosyanın istediğiniz bir bölümüne okuma veya yazma amacıyla dosya konumunu istediğiniz gibi değiştirebilirsiniz. Diğer bazı dosya çeşitlerinde de buna izin verilebilmektedir. Dosya konumu değiştirmeyi destekleyen dosyalara bazan ***rasgele erişimli*** dosyalar da denir.

Bu bölümdeki işlevleri bir akım ile ilişkilendirilmiş dosya konumlayıcıyı değiştirmek ya da durumunu saptamak amacıyla kullanabilirsiniz. Aşağıda listelenen semboller `stdio.h` başlık dosyasında bildirilmiştir.

`long int` **`ftell`**`(FILE *`*akım*`)` işlev

Bu işlev *akım* akımının o andaki dosya konumu ile döner.

Bu işlev, eğer akım dosya konumlamayı desteklemiyorsa veya dosya konumu bir **`long int`** ile ifade edilemiyorsa ve olası diğer sebeplerle başarısız olabilir. Bir başarısızlık durumunda **`-1`** ile döner.

`off_t` **`ftello`**`(FILE *`*akım*`)` işlev

`ftello` işlevi **`off_t`** türünden bir dosya konumu ile dönmesi dışında **`ftell`** işlevi gibidir. POSIX belirtimi tarafından kullanılan **`long int`**'in aksine, bu veri türünü destekleyen sistemler, tüm dosya konumlarını açıklamakta onu kullanırlar. Bu ikisinin aynı boyutta olması gerekli değildir. Bu nedenle, gerçekleme, tepede POSIX uyumlu düşük seviyeli bir G/Ç gerçeklemesi olarak yazılmışsa **`ftell`** kullanımı bazı sorunlara yolaçabilir, bu durumda mümkün olduğunca **`ftello`** kullanımı tercih edilir.

Bu işlev başarısız olduğunda **`(off_t) -1`** ile döner. Bu, dosya konumlama desteği olmamasından ya da bir dahili hatanın sonucu olabilir. Aksi takdirde, dönüş değeri o anki dosya konumudur.

Bu işlev Tek Unix Belirtiminin 2. sürümünde tanımlı bir genişletmedir.

Kaynaklar, 32 bitlik sistemlerde **`_FILE_OFFSET_BITS == 64`** ile derlendiğinde bu işlev **`ftello64`** olarak davranır. Yani büyük dosya desteği arayüzü şeffaf olarak eski arayüzle yer değiştirir.

```
off64_t ftello64(FILE *akım)                                                    işlev
```

Bu işlev dönüş değerinin **off64_t** türünde olması dışında **ftello** gibidir. Bu ayrıca, *akım* akımının, 2^31 baytlık sınırın üzerindeki dosya konumlarına konumlayan dosya işlemlerinin başarılı olabildiği **fopen64**, **freopen64** veya **tmpfile64** işlevlerinin kullanılarak açılmasını gerektirir.

Kaynaklar, 32 bitlik sistemlerde **_FILE_OFFSET_BITS == 64** ile derlendiğinde bu işlev **ftello** ismiyle de kullanılabilir. Yani büyük dosya desteği arayüzü şeffaf olarak eski arayüzle yer değiştirir.

```
int fseek(FILE     *akım,                                                       işlev
          long int konum,
          int      nereden)
```

fseek işlevi *akım* akımının dosya konumunu değiştirmekte kullanılır. *nereden* parametresinin değeri, *konum* değerinin dosyanın başlangıcına mı, o anki dosya konumuna göre mi yoksa dosyanın sonuna göremi konumlanacağına bağlı olarak sırasıyla **SEEK_SET**, **SEEK_CUR** ya da **SEEK_END** sabitlerinden biri olmalıdır.

Eğer işlem başarılı olursa dönüş değeri sıfırdır. Sıfırdan farklı bir dönüş değeri işlemin başarısız olduğunu gösterir. Bir başarılı çağrı ayrıca *akım* akımının dosyasonu göstergesini temizler ve **ungetc** kullanımıyla "geriye basılan" karakterleri iptal eder.

fseek dosya konumunu değiştirmeden önce tamponlanmış çıktıyı ya boşaltır ya da aksine daha sonra dosyadaki yerine yazılmak üzere onu hatırlar.

```
int fseeko(FILE *akım,                                                          işlev
           off_t konum,
           int   nerden)
```

Bu işlev **fseek** işlevi gibidir, ancak POSIX türleri kullanılan sistemlerde **fseek** kullanımından kaynaklanan bir sorunu düzeltir. Konum için **long int** türünde bir değerin kullanılması POSIX ile uyumlu değildir. **fseeko** işlevi *konum* parametresi için doğru tür olan **off_t** türünü kullanır.

Bu sebeple, işlevselliği ilgili tanımlamaya daha yakın olduğundan (tamamen farklı bile olsa) mümkünse **ftello** kullanımı tercih edilmelidir.

İşlevsellik ve dönüş değeri **fseek** ile aynıdır.

Bu işlev Tek Unix Belirtiminin 2. sürümünde tanımlı bir genişletmedir.

Kaynaklar, 32 bitlik sistemlerde **_FILE_OFFSET_BITS == 64** ile derlendiğinde bu işlev **fseeko64** olarak davranır. Yani büyük dosya desteği arayüzü şeffaf olarak eski arayüzle yer değiştirir.

```
int fseeko64(FILE    *akım,                                                     işlev
             off64_t konum,
             int     nereden)
```

Bu işlev dönüş değerinin **off64_t** türünde olması dışında **fseeko** gibidir. Bu ayrıca, *akım* akımının, 2^31 baytlık sınırın üzerindeki dosya konumlarına konumlayan dosya işlemlerinin başarılı olabildiği **fopen64**, **freopen64** veya **tmpfile64** işlevlerinin kullanılarak açılmasını gerektirir.

Kaynaklar, 32 bitlik sistemlerde **_FILE_OFFSET_BITS == 64** ile derlendiğinde bu işlev **fseeko** ismiyle de kullanılabilir. Yani büyük dosya desteği arayüzü şeffaf olarak eski arayüzle yer değiştirir.

Uyumluluk Bilgisi:

POSIX dışı sistemlerde **ftell**, **ftello**, **fseek** ve **fseeko** işlevleri sadece ikilik akımlarla düzgün çalışabilir. Bkz. *İkilik ve Metin Akımları* (sayfa: 287).

Aşağıdaki sembolik sabitler **fseek**. işlevinin *nereden* argümanında kullanmak için tanımlanmışır. Bunlar ayrıca, **lseek** işleviyle kullanmak (*Girdi ve Çıktı İlkelleri* (sayfa: 308) ve dosya kilitleri için konum belirtmek (*Dosyalar Üzerindeki Denetim İşlemleri* (sayfa: 338)) için de kullanılır.

`int` **SEEK_SET** makro

Bu tamsayı sabit, **fseek** veya **fseeko** işlevinin *nereden* argümanında dosyanın başlangıcına göre konum belirtmek için kullanılır.

`int` **SEEK_CUR** makro

Bu tamsayı sabit, **fseek** veya **fseeko** işlevinin *nereden* argümanında dosyanın o anki dosya konumuna göre konum belirtmek için kullanılır.

`int` **SEEK_END** makro

Bu tamsayı sabit, **fseek** veya **fseeko** işlevinin *nereden* argümanında dosyanın sonuna göre konum belirtmek için kullanılır.

`void` **rewind**`(FILE *`*akım*`)` işlev

rewind işlevi *akım* akımını dosyanın başlangıcına konumlar. **fseek** veya **fseeko** işlevinin *nereden* argümanında **SEEK_SET** ve *konum* argümanında **0L** belirtilerek çağrılmasına eşdeğerdedir. Bu işlevlerin aksine dönüş değeri yoktur ve akımın hata göstergesi de sıfırlanır.

Eski BSD sistemleri ile uyumluluk adına **SEEK_…** sabitlerine karşılık olarak aşağıdaki sabitler de desteklenmektedir. Bu sabitler iki farklı başlık dosyasında tanımlıdır: `fcntl.h` and `sys/file.h`.

`L_SET`
: `SEEK_SET` ile aynıdır.

`L_INCR`
: `SEEK_CUR` ile aynıdır.

`L_XTND`
: `SEEK_END` ile aynıdır.

19. Taşınabilir Dosya Konumlama İşlevleri

GNU sistemlerinde dosya konumlaması tamamen karakter sayısıdır. **fseek** veya **fseeko** işlevine konumu karakter sayısı olarak belirtebilir ve herhangi bir rasgele erişimli dosyada düzgün sonuçlar alaabilirsiniz. Ancak ISO C sistemleride dosya konumlaması bu şekilde değildir.

Bazı sistemlerde metin akımları ikilik akımlardan tamamen farklıdır ve bir metin akımının dosya konumunu dosyanın başlangıcından itibaren karakterlerin sayısı olarak belirmek mümkün değildir. Örneğin bazı sistemlerde önce dosya içindeki kaydın konumuna oradanda kayıt içindeki karakter konumuna erişilir.

Sonuç olarak, eğer yazılımınızın bu sistemlere taşınabilir olmasını isterseniz bazı kurallara uymanız gerekir:

- Bir metin akımında **ftell** işlevinden dönen değer, o ana kadar okunan karakterlerin sayısıyla birebir ilişkili değildir. Tek bir şeyden emin olunabilir: **fseek** veya **fseeko** işlevinin *konum* argümanına aynı değerleri ardışık kullanarak geriye aynı dosya konumuna gidebilirsiniz.
- Bir metin akımında bir **fseek** veya **fseeko** çağrısında ya *konum* sıfır olmalı ya da *nereden* argümanının değeri **SEEK_SET** olmalı ve *konum* aynı akım üzerinde bir önceki **ftell** çağrısının sonucu olmalıdır.

- Bir metin akımının dosya konumlayıcı değeri, bu karakterler, onları okunmamış ya da iptal edilmemiş yapan **ungetc** ile geriye basılmışsa tanımsızdır. Bkz. *Okunmamış Yapmak* (sayfa: 252).

Bu kurallara uysanız bile uzun dosyalarda hala bazı sorunlarınız olabilir, çünkü **ftell** ve **fseek** dosya konumu için **long int** değer kullanır. Bu tür, bir büyük dosyadaki tüm dosya konumlarına erişmek için yeterli olmayabilir. **ftello** ve **fseeko** işlevlerinin kullanımı **off_t** türünü kullanmalarından ötürü tüm dosya konumlarına erişmeya yardımcı olabileceği umulsa bile hala, bir dosya konumu ile ilişkili ek bilgileri elde etmekte yardımı olmayacaktır.

Bu durumda, dosya konumu için özel kodlamalar kullanılan sistemlere destek vermek isterseniz, bunlar yerine **fgetpos** ve **fsetpos** işlevlerini kullanmanız daha iyi olur. Bu işlevler dosya konumunu belirtmek için dahili genişliği sistemden sisteme değişiklik gösteren **fpos_t** veri türünü kullanırlar.

Bu semboller `stdio.h` başlık dosyasında bildirilmiştir.

fpos_t veri türü

fgetpos ve **fsetpos** işlevlerinde kullanmak üzere, bir akımın dosya konumu hakkındaki bilgileri kodlayan bir nesnenin türüdür.

GNU sisteminde, **fpos_t**, dosya konumunu içeren dahili veriyi ve dönüşüm durum bilgilerini tutan bir veri yapısıdır. Diğer sistemlerdeki görüntüsü farklı olabilir.

Kaynaklar, 32 bitlik sistemlerde **_FILE_OFFSET_BITS == 64** ile derlendiğinde büyük dosya desteği arayüzü eski arayüzün yerine geçtiğinden bu veri türü **fpos64_t** türüne eşdeğer olur.

fpos64_t veri türü

fgetpos64 ve **fsetpos64** işlevlerinde kullanmak üzere, bir akımın dosya konumu hakkındaki bilgileri kodlayan bir nesnenin türüdür.

GNU sisteminde, **fpos64_t**, dosya konumunu içeren dahili veriyi ve dönüşüm durum bilgilerini tutan bir veri yapısıdır. Diğer sistemlerdeki görüntüsü farklı olabilir.

```
int fgetpos(FILE    *akım,
            fpos_t  *konum)
```
işlev

Bu işlev *akım* akımının dosya konum değerini *konum* ile gösterilen **fpos_t** nesnesinde saklar. Başarı durumunda **fgetpos** sıfır ile döner, aksi halde gerçeklemeye bağlı bir pozitif değeri **errno** değişkeninde saklayarak sıfırdan farklı bir değerle döner.

Kaynaklar, 32 bitlik sistemlerde **_FILE_OFFSET_BITS == 64** ile derlendiğinde büyük dosya desteği arayüzü eski arayüzün yerine geçtiğinden bu işlev **fgetpos64** işlevine eşdeğer olur.

```
int fgetpos64(FILE      *akım,
              fpos64_t  *konum)
```
işlev

Bu işlev dosya konumunu *konum* ile gösterilen **fpos64_t** türünde bir değişken içinde döndürmesi dışında **fgetpos** işlevi gibidir.

Kaynaklar, 32 bitlik sistemlerde **_FILE_OFFSET_BITS == 64** ile derlendiğinde büyük dosya desteği arayüzü eski arayüzün yerine geçtiğinden bu işlev **fgetpos** ismiyle de kullanılabilir.

```
int fsetpos(FILE          *akım,
            const fpos_t  *konum)
```
işlev

Bu işlev, aynı akım üzerinde bir önceki **fgetpos** işlevinden dönen *konum* değeriyle *akım* akımının dosya göstericisini konumlandırır. Başarı durumunda **fsetpos** akım üzerindeki dosyasonu göstergesini temizler, **ungetc** kullanımıyla geriye basılan karakterleri iptal eder ve sıfır değeriyle döner. Aksi takdirde, gerçeklemeye göre **errno** değişkenine bir pozitif değer atar ve sıfırdan farklı bir değerle döner.

Kaynaklar, 32 bitlik sistemlerde **_FILE_OFFSET_BITS == 64** ile derlendiğinde büyük dosya desteği arayüzü eski arayüzün yerine geçtiğinden bu işlev **fsetpos64** işlevine eşdeğer olur.

```
int fsetpos64(FILE           *akım,                                    işlev
              const fpos64_t *konum)
```

Bu işlev dosya konumlamasında kullanılacak *konum* parametresinin **fpos64_t** türünde bir değişken olarak verilmesi dışında **fsetpos** ile aynıdır.

Kaynaklar, 32 bitlik sistemlerde **_FILE_OFFSET_BITS == 64** ile derlendiğinde büyük dosya desteği arayüzü eski arayüzün yerine geçtiğinden bu işlev **fsetpos** ismiyle de kullanılabilir.

20. Akım Tamponlama

Bir akıma yazılan karakterler normalde bir araya getirilir ve uygulama tarafından bir çıktı olarak gösterilmeden bir blok olarak dosyaya eşzamansız olarak aktarılır. Benze olarak, akımlar çoğunlukla girdiyi karakter karakter değil bir blok olarak konak ortamından alırlar. Bu işleme ***tamponlama*** adı verilir.

Girdi ve çıktıyı akımları kullanarak etkileşimli yapan uygulamalar yazıyorsanız, uygulamanızın kullanıcı arayüzünü tasarlayabilmek için tamponlamanın nasıl çalıştığını bilmek zorundasınız. Aksi takdirde, karşınızda ummadığınız bir çıktı ya da umulmadık bir takım davranışlar görebilirsiniz.

Bu bölümde sadece karakterler, akımlarla yankılama ve akış denetimi gibi ve aygıtların belirli sınıflarında elde edilen şeyler arasında değil dosya ya da aygıtlar arasında nasıl aktarılacağı konusu işlenecektir. Uçbirimler üzerindeki denetim işlemleri ile ilgili bilgiler için *Düşük Seviyeli Uçbirim Arayüzü* (sayfa: 442) bölümüne bakınız.

Akım tamponlama oluşumlarını kullanmak yerine dosya tanımlayıcıları ile çalışan düşük seviye girdi ve çıktı işlevlerini birlikte kullanabilirsiniz. Bkz. *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305).

20.1. Tamponlama Kavramları

Tamponlama stratejilerinin üç türü vardır:

- *Tamponlanmamış* bir akıma yazılan veya okunan karakterler dosyaya mümkün olduğunca tek tek aktarılır.

- Bir *satır tamponlu* akıma yazılan karakterler bir satırsonu karakterine rastlandığında dosyaya blok olarak aktarılır.

- *Tamamı tamponlu* olarak bir akıma yazılan ya da okunan karakterler bir dosyaya keyfi uzunlukta bloklar halinde yazılır ya da okunur.

Yeni açılmış akımların normalde tamamı tamponludur, bir şey dışında: bir uçbirim gibi bir etkileşimli aygıtla bağlantılı akımlar dahili olarak satır tamponludur. Tamponlama türünün seçimi hakkında daha ayrıntılı bilgi için *Tamponlama Çeşidinin Seçimi* (sayfa: 294) bölümüne bakınız. Genellikle, özdevinimli seçim açtığınız dosya ya da aygıt için tamponlamanın en uygun çeşidini sağlar.

Etkileşimli aygıtlar için satır tamponlaması çıktılanan iletilerin sonuna (tam da istediğiniz şey) hemen bir satırsonu karakteri ekler. Bir satırsonu karakteri ile bitmeyen çıktılar hemen gösterilebileceği gibi gösterilmeyebilir de. Hemen görüntülenmesini isterseniz, *Tamponların Boşaltılması* (sayfa: 292) bölümünde açıklandığı gibi **fflush** ile tamponlu çıktıyı doğrudan doğruya aygıta boşaltabilirsiniz.

20.2. Tamponların Boşaltılması

Boşaltma, bir tamponlu akımın biriken karakterleri bir dosyaya çıktılamasıdır. Bir akım üzerindeki tamponlu çıktının özdevinimli olarak boşaltılması çeşitli durumlarda ortaya çıkar:

- Çıktı tamponu doludur ve çıktılama yapmayı deniyorsunuzdur.
- Akım kapatılırken. Bkz. *Akımların Kapatılması* (sayfa: 241).
- **exit** çağrısı ile uygulamayı sonlandırırken. Bkz. *Normal Sonlandırma* (sayfa: 681).
- Akım satır tamponludur ve bir satırsonu karakteri yazılmıştır.
- Bir akımın bir veriyi dosyadan okurken bir girdi işleminin varlığında.

Bunların dışında bir tamponlu çıktıyı boşaltmak isterseniz `stdio.h` başlık dosyasında bildirilmiş olan **fflush** işlevi çağrılır.

`int` **fflush**`(FILE *`*akım*`)` işlev

Bu işlev *akım* üzerindeki herhangi bir tamponlu çıktının dosyaya boşaltılmasına sebep olur. Eğer *akım* bir boş gösterici ise **fflush**, tamponlu çıktının tüm açık çıktı akımlarına boşaltılmasına sebep olur.

Bir yazma hatası oluşursa, bu işlev **EOF** döndürür. Aksi takdirde sıfır döner.

`int` **fflush_unlocked**`(FILE *`*akım*`)` işlev

Akımı kilitlememesi dışında **fflush** işlevi ile aynıdır.

fflush işlevi o an açık olan tüm akımları boşaltmak için kullanılabilir. Bu bazı durumlarda kullanışlıdır ama bazı durumlarda da gereklidir. Örneğin, uçbirimden girdi bekleyen bir uygulama için, uçbirimde tüm çıktının görünür olmasını istenir. Fakat bu sadece satır tamponlu akımlar için anlamlıdır. Solaris özellikle bu durum için bir işlev içerir. Bu işlev GNU C kütüphanesinde bir takım şekillerde hep vardı ama hiçbir zaman resmen var denilmedi.

`void` **_flushlbf**`(void)` işlev

_flushlbf işlevi o an açık bulunan tüm satır tamponlu akımları boşaltır.

Bu işlev `stdio_ext.h` başlık dosyasında bildirilmiştir.

Uyumluluk Bilgisi

Satır yönlenimli girdi ve çıktıya saplantılı olduğu bilinen kafa travması geçirmiş bazı işletim sistemlerinde satır tamponlu çıktının boşaltılması bir satır sonu karakterinin de yazılmasına sebep olur. Bereket versin ki bu "özellik" giderek daha az kullanılır olmaya doğru gidiyor. GNU sistemlerinde bundan dolayı kaygılanmanıza gerek yok.

Bazı durumlarda bekleyen çıktının boşaltılmak yerine unutulması daha kullanışlı olabilir. Eğer aktarımın bedeli yüksekse vegeçerli bir sebep yoksa çıktılama gerekmez. Bu gibi durumlar için Solaris'de standart dışı bir işlev vardır ve GNU kütüphanesinde de bulunmaktadır.

`void` **__fpurge**`(FILE *`*akım*`)` işlev

__fpurge işlevi *akım* akımının tamponunun temizlenmesini sağlar. Akım zaten okuma kipindeyse tampondaki tüm girdi kaybolur. Çıktılama kipindeyse tamponlu çıktı aygıta (veya igili saklama alanına) yazılmaz ve tampon temizlenir.

Bu işlev `stdio_ext.h` başlık dosyasında bildirilmiştir.

20.3. Tamponlama Çeşidinin Seçimi

Bir akım açıldıktan sonra (ancak henüz başka hiçbir işlem yapılmadan), Hangi tamponlama çeşidini kullanacağınızı **setvbuf** işlevini kullanarak belirtebilirsiniz. Bu bölümde sözü edilen oluşumlar `stdio.h` başlık dosyasında bildirilmiştir.

```
int setvbuf(FILE  *akım,                                                    işlev
            char  *tampon,
            int    kip,
            size_t boyut)
```

Bu işlev *akım* akımının hangi tamponlama kipini belirtmek için kullanılır. *kip* parametresinde, akımın tümünün tamponlanması isteniyorsa **_IOFBF**, satır tamponlu olması isteniyorsa **_IOLBF**, girdi/çıktının tamponlanması istenmiyorsa **_IONBF** sabiti kullanılır.

tampon argümanına boş gösterici belirtilirse, işlev **malloc** kullanarak tamponu kendisi ayırır. Akımı serbest bıraktığınızda bu tamponu da serbest bırakmış olacaksınız.

Aksi takdirde, *tampon* en az *boyut* karakterlik bir karakter dizisini tutacak büyüklükte seçilmelidir. Bu dizi akım açık olduğu akım tarafından kullanılacağından akım açık olduğu sürece serbest bırakmamalısınız. Tamponu ya durağan olarak ya da **malloc** (*Özgür Bellek Ayırma* (sayfa: 50)) kullanarak ayırmalısınız. Dizinin bildirildiği blok çıkmadan önce dosyayı kapatmadıkça özdevinimli bir dizinin kullanılması önerilmez.

Dizi bir akım tamponu olarak kaldığı sürece, akım G/Ç işlevleri dahili amaçları için tamponu kullanacaklardır. Akım diziyi tamponlama amacıyla kullanırken dizinin içeriğine doğrudan erişmeyi denememelisiniz.

setvbuf işlevi başarılı olduğunda sıfır ile döner. *kip* değeri geçersizse ya da istek yerine getirilememişse sıfırdan farklı bir değerle döner.

```
int _IOFBF                                                                  makro
```

Bu makronun değeri **setvbuf** işlevinin *kip* argümanında kullanılarak akımın tamamen tamponlanacağının belirtilmesini sağlayan bir tamsayı sabit ifadesidir.

```
int _IOLBF                                                                  makro
```

Bu makronun değeri **setvbuf** işlevinin *kip* argümanında kullanılarak akımın satır tamponlu olacağının belirtilmesini sağlayan bir tamsayı sabit ifadesidir.

```
int _IONBF                                                                  makro
```

Bu makronun değeri **setvbuf** işlevinin *kip* argümanında kullanılarak akımın tamponlanmayacağının belirtilmesini sağlayan bir tamsayı sabit ifadesidir.

```
int BUFSIZ                                                                  makro
```

Bu makronun değeri **setvbuf** işlevinin *boyut* argümanında kullanılmasının iyi olacağı bir tamsayı sabit ifadesidir. Bu değer en azından **256** değerini garantiler.

BUFSIZ değeri her sistemde akımın G/Ç verimliliğine uygun olarak seçilir. Bu bakımdan **setvbuf** çağrısında tampon boyu olarak **BUFSIZ** kullanmak iyi bir fikirdir.

Aslında, **fstat** sistem çağrısından elde edilen bir değerin tampon boyu olarak kullanılması daha da iyi olacaktır: dosya özniteliklerinin **st_blksize** alanı bu değeri içerir. Bkz. *Dosya Özniteliklerinin Anlamları* (sayfa: 371).

Bazıları **fgets** (*Karakter Girdilerinin Alınması* (sayfa: 248)) ile girdi alınması gibi durumlarda tamponların ayırma boyu olarak ayrıca **BUFSIZ** değerini de kullanırlar. G/Ç işlemlerinin verimli olmasını sağlamak dışında herhangi bir tamsayı değer yerine **BUFSIZ** kullanmanın geçerli bir sebebi yoktur.

```
void setbuf(FILE *akım,                                                    işlev
            char *tampon)
```

tampon argümanı bir boş gösterici olduğunda, bu işlev, **setvbuf** işlevinin *kip* argümanına **_IONBF** belirtilerek çağrılmasına eşdeğerdedir. Aksi takdirde **setvbuf** işlevinin *boyut* argümanında **BUFSIZ** ve *kip* argümanında **_IOFBF** kullanılarak *tampon* istendiği duruma eşdeğerdir.

setbuf işlevi eski kod ile uyumluluk için vardır. Yeni yazılımlarda **setvbuf** işlevini kullanmalısınız.

```
void setbuffer(FILE  *akım,                                                işlev
               char  *tampon,
               size_t boyut)
```

tampon bir boş gösterici ise bu işlev, *akım* akımını tamponsuz yapar. Aksi takdirde, *akım* akımı *tampon* tamponuna tamamen tamponlanacaktır. *boyut* argümanı *tampon* tamponunun boyunu belirtmekte kullanılır.

Bu işlev eski BSD kodu ile uyumluluk için vardır. Yeni yazılımlarda **setvbuf** işlevini kullanmalısınız.

```
void setlinebuf(FILE *akım)                                                işlev
```

Bu işlev *akım* akımını satır tamponlu yapar ve tamponu sizin için ayırır.

Bu işlev eski BSD kodu ile uyumluluk için vardır. Yeni yazılımlarda **setvbuf** işlevini kullanmalısınız.

Verilen bir akımın satır tamponlu mu yoksa Solaris'de standart–dışı ama GNU C kütüphanesinde mevcut işlevin kulanılmamış mı olduğu sorgulanabilir.

```
int __flbf(FILE *akım)                                                     işlev
```

__flbf işlevi, *akım* akımı satır tamponlu ise sıfırdan farklı bir değerle döner. Aksi takdirde sıfır ile döner.

Bu işlev **stdio_ext.h** başlık dosyasında bildirilmiştir.

Tampon boyunu ve ne kadarının kullanılmş olsuğunu sorgulayan iki işlev daha vardır ve bunlar Solaris'de de vardır.

```
size_t __fbufsize(FILE *akım)                                              işlev
```

__fbufsize işlevi, *akım* akımının tampon boyu ile döner. Bu değer akım kullanımını eniyilemek için kullanılabilir.

Bu işlev **stdio_ext.h** başlık dosyasında bildirilmiştir.

```
size_t __fpending(FILE *akım)                                              işlev
```

__fpending işlevi, çıktı tamponunda o an bulunan karakterlerin sayısı ile döner. Geniş karakterli tamponlar için bu boyut geniş karakter cinsindendir. Bu işlev okuma kipindeki ya da salt okunur açılmış tamponlarda kulllanılmamalıdır.

Bu işlev **stdio_ext.h** başlık dosyasında bildirilmiştir.

21. Diğer Akım Çeşitleri

GNU kütüphanesinde bir açık dosyanın karşılığı olması gerekli olmayan akım çeşitleri tanımlayabilme imkanı da vardır.

Bu tür bir akım bir girdiyi bir dizgeden alabilir veya bir dizgeye yazabilir. Bu çeşit akımlar dahili olarak **sprintf** ve **sscanf** işlevlerinin gerçeklemelerinde kullanılmaktadır. Siz *Dizge Akımları* (sayfa: 296) bölümünde açıklanan işlevleri kullanarak bu çeşit akımları kendiniz de oluşturabilirsiniz.

Daha genel olarak keyfi nesnelerde giriş/çıkış işlemi yapmakta kullanılan akımlar da tanımlanabilir. Bu protokol *Kendi Özel Akımlarınızı Oluşturun* (sayfa: 298) bölümünde anlatılmıştır.

Taşınabilirlik Bilgisi

Bu bölümde açıklanan oluşumlar GNU'ya özeldir. Diğer sistemler ve diğer C gerçeklemeleri bunlara eşdeğer işlevleri sağlıyor/sağlamıyor olabilir.

21.1. Dizge Akımları

Bir dizge ya da bellek tamponu ile veri alışverişini mümkün kılan **fmemopen** ve **open_memstream** işlevleri `stdio.h` başlık dosyasında bildirilmiştir.

```
FILE *fmemopen(void        *tampon,                                    işlev
               size_t       boyut,
               const char *açıştürü)
```

Bu işlev *tampon* tamponu ile belirtilen tamponu kullanan ve *açıştürü* argümanında belirtilen erişimi mümkün kılan bir akım açar. *tampon* olarak belirtilen dizi en az *boyut* bayt uzunlukta olmalıdır.

tampon olarak bir boş gösterici belirtirseniz **fmemopen** işlevi *boyut* bayt uzunlukta bir diziyi (**malloc** kullanarak; bkz. *Özgür Bellek Ayırma* (sayfa: 50)) özdevimli ayırırır. Bu genelikle sadece tampona bazı şeyleri yadıktan hemen sonra okumak sisterseniz kullanışlıdır. Çünkü aslında tampona bir gösterici almanın bir yolu yoktur (bunun için aşağıdaki **open_memstream** işlevini deneyin). Tampon, akım kapatıldığında serbest bırakılır.

açıştürü argümanı **fopen** işlevindeki gibi belirtilir (bkz. *Akımların Açılması* (sayfa: 238)). *açıştürü* ekleme kipinde belirtilirse, ilk dosya konumu tampondaki ilk boş karaktere ayarlanır. Aksi takdirde ilk dosya konumu tamponun başlangıcıdır.

Bir akım yazmak için açılırken boşaltılır ve kapatılır, bir boş karakter (sıfır baytı) yer varsa tamponun sonuna yazılır. Bunun için *boyut* argümanını tasarlarken bir ek baytı gözönüne almalısınız. Tampona *boyut* bayttan daha fazlası yazılmak istendiğinde bir hata oluşacaktır.

Okumak için açılan bir akım için tampondaki boş karakterler (sıfır baytları) dosyanın sonu olarak ele alınmaz. Okuma işlemlerinde sadece dosya konumu *boyut* bayt ilerlediğinde dosya sonu olarak değerlendirilir. Bu durumda karakterleri bir boş karakter sonlandırmalı dizgeden okumak isterseniz *boyut* baytlık bir dizge sağlamanız gerekir.

Aşağıdaki örnekte bir dizgeden okuma yapmak için **fmemopen** kullanarak bir akım açılmaktadır:

```
#include <stdio.h>

static char tampon[] = "deneme";

int
main (void)
{
  int ch;
  FILE *akim;

  akim = fmemopen (tampon, strlen (tampon), "r");
  while ((ch = fgetc (akim)) != EOF)
    printf ("%c okundu\n", ch);
  fclose (akim);
```

```
  return 0;
}
```

Bu kod parçası aşağıdaki çıktıyı üretir:

```
d okundu
e okundu
n okundu
e okundu
m okundu
e okundu
```

```
FILE *open_memstream(char  **gstr,                                    işlev
                     size_t *boyut)
```

Bu işlev bir tampona yazmak için bir akım açar. Tampon (**malloc** kullanılarak özdevimli ayrılır ve gerektikçe büyütülür. Akımı kapattıktan sonra bu tamponun **free** veya **realloc** kullanarak temizlenmesi sizin sorumluluğunuzdadır. Bkz. *Özgür Bellek Ayırma* (sayfa: 50).

Akım **fclose** ile kapatıldığında ya da **fflush** ile boşaltıldığında, *gstr* tampona bir gösterici olarak ve boyutu da *boyut* değişkenine güncellenir. Buralarda saklanmış olan değerler akımda başka bir değer yer alıncaya kadar geçerli kalır. Başka çıktılar almak için kullanmadan önce akımı boşaltmalısınız.

Tamponun sonuna bir boş karakter yazılır. Bu boş karakter *boyut* değerine dahil edilmez.

Akımın dosya konumunu **fseek** veya **fseeko** (*Dosyalarda Konumlama* (sayfa: 288)) işlevlerini kulanarak değiştirebilirsiniz. Dosya konumunun yazılmış verinin sonrasına taşınması halinde arada kalan boşluklar sıfırlarla doldurulur.

Aşağıda **open_memstream** işlevi kullanılan bir örnek görüyorsunuz:

```
#include <stdio.h>

int
main (void)
{
  char *bp;
  size_t size;
  FILE *stream;

  stream = open_memstream (&bp, &size);
  fprintf (stream, "merhaba");
  fflush (stream);
  printf ("buf = `%s', size = %d\n", bp, size);
  fprintf (stream, ", dostlar");
  fclose (stream);
  printf ("buf = `%s', size = %d\n", bp, size);

  return 0;
}
```

Bu kod parçası aşağıdaki çıktıyı üretir:

```
buf = `merhaba', size = 7
buf = `merhaba, dostlar', size = 16
```

21.2. Yığınak Akımları

Bir çıkış akımı açabilir ve verisini bir yığınağa koyabilirsiniz. Bkz. *Yığınaklar (Obstacks)* (sayfa: 64).

`FILE *open_obstack_stream(struct obstack *yığınak)` işlev

Bu işlev *yığınak* yığınağına veri yazmak için bir akım açar. Bu yığınakta bir nesne başlatır ve onu veri yazıldıkça büyütür (bkz. *Büyüyen Nesneler* (sayfa: 69)).

Bu akım üzerinde yapılan bir **fflush** çağrısı nesnenin o anki boyunu yazılmış olan veriye eşitler. Bir **fflush** çağrısından sonra nesneyi geçici olarak inceleyebilirsiniz.

Bir yığınak akımının dosya konumunu **fseek** veya **fseeko** (*Dosyalarda Konumlama* (sayfa: 288)) işlevlerini kullanarak değiştirebilirsiniz. Dosya konumunun yazılmış olan verinin sonrasına taşınması arada kalan boşlukların sıfırlarla doldurulmasına sebep olur.

Nesnyi kalıcı yapmak için yığınağı **fflush** ile güncelleyin ve **obstack_finish** ile nesneyi bitirip adresini alın. Bunun ardından akıma yazma işlemleri yığınakta yeni bir nesne başlatır ve sonraki **fflush** ve **obstack_finish** çağrılarına kadar bu nesne kullanılır.

Fakat ne uzunlukta bir nesne kullanıldığını nasıl bulacaksınız? Nesne uzunluğunu **obstack_object_size** (*Bir Yığınağın Durumu* (sayfa: 71)) çağrısı ile alabileceğiniz gibi nesneyi aşağıdaki gibi boş karakterle sonlandırabilirsiniz:

```
obstack_1grow (yığınak, 0);
```

Hangi yöntemi kullanırsanız kullanın, bunu **obstack_finish** çağrısından **önce** yapmalısınız (isterseniz ikisini de yapabilirsiniz).

Örnekte **obstack_finish** işlevinin kullanımı görülmektedir:

```
char *
ileti_dizgesi_yap (const char *a, int b)
{
  FILE *akim = open_obstack_stream (&ileti_yigini);
  output_task (akim);
  fprintf (akim, ": ");
  fprintf (akim, a, b);
  fprintf (akim, "\n");
  fclose (akim);
  obstack_1grow (&ileti_yigini, 0);
  return obstack_finish (&ileti_yigini);
}
```

21.3. Kendi Özel Akımlarınızı Oluşturun

Bu bölünde girdiyi bir keyfi veri kaynağından alan ve sizin tarafınızdan tasarlanan bir veri alıcısına çıktılayan bir akımı nasıl oluşturabileceğiniz açıklanmaktadır. Buna ***özel akımlar*** diyoruz. Burada açıklanan işlev ve veri türlerinin hepsi GNU oluşumudur.

21.3.1. Özel Akımlar ve Çerezler

Her özel akımın içeriği ***çerez*** adı verilen özel bir nesnedir. Bu nesne sizin tarafınızdan sağlanır ve okunan veya yazılan verinin nereden alınacağı ve nereye yazılacağının kayıtlarından oluşur. Kütüphanedeki akım işlevleri doğrudan bu akımların içeriği ile çalışamaz ve hatta veri türünü bile bilmez; bunların adresleri **void *** türünde kaydedilir.

Bir özel akımı gerçekleştirmek için, verinin belirtilen yerden NASIL alınacağı veya nerede saklanacağını belirtmelisiniz. Bunu, akıma yazan, onu okuyan, dosya konumunu değiştiren ve kapatan *kanca işlevleri* tanımlayarak yaparsınız. Bu dirt işlev akımın çerezine aktarılacak, böylece onlar verinin nereden alınıp nerede saklanacağını söyleyebilecektir. Kütüphane işlevleri çerezin içinde neler olduğunu bilmez, ama sizin işlevleriniz bilmelidir.

Bir özel akımı oluştururken bir çerez göstericisi ve ayrıca **cookie_io_functions_t** yapısının kayıtlarında saklanan dört kanca işlevi belirtmelisiniz.

Burada sözü edilen oluşumlar stdio.h başlık dosyasında bildirilmiştir.

cookie_io_functions_t veri türü

Bu veri yapısı, akım ile onun çerezi arasındaki iletişim protokolünü tanımlayan işlevleri saklar. Aşağıdaki üyelere sahiptir:

cookie_read_function_t ***read**
: Bu, çerezden veriyi okuyan işlevdir. Değeri bir işlev değil de bir boş gösterici ise bu akımdan yapılan okuma işlemleri daima **EOF** döndürür.

cookie_write_function_t ***write**
: Bu, çereze veriyi yazan işlevdir. Değeri bir işlev değil de bir boş gösterici ise bu akıma yazılan veriler iptal edilir.

cookie_seek_function_t ***seek**
: Bu, çerez üzerindeki dosya konumlamasına eşdeğer işlemleri uygulayan işlevdir. Değeri bir işlev değil de bir boş gösterici ise bu akım üzerinde yapılan **fseek** veya **fseeko** çağrıları sadece tampon içinde konumlama yapar; tamponun dışına çıkan tüm konumlama istekleri bir **ESPIPE** hatası ile sonuçlanacaktır.

cookie_close_function_t ***close**
: Bu, akım kapatılırken çerez üzerinde temizlik yapan işlevdir. Değeri bir işlev değil de bir boş gösterici ise bu akım kapatılırken çerezi kapatacak hiçbir özel işlem yapılmaz.

```
FILE *fopencookie(void                    *çerez,                    işlev
                  const char              *açıştürü,
                  cookie_io_functions_t   gç–işlevleri)
```

Bu işlev, *gç–işlevleri* argümanında belirtilen işlevleri kullanarak *çerez* ile haberleşen bir akım oluşturur. *açıştürü* argümanı **fopen** işlevindeki gibi kullanılır; bkz. *Akımların Açılması* (sayfa: 238). (Fakat "açılışta kırp" seçeneği gözardı edilir.) Oluşturulan akımın tamamı tamponlanır.

fopencookie işlevi ya yeni oluşturulan akım ile ya da bir hata durumunda bir boş gösterici ile döner.

21.3.2. Özel Akım Kanca İşlevleri

Bu bölümde bir özel akımın gerektirdiği dört kanca işlevin tanımlanması ayrıntılı olarak incelenmiştir.

Çerezden veri okuyan işlevi şöyle tanımlamalısınız:

```
ssize_t okuyucu (void *çerez, char *tampon, size_t boyut)
```

Bu işlev **read** işlevine çok benzer; bkz. *Girdi ve Çıktı İlkelleri* (sayfa: 308). İşleviniz *tampon* tamponuna *boyut* bayt aktarmalı ve okunan bayt sayısı ile ya da dosyasonunu belirtmek üzere sıfır ile dönmelidir. Hata durumunu belirtmek için **-1** de döndürebilirsiniz.

Çereze veri yazan işlevi şöyle tanımlamalısınız:

```
ssize_t yazıcı (void *çerez, const char *tampon, size_t boyut)
```

Bu işlev **write** işlevine çok benzer; bkz. *Girdi ve Çıktı İlkelleri* (sayfa: 308). İşleviniz *tampon* tamponundan *boyut* bayt aktarmalı ve yazılan bayt sayısı ile dönmelidir. Hata durumunu belirtmek için **-1** de döndürebilirsiniz.

Çerez üzerinde konumlama yapan işlevi şöyle tanımlamalısınız:

```
int konumlayıcı (void *çerez, off64_t *konum, int nereden)
```

Bu işlev için *konum* ve *nereden* argümanları **fgetpos** işlevindeki gibi yorumlanır; bkz. *Taşınabilir Dosya Konumlama İşlevleri* (sayfa: 290).

Konumlama işleminden sonra, işleviniz sonuçlanan dosya konumunu dosyanın başlangıcına göre *konum* argümanında saklamalıdır. İşleviniz başarı durumunda **0** ve hata durumunda **-1** ile dönmelidir.

Akımı kapatırken çerez üzerinde uygulanacak temizlik işlemlerini yapacak işlevi şöyle tanımlamalısınız:

```
int temizleyici (void *çerez)
```

İşleviniz başarı durumunda **0** ve hata durumunda **-1** ile dönmelidir.

cookie_read_function veri türü

Bu, bir özel akımın gerektirdiği okuyucu işlevin veri türüdür. İşlevi yukarıdaki gibi tanımlarsanız, bu, o işlevin veri türü olacaktır.

cookie_write_function veri türü

Bu, bir özel akımın gerektirdiği yazıcı işlevin veri türüdür. İşlevi yukarıdaki gibi tanımlarsanız, bu, o işlevin veri türü olacaktır.

cookie_seek_function veri türü

Bu, bir özel akımın gerektirdiği konumlayıcı işlevin veri türüdür.

cookie_close_function veri türü

Bu, bir özel akımın gerektirdiği kapatma işlevin veri türüdür.

22. Biçimli İletiler

System V tabanlı sistemlerde uygulamalar (özellikle sistem araçları) **fmtmsg** işlevini kullanarak iletilerini daima belli bir biçimde basar. Bu tek biçimlilik bazan iletileri yorumlayan kullanıcılara yardımcı olur ve **fmtmsg** işlevinin belirli biçimine bağlı sınamalarla yazılımcıyı daha az sayıda gereksinimi karşılamak zorunda bırakır.

22.1. Biçimli İletilerin Basılması

İletiler standart hataya ve/veya konsola basılabilir. Hedefi seçerken yazılımcı **fmtmsg** işlevinin *sınıflama* argümanında aşağıdaki iki değeri kullanabilir, isterse bit bit VEYAlayarak onları birleştirebilir:

`MM_PRINT`
İleti standart hatada gösterilir.

`MM_CONSOLE`
İleti sistem konsolunda gösterilir.

Sistemdeki hata kaynakları **fmtmsg** işlevinin *sınıflama* argümanında aşağıdaki değerler bit bit VEYAlanarak sinyallenebilir:

`MM_HARD`

Hata kaynağı bazı donanımlardır.

MM_SOFT
Hata kaynağı bazı yazılımlardır.

MM_FIRM
Hata kaynağı bazı donanımlara gömülü yazılımlardır.

fmtmsg işlevinin *sınıflama* argümanında belirtilebilecek üçüncü eleman sorunun saptandığı sistem parçalarını açıklamakta kullanılır. Bu aşağıdaki değerlerden sadece birini kullanarak yapılır:

MM_APPL
Sorun uygulama tarafından saptanmıştır.

MM_UTIL
Sorun başka bir uygulama tarafından saptanmıştır.

MM_OPSYS
Sorun işletim sistemi tarafından saptanmıştır.

fmtmsg işlevinin *sınıflama* argümanında belirtilebilecek son eleman bu iletinin sonucunu gösterir. Aşağıdaki değerlerden sadece biri kullanılabilir:

MM_RECOVER
Kurtarılabilir bir hata.

MM_NRECOV
Kurtarılamayacak bir hata.

```
int fmtmsg(long int     sınıflama,                                    işlev
           const char *isim,
           int          önem,
           const char *metin,
           const char *eylem,
           const char *etiket)
```

Parametreleriyle açıklanan bir iletiyi *sınıflama* parametresinde belirtilen aygıt(lar) üzerinde görüntüler. *isim* parametresi ileti kaynağını betimler. Dizge iki sütunu oluşturan iki parçadan oluşur. Birinci parça 10 karakterden, ikinci parça ise 14 karakterden uzun olamaz. *metin* parametresi hata durumunu, *eylem* parametresi hatadan kurtulmak için olası adımları açıklar, *etiket* parametresi ise daha fazla bilgi edinilebilecek belgeyi gösterir. is a reference to the online documentation where more information can be found. *etiket* parametresi *isim* değerini ve eşsiz bir kimlik numarası içermelidir.

Parametreleri her biri istenirse yoksayılabilecek özel bir değer olabilir. Bu değerler için kullanılabilecek sembolik isimler:

MM_NULLLBL
isim parametresi yoksayılır.

MM_NULLSEV
önem parametresi yoksayılır.

MM_NULLMC
sınıflama parametresi yoksayılır. Bu hiçbir şey basılmamasına sebep olur.

MM_NULLTXT

metin parametresi yoksayılır.

MM_NULLACT
eylem parametresi yoksayılır.

MM_NULLTAG
etiket parametresi yoksayılır.

Bu alanların yoksayılması için başka bir yöntem de çıktının standart hataya gönderilmesinde kullanılabilir. Bu yöntemde davranışı aşağıda açıkalanan ortam değişkenleri belirler.

önem parametresi aşağıdaki değerlerden biri olabilir:

MM_NOSEV
Hiçbir şey basılmaz, **MM_NULLSEV** ile aynıdır.

MM_HALT
Değer **ÇÖKME** (HALT) anlamında basılır.

MM_ERROR
Değer **HATA** (ERROR) olarak basılır.

MM_WARNING
Değer **UYARI** (WARNING) olarak basılır.

MM_INFO
Değer **BİLGİ** (INFO) olarak basılır.

Bu beş makronun değeri **0** ile **4** arasındadır. SEV_LEVEL ortam değişkenini kullanarak ya da **addseverity** işleviyle basılacak dizgeye karşılk olarak başka önem dereceleri de belirlenebilir. Bu konu *Önem Derecelerinin Eklenmesi* (sayfa: 303) bölümünde açıklanmıştır.

Hiçbir parametre yoksayılmadığında çıktı şuna benzer:

```
isim: önem–dizgesi: metin
TO FIX: eylem etiket
```

İkinokta üstüste, satırsonu ve **TO FIX** (DÜZELTMEK İÇİN) dizgesi üretilmesini sağlayan parametreler yoksayılarak görüntülenmeyebilir.

Bu işlev X/Open Taşınabilirlik Kılavuzunda belirtilmiştir. Ayrıca System V'den türetilmiş tüm sistemlerde bulunur.

GBir hata oluşmazsa işlev **MM_OK** değeri ile döner. Başarısızlık standart hataya basma ile ilgiliyse, **MM_NOMSG** ile döner. Çıktılama konsolda başarısız olursa, **MM_NOCON** ile döner. Hiçbirine çıktılama yapılamıyorsa, **MM_NOTOK** ile döner. Tüm çıktılamanın başarısız olduğu son durumda ayrıca bir parametrenin değeri yanlışsa bu da döner.

fmtmsg işlevinin davranışını açıklayan iki ortam değişkeni vardır. İlki olan **MSGVERB** çıktının gerçekte standart hataya (konsola değil) gönderilmesinde kullanılır. Beş alanın herbiri açıkça etkinleştirilir. Bunu yapmak için, işlev çağrılmadan önce ortam değişkeni aşağıdaki gibi düzenlenir:

```
MSGVERB=anahtar–sözcük[:anahtar–sözcük[:...]]
```

Geçerli *anahtar–sözcük*ler **label** (isim), **severity** (önem), **text** (metin), **action** (eylem) ve **tag** (etiket) sözcükleridir. Ortam değişkeni yoksa veya boşsa, desteklenmeyen bir anahtar sözcük kullanılmışsa ya da değer iletinin bir parçası olarak geçersizse o alan(lar) çıktılanmaz.

fmtmsg işlevinin davranışını açıklayan ikinci ortam değişkeni **SEV_LEVEL** değişkenidir. Bu değişken ve **fmtmsg** işlevinin davranış değiştirmesi X/Open Taşınabilirlik Kılavuzunda belirtilmemiştir. Yine de System V sistemlerinde bulunmaktadır. Yeni önem derecelerini belirtmekte kullanılır. Öntanımlı olarak yukarıda açıklanan beş önem derecesi vardır. Diğer sayısal değerler **fmtmsg** işlevinin bir şey basmasını sağlamaz.

Kullanıcı **SEV_LEVEL** değişkenini **fmtmsg** işlevini çağırmadan önce şöyle tanımlayabilir:

```
SEV_LEVEL=[açıklama[:açıklama[:...]]]
```

Buradaki *açıklama* alanları aşağıdaki değerleri alabilir:

```
önem–sözcüğü, seviye, basılacak–dizge
```

önem–sözcüğü parçası **fmtmsg** tarafından kıllanılmaz ama yine de vardır. *seviye* bir numarayı ifade eden bir dizgedir Sayısal değer 4 den büyük olmalıdır. Bu değer, **fmtmsg** işlevinin bu sınıfı seçmesini sağlayan *önem* parametresinde kullanılır. Öntanımlı önem derecelerinin yerini alacak bir değer mümkün değildir. *basılacak–dizge* parçası ise **fmtmsg** tarafından bu sınıf işleme alındığında gösterilecek iletidir (yukarıya bakarsanız, **fmtsmg** işlevinin sayısal değeri değil, onun dizge karşılığını bastığını görürsünüz).

22.2. Önem Derecelerinin Eklenmesi

SEV_LEVEL ortam değişkenini kullanmanın yanında yeni önem derecelerini belirlemek için bir olasılık daha vardır. Bu yöntem çalışan bir yazılım içinde başka önem derecelerinden bahsedilebilmesini kolaylaştırır. **setenv** veya **putenv** işlevleri ortam değişkenlerini ayarlamakta kullanılabilir ama bu zahmetlidir.

```
int addseverity(int          önem,                                    işlev
                const char *dizge)
```

Bu işlev, **fmtmsg** işlevinin *önem* parametresi tarafından adreslenebilecek yeni önem derecesini belrlemekte kullanılır. **addseverity** işlevinin *önem* parametresi **fmtmsg** işlevinin *önem* parametresinin değeri ile eşleşmeli ve *dizge* de asıl iletide sayısal değer yerine gösterilecek dizge olmalıdır.

dizge'nin değeri **NULL** ise *önem*'e karşılık olan sayısal değerle ilgili önem derecesi kaldırılır.

Öntanımlı önem derecelerini kaldırmak ya da değiştirmek mümkün değildir. *önem* parametresinde bu değerlerlerden biri kullanılarak yapılan **addseverity** çağrıları başarısız olacaktır.

Görev yerine getirilmişse işlev **MM_OK** ile döner. **MM_NOTOK** dönmüşse birşeyler yanlış gitmiş demektir. Bu durumda ya yeterli bellek ya da kaldırılmak istenen sınıf yoktur.

fmtsmg olduğu halde bu işlev X/Open Taşınabilirlik Kılavuzunda yoktur. System V sistemlerinde bulunur.

22.3. Örnek

Burada bu kısımda bahsedilmiş olan **fmtmsg** ve **addseverity** işlevlerinin kullanımı örneklenmiştir (Parametre değerleri değişmez alanları göstermesi açısından türkçeye çevrilmiştir.).

```
#include <fmtmsg.h>

int
main (void)
{
  addseverity (5, "BİLGİ-2");
  fmtmsg (MM_PRINT, "tek1alan", MM_INFO, "metin-2", "eylem-2", "etiket-2");
  fmtmsg (MM_PRINT, "UX:cat", 5, "sözdizimi geçersiz", "belgelere bakınız",
          "UX:cat:001");
```

```
  fmtmsg (MM_PRINT, "isim:foo", 6, "metin", "eylem", "etiket");
  return 0;
}
```

fmtmsg işlevinin ikinci çağrısı genelde bu işlevin ağırlıkla kullanıldığı System V sistemlerinde görülen kullanımını örneklemektedir. Bu sistemin System V üzerinde nasıl çalıştığına burada kısaca değinmek iyi olacaktır. *isim* alanındaki **UX:cat**, bir Unix uygulaması olan **cat**'in bir hata verdiğini belirtir. Devamında hatanın açıklaması ile *eylem* parametresi olarak **"belgeye bakınız"** değeri yer alır. Gerekiyorsa daha özel birşeyler yazılabilirdi. *etiket* alanı evvlce de bahsedildiği gibi *isim* parametresinin değerine ek olarak bir eşsiz kimlik (burada **001**) dizgesinden oluşur. GNU ortamında bu dizge uygulamanın info belgesindeki bu konuya karşılık düşen düğümü içermeliydi.

MSGVERB ve **SEV_LEVEL** değişkenleri ayarlanmadan yazılım çalıştırılırsa aşağıdaki çıktı üretilir:

```
UX:cat: BİLGİ-2: sözdizimi geçersiz
TO FIX: belgelere bakınız UX:cat:001
```

Burada iletinin farklı alanlarının iki nokta üstüste ve **TO FIX** dizgesi ile birlikte basıl basıldığını görüyoruz. Ama üç **fmtmsg** çağrısından yalnız biri bu çıktıyı üretti. İlk çağrı *isim* parametresi doğru biçimde olmadıığından hiçbir çıktı üretmedi. Dizge iki nokta üstüste ile ayrılmış iki alan içermeliydi (*Biçimli İletilerin Basılması* (sayfa: 300)). Üçüncü **fmtmsg** çağrısı da sayısal değeri **6** olan sınıf tanımlanmadığı çıktı üretmedi. Sayısal değeri **5** olan bir sınıf ayrıca öntanımlı olarak tanımlanmadığı halde **addseverity** çağrısı ile tanımlanmış ve ikinci **fmtmsg** çağrısı yukarıdaki çıktıyı üretmiştir.

Uygulamamız çalışırken ortamı **SEV_LEVEL=XXX,6,BİLGİ** ile değiştirirsek aşağıdaki sonucu alırız:

```
UX:cat: BİLGİ-2: sözdizimi geçersiz
TO FIX: belgelere bakınız UX:cat:001
isim:foo: BİLGİ: metin
TO FIX: eylem etiket
```

Şimdi üçüncü **fmtmsg** çağrısı çıktı üretti. Böylece ortam değişkenindeki **NOTE** dizgesinin ileti içinde nasıl kullanıldığını görmüş olduk.

Şimdi sadece bize gerekli alanlarla bir çıktı üretelim. Eğer, **MSGVERB** ortam değişkenine **severity:label:action** (önem:isim:eylem) değerini atarsak aşağıdaki çıktıyı alırız:

```
UX:cat: BİLGİ-2
TO FIX: belgelere bakınız
isim:foo: BİLGİ
TO FIX: eylem
```

Böylece, I.e., **fmtmsg** işlevinin *metin* ve *etiket* parametreleri kullanılmamış oldu. Dikkat ederseniz **BİLGİ** ve **BİLGİ-2** dizgelerinden sonra iki nokta üstüste kullanılmadı. Bu satırlarda bu dizgelerden sonra gösterilecek metin çıktılanmayacağından bu karakter kullanılmamıştır.

XIII. Düşük Seviyeli Girdi ve Çıktı

İçindekiler

Bu oylumda dosya tanıtıcıları üzerinde düşük seviyeli girdi ve çıktı işlemlerini gerçekleştiren işlevlerden bahsedilecektir. Bu işlevler *Akımlar Üzerinde Giriş/Çıkış* (sayfa: 236) bölümünde açıklanan daha yüksek seviyeli G/Ç işlemleri ile ilgili ilkeller ile akımlarda eşdeğeri olmayan düşük seviyeli denetim işlemlerini gerçekleştiren işlevlerdir.

Akım seviyesindeki G/Ç daha esnek ve daha kullanışlıdır; bununla birlikte, yazılımcılar gerektiğinde dosya tanıtıcı seviyesindeki işlevleri de kullanırlar. Bunlarun kullanım sebepleri genellikle şunlardır:

- İkilik dosyaları büyük tomarlar halinde okumak için.
- Dosyanın tamamını çözümlemek amacıyla core dosyasına okumak için.
- Sadece dosya tanıtıcıları ile yapılabilen veri aktarımı işlemlerini gerçekleştirmek için. (Bir akıma karşılık olan tanıtıcıyı `fileno` kullanarak alabilirsiniz.)
- Tanıtıcıları alt süreçlere aktarmak için. (Bir alt süreç bir akımı miras alamadığından bir tanıtıcıyı miras alıp onu kendi akımını oluşturmak için kullanır.)

1. Dosyaların Açılması ve Kapatılması

Bu kısımda dosya tanıtıcılar kullanılarak dosyaların açılması ve kapatılması için kullanılan ilkeller açıklanacaktır. **open** ve **creat** işlevleri `fcntl.h` dosyasında bildirilmişken **close** işlevi `unistd.h` dosyasında bildirilmiştir.

```
int open(const char *dosyaismi,                                          işlev
         int          seçenekler[, mode_t kip])
```

open işlevi, *dosyaismi* isimli dosya için bir dosya tanıtıcı oluşturur ve bunu döndürür. İlk olarak dosya konum göstergesi dosyanın başlangıcındadır. *kip* argümanı sadece dosya oluşturulurken kullanılır, ancak herhangi bir durumda argümanın bir zararı yoktur.

seçenekler argümanı dosyanın nasıl açılacağını belirler. Bu bir bit maskesidir ve ilgili parametreleri bit bit veyalayarak (C'de **|** işleci ile) değer oluşturabilirsiniz. Kullanılabilecek parametreleri *Dosya Durum Seçenekleri* (sayfa: 341) bölümünde bulabilirsiniz.

open işlevinin normal dönüş değeri negatif olmayan bir tamsayı olarak dosya tanıtıcısıdır. Bir hata durumunda –1 değeri döner. Olağan *dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EACCES
: Dosya var ancak *seçenekler* argümanında istendiği gibi okunabilir/yazılabilir değil; dosya yok, dizine de yazılamadığından dosya oluşturulamıyor.

EEXIST
: **O_CREAT** ve **O_EXCL** ikisi de belirtilmiş ve isimli dosya zaten var.

EINTR
: **open** işlemi bir sinyal tarafından durduruldu. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

EISDIR
: *seçenekler* argümanında yazma erişimi belirtilmiş ve dosya bir dizin.

EMFILE
: Sürecin çok fazla açık dosyası var. Dosya tanıtıcılarının azami sayısı **RLIMIT_NOFILE** özkaynak sınırı tarafından denetlenir; bkz. *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

ENFILE
: Dizini içeren sistemin tamamı ya da ihtimal ki dosya sistemi, bu anda hiçbir ek açık dosyayı destekleyemiyor (Böyle bir sorun GNU sisteminde asla olmaz).

ENOENT
: İsimli dosya yok ve **O_CREAT** belirtilmemiş.

ENOSPC
: Yeni dosyayı içerecek dizin ya da dosya sistemi genişletilemiyor çünkü diskte yer yok.

ENXIO
: **O_NONBLOCK** ve **O_WRONLY** ikisi de *seçenekler* argümanında belirtilmiş ve okumak için dosya açmış bir süreç yok.

EROFS
: Dosya bir salt okunur dosya istemi üzerinde ve *seçenekler* argümanında **O_WRONLY**, **O_RDWR** ve **O_TRUNC**'dan biri belirtilmiş; veya **O_CREAT** belirtilmiş ve dosya mevcut değil.

Bir 32 bitlik makinada kaynaklar **_FILE_OFFSET_BITS == 64** ile dönüştürülürse, **open** işlevi, 2^63 baytlık ve –2^63 ile 2^63 arasındaki konum sınırlı dosyalarda kullanılan dosya işleme işlevlerini etkinleştiren büyük dosya kipinde açılmış bir dosya tanıtıcı ile döner. Bu, tüm düşük seviyeli dosya işleme işlevleri büyük dosya kipindeki eşdeğerleri ile değiştirilerek kullanıcı bakımından şeffaf olarak yapılır.

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **open** çağrısı sırasında evre bazı özkaynakları (bellek, dosya tanıtıcı, semafor, vb.) ayırdığında bu bir sorun olur. Evre tam bu anda bir iptal alırsa ayrılan özkaynaklar yazılım sonlanana kadar ayrılmış olarak kalır. Bu tür **open** çağrılarından kaçınmak için iptal eylemcileri kullanılarak korunulmalıdır.

open işlevi, akımları oluşturan **fopen** ve **freopen** işlevlerinin düşük seviyedeki karşılığıdır.

```
int open64(const char *dosyaismi,                                    işlev
           int         seçenekler[, mode_t kip])
```

Bu işlev **open** işlevine benzer. *dosyaismi* isimli dosyaya erişim için kullanılabilen bir dosya tanıtıcı ile döner. Tek farkı bu işlevin 32 bitlik sistemlerde dosyayı büyük dosya (dosya uzunluğunun ve konum değerlerinin 63 bit genişlikte olduğu dosyalar) kipinde açmasıdır.

Kaynakların **_FILE_OFFSET_BITS == 64** ile dönüştürülürdüğü durumda, bu işlev aslında **open** ismi altında kullanılır. Yani 64 bitlik yeni, genişletilmiş arayüz eski arayüzle seffaf olarak değiştirilir.

```
int creat(const char *dosyaismi,                                     artık kullanılmayan işlev
          mode_t      kip)
```

Bu işlev atıl olmuştur.

```
creat (dosyaismi, kip)
```

gibi bir çağrı:

```
open (dosyaismi, O_WRONLY | O_CREAT | O_TRUNC, kip)
```

çağrısına eşdeğerdir.

Bir 32 bitlik makinada kaynaklar **_FILE_OFFSET_BITS == 64** ile dönüştürülürse, **creat** işlevi, 2^63 baytlık ve –2^63 ile 2^63 arasındaki konum sınırlı dosyalarda kullanılan dosya işleme işlevlerini etkinleştiren büyük dosya kipinde açılmış bir dosya tanıtıcı ile döner. Bu, tüm düşük seviyeli dosya işleme işlevleri büyük dosya kipindeki eşdeğerleri ile değiştirilerek kullanıcı bakımından şeffaf olarak yapılır.

```
int creat64(const char *dosyaismi,                                   artık kullanılmayan işlev
            mode_t      kip)
```

Bu işlev **creat** işlevine benzer. *dosyaismi* isimli dosyaya erişim için kullanılabilen bir dosya tanıtıcı ile döner. Tek farkı bu işlevin 32 bitlik sistemlerde dosyayı büyük dosya (dosya uzunluğunun ve konum değerlerinin 31 bit genişlikte olduğu dosyalar) kipinde açmasıdır.

Bu dosya tanıtıcıları ***64** biçiminde isimlendirilmiş işlevler (**read64** gibi) dışında normal dosya işlemlerinde kullanılmamalıdır.

Kaynakların **_FILE_OFFSET_BITS == 64** ile dönüştürülürdüğü durumda, bu işlev aslında **creat** ismi altında kullanılır. Yani 64 bitlik yeni, genişletilmiş arayüz eski arayüzle seffaf olarak değiştirilir.

```
int close(int dosyatanıtıcı)                                         işlev
```

Bu işlev dosya tanıtıcısı *dosyatanıtıcı* ile belirtilen dosyayı kapatır. Bir dosyanın kapatılma işlemi şöyle yürütülür:

- Dsya tanıtıcı serbest bırakılır.
- Dosyadaki süreç tarafından sahiplenilmiş kayıt kilitleri kaldırılır.
- Bir boru ya da FIFO ile ilişkili tüm dosya tanıtıcılar kapatılır, okunmamış veri varsa iptal edilir.

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **close** çağrısı sırasında evre bazı özkaynakları (bellek, dosya tanıtıcı, semafor, vb.) ayırdığında bu bir sorun olur. Evre tam bu anda bir iptal alırsa ayrılan özkaynaklar yazılım sonlanana kadar ayrılmış olarak kalır. Bu tür **close** çağrılarından kaçınmak için iptal eylemcileri kullanılarak korunulmalıdır.

close işlevinin normal dönüş değeri sıfırdır. Bir hata durumunda –1 değeri döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* geçerli bir dosya tanıtıcı değil

EINTR
: Çağrı bir sinyalle durduruldu Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626). **EINTR** hatasının düzgün olarak elde edilişine bir örnek:

```
TEMP_FAILURE_RETRY (close (desc));
```

ENOSPC
EIO
EDQUOT
: Dosyaya NFS üzerinden erişilirken, **write** işlevinden kaynaklanan bu hatalar bazan bir **close** çağrısına kadar saptanamaz. Bunun ne anlama geldiği *Girdi ve Çıktı İlkelleri* (sayfa: 308) bölümünde açıklanmıştır.

Ayrı bir **close64** işlevinin bulunmadığını aklınızdan çıkarmayın. Çünkü bu işlev dosya kipine bağımlı olmadığı gibi bunu saptamaya da çalışmaz. Bunu çekirdek bilir ve bir **close** çağrısı olduğunda gerekeni yapar.

Bir akımı kapatırken, **fclose** (bkz. *Akımların Kapatılması* (sayfa: 241)) işlevini kullanmalısınız, işlevin düşük seviye eşdeğeri olan **close** ile kapatmaya çalışmayın. **fclose** işlevi tamponda bir veri varsa bunu boşaltır ve akım nesnesini kapandığını belirtecek şekilde günceller.

2. Girdi ve Çıktı İlkelleri

Bu kısımda dosya tanıtıcılar üzerinde düşük seviyeli girdi ve çıktı işlemlerini gerçekleştiren **read**, **write** ve **lseek** işlevlerinden bahsedilecektir. Bu işlevler `unistd.h` başlık dosyasında bildirilmiştir.

ssize_t — veri türü

Bu veri türü tek bir işlem olarak okunup yazılabilen veri parçalarınını uzunluğunu göstermekte kullanılır. **size_t** türüne benzemekle birlikte bu bir işaretsiz bir türdür.

```
ssize_t read(int     dosyatanıtıcı,
             void   *tampon,
             size_t boyut)
```
işlev

read işlevi *dosyatanıtıcı* tanıtıcılı dosyadan *boyut* baytlık okuma yapar ve sonucu *tampon* içinde döndürür. (Bunun bir karakter dizgesi olması gerekmediği gibi sonlandırıcı boş karakter de eklenmez.)

İşlevin normal dönüş değeri okunan baytların sayısıdır. Bu *boyut* bayttan küçük olabilir; örneğin dosyada kalan baytlar *boyut* bayttan az olabilir ya da o an için bu dosyada bu kadar bayt olmayabilir. Davranış aslında dosyanın çeşidine bağlıdır. *boyut* bayttan daha az verinin okunmuş olmasının bir hata olarak değerlendirilmediğini unutmayın.

Sıfır dönüş değeri dosyasonunu belirtir (*boyut* argümanının sıfır olduğu durum dışında). Bu bir hata olarak sayılmaz. Dosya sonunda **read** çağrısı yapmayı sürdürürseniz, işlev de başka bir şey yapmadan sıfır döndürmeye devam eder.

read çağrısı ile zaten tek bir karakter döndürüyorsanız dosya sonuna eriştiğinizde bunu anlayamayacaksınız. Ancak dosya sonunda, sonraki **read** çağrıları hep sıfır döndürecektir.

Bir hata durumunda işlev –1 ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EAGAIN
: Normalde, bir girdi yoksa **read** girdi varolana kadar bekler. Fakat dosya için **O_NONBLOCK** seçeneği etkinse (bkz. *Dosya Durum Seçenekleri* (sayfa: 341)), böyle bir durumda **read** hiç veri okumadan hemen döner ve bunu bu hatayla raporlar.

  

  Uyumluluk Bilgisi

  BSD Unix'lerin çoğu sürümü bunun için farklı bir hata kodu kullanır: **EWOULDBLOCK**. GNU kütüphanesinde **EWOULDBLOCK**, **EAGAIN** için bir takma addır, dolayısıyla hangi ismi kullandığınızın bir önemi yoktur.

  Bazı sistemlerde, çekirdek kullanıcı sayfaları için yeterli fiziksel bellek bulamazsa, bir karakter özel dosyasından büyük miktarda veri okunması da **EAGAIN** hata koduna sebep olur. Bu, kullanıcı belleğine doğrudan bellek erişimi ile iletim yapan aygıtlarla sınırlıdır. Bunlar uçbirimleri kapsamaz, çünkü uçbirimler için çekirdek içinde daima ayrı tamponlar vardır. GNU kütüphanesinde böyle bir sorunla asla karşılaşmayacaksınız.

  EAGAIN ile sonuçlanabilecek bir durumda istenenden daha az bayt döndürerek **read** başarılı olabilir. Hemen ardından yapılan bir **read** çağrısı **EAGAIN** ile sonuçlanırdı.

EBADF
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcı değil ya da okumak için açılmamış

EINTR
: **read** çağrısı girdi beklerken bir sinyal ile durduruldu. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626). Bir sinyalin **read** çağrısının **EINTR** döndürmesine sebep olması şart değildir; istenenden daha az bayt döndürerek **read** başarılı olabilir.

EIO
: Çoğu aygıt ve disk dosyası için bu hata kodu bir donanım hatasına işaret eder.

  EIO ayrıca, bir artalan süreci denetim uçbiriminden okuma yapmaya çalışırken ve sürecin bir **SIGTTIN** sinyali gönderilerek durdurulmasında normal eylemin çalışmaması durumunda oluşabilir. Bu **SIGTTIN** sinyalinin engellenmesi ya da yoksayılmasından kaynaklanabileceği gibi süreç grubunun öksüz kalması nedeniyle de olabilir. Sinyaller hakkında daha fazla bilgi almak için *Sinyal İşleme* (sayfa: 601) bölümüne ve iş denetimi için *İş Denetimi* (sayfa: 716) bölümüne bakabilirsiniz.

EINVAL

Bazı sistemlerde bir karakter veya blok aygıtından okuma yapılırken, konum ve boyut başlangıçları belli bir blok boyuna hizalanmalıdır. Bu hata başlangıçların gerektiği gibi hizalanmadığını belirtir.

read64 isminde bir işlevin olmadığını unutmayın. İşlev olası geniş dosya konumlarını işlemek ya da değiştirmek için bir işlem yapmadığından bu gerekli değidir. Çekirdek gerekeni kendi içinde hallettiğinden **read** işlevi her durumda kullanılabilir.

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **read** çağrısı sırasında evre bazı özkaynakları (bellek, dosya tanıtıcı, semafor, vb.) ayırdığında bu bir sorun olur. Evre tam bu anda bir iptal alırsa ayrılan özkaynaklar yazılım sonlanana kadar ayrılmış olarak kalır. Bu tür **read** çağrılarından kaçınmak için iptal eylemcileri kullanılarak korunulmalıdır.

read işlevi, akımlardan okuma yapan **fgetc** gibi işlevlerin düşük seviyedeki karşılığıdır.

```
ssize_t pread(int    dosyatanıtıcı,                                    işlev
              void  *tampon,
              size_t boyut,
              off_t  konum)
```

pread işlevi ilk üç argümanının ve dönüş değeri ile hata durumlarının aynı olmasıyla **read** işlevine çok benzer.

Fark dördüncü argümanda ve onun elde edilişindedir. Okuma işlemi *dosyatanıtıcı* dosya tanıtıcısının o anki konumundan değil, *konum* ile belirtilen konumdan başlar ve dosya tanıtıcısının konumu bu işlemden etkilenmez; değeri çağrı öncesindeki değerinde kalır.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse **pread** işlevi aslında 2^63 bayta kadar dosyalarla çalışabilen ve 64 bitlik **off_t** türünde olan **pread64** işlevi olur.

pread işlevinin normal dönüş değeri okunan baytların sayısıdır. Hata durumunda **read** gibi –1 ile döndüğü gibi hata kodları aşağıdakiler dışında aynıdır:

EINVAL
: *konum* değeri negatif dolayısıyla kuraldışı

ESPIPE
: *dosyatanıtıcı* bir boru ya da FIFO ile ilişkili ve bu aygıtlar dosya içinde konum belirtilmesine izin vermez.

İşlev Unix Tek Belirtiminin 2. sürümünde tanımlı bir oluşumdur.

```
ssize_t pread64(int      dosyatanıtıcı,                                 işlev
                void    *tampon,
                size_t   boyut,
                off64_t  konum)
```

Bu işlev **pread** işlevinin benzeridir. Farkı *konum* parametresinin 2^31 bayta kadar dosyalar için olan **off_t** türünde değil, 2^63 bayta kadar dosyalar için olan **off64_t** türünde olmasıdır. Bu işlev için kullanılan *dosyatanıtıcı* tanıtıcısının **open64** ile açılması önemlidir. Aksi takdirde, küçük dosya kipinde açılmış dosya tanıtıcılarla **off64_t** türündeki dosya konumları hatalara yol açacaktır.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **pread** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

```
ssize_t write(int    dosyatanıtıcı,                                     işlev
              void  *tampon,
              size_t boyut)
```

write işlevi *dosyatanıtıcı* tanıtıcılı dosyaya *tampon* tamponundaki *boyut* baytı yazar. *tampon* içindeki veri bir karakter dizgesi olması gerekmediği gibi bir boş karakter de sıradan bir karakter olarak ele alınır.

İşlevin normal dönüş değeri yazılabilen baytların sayısıdır. Bu *boyut* sayıda olabileceği halde hep daha küçük olur. Yazılımınızda işlevi, tüm veriyi yazana kadar tekrarlanan **write** çağrıları şeklinde bir döngü içinde kullanmalısınız.

write döndükten hemen sonra kuyruklanan veri okunabilir. Bunun için verinin kalıcı bir saklama alanına yazılması şart değildir. Devam etmeden önce verinin kalıcı saklama alanına yazılmasını sağlamak için **fsync** kullanablirsiniz. (Yazma işlemini peşpeşe çağrılar şeklinde bir defada gerçekleştirip saklama alanına yazma işini sisteme bırakmak daha verimlidir. Normalde bu veri diske bir dakikadan daha geç yazılmaz.) Günümüz sistemlerinde **fdatasync** adında bir işlev daha vardır ve bununla dosya verisinin bütünlüğü garanti edilmiştir ve daha hızlıdır. Dosyayı **O_FSYNC** kipinde açarsanız **write** çağrıları veri diske yazılmadan dönmeyecektir; bkz. *G/Ç İşlem Kipleri* (sayfa: 344).

Hata durumunda işlev –1 ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EAGAIN
: Normalde, yazma işlemi tamamlanana kadar **write** dönmez. Fakat **O_NONBLOCK** seçeneği etkinse (bkz. *Dosyalar Üzerindeki Denetim İşlemleri* (sayfa: 338)), hiçbir veri yazılmadan işlev hemen bu hata durumu ile döner. Bu duruma bir örnek vermek gerekirse, sürecin bir STOP karakteri aldığında, akış denetimini destekleyen bir uçbirim aygıtına çıktının yazılmasını engellemesi verilebilir.

Uyumluluk Bilgisi

BSD Unix'in çoğu sürümü bu hata kodu için farklı bir hata kodu kullanır: **EWOULDBLOCK**. GNU kütüphanesinde **EWOULDBLOCK**, **EAGAIN** için bir takma addır, dolayısıyla hangi ismin kullanıldığının bir önemi yoktur.

Bazı sistemlerde, çekirdek kullanıcı sayfaları için yeterli fiziksel bellek bulamazsa, bir karakter özel dosyasına büyük miktarda veri yazılması da **EAGAIN** hata koduna sebep olur. Bu, kullanıcı belleğine doğrudan bellek erişimi ile iletim yapan aygıtlarla sınırlıdır. Bunlar uçbirimleri kapsamaz, çünkü uçbirimler için çekirdek içinde daima ayrı tamponlar vardır. GNU kütüphanesinde böyle bir sorunla asla karşılaşmayacaksınız.

EBADF
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcı değil ya da yazma amacıyla açılmamış

EFBIG
: Dosya boyutu gerçeklemenin desteklediğinden büyük

EINTR
: **write** işlemi tamamlanmadan önce bir sinyal tarafından durduruldu. Bir sinyal her zaman **write** işlevinin **EINTR** döndürmesine sebep olmaz; istenenden daha az baytı yazarak da işlev başarılı olabilir. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

EIO
: Çoğu aygıt ve disk dosyası için bu hata kodu bir donanım hatasını işaret eder.

ENOSPC
: Aygıtın dosyası dolu.

EPIPE

Bu hata bir süreç tarafından okumak için açılmamış bir boru ya da FIFO'ya yazmaya çalışırsanız oluşur. Bu oluştuğu zaman sürece bir **SIGPIPE** sinyali gönderilir; bkz. *Sinyal İşleme* (sayfa: 601).

EINVAL
Bazı sistemlerde bir karakter veya blok aygıtından okuma yapılırken, konum ve boyut başlangıçları belli bir blok boyuna hizalanmalıdır. Bu hata başlangıçların gerektiği gibi hizalanmadığını belirtir.

EINTR başarısızlıklarından korunmak için bir düzenleme yapmadıkça, her başarısız **write** çağrısından sonra **errno** değişkenine bakmalı ve bu **EINTR** hatası ise çağrıyı tekrarlamalısınız. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626). Bunu yapmanın kolay bir yolu **TEMP_FAILURE_RETRY** makrosunu kullanmaktır:

```
nbytes = TEMP_FAILURE_RETRY (write (desc, buffer, count));
```

write64 isminde bir işlevin olmadığını unutmayın. İşlev olası geniş dosya konumlarını işlemek ya da değiştirmek için bir işlem yapmadığından bu gerekli değidir. Çekirdek gerekeni kendi içinde hallettiğinden **write** işlevi her durumda kullanılabilir.

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **write** çağrısı sırasında evre bazı özkaynakları (bellek, dosya tanıtıcı, semafor, vb.) ayırdığında bu bir sorun olur. Evre tam bu anda bir iptal alırsa ayrılan özkaynaklar yazılım sonlanana kadar ayrılmış olarak kalır. Bu tür **write** çağrılarından kaçınmak için iptal eylemcileri kullanılarak korunulmalıdır.

write işlevi, akımlardan okuma yapan **fputc** gibi işlevlerin düşük seviyedeki karşılığıdır.

```
ssize_t pwrite(int         dosyatanıtıcı,                                    işlev
               const void *tampon,
               size_t      boyut,
               off_t       konum)
```

pwrite işlevi ilk üç argümanının ve dönüş değeri ile hata durumlarının aynı olmasıyla **write** işlevine çok benzer.

Fark dördüncü argümanda ve onun elde edilişindedir. Yazma işlemi *dosyatanıtıcı* dosya tanıtıcısının o anki konumundan değil, *konum* ile belirtilen konumdan başlar ve dosya tanıtıcısının konumu bu işlemden etkilenmez; değeri çağrı öncesindeki değerinde kalır.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse **pwrite** işlevi aslında 2^63 bayta kadar dosyalarla çalışabilen ve 64 bitlik **off_t** türünde olan **pwrite64** işlevi olur.

pwrite işlevinin normal dönüş değeri yazılan baytların sayısıdır. Hata durumunda **write** gibi –1 ile döndüğü gibi hata kodları aşağıdakiler dışında aynıdır:

EINVAL
konum değeri negatif dolayısıyla kuraldışı

ESPIPE
dosyatanıtıcı bir boru ya da FIFO ile ilişkili ve bu aygıtlar dosya içinde konum belirtilmesine izin vermez.

İşlev Unix Tek Belirtiminin 2. sürümünde tanımlı bir oluşumdur.

```
ssize_t pwrite64(int         dosyatanıtıcı,                                  işlev
                 const void *tampon,
                 size_t      boyut,
                 off64_t     konum)
```

Bu işlev **pwrite** işlevinin benzeridir. Farkı *konum* parametresinin 2^31 bayta kadar dosyalar için olan **off_t** türünde değil, 2^63 bayta kadar dosyalar için olan **off64_t** türünde olmasıdır. Bu işlev için kullanılan *dosyatanıtıcı* tanıtıcısının **open64** ile açılması önemlidir. Aksi takdirde, küçük dosya kipinde açılmış dosya tanıtıcılarla **off64_t** türündeki dosya konumları hatalara yol açacaktır.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **pwrite** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

3. Dosya Konumu İlkeli

Bir akımın dosya konumunu **fseek** ile belirtiğiniz gibi, bir tanıtıcının dosya konumunu da **lseek** ile sonraki **read** veya **write** işlemleri için belirtebilirsiniz. Dosya konumlamanın ne olduğu ve daha fazlası için *Dosyalarda Konumlama* (sayfa: 288) bölümüne bakabilirsiniz.

Bir tanıtıcıdan o anki dosya konumunu okumak için
lseek (*tanıtıcı***, 0, SEEK_CUR)** çağrısını kullanabilirsiniz.

```
off_t lseek(int    dosyatanıtıcı,                                    işlev
            off_t  konum,
            int    nereye)
```

lseek işlevi *dosyatanıtıcı* tanıtıcısında dosya konumunu değiştirmek için kullanılır.

nereye argümanı **fseek** işlevindeki gibi *konum*'un nasıl yorumlanacağını belirtir. Değeri **SEEK_SET**, **SEEK_CUR** veya **SEEK_END** sembolik sabitlerinden biri olabilir.

SEEK_SET
: *nereye* dosyanın başlangıcından itibaren karakter sayısını belirtir.

SEEK_CUR
: *nereye* o anki dosya konumundan itibaren karakter sayısını belirtir.

SEEK_END
: *nereye* dosya sonundan kaç karakter sayılacağını belirtir. Negatif bir değer dosya sonundan ileriye doğru, pozitif bir değer dosya sonundan geriye doğru karakter sayısıdır. Konumu sondan başa doğru belirtirseniz ve yazma yapıyorsanız, dosya, konuma kadar sıfırlarla doldurularak büyütülecektir.

lseek işlevinin normal dönüş değeri sonuçlanan dosya başlangıcından itibaren karakter sayısı olarak dosya konumudur. Bu özelliğini, o anki dosya konumunu öğrenmek için **SEEK_CUR** ile kullanabilirsiniz.

Dosyaya eklema yapmak isterseniz, dosya konumunu o anki dosya sonundan itibaren **SEEK_END** ile belirtmek yeterli olmaz. Siz dosya konumunu değiştirdikten sonra ancak yazmaya başlamadan önce başka bir süreç dosyaya bir miktar veri yazmış olabilir, dolayısıyla böyle bir konuma yapacağınız bir yazma işlemi mevcut verinin kaybına sebep olur. Bunun olmaması için **O_APPEND** işletim kipini kullanmalısınız; bkz. *G/Ç İşlem Kipleri* (sayfa: 344).

Dosya konumunu dosyanın o anki dosya sonundan başa doğru belirtebilirsiniz. Bu dosyanı kendisini uzatmaz; **lseek** dosyanın boyunu asla değiştirmez. Fakat aynı konuma daha sonra yapılan çıktılama dosyayı uzatır. Dosyanın önceki sonu ile yeni konum arası sıfırlarla doldurulur. Bu yolla uzatılan dosyada sıfırlardan oluşan bir "delik" oluşur, disk üzerinde yer ayırma yapılmaz, bu durumda dosya göründüğünden daha az alanı kapsar; bu durumdaki bir dosyaya "seyrek dosya" (sparse file) denir. Dosya konumu değiştirilemezse ya da işlem bir şekilde geçersiz olmuşsa, işlev –1 ile döner. Aşağıdaki **errno** hata durmları bu işlev için tanımlanmıştır:

EBADF

dosyatanıtıcı geçerli bir dosya tanıtıcı değil

EINVAL
nereye argümanının değeri geçersiz ya da sonuçlanan dosya konumu geçersiz. Bir dosya konumu geçersiz.

ESPIPE
dosyatanıtıcı, bir boru, FIFO ya da uçbirim gibi dosya konumlamasına izin verilmeyen bir nesneye karşılık olduğundan konunlama yapılamıyor. (POSIX.1 bu hatayı sadece borular ve FIFO'lar için belirtir, ancak GNU sisteminde konumlama yapılamayan her nesne için daima **ESPIPE** hatasını alırsınız.)

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse **lseek** işlevi aslında 2^63 bayta kadar dosyalarla çalışabilen ve 64 bitlik **off_t** türünde olan **lseek64** işlevi olur.

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **lseek** çağrısı sırasında evre bazı özkaynakları (bellek, dosya tanıtıcı, semafor, vb.) ayırdığında bu bir sorun olur. Evre tam bu anda bir iptal alırsa ayrılan özkaynaklar yazılım sonlanana kadar ayrılmış olarak kalır. Bu tür **lseek** çağrılarından kaçınmak için iptal eylemcileri kullanılarak korunulmalıdır.

lseek işlevi, akımlarda konumlama yapan **fseek**, **fseeko**, **ftell**, **ftello** ve **rewind** işlevlerinin düşük seviyedeki karşılığıdır.

```
off64_t lseek64(int      dosyatanıtıcı,                                  işlev
                off64_t  konum,
                int      nereye)
```

Bu işlev **lseek** işlevinin benzeridir. Farkı *konum* parametresinin 2^31 bayta kadar dosyalar için olan **off_t** türünde değil, 2^63 bayta kadar dosyalar için olan **off64_t** türünde olmasıdır. Bu işlev için kullanılan *dosyatanıtıcı* tanıtıcısının **open64** ile açılması önemlidir. Aksi takdirde, küçük dosya kipinde açılmış dosya tanıtıcılarla **off64_t** türündeki dosya konumları hatalara yol açacaktır.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **lseek** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

Bir dosyayı defalarca açarsanız ya da **dup** ile tanıtıcıyı çoğaltırsanız aynı dosya için çok sayıda tanıtıcınız olabilir. **open** çağrıları ile elde edilen tanıtıcıların dosya konumları birbirinden bağımsızdır dolayısıyla biri üzerinde kullanacağınız **lseek** diğerlerini etkilemez. Örneğin,

```
{
  int d1, d2;
  char buf[4];
  d1 = open ("foo", O_RDONLY);
  d2 = open ("foo", O_RDONLY);
  lseek (d1, 1024, SEEK_SET);
  read (d2, buf, 4);
}
```

foo dosyasının ilk dört karakterini okuyacaktır. (Gerçek br yazılımda gerekli olan hata denetimine, örneği karıştırmaması için yer verilmemiştir.)

Tersine olarak, çoğaltma yoluyla elde edilen dosya tanıtıcılar ortak bir dosya konumunu paylaşırlar. Biri üzerinde yapılan okuma, yazma, dosya konumu değiştirme gibi her işlem diğerlerini etkiler. Örneğin,

```
{
```

```
  int d1, d2, d3;
  char buf1[4], buf2[4];
  d1 = open ("foo", O_RDONLY);
  d2 = dup (d1);
  d3 = dup (d2);
  lseek (d3, 1024, SEEK_SET);
  read (d1, buf1, 4);
  read (d2, buf2, 4);
}
```

foo dosyasının 1024. karakterinden başlayarak dört karakter okur ve 1028. karakterden başlayarak dört karakter daha okur.

off_t veri türü

Bu dosya boyutlarını göstermekte kullanıan bir veri türüdür. GNU sisteminde **fpos_t** veya **long int** türüne eşdeğerdir.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse, bu veri türü **off64_t** veri türüne eşdeğer olur.

off64_t veri türü

off_t veri türüne benzer. Bir farkla: 32 bitlik makinalarda **off_t** 32 bitlikken **off64_t** 64 bitliktir. Böylece dosyalar 2^63 bayta kadar adreslenebilir.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse, bu veri türü **off_t** ismiyle kullanılır.

SEEK_... sabitleri için daha eski BSD sistemleri ile uyumluluk adına takma adlar tanımlanıştır. Bunlar iki ayrı başlık dosyasında tanımlıdır: `fcntl.h` ve `sys/file.h`.

L_SET
: **SEEK_SET** için takma addır.

L_INCR
: **SEEK_CUR** için takma addır.

L_XTND
: **SEEK_END** için takma addır.

4. Tanıtıcılar ve Akımlar

Belirtilen bir dosya tanıtıcı için **fdopen** ile bir akım oluşturabileceğiniz gibi **fileno** ile mevcut bir akımın dosya tanıtıcısını elde edebilirsiniz. Bu işlevler `stdio.h` başlık dosyasında bildirilmiştir.

```
FILE *fdopen(int           dosyatanıtıcı,
             const char *açıştürü)
```
işlev

fdopen işlevi *dosyatanıtıcı* tanıtıcısı için yeni bir akım döndürür.

açıştürü argümanı **fopen** işlevindeki gibi yorumlanır (bkz. *Akımların Açılması* (sayfa: 238)). Bir farkla: GNU sisteminde ikilik ve metin dosyalar arasında bir fark gözetilmediğinden **b** seçeneğine izin verilmez. Ayrıca **"w"** ve **"w+"** dosyanın kırpılmasına sebep olmaz; bu sadece dosyayı açarken etkilidir ve bu durumda zaten dosya açıktır. *açıştürü* argümanı açık dosyanın kipiyle eşleşmek zorundadır, buna dikkat etmelisiniz.

İşlevin normal dönüş değeri yeni akımdır. Akım oluşturulamazsa bir boş gösterici döner. Bu duruma bir örnek: dosya tanıtıcının dosya kipinin *açıştürü* argümanında belirtilen erişim türüne izin vermemesi.

Bazı sistemlerde, *açıştürü* argümanında belirtilen erişim türüne izin vermeyen dosya tanıtıcı kipi saptanırken **fdopen** başarısız olabilir. GNU C kütüphanesi daima bunun için denetlenir.

fdopen işlevinin bir kullanım örneğini *Bir Borunun Oluşturulması* (sayfa: 393) bölümünde bulabilirsiniz.

`int` **fileno**`(FILE *`*akım*`)` işlev

Bu işlev *akım* akımıyla ilişkili dosya tanıtıcı ile döner. Bir hata saptanırsa (örn, *akım* geçersizse) ya da *akım* bir dosyayla G/Ç yapmıyorsa, işlev –1 ile döner.

`int` **fileno_unlocked**`(FILE *`*akım*`)` işlev

fileno_unlocked işlevi, durum **FSETLOCKING_INTERNAL** ise akımı doğrudan kilitlememesi dışında **fileno** işlevine eşdeğerdir.

Bu işlev bir GNU oluşumudur.

stdin, **stdout** ve **stderr** *standart akımlarının* (sayfa: 237) dosya tanıtıcıları için sembolik sabitleri vardır ve bunlar `unistd.h` başlık dosyasında tanımlıdır.

STDIN_FILENO makro

Bu makro standart girdi için dosya tanıtıcı olarak **0** değerine sahiptir.

STDOUT_FILENO makro

Bu makro standart çıktı için dosya tanıtıcı olarak **1** değerine sahiptir.

STDERR_FILENO makro

Bu makro standart hata için dosya tanıtıcı olarak **2** değerine sahiptir.

5. Akımlarla Tanıtıcıları Karıştırmanın Tehlikeleri

Aynı dosyaya bağlı çok sayıda dosya tanıtıcısı ve akıma (bunların ikisine birden kanallar diyebiliriz) sahip olabilirsiniz ama kanalları birbiri yerine kullanmaktan kaçınmanız gerekir. Ele alınacak iki durum vardır: tek bir dosya konumunu paylaşan ***ilintili kanallar*** ile kendi dosya konumları olan ***bağımsız kanallar***.

Tüm erişimin girdi amacıyla olması dışında, bir dosyayla veri iletimi için yazılımınızda sadece bir kanalı kullanmanız en iyisidir. Örneğin, bazı şeylerin sadece dosya tanıtıcıları ile yapılabildiği bir boru açarsanız, tüm G/Ç işlemlerini hem dosya tanıtıcı ile hem de bu dosya tanıtıcı için **fdopen** ile oluşturacağınız akım ile yapabilirsiniz.

5.1. İlintili Kanallar

Tek bir açılışta aynı dosya konumunu paylaşan kanallara ***ilintili kanallar*** diyoruz. **fdopen** kullanarak bir tanıtıcıdan bir akım yaptığınızda, **fileno** ile bir akım ile ilişkili dosya tanıtıcısına eriştiğinizde, **dup** veya **dup2** ile bir dosya tanıtıcısını çoğalttığınızda ve **fork** ile dosya tanıtıcıları alt süreçte miras aldığınızda bir ilintili kanalınız olur. Uçbirimler ve borular gibi rasgele erişimi desteklemeyen dosyalar için *tüm* kanallar zorunlu olarak ilintilidir. Rasgele erişimli dosyalarda, sona ekleme yapılan akımlar da zorunlu olarak diğeri ile ilintilidir.

G/Ç için bir akım kullanıyorsanız (sadece akım açmışsanız) ve onunla ilintili diğer kanalla (bir akım ya da tanıtıcı olabilir) G/Ç yapmak isterseniz, kullanmakta olduğunuz akımı önce *temizlemelisiniz* (sayfa: 317).

Bir sürecin sonladırılması ya da süreç içinde yeni bir yazılımın çalıştırılması süreç içindeki tüm akımların ortadan kaldırılmasına sebep olur. Bu akımlarla ilintili tanıtıcılar diğer süreçte kalır, dolayısıyla dosya konumları tanımsız duruma gelir. Bundan kaçınmak için ortadan kaldırmadan önce akımları temizlemeniz gerekir.

5.2. Bağımsız Kanallar

Dosya konumlaması yapılabilen bir dosya için ayrı kanallar (akım ya da tanıtıcı) açtığınızda her kanalın kendi dosya konumu olur. Bu tür kanallara ***bağımsız kanallar*** denir.

Sistem her kanalı bağımsız olarak elde eder. Çoğu zaman, bu tahmin edilebilir ve (özellikle girdi için) doğal bir sonuçtur: her kanal, kendi dosyasında ve kendi konumunda sıralı okuma ya da yazma yapabilir. Bununla birlikte, birer akım olan bazı kanallarda şunlara dikkat etmeniz gerekir:

- Dosyanın aynı parçasında okuma ya da yazma yapacaksanız, birşey yapmadan önce ve kullandıktan sonra çıktı akımını temizlemelisiniz.
- Bir bağımsız kanal kullanılarak değiştirilmekte olan verinin okunmasından önce bir girdi akımını temizlemelisiniz. Aksi takdirde, akımın tamponunda bulunan atıl veriyi okuyabilirsiniz.

Bir dosyanın sonunda tek bir kanala çıktılama yaparsanız, diğer bağımsız kanalların dosya sonuna göre yaptıkları konumlamayı alakasız bir noktaya taşımış olacaktır. Sizin dosya konumlaması yapmanız ile dosyaya yazmanız arasında başka bir sürecin aynı şeyi yapmasını engelleyemediğiniz gibi sizinde bunun tersine olarak onların dosya konumlarını doğru yere kaydıramazsınız. Ama tanıtıcı ya da akımı dosya sonuna ekleme yapan türde açarsanız, daima o anki dosya sonuna ekleme yaparsınız. Dosya sonu konumlamasını doğru yapabilmek için çıktı kanalını, kanal bir akımsa temizlemelisiniz.

Rasgele erişimi desteklemeyen bir dosya için farklı dosya konumları olan iki kanal açmak mümkündür. Bu şekilde dosyaya okuma ve yazma amacıyla açılan kanallar bağımsız açılsa bile ilintilidir. Dosya sonuna ekleme türünde açılan kanallar daima ilintilidir. Bu kanallar için *ilintili kanallarla ilgili kurallar* (sayfa: 316) uygulanır.

5.3. Akımların Temizlenmesi

GNU sisteminde, herhangi bir akımı **fclean** ile temizleyebilirsiniz:

```
int fclean(FILE *akım)                                                    işlev
```

akım akımını temizler yani tamponunu boşaltır. *akım* çıktılama yapıyorsa, çıktılamaya zorlar; giriş yapıyorsa, tampondaki veriyi yeniden okumak için düzenleyerek sisteme iade eder.

Diğer sistemlerde, çoğu durumda bir akımı temizlemek için **fflush** kullanabilirsiniz.

Bir akımın zaten temiz olduğunu biliyorsanız **fclean** veya **fflush** çağrısı yapmayabilirsiniz. Örneğin, bir tamponsuz akım daima temizdir. Dosya sonundaki bir girdi akımı daima temizdir. Son çıktılanan karakter bir satırsonu karakteri ise bir satır tamponlu akım daima temizdir. Bununla birlikte, açıldığı anda bir girdi akımı, girdi tamponu boş olmayabileceğinden temiz olmayabilir.

Çoğu sistemde bir akımı temizlemenin mümkün olmadığı bir durum vardır. Bu, rasgele erişimli olmayan bir dosyadan girdi yapan bir akımın varlığıdır. Böyle akımlar genellikle sürekli okur ama dosya rasgele erişimli olmadığında, okunmuş olan veriye tekrar erişmenin bir yolu yoktur. Bir girdi akımı bir rasgele erişimli dosyadan okuduğu zaman, **fflush** akımı temizler ama dosya konumlayıcıyı alakasız bir yerde bırakır; bu bakımdan, herhangi bir G/Ç işlemi yapmadan öce dosya konumlayıcıyı doğru yere ayarlamalısınız. GNU sisteminde **fclean** kullandığınızda her iki sorun da ortaya çıkmaz.

Sadece çıktılama yapan bir akımın kapatılması da **fflush** yapar, dolayısıyla bir çıktı akımının temizlenmesinde bu yöntem de kullanılabilir. GNU sisteminde bir girdi akımının kapatılması **fclean** yapar.

Uçbirim kipini ayarlamak gibi denetim işlemleri için tanıtıcısını kullanmadan önce bir akımı temizleyemezsiniz; bu işlemler dosya konumunu etkilemez ve dosya konumundan etkilenmez. Bu işlemler için her tanımlayıcıyı kullanabilirsiniz ve tüm kanallar aynı anda etkilenir. Bununla birlikte, metin zaten bir akıma çıktılanır ama hemen ardından boşaltılırken yeni uçbirim kiplerine konu olacak akım tarafından hala tamponlu olur. Yapılan çıktılamanın o anda etkili olan uçbirim ayarları tarafından kapsandığından emin olmak için kipi ayarlanmadan önce uçbirimin çıktı akımları boşaltılır. Bkz. *Uçbirim Kipleri* (sayfa: 444).

6. G/Ç'yı Hızlı Dağıtıp Toplama

Bazı uygulamalarda bellekte ayrı yerlerde duran çok sayıda tampona yazmak ya da okumak gerekebilir. Bu işlem çok sayıda **read** ve **write** çağrısı ile kolayca yapılabildiği halde, her çekirdek çağrısının sabit bir maliyeti olduğundan bu verimli olmaz.

Bunun yerine, çoğu platformada tek bir çekirdek çağrısında her iki işlemi birlikte yapan yüksek hızlı özel ilkeller vardır. GNU C kütüphanesi bu ilkellerin her sistemde benzetilmesini sağlayarak bunların taşınabilirliğe konu olmamasını sağlamıştır. Bu ilkeller `sys/uio.h` başlık dosyasında tanımlıdır.

Bu işlevler, her tamponun boyutunun ve konumunun belirtildiği **iovec** yapılarından oluşan bir diziyle çalışırlar.

`struct` **iovec** veri türü

iovec yapısı bir tampon ile ilgili bilgileri içerir. İki alanı vardır:

`void *`**iov_base**
: Tamponun adresidir

`size_t` **iov_len**
: Tamponun uzunluğudur.

```
ssize_t readv(int                  dosyatanıtıcı,
              const struct iovec *vektör,
              int                  sayı)
```
işlev

readv işlevi veriyi *dosyatanıtıcı* tanıtıcısından okuyup *sayı* yapılık *vektör* dizisindeki tamponlara bir tampon dolduktan sonra diğerine geçerek dağıtır.

readv işlevinin tamponların tümünü dolduracağı garanti edilmemiştir. **read** işlevinde olduğu gibi aynı sebeplerle işlem bir noktada durabilir.

Normal dönüş değeri okunan (tamponlara yazılan değil) baytları sayısıdır. 0 değeri dosya sonunu belirtir. –1 değeri ise bir hata saptandığını gösterir. Olası hatalar **read** işlevindekilerle aynıdır.

```
ssize_t writev(int                  dosyatanıtıcı,
               const struct iovec *vektör,
               int                  sayı)
```
işlev

writev işlevi veriyi *sayı* yapılık *vektör* dizisindeki tamponlardan onları sırayla okuyarak toplayıp *dosyatanıtıcı* tanıtıcısına yazar.

readv gibi, **writev** işlevi de **write** işlevindeki aynı koşullarda işlemin ortasında durabilir.

Normal dönüş değeri yazılan baytların sayısıdır. –1 değeri hata saptandığını belirtir. Olası hatalar **write** işlevindekilerle aynıdır.

Bu tamponların küçük (1kB'ın altında) olması halinde yüksek seviyeli akımların bu işlevlerden daha hızlı olacaklarını unutmayın. Bununla birlikte, **readv** ve **writev** işlevleri tamponların büyük olması (toplam çıktının değil) durumunda daha verimlidir. Bu durumda bir yüksek seviyeli akımın veriyi verimli olarak arabelleklemesi mümkün olmazdı.

7. Bellek Eşlemli G/Ç

Günümüz işletim sistemlerinde, bir dosyayı bir bellek bölgesine eşlemek mümkündür. Bu yapıldığında dosyaya yazılım içinden bir dizi gibi erişilebilir.

Yazılımın bir dosyanın sadece yüklü bölümlerine erişim anlamında bu işlem **read** veya **write** ile yapılan işlemlerden daha verimlidir. Henüz belleğe yüklenmemiş parçalara bellek sayfalarının takaslanmasına benzer bir yolla erişilir.

Belleğe eşlenmiş sayfalar, fiziksel bellek azaldığında tekrar dosyasında saklanabilir, belleğe eşlenmiş dosyaların boyutları büyüdükçe hem fiziksel bellekte hem de takas alanında eşlenebilir. Tek sınır adres alanıdır. Teorik sınır 32 bitklik makinalarda 4GB'dır. Başka amaçlarla ayrılan alanlardan dolayı gerçek sınır daha küçük olabilir. LFS arayüzü kullanan 32 bitlik dosya sistemlerinde dosya boyu 64 bitlik olabildiğinden 2GB ile sınırlı değildir (konumların işaretli tamsayılar olması halinde adreslenebilir alan 4GB'ın yarısına düşebilir).

Bellek eşleme, bellekte sadece sayfalarla çalışır. Bu bakımdam, eşleşme adresleri sayfalara hizalanmış ve uzunluk değerleri de buna göre yuvarlanmış olmalıdır. Makinada kullanılan sayfa boyutlarını saptamak için

```
size_t page_size = (size_t) sysconf (_SC_PAGESIZE);
```

kullanılabilir. Bu işlevler `sys/mman.h` başlık dosyasında bildirilmiştir.

```
void *mmap(void  *adres,                                                    işlev
           size_t uzunluk,
           int    izinler,
           int    seçenekler,
           int    dosyatanıtıcı,
           off_t  konum)
```

mmap işlevi *dosyatanıtıcı* ile açılan dosya için yeni bir eşlem oluşturur. Eşlem (*konum*)'dan başlar, (*konum* + *uzunluk* – 1) de biter. *dosyatanıtıcı* ile belirtilen dosya için dosya kapatıldığında bile kaldırılmayan yeni bir referans oluşturur.

adres ile eşlem için tercih edilen adres belirtilir. **NULL** tercih belirtilmediğini gösterir. Adreste evvelce bir eşlem varsa özdevinimli olarak kaldırılır. **MAP_FIXED** seçeneği kullanmasanızda belirttiğiniz adres yine de değiştirilmeyebilir.

izinler ne çeşit erişime izin verildiğini belirten seçenekleri içerir. Bunlar sırayla okuma, yazma ve çalıştırma izinlerini belirten **PROT_READ**, **PROT_WRITE** ve **PROT_EXEC** seçenekleri olabilir. Belirtilmeyen bir iznin kullanılmaya çalışılması bir parçalama arızasına (segfault) yol açacaktır (bkz. *Yazılım Hatalarının Sinyalleri* (sayfa: 604)).

Çoğu donanım tasarımının okuma izni olmadan yazma iznini desteklemediğini ve çoğunun okuma ve çalıştırma izinleri bakımından bir ayrım yapmadığını aklınızdan çıkarmayın. Bu bakımdan, istediğinizden daha geniş yetkilere sahip olabileceğiniz gibi sadece yazılabilir olarak belirttiğiniz dosyalara **PROT_READ** kullanmadığınız takdirde erişemeyebileceğinizi de unutmayın.

seçenekler eşlemin doğasını denetleyen seçenekleri içerir. Burada **MAP_SHARED** veya **MAP_PRIVATE** seçeneklerinden biri belirtilmelidir.

Bunlar:

`MAP_PRIVATE`
: Eşlemle ilgili dosyaya yazılamasa da bellek bölgesine yazılabileceğini belirtir. Bunun yerine, süreç için bir kopya yapılır ve normal olarak bellek azsa bölge takaslanır. Başka hiçbir süreç değişiklikleri görmez.

  Özel eşlemler üzerlerine yazıldıklarında fiilen sıradan belleğe konulduğundan eğer bu kipi **`PROT_WRITE`** ile kullanıyorsanız eşlemlenmiş bölgenin tamamının kopyası için yeterli sanal bellek olmalıdır.

`MAP_SHARED`
: bölgeye yapılan yazmaların dosyaya da yazılacağını belirtir. Değişiklikler aynı dosyayı eşlemlemiş olan diğer süreçlerle anında paylaşılacaktır.

  Asıl yazma işlemlerinin herhangi bir anda olabileceğini unutmayın. Diğer süreçlerin geleneksel G/Ç işlemlerini kullanarak dosyanın tutarlı bir görünümünü almaları önemliyse, aşağıda açıklanan **`msync`** işlevini kullanmanız gerekir.

`MAP_FIXED`
: Sistemin eşlem adresi olarak *adres* ile belirtilen adresi kullanması için zorlar. Bu olmazsa işlev başarısız olur.

`MAP_ANONYMOUS`
`MAP_ANON`
: Sisteme, bir dosyaya bağlı olmayan bir anonim eşlem oluşturmasını söyler. *dosyatanıtıcı* ve *konum* yoksayılır ve bölge sıfırlarla ilklendirilir.

  Anonim eşlemler bazı sistemlerde özdevimli ayırma yapılabilen bellek bölgesinin genişletilmesinde temel ilkel olarak kullanılır. Ayrıca bir dosya oluşturmadan çok sayıda görev arasında veri paylaşmak için de kullanışlıdır.

  Bazı sistemlerde büyük bellek blokları ile çalışırken özel anonim eşlemleri kullanmak **`malloc`** kullanmaktan daha verimlidir. GNU C sisteminde **`malloc`** gerektiği takdirde özdevinimli olarak **`mmap`** işlevini kullanır.

`mmap` normalde yeni eşlemin adresi ile döner. –1 dönüş değeri bir hata oluştuğunu belirtir.

Olası hatalar:

`EINVAL`
: Ya *adres* işe yaramaz ya da belirtilen *seçenekler* tutarsız

`EACCES`
: *dosyatanıtıcı* belirtilen *izinler* ile açılmadı

`ENOMEM`
: Ya işlem için yeterli bellek yok ya da süreç, adres alanı dışında

`ENODEV`
: Bu dosya belleğe eşlemlenen türde değil

`ENOEXEC`
: Dosya belleğe eşlemlemeyi desteklemeyen bir dosya sisteminde

```
void *mmap64(void   *adres,                                                    işlev
             size_t  uzunluk,
             int     izinler,
             int     seçenekler,
             int     dosyatanıtıcı,
             off64_t konum)
```

mmap64 işlevi *konum* parametresinin **off64_t** türünde olması dışında **mmap** işlevine eşdeğerdir. 32 bitlik sistemlerde bu, tanıtıcısı *dosyatanıtıcı* olan dosyanın 2GB'dan büyük olabilmesini mümkün kılar. *dosyatanıtıcı*, **open64** veya **fopen64** ve **freopen64** çağrılarından dönen bir tanıtıcı olmalıdır.

Kaynak dosyaları **_FILE_OFFSET_BITS == 64** ile derlenmişse bu işleve **mmap** ismiyle erişilir. Yani, derleme sırasında 64 bitlik yeni arayüz eski arayüzün yerine geçer.

```
int munmap(void  *adres,                                                       işlev
           size_t uzunluk)
```

munmap işlevi (*adres*)'ten (*adres* + *uzunluk*)'a kadar olan eşlemi siler. *uzunluk* eşlemin uzunluğu olmalıdır.

Aralık içinde eşlenmemiş alanlar olabilir, bu şekilde çok sayıda eşlem tek bir komutla silinebilir. Ayrıca mevcut eşlemin sadece bir bölümünü silmek de mümkündür. Ancak sadece tam sayfalar silinebilir. *uzunluk* sayfa sayısına denk değilse bile üste yuvarlanır.

Normal dönüş değeri sıfırdır, –1 bir hata belirtir.

Olası hata:

EINVAL
: Belirtilen bellek adresi aralığı kullanıcının mmap aralığının dışında ya da sayfa hizalı değil

```
int msync(void  *adres,                                                        işlev
          size_t uzunluk,
          int    seçenekler)
```

Paylaşımlı eşlemler kullanılırken, silmeden önce çekirdek herhangi bir zamanda eşlemi dosyaya yazabilir. Değişmiş bir veri varsa, dosyaya bellek eşlemli olmayan G/Ç ile erişen süreçler veri dosyaya fiilen yazılana kadar bunları göremezler. Bunun olmaması için bu işlevi kullanmak gerekir.

İşlev *adres*'ten (*adres* + *uzunluk*)'a kadar olan bölgede çalışır. Bu bölge çok sayıda dosyanın eşlemlerini içerebileceği gibi bir dosyanın bir bölümünü içeren bir eşlem bölgesi de olabilir.

seçenekler şunları içerebilir:

MS_SYNC
: Bu seçenek verinin "disk"e yazılmasını sağlar. Normalde **msync** işlevi geleneksel G/Ç işlemleri ile bir dosyaya erişimde son yapılan değişikliklere dosyada mevcutmuş gibi erişilmesini sağlar.

MS_ASYNC
: **msync**'e eşzamanlamaya başlamasını söyler ama başlaması için beklemez.

msync normalde sıfır ile hata oluştuğunda –1 ile döner. Olası hatalar:

EINVAL
: Ya geçersiz bir bölge belirtilmiş ya da *seçenekler* geçersiz

EFAULT

Bir bellek eşlemin parçası olarak bile bir eşlem yok

```
void *mremap(void  *adres,                                                  işlev
             size_t uzunluk,
             size_t yeni_uzunluk,
             int    seçenek)
```

Bu işlev mevcut bir bellek alanının boyunu değiştirmekte kullanılabilir. *adres* ve *uzunluk* tamamı aynı **mmap** deyiminde eşlemlenmiş bir bölgeyi belirtmelidir. *yeni_uzunluk* ile belirtilen yeni eşlem aynı karakteristik özelliklerle dönecektir.

Tek bir seçenek, **MREMAP_MAYMOVE** belirtilebilir. Bu, *seçenek* içinde belirtilmişse, sistem mevcut eşlemi silip başka bir yerde belirtilen uzunlukta yeni bir eşlem oluşturur.

Normalde yeni eşlemin adresi ile bir hata oluştuğunda ise –1 ile döner. Olası hatalar:

EFAULT
: Özgün eşlemin parçası olarak bile bir eşlem yok veya bölge birden fazla farklı eşlem içeriyor

EINVAL
: Belirtilen adres hizalı değil ya da uygunsuz

EAGAIN
: Bölge kilitli sayfalar içeriyor, eğer genişletmek gerekirse kilitli sayfalar için *sürecin özkaynak sınırı* (sayfa: 575) aşılabilir.

ENOMEM
: Bölge yazılabilir ama özel, ayrıca genişletmek için sanal bellek yetersiz. Bundan başka, **MREMAP_MAYMOVE** belirtilmemişse ve genişletme başka bir eşlemli bölge ile çatışacaksa bu hata oluşacaktır.

Bu işlev sadece bir kaç sistemde kullanılabilir. İsteğe bağlı eniyilemeler gerçekleştirmek dışında bu işlev kullanılmamalıdır.

Tüm dosya tanıtıcılar bellek eşlemli olamaz. Soketler, borular ve çoğu aygıta sadece ardışık erişim mümkündür ve ayrılabilecek bir eşleme sığmaz. Ek olarak, bazı normal dosyalar da eşlemlenemez ve eski çekirdekler eşlemlemeyi hiç desteklemeyebilir. Bu bakımdan bu işlevi kullanacak yazılımların işlev başarısız olduğunda kullanılacak bir sonçare yöntemi olmalıdır. Bkz. .

```
int madvise(void  *adres,                                                   işlev
            size_t uzunluk,
            int    öneri)
```

Bu işlev, sisteme *adres*'tan başlayan *uzunluk* baytlık bellek bölgesinin düşünülen kullanım şekliyle ilgili bir öneri yapmak için kullanılabilir.

öneri için geçerli BSD değerleri şunlardır:

MADV_NORMAL
: Bölge özel birşey yapmadan alınmalı.

MADV_RANDOM
: Bölge rasgele sayfa referansları üzerinde erişilebilir olacak. Çekirdek her sayfalama hatası için en az sayıda sayfayı gerçek belleğe sayfalamalıdır.

MADV_SEQUENTIAL

Bölge ardışık sayfa referansları üzerinde erişilebilir olacak. Bu çekirdeğin bölge içindeki her sayfalama hatasından sonra bir ardışık referans umarak sürekli ileri okumaya zorlanmasına sebep olur.

`MADV_WILLNEED`
Bölge gerekli olacaktır. Bu bölge içindeki sayfalar çekirdek tarafından önceden gerçek belleğe sayfalanmış olmalıdır.

`MADV_DONTNEED`
Bölge artık gerekmeyecektir. Sayfa kaybına sebep olabilecek bir değişiklikte ya da sanal belleğe kopyalanması gerektiğinde bunun yapılmaması suretiyle çekirdek bu sayfaları serbest bırakabilir.

POSIX isimleri birazcık farklıdır ama isimler aynı anlama gelir:

`POSIX_MADV_NORMAL`
BSD'nin **`MADV_NORMAL`** seçeneğine karşılıktır.

`POSIX_MADV_RANDOM`
BSD'nin **`MADV_RANDOM`** seçeneğine karşılıktır..

`POSIX_MADV_SEQUENTIAL`
BSD'nin **`MADV_SEQUENTIAL`** seçeneğine karşılıktır.

`POSIX_MADV_WILLNEED`
BSD'nin **`MADV_WILLNEED`** seçeneğine karşılıktır.

`POSIX_MADV_DONTNEED`
BSD'nin **`MADV_DONTNEED`** seçeneğine karşılıktır.

Normalde **`msync`** sıfır ile döner, –1 bir hata saptandığını belirtir. Olası hatalar:

`EINVAL`
Ya belirtilen bölge ya da *öneri* geçersiz

`EFAULT`
Belirtilen bölgenin bir parçası olabilecek kadar bile eşlem yok

8. Girdi ve Çıktının Beklenmesi

Bazan bir yazılımın girdiyi gelişine bağlı olarak çok sayıda girdi kanalından kabul etmesi gerekir. Örneğin bazı iş istasyonları tablet, işlev düğmeleri kutusu, normal eşzamansız seri arayüz üzerinden bağlanılan çevirmeli ağ gibi çok sayıda aygıttan aynı anda ve anında yanıt verecek iyi bir kullanıcı arayüzü gerektirmektedir. Başka bir örnek de bir yazılımın başka süreçlere borular ve FIFO'lar üzerinden bir sunucu olarak hizmet vermesidir.

Normalde bu amaçla **`read`** kullanamazsınız, çünkü işlev bir dosya tanıtıcıdan bir girdi gelene kadar beklerken diğer kanallardaki girdi fazladan bekletilir. Engellenmeyen kipe geçmeniz ve dosya tanıtıcılarını sürekli taramanız gerekir ki bu da pek verimli değildir.

Daha iyi bir çözüm **`select`** işlevini kullanmaktır. Bu, belirtilen dosya tanıtıcı kümesinde bir girdi ya da çıktı hazır olana kadar ya da bir zamanlayıcı zamanaşımına uğrayıncaya kadar (hangisi önce gerçekleşirse), bekler. Bu oluşum `sys/types.h` başlık dosyasında bildirilmiştir.

Bir *sunucu soketi* (sayfa: 423) için konuşursak, askıda *kabul edilmiş bağlantılar* (sayfa: 424) olduğunda girdinin var olabileceğinden söz edilebilir. **`accept`** işlevi sunucu soketini beklemeye alır ve **`read`** işlevinin normal girdi için yaptığı gibi **`select`** ile etkileşir.

select işlevi için dosya tanıtıcı kümeleri **fd_set** nesneleri olarak belirtilir. Burada bu nesnelerin veri türü ve bunlarla ilgili makrolara değinilecektir.

fd_set veri türü

fd_set veri türü **select** işlevinin üzerlerinde işlem yaptığı dosya tanıtıcılarının kümesini içerir. Aslında bir bit dizisidir.

`int` **FD_SETSIZE** makro

fd_set nesnesinde saklanabilen dosya tanıtıcılarının azami sayısıdır. Azami sayının sabit olduğu sistemlerde **FD_SETSIZE** bu sayıya eşittir. GNU sistemininde dahil olduğu bazı sistemlerde açık tanıtıcıların azami sayısı için mutlak bir üst sınır olmamasına rağmen **fd_set** içindeki bitlerin sayısını belirten sabit bir değerdir; **FD_SETSIZE**'ıncıdan sonra bir dosya tanıtıcıyı daha **fd_set** içine koyamazsınız.

```
void FD_ZERO(fd_set *küme)
```
makro

Bu makro *küme* dosya tanıtıcı kümesini boş bir küme olarak ilklendirir.

```
void FD_SET(int      dosyatanıtıcı,
            fd_set *küme)
```
makro

Bu makro *dosyatanıtıcı* dosya tanıtıcısını *küme* dosya tanıtıcı kümesine dahil eder.

dosyatanıtıcı parametresi birden fazla değerlendirildiğinden yan etkilere sahip olmamalıdır .

```
void FD_CLR(int      dosyatanıtıcı,
            fd_set *küme)
```
makro

Bu makro *dosyatanıtıcı* dosya tanıtıcısını *küme* dosya tanıtıcı kümesinden kaldırır.

dosyatanıtıcı parametresi birden fazla değerlendirildiğinden yan etkilere sahip olmamalıdır .

```
int FD_ISSET(int            dosyatanıtıcı,
             const fd_set *küme)
```
makro

Bu makro *dosyatanıtıcı* dosya tanıtıcısını *küme* dosya tanıtıcı kümesinin bir üyesi ise sıfırdan farklı bir değerle aksi takdirde sıfırla döner.

dosyatanıtıcı parametresi birden fazla değerlendirildiğinden yan etkilere sahip olmamalıdır .

```
int select(int             dtsayısı,
           fd_set          *oku–dt,
           fd_set          *yaz–dt,
           fd_set          *diğer–dt,
           struct timeval *süre)
```
işlev

select işlevi, çağrıldığı süreci belirtilen dosya tananıtıcı kümesindeki tanıtıcılarda bir etkinlik olana kadar ya da belirtilen zamanaşımı süresi dolana kadar bekletir.

oku–dt argümanı ile okumaya hazır tanıtıcılar, *yaz–dt* argümanı ile yazmaya hazır tanıtıcılar belirtilir. *diğer–dt* ile belirtilen tanıtıcılar ise olağandışı durumlara göre denetlenir. İlgilenmediğiniz durumla ilgili olan argümana boş gösterici atayabilirsiniz.

Bir dosya gösterici eğer bir **read** çağrısı engellenmeyecekse okumaya hazır olarak kabul edilir. Engellenme durumları olarak okuma başlangıcının dosyanın sonunda olması veya raporlanacak bir hatanın varlığından bahsedilebilir. Bir sunucu soketi de **accept** ile *kabul edilebilen bir bağlantı* (sayfa: 424) askıdaysa okumaya hazır kabul edilir. Bir istemci soketi ise *bağlantı tamamen kurulduğunda* (sayfa: 422) yazmaya hazır olur.

"Olağandışı durumlar" hata anlamında değildir; hatalar oluştuğunda sistem çağrıları tarafından raporlanır ve bunların tanıtıcının durumu ile ilgisi yoktur. Olağandışı durumlar bir soket üzerinde acil bir iletinin varlığı gibi durumlardır. (Acil iletiler hakkında *Soketler* (sayfa: 398) bölümünde bilgi bulabilirsiniz.)

select işlevi sadece ilk *dtsayısı* dosya tanıtıcısını denetler. *dtsayısı* olarak **FD_SETSIZE** çok kullanışlıdır.

süre azami bekleme süresini belirtir. Bir boş gösterici belirtmişseniz bir süre sınırı olmaksızın bir dosya tanıtıcı hazır olana kadar işlev bekleyecektir. Bunun olmaması için **struct timeval** biçiminde bir *zamanaşımı süresi belirtmelisiniz* (sayfa: 543). Beklemeden hangi dosya tanıtıcıların hazır olduğuna bakmak isterseniz buraya süre olarak sıfır (**struct timeval**'ın üyelerinin hepsi sıfır) belirtebilirsiniz.

İşlevin normal dönüş değeri tüm kümelerde hazır olan dosya tanıtıcıların sayısıdır. Küme argümanlarının her birinde hazır olan tanıtıcılarla ilgili bilgi bulunur. **select** döndükten sonra belli bir dosya tanıtıcının girdi için hazır olup olmadığını
FD_ISSET (*dosyatanıtıcı***,** *oku–dt***)** ile öğrenebilirsiniz.

select zamanaşımına uğramışsa sıfır ile döner.

Herhangi bir sinyal **select** işlevinin anında dönmesine sebep olur. Yazılımınızda sinyaller kullanılıyorsa belirttiğiniz zamanaşımı süresince işlevin beklemede kalması mümkün olmayabilir. Bu sürenin mutlaka beklenmesini istiyorsanız **EINTR** durumunun varlığına bakarak o anki zaman değeri ile karşılaştırarak yeni bekleme süresini hesaplayıp çağrıyı yinelemelisiniz. Bunun bir örneği aşağıda verilmiştir, ayrıca *Sinyallerle Kesilen İlkeller* (sayfa: 626) bölümüne de bakın.

Bir hata oluşursa işlev –1 ile döner ve dosya tanıtıcı kümesi argümanlarında bir değişiklik yapmaz. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: Dosya tanıtıcı kümelerinden biri geçersiz bir dosya tanıtıcı içeriyor

EINTR
: İşlem bir sinyalle durduruldu. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

EINVAL
: *süre* argümanı geçersiz; üyelerinden biri ya negatif ya da çok büyük

Taşınabilirlik Bilgisi

select işlevi bir BSD Unix özelliğidir.

Bu örnekte bir dosya tanıtıcısının okumaya hazır olmasını belli bir süre beklemek için **select** işlevinin kullanımı gösterilmiştir. **input_timeout** işlevi çağrıldığı süreci dosya tanıtıcı üzerinde bir girdi olana kadar ya da belli bir zamanaşımına kadar bekletir.

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>

int
input_timeout (int filedes, unsigned int seconds)
{
  fd_set set;
```

```
  struct timeval timeout;

  /* Dosya tanıtıcı kümesini ilklendirelim. */
  FD_ZERO (&set);
  FD_SET (filedes, &set);

  /* Zamanaşımı yapısını ilklendirelim. */
  timeout.tv_sec = seconds;
  timeout.tv_usec = 0;

  /* select, zamanaşımına uğrarsa 0 ile,
     girdi varsa 1 ile, hata oluşursa -1 ile döner. */
  return TEMP_FAILURE_RETRY (select (FD_SETSIZE,
                                     &set, NULL, NULL,
                                     &timeout));
}

int
main (void)
{
  fprintf (stderr, "select %d ile döndü.\n",
           input_timeout (STDIN_FILENO, 5));
  return 0;
}
```

select işlevinin çok sayıda soketten çoklu girdi alınması ile ilgili kullanım örneğini *Bayt Akımlı Bağlantı Sunucusu Örneği* (sayfa: 429) bölümünde bulabilirsiniz.

9. G/Ç İşlemlerinin Eşzamanlanması

Günümüzdeki çoğu işletim sisteminde normal G/Ç işlemleri eşzamanlı yapılmaz. Örneğin, bir **write** çağrısı normal olarak dönse bile bu, verinin ilgili ortama (örn, disk) yazılmış olduğu anlamına gelmez.

Eşzamanlamanın gerektiği durumlarda, işlev dönmeden önce tüm işlemlerin tamamlanmış olduğundan emin olunmasını sağlayan özel işlevler vardır.

`int` **sync**`(void)` işlev

Bu işleve yapılacak bir çağrı, veri aygıta yazılana kadar dönmez. Çekirdekte içinde veri bulunan tüm tamponlar boşaltılır (veri yerine yazıldıktan sonra tampon silinir), böylece sistemin tamamı tutarlı duruma gelir (veriyi paralel yazan bir süreç yoksa).

sync işlevinin prototipi `unistd.h` başlık dosyasında bulunur.

Normal dönüş değeri sıfırdır.

Yazılımlar çoğunlukla sistemdeki tüm verinin değil, bir dosya ile ilgili bir verinin o dosyaya yazıldığından emin olmak ister. Bu bakımdan **sync** fazla gelir.

`int` **fsync**`(int` *dosyatanıtıcı*`)` işlev

fsync işlevi yazmak amacıyla açılmış ve tanıtıcısı *dosyatanıtıcı* olan dosyaya tüm veri fiziksel olarak yazılıncaya kadar dönmez.

fsync işlevinin prototip `unistd.h` başlık dosyasında bulunur.

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **fsync** çağrısı sırasında evre bazı özkaynakları (bellek, dosya tanıtıcı, semafor, vb.) ayırdığında bu bir sorun olur. Evre tam bu anda bir iptal alırsa ayrılan özkaynaklar yazılım sonlanana kadar ayrılmış olarak kalır. Bu tür **fsync** çağrılarından kaçınmak için iptal eylemcileri kullanılarak korunulmalıdır.

İşlevin normal dönüş değeri sıfırdır, bir hata oluşmuşsa –1 ile döner. Aşağıdaki *errno* hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* tanıtıcısı geçersiz

EINVAL
: Sistemde ilgili oluşum gerçeklenmediğinden eşzamanlama mümkün değil

Bazan bir dosya tanıtıcı ile ilgili verinin tamamını yazmak gerekmez. Örneğin, veritabanı dosyalarına veri yazarken dosya boyutu değişmeyeceğinden dosyanın içerdiği verinin aygıta yazılması yeterlidir. Dosya ile ilgili değişiklik zamanı gibi temel veriler önemli değildir ve bu bilgilerin olduğu gibi bırakılması bir sorun çıktığında dosyanın başarıyla kurtarılmasını engellemez.

`int` **fdatasync**`(int` *dosyatanıtıcı*`)` işlev

fdatasync çağrıldığında, dosya verisinin tamamı aygıta yazılmadan dönmez. Bekleyen tüm G/Ç işlemleri için parçalar veri bütünlüğünü sağlayacak şekilde birleştirilir.

Tüm sistemler **fdatasync** işlemini gerçekleştirmez. Bu işlevselleğin olmadığı sistemlerde **fdatasync** işlemleri, gerçekleştirilen eylemler **fdatasync** için gereken işlemlerin bir üst kümesi olarak bir **fsync** çağrısı ile benzeştirilir.

fdatasync işlevinin prototip `unistd.h` başlık dosyasında bulunur.

İşlevin normal dönüş değeri sıfırdır, bir hata oluşmuşsa –1 ile döner. Aşağıdaki *errno* hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* tanıtıcısı geçersiz

EINVAL
: Sistemde ilgili oluşum gerçeklenmediğinden eşzamanlama mümkün değil

10. Eşzamansız G/Ç

POSIX.1b standardı G/Ç işlemlerinde beklemelerden kaynaklanan zaman kaybını en aza indiren yeni bir G/Ç işlemleri kümesi tanımlar. Yeni işlevler bir yazılımın birden fazla G/Ç işlemini ilklendirmesini ve G/Ç işlemlerini birarada (paralel) gerçekleştirdikten hemen sonra yazılımın normal çalışmasına dönmesini mümkün kılar. Bu işlevsellik varsa **unistd.h** dosyasında **_POSIX_ASYNCHRONOUS_IO** sembolü tanımlıdır.

Bu işlevler, gerçek zamanlı işlevler içeren **librt** isimli kütüphanenin bir parçasıdır. Aslında **libc** kodunun parçası değildir. Bu işlevlerin gerçeklenmesi çekirdekteki destek (varsa) kullanılarak ya da kullanıcı seviyesinde evrelere tabanlanmış bir gerçekleme kullanılarak yapılabilir. Son durumda uygulamaları **librt** kütüphanesinden başka **libpthread** evre kütüphanesi ile de ilintilemek gerekir.

Tüm eşzamansız G/Ç işlemleri önceden açılmış dosyalar üzerinde yapılır. Bir dosya üzerinde keyfi manada çok sayıda işlem yapılıyor olabilir. Eşzamansız G/Ç işlemleri **struct aiocb** isimli ("AIO control block" kısaltması) bir veri yapısı kullanılarak denetlenir. **struct aiocb** yapısı `aio.h` başlık dosyasında tanımlıdır.

`struct` **aiocb** veri türü

POSIX.1b standardı **struct aiocb** yapısının en azından aşağıdaki listede açıklanan üyelere sahip olmasını zorunlu kılar. Gerçekleme tarafından kullanılan daha fazla eleman olabilir, ancak bu elemanlara bağımlılık taşınabilir olmayacağından buna şiddetle karşı çıkılır.

`int` **aio_fildes**
: Bu eleman işlem için kullanılan dosya tanıtıcıyı içerir. Geçerli bir tanıtıcı olmalıdır, aksi takdirde işlem başarısız olur.

  Dosya üzerinde açılan aygıt konumlama işlemlerine izin vermelidir. Örneğin, **lseek** çağrılarının hataya yolaçtığı uçbirim benzeri aygıtlar üzerinde eşzamansız G/Ç işlemleri yapılması mümkün değildir.

`off_t` **aio_offset**
: Dosyada işlem (girdi ya da çıktı) yapılacak dosya konumunu belirtir. İşlem keyfi sırada yapıldığından ve bir dosya tanıtıcı üzerinde birden fazla işlem başlatıldığından bunun dosya tanıtıcısının o anki okuma/yazma konumu olduğundan bahsedilemez.

`volatile void *`**aio_buf**
: Verinin yazıldığı ya da okunan verinin saklandığı tampona göstericidir.

`size_t` **aio_nbytes**
: **aio_buf** ile gösterilen tamponun uzunluğudur.

`int` **aio_reqprio**
: Eğer platform **_POSIX_PRIORITIZED_IO** ve **_POSIX_PRIORITY_SCHEDULING** ile tanımlanmışsa, eşzamansız G/Ç istekleri o anki zamanlama önceliğine göre işlenir. **aio_reqprio** elemanı eşzamansız G/Ç işlemini daha düşük önceliğe ayarlamakta kullanılabilir.

`struct sigevent` **aio_sigevent**
: Çağıran sürecin işlem sonlandığında nasıl uyarılacağını belirtir. **sigev_notify** elemanının değeri **SIGEV_NONE** ise uyarı gönderilmez. **SIGEV_SIGNAL** ise **sigev_signo** tarafından saptanan sinyal gönderilir. Aksi takdirde, **sigev_notify** elemanının değeri **SIGEV_THREAD** olmalıdır. Bu durumda, **sigev_notify_function** tarafından gösterilen işlev çalıştırılarak başlatılan bir evre oluşturulur.

`int` **aio_lio_opcode**
: Bu eleman sadece **lio_listio** ve **lio_listio64** işlevleri tarafından kullanılır. Bu işlevler bir kerede birden fazla keyfi işlemin başlatılmasını mümkün kıldığından ve her işlem bir girdi ya da bir çıktı (ya da hiçbir şey) olabildiğinden, bilgi denetim bloğunda saklanmalıdır. Olası değerler şunlardır:

  `LIO_READ`
  : Bir okuma işlemi başlatır. Okuma **aio_offset**'deki konumdan başlar ve okunan ilk **aio_nbytes** bayt **aio_buf** ile gösterilen tamponda saklanır.

  `LIO_WRITE`
  : Bir yazma işlemi başlatır. **aio_buf**'dan başlayan **aio_nbytes** bayt, dosyaya **aio_offset** konumdan itibaren yazılır.

  `LIO_NOP`
  : Hiçbir şey yapılmaz. Bu değer **struct aiocb** dizisi delikler içerdiğinde bazan kullanışlı olur; örneğin, dizinin tamamında elde edilememiş bazı değerlerle **lio_listio** çağrısı yapmak.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu tür aslında **struct aiocb64** yapısına karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

LFS'de tanımlı eşzamansız G/Ç işlevleri ile kullanmak amacıyla tanımlanmış benzer bir veri türü vardır. Bu yapının üyelerinin türleri daha geniş veri türleridir. Bunun dışında her iki yapının üyelerinin isimleri aynıdır.

`struct` **`aiocb64`** veri türü

`int` **`aio_fildes`**
: Bu eleman işlem için kullanılan dosya tanıtıcıyı içerir. Geçerli bir tanıtıcı olmalıdır, aksi takdirde işlem başarısız olur.

  Dosya üzerinde açılan aygıt konumlama işlemlerine izin vermelidir. Örneğin, **`lseek`** çağrılarının hataya yolaçtığı uçbirim benzeri aygıtlar üzerinde eşzamansız G/Ç işlemleri yapılması mümkün değildir.

`off64_t` **`aio_offset`**
: Dosyada işlem (girdi ya da çıktı) yapılacak dosya konumunu belirtir. İşlem keyfi sırada yapıldığından ve bir dosya tanıtıcı üzerinde birden fazla işlem başlatıldığından bunun dosya tanıtıcısının o anki okuma/yazma konumu olduğundan bahsedilemez.

`volatile void *`**`aio_buf`**
: Verinin yazıldığı ya da okunan verinin saklandığı tampona göstericidir.

`size_t` **`aio_nbytes`**
: **`aio_buf`** ile gösterilen tamponun uzunluğudur.

`int` **`aio_reqprio`**
: **`_POSIX_PRIORITIZED_IO`** ve **`_POSIX_PRIORITY_SCHEDULING`** ile tanımlanmışsa, eşzamansız G/Ç istekleri o anki zamanlama önceliğine göre işlenir. **`aio_reqprio`** elemanı eşzamansız G/Ç işlemini daha düşük önceliğe ayarlamakta kullanılabilir.

`struct sigevent` **`aio_sigevent`**
: Çağıran sürecin işlem sonlandığında nasıl uyarılacağını belirtir. **`sigev_notify`** elemanının değeri **`SIGEV_NONE`** ise uyarı gönderilmez. **`SIGEV_SIGNAL`** ise **`sigev_signo`** tarafından saptanan sinyal gönderilir. Aksi takdirde, **`sigev_notify`** elemanının değeri **`SIGEV_THREAD`** olmalıdır. Bu durumda, **`sigev_notify_function`** tarafından gösterilen işlev çalıştırılarak başlatılan bir evre oluşturulur.

`int` **`aio_lio_opcode`**
: Bu eleman sadece **`lio_listio`** ve **`lio_listio64`** işlevleri tarafından kullanılır. Bu işlevler bir kerede birden fazla keyfi işlemin başlatılmasını mümkün kıldığından ve her işlem bir girdi ya da bir çıktı (ya da hiçbir şey) olabildiğinden, bilgi denetim bloğunda saklanmalıdır. Olası değerlerin açıklamaları için **`struct aiocb`** yapısında bu üyenin açıklamasına bakınız.

Kaynak dosyası **`_FILE_OFFSET_BITS == 64`** ile derlenmişse 32 bitlik sistemlerde bu türe **`struct aiocb`** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

10.1. Eşzamansız Okuma ve Yazma İşlemleri

`int` **`aio_read`**`(struct aiocb *`*aiocbp*`)` işlev

Bu işlev bir eşzamansız okuma işlemini ilklendirir. İşlem kuyruğa alındığında ya da bir hata oluştuğunda işlev beklemeksizin döner.

Tanıtıcısı **`aiocbp->aio_fildes`** olan dosyanın **`aiocbp->aio_offset`** bayttan itibaren ilk **`aiocbp->aio_nbytes`** baytı **`aiocbp->aio_buf`**'dan başlayan tampona yazılır.

Öncelikli G/Ç destekleyen platformlarda **aiocbp->aio_reqprio** değeri, istek kuyruğa alınmadan önceki önceliği ayarlamakta kullanılır.

İşlevin çağrıldığı süreç okuma isteğinin sonlaması halinde **aiocbp->aio_sigevent** değerine göre uyarılır.

aio_read işlevinin normal dönüş değeri sıfırdır. İşlem kuyruğa alınmadan önce bir hata oluşmuşsa işlev –1 ile döner ve bu durumda **errno** değişkenine şu değerlerden biri atanır:

EAGAIN
: Özkaynak sınırları (geçici olarak) aşıldığı için istek kuyruğa alınmadı

ENOSYS
: **aio_read** işlevi gerçeklenmedi

EBADF
: **aiocbp->aio_fildes** tanıtıcısı geçersiz. Bu hata durumu isteğin kuyruğa alınmasından önce tanınmamış olabilir ve bu bakımdan bu hata ayrıca eşzamansız olarak sinyallenir.

EINVAL
: **aiocbp->aio_offset** ya da **aiocbp->aio_reqpiro** değeri geçersiz. Bu hata durumu isteğin kuyruğa alınmasından önce tanınmamış olabilir ve bu bakımdan bu hata ayrıca eşzamansız olarak sinyallenir.

aio_read sıfırla dönerse, isteğin o anki durumu **aio_error** ve **aio_return** işlevleri ile sorgulanabilir. **aio_error** işlevinden dönen değer **EINPROGRESS** oldukça işlem henüz tamamlanmamış demektir. Eğer **aio_error** sıfırla dönerse işlem başarıyla bitmiş demektir; aksi takdirde, dönen değer bir hata kodu olarak değerlendirilmelidir. İşlem sonlanmışsa, işlemin sonucu **aio_return** çağrısı ile sağlanabilir. Dönen değer, eşdeğer **read** çağrısından dönen değerle aynıdır. **aio_error** çağrısından dönebilecek olası hata durumları şunlardır:

EBADF
: **aiocbp->aio_fildes** tanıtıcısı geçersiz

ECANCELED
: İşlem bitmeden durduruldu (bkz. *Eşzamansız G/Ç İşlemlerinin İptal Edilmesi* (sayfa: 336))

EINVAL
: **aiocbp->aio_offset** değeri geçersiz

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işlev aslında **aio_read64** işlevine karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

```
int aio_read64(struct aiocb *aiocbp)                                    işlev
```

Bu işlev **aio_read** işlevine çok benzer. Tek fark, 32 bitlik makinalarda dosya tanıtıcısının büyük dosya kipinde açılmış olmasıdır. İçsel olarak, **aio_read64** işlevi okumak için dosya konumlayıcıyı doğru yere konumlandırırken **lseek64** (*Dosya Konumu İlkeli* (sayfa: 313)) işlevselliğini kullanır, benzer şekilde **aio_read** işlevi de **lseek** işlevselliğini kullanır.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **aio_read** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

Veriyi bir dosyaya eşzamansız yazmak için çok benzer bir arayüze sahip eşdeğer bir işlev çifti vardır.

```
int aio_write(struct aiocb *aiocbp)                                     işlev
```

Bu işlev bir eşzamansız yazma işlemini ilklendirir. İşlem kuyruğa alındığında ya da bir hata oluştuğunda işlev beklemeksizin döner.

aiocbp->aio_buf'dan başlayan tampondaki ilk **aiocbp->aio_offset** bayt tanıtıcısı **aiocbp->aio_fildes** olan dosyaya **aiocbp->aio_offset** bayttan itibaren yazılır.

Öncelikli G/Ç destekleyen platformlarda **aiocbp->aio_reqprio** değeri, istek kuyruğa alınmadan önceki önceliği ayarlamakta kullanılır.

İşlevin çağrıldığı süreç yazma isteğinin sonlaması halinde **aiocbp->aio_sigevent** değerine göre uyarılır.

aio_write işlevinin normal dönüş değeri sıfırdır. İşlem kuyruğa alınmadan önce bir hata oluşmuşsa işlev –1 ile döner ve bu durumda **errno** değişkenine şu değerlerden biri atanır:

EAGAIN
: Özkaynak sınırları (geçici olarak) aşıldığı için istek kuyruğa alınmadı

ENOSYS
: **aio_write** işlevi gerçeklenmedi

EBADF
: **aiocbp->aio_fildes** tanıtıcısı geçersiz. Bu hata durumu isteğin kuyruğa alınmasından önce tanınmamış olabilir ve bu bakımdan bu hata ayrıca eşzamansız olarak sinyallenir.

EINVAL
: **aiocbp->aio_offset** ya da **aiocbp->aio_reqpiro** değeri geçersiz. Bu hata durumu isteğin kuyruğa alınmasından önce tanınmamış olabilir ve bu bakımdan bu hata ayrıca eşzamansız olarak sinyallenir.

aio_write sıfırla dönerse, isteğin o anki durumu **aio_error** ve **aio_return** işlevleri ile sorgulanabilir. **aio_error** işlevinden dönen değer **EINPROGRESS** oldukça işlem henüz tamamlanmamış demektir. Eğer **aio_error** sıfırla dönerse işlem başarıyla bitmiş demektir; aksi takdirde, dönen değer bir hata kodu olarak değerlendirilmelidir. İşlem sonlanmışsa, işlemin sonucu **aio_return** çağrısı ile sağlanabilir. Dönen değer, eşdeğer **write** çağrısından dönen değerle aynıdır. **aio_error** çağrısından dönebilecek olası hata durumları şunlardır:

EBADF
: **aiocbp->aio_fildes** tanıtıcısı geçersiz

ECANCELED
: İşlem bitmeden durduruldu (bkz. *Eşzamansız G/Ç İşlemlerinin İptal Edilmesi* (sayfa: 336))

EINVAL
: **aiocbp->aio_offset** değeri geçersiz

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işlev aslında **aio_write64** işlevine karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

`int` **aio_write64**`(struct aiocb *`*aiocbp*`)` işlev

Bu işlev **aio_write** işlevine çok benzer. Tek fark, 32 bitlik makinalarda dosya tanıtıcısının büyük dosya kipinde açılmış olmasıdır. İçsel olarak, **aio_write64** işlevi okumak için dosya konumlayıcıyı doğru yere konumlandırırken **lseek64** (*Dosya Konumu İlkeli* (sayfa: 313)) işlevselliğini kullanır, benzer şekilde **aio_write** işlevi de **lseek** işlevselliğini kullanır.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **aio_write** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

Az ya da çok geleneksel arayüzle bu işlevlerden başka, POSIX.1b bir defada birden fazla karışık okuma ve yazma işlemini ilklendiren bir işlev daha tanımlar. Bu işlev **readv** ve **writev** işlevlerinin bir birleşimi gibidir.

işlev
```
int lio_listio(int                  kip,
               struct aiocb *const  liste[],
               int                  isteksayısı,
               struct sigevent     *sinyal)
```

lio_listio işlevi bir defada birden fazla okuma ve yazma isteğini kuyruğa almakta kullanılabilir. İsteklerin hepsi aynı dosya için, farklı dosyalar için ya da bunlar arasındaki işlemler için olabilir.

lio_listio işlevi *isteksayısı* isteği *liste* ile gösterilen diziden alır. Uygulanacak işlem *liste* dizisinin her elemenındaki **aio_lio_opcode** elemanından saptanır. Bu alandaki değer **LIO_READ** ise dizinin bu elemenına bir **aio_read** çağrısı yapılmış gibi (bir farkla, sonlanma aşağıda belirtileceği gibi farklı bir yolla sinyallenir) okuma işlemi kuyruğa alınır. **aio_lio_opcode** üyesinin değeri **LIO_WRITE** ise yazma işlemi kuyruğa alınır. Bunlar dışında üyenin değeri **LIO_NOP** olmalıdır, bu durumda dizinin bu elemanı basitçe yoksayılır. Yoksayma işlemi, bütün elemenlar için işlem yapılmayacaksa istekleri içeren dizinin eleman sayısını değiştirmeden isteklerin ele alınması için faydalıdır. Başka durumda, **lio_listio** çağrısının tüm istekleri işleme sokmadan durdurulduğu durumdur (bkz. *Eşzamansız G/Ç İşlemlerinin İptal Edilmesi* (sayfa: 336)). Bu durumda işleme sokulmayan istekler yinelenirken işleme sokulmuş istekler yoksayılabilir.

liste ile gösterilen dizinin yoksayılmayan elemanlarının üyeleri, evvelce **aio_read** ve **aio_write** işlevlerinin açıklamasında belirtilen işlemlere uygun değerlere sahip olmalıdır.

kip argümanı **lio_listio** işlevinin tüm istekler kuyruğa alındıktan sonra nasıl davranacağını saptamakta kullanılır. *kip* olarak **LIO_WAIT** belirtilmişse, tüm istekler tamamlanana kadar işlev bekler. Aksi takdirde, *kip* olarak **LIO_NOWAIT** verilebilir ki, bu durumda işlev, işlemler kuyruğa alındıktan hemen sonra işlemlerin bitmesini beklenmeden döner. Bu durumda, işlevi çağıran süreç tüm istekler için *sinyal* ile belirtilen değere bağlı olarak sonlanmaları ile ilgili bir uyarı alır. *sinyal* olarak **NULL** belirtilmişse herhangi bir uyarı gönderilmez. Aksi takdirde ya bir sinyal gönderilir ya da **aio_read** ve **aio_write** işlevlerinin açıklamalarında belirtildiği gibi bir evre başlatılır.

kip değeri **LIO_WAIT** ise ve tüm istekler yerine getirilmişse işlevin dönüş değeri sıfır olur. Aksi takdirde işlev –1 ile döner ve hata durumu **errno** değişkenine atanır. Hangi isteklerin başarısız olduğunu bulmak için *liste* dizisindeki her eleman için bir **aio_error** çağrısı yapılmalıdır.

kip değeri **LIO_NOWAIT** ise tüm istekler düzgün bir şekilde kuyruğa alınmışsa işlevin dönüş değeri sıfır olur. İsteklerin mevcut durumu **aio_error** ve **aio_return** çağrıları ile saptanabilir. İşlev bu kipte –1 ile dönerse, hata durumu **errno** değişkenine atanır. Bir istek henüz sonlanmamışsa bir **aio_error** çağrısı **EINPROGRESS** döndürür. Değer farklı olursa, istek bitmiş demektir, **aio_error** ya bir hata değeri ya da sıfır ile döner, bu durumda işlemin sonucu **aio_return** kullanılarak saptanabilir.

errno için olası değerler şunlardır:

EAGAIN
: Tüm istekleri kuyruğa almak için gereken özkaynaklar şu anda yok. Hangi isteğin başarısız olduğunu bulmak için *liste* dizisinin tüm elemanları için hata durumuna bakmalısınız.

  Bu hatanın başka bir sebebi de eşzamansız G/Ç isteklerinin sistem çapında sınırları aşması olabilirdi. Bu durum GNU sisteminde keyfi sınırlar olmadığından mümkün değildir.

EINVAL

Ya *kip* parametresi geçersiz ya da *isteksayısı* > **AIO_LISTIO_MAX**.

EIO
: Bir ya da daha fazla G/Ç isteği başarısız oldu. Hangi isteğin başarısız olduğunu bulmak için *liste* dizisinin tüm elemanları için hata durumuna bakmalısınız.

ENOSYS
: **lio_listio** işlevi desteklenmiyor

kip değeri **LIO_NOWAIT** ise ve istek iptal edilmişse bu istek için **aio_error** çağrısından dönen hata durumu **ECANCELED** olur.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işlev aslında **lio_listio64** işlevine karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

```
int lio_listio(int                  kip,                                  işlev
               struct aiocb *const  liste,
               int                  isteksayısı,
               struct sigevent     *sinyal)
```

Bu işlev **lio_listio** işlevinin benzeridir. Tek fark, 32 bitlik makinalarda dosya tanıtıcısının büyük dosya kipinde açılmış olmasıdır. İçsel olarak, **lio_listio64** işlevi okumak için dosya konumlayıcıyı doğru yere konumlandırırken **lseek64** (*Dosya Konumu İlkeli* (sayfa: 313)) işlevselliğini kullanır, benzer şekilde **lio_listio** işlevi de **lseek** işlevselliğini kullanır.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **lio_listio** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

10.2. Eşzamansız G/Ç İşlemlerinin Durumu

Geçtiğimiz bölümde değinilen işlevlerin açıklamalarında da belirtildiği gibi, bir G/Ç isteğinin durumu hakkında bilgi edinilmesi mümkün olmalıdır. İşlem gerçekten eşzamansız olarak gerçekleştirildiğinde (**aio_read** ve **aio_write** ile ilgili olarak **lio_listio** işlevinde kip olarak **LIO_NOWAIT** belirtilmesi durumu), bazan bir isteğin sonlanmış olup olmadığı, sonlanmışsa sonucun ne olduğunu bilmek gerekir. Aşağıdaki iki işlev bu çeşit bilgileri almak için tasarlanmıştır.

```
int aio_error(const struct aiocb *aiocbp)                                  işlev
```

Bu işlev *aiocbp* ile gösterilen **struct aiocb** yapısında açıklanan isteğin hata durumunu saptar. İstek henüz sonlanmamışsa daima **EINPROGRESS** döner. İstek sonlandıktan sonra **aio_error** işlevi istek başarıyla tamamlanmışsa sıfır ile döner, aksi takdirde isteğin **read**, **write** ya da **fsync** işlevlerinin sonucu olarak **errno** hata durumlarının karşılığı olan bir değer döner.

İşlev gerçeklenmemişse **ENOSYS** dönebilir. Eğer *aiocbp* parametresi dönüş durumu bilinmeyen bir eşzamansız işlemi belirtiyorsa işlev **EINVAL** değeriyle de dönebilir.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işlev aslında **aio_error64** işlevine karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

```
int aio_error64(const struct aiocb64 *aiocbp)                              işlev
```

Bu işlev, argümanının **struct aiocb64** türünde bir değişken olması dışında **aio_error** işlevine benzer.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **aio_error** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

```
ssize_t aio_return(const struct aiocb *aiocbp)                    işlev
```

Bu işlev *aiocbp* ile gösterilen **struct aiocb** yapısında açıklanan isteğin dönüş durumunu saptar. **aio_error** işlevinin bu istek için **EINPROGRESS** döndürdüğü durumda bu işlevin dönüş değeri tanımsızdır.

İstek bittikten hemen sonra bu işlev dönüş durumunu saptamak için kullanılır. Aşağıdaki çağrılar tanımlanmamış bir davranışla sonuçlanabilir. Dönüş değeri yapılan işleme göre **read**, **write** ya da **fsync** çağrılarının döndürdüğü değerdir.

İşlev gerçeklenmemişse **ENOSYS** dönebilir. Eğer *aiocbp* parametresi dönüş durumu bilinmeyen bir eşzamansız işlemi belirtiyorsa işlev **EINVAL** değeriyle de dönebilir.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işlev aslında **aio_return64** işlevine karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

```
ssize_t aio_return64(const struct aiocb64 *aiocbp)                işlev
```

Bu işlev, argümanının **struct aiocb64** türünde bir değişken olması dışında **aio_return** işlevine benzer.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **aio_return** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

10.3. Eşzamansız G/Ç İşlemlerinin Eşzamanlanması

Eşzamansız işlemlerle çalışırken bazan istikrarlı duruma geçmek gerekir. Bu, eşzamansız G/Ç işlemlerinde belli bir isteğin ya da belli bir istek grubunun işlenip işlenmediğinin bilinmesinin istendiği anlamına gelir. Bu, işlem sonlandıktan sonra sistem tarafından gönderilecek bir uyarı beklenerek de yapılabilirdi, fakat bu bazan özkaynakların kirlenmesi anlamına gelir (özellikle hesaplama sırasında). Bunun yerine istikrar gerektiren çoğu durumda yardımcı olmak üzere POSIX.1b iki işlev tanımlamıştır.

aio_fsync ve **aio_fsync64** işlevleri sadece `unistd.h` dosyasında **_POSIX_SYNCHRONIZED_IO** sembolü tanımlıysa kullanılabilir.

```
int aio_fsync(int kip,                                            işlev
              struct aiocb *aiocbp)
```

Bu işlev çağrıldığında **aiocbp->aio_fildes** tanıtıcısı üzerinde işlev çağrısının çalışması esnasında kuyruktaki tüm G/Ç işlemleri eşzamanlı G/Ç tamamlama durumuna (bkz. *G/Ç İşlemlerinin Eşzamanlanması* (sayfa: 326)) sokulmaya zorlanır. **aio_fsync** işlevi beklemeden hemen döner ancak, **aiocbp->aio_sigevent** ile belirtilen yöntem üzerinden uyarı sadece dosya tanıtıcısı sonlandığında ve dosya eşzamanlandığında verilir. Bu ayrıca, eşzamanlama isteğinden sonra aynı dosya tanıtıcısı için yapılan bu isteklerin etkili olmadığı anlamına gelir.

kip değeri **O_DSYNC** ise eşzamanlama bir **fdatasync** çağrısı olarak, **O_SYNC** ise bir **fsync** çağrısı olarak gerçekleştirilir.

Eşzamanlama oluşmadığı sürece, *aiocbp* ile gösterilen nesne ile yapılan **aio_error** çağrıları **EINPROGRESS** ile döner. Eşzamanlama oluştuktan sonra yapılan bir **aio_error** çağrısı eşzamanlama gerçekleşmişse sıfır ile döner; aksi takdirde, **fsync** ya da **fdatasync** çağrılarının hata durumunda **errno** değişkenine atadıkları değer ile döner. Bu durumda dosya tanıtıcısına veri yazmada istikrar anlamında hiçbir şey yapılmamış olabilir.

İstek başarıyla kuyruğa alınmışsa bu işlevin dönüş değeri sıfır olur. Aksi takdirde –1 ile döner ve **errno** değişkenine şu değerlerden biri atanır:

EAGAIN
İstek, özkaynakların geçici yokluğundan dolayı kuyruğa alınmadı

EBADF
aiocbp->aio_fildes tanıtıcı ya geçersiz ya da yazmak için açılmamış

EINVAL
Ya gerçekleme G/Ç eşzamanlamasını gerçeklemiyor ya da *kip* değeri **O_DSYNC** veya **O_SYNC** değil

ENOSYS
İşlev desteklenmiyor

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işlev aslında **aio_fsync64** işlevine karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

```
int aio_fsync64(int               kip,                                    işlev
                struct aiocb64 *aiocbp)
```

Bu işlev, argümanının **struct aiocb64** türünde bir değişken olması dışında **aio_fsync** işlevine benzer.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **aio_fsync** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

Başka bir eşzamanlama yöntemi de bir ya da daha fazla istekten oluşan belli bir küme sonlanana kadar beklemektir. Bu, sonlanma hakkında işlemi başlatan sürecin uyarılmasında **aio_*** işlevleri kullanılmasıyla mümkün olsa da bazı durumlarda bu ideal bir çözüm değildir. Sunucuya bağlı istemcileri sürekli güncel tutan bir yazılımda bazı bağlantıların yavaş bağlantılar olması nedeniyle istemcileri sırayla taranması da en iyi çözüm olmaz. Diğer taraftan, bir istemci güncellenmeden bir uyarı alan süreç yeni istemciye geçemeyeceğinden bir **aio_*** işlevinden bir uyarı ile işlemin durdurulması birşey ifade etmeyeceğinden bu da iyi bir çözüm olmaz. Bu gibi durumlar için **aio_suspend** kullanılmalıdır.

```
int aio_suspend(const struct aiocb *const liste[],                        işlev
                int                        isteksayısı,
                const struct timespec     *süre)
```

Bu işlev çağrıldığında, *liste* dizisinin *isteksayısı* elemanı tarafından yapılan isteklerden en az biri tamamlanana kadar çağıran evre bekletilir. **aio_suspend** çağrısı sırasında zaten tamamlanmış bir istek varsa işlev beklemeksizin döner. Bir isteğin sonlanıp sonlanmadığı isteğin hata durumunun **EINPROGRESS** olup olmamasına bağlıdır. *liste*'nin **NULL** bir elemanı varsa bu girdi yoksayılır.

Tamamlanan bir istek yoksa, çağıran süreç beklemeye alınır. *süre* argümanında **NULL** belirtilmişse, bir istek tamamlanana kadar süreç ilerlemez. *süre* argümanında **NULL** belirtilmemişse, belirtilen süre kadar süreç bekletilir. Bu durumda **aio_suspend** bir hata ile döner.

liste'deki elemanlardan en az biri sonlanmışsa işlev sıfır değeri ile döner. Aksi takdirde –1 ile döner, bu durumda **errno** değişkeninde şu durumlardan biri olabilir:

EAGAIN
liste'deki elemanlardan hiçbiri belirtilen *süre*'de tamamlanmadı

EINTR
aio_suspend işlevini bir sinyal durdurdu. Bu sinyal eşzamansız G/Ç gerçeklemesi tarafından isteklerden birinin sonlanması sinyallenirken de gönderilmiş olabilir.

ENOSYS
: **aio_suspend** işlevi desteklenmiyor

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işlev aslında **aio_suspend64** işlevine karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

```
int aio_suspend64(const struct aiocb64 *const liste[],                    işlev
                  int                          isteksayısı,
                  const struct timespec       *süre)
```

Bu işlev, argümanının **struct aiocb64** türünde bir değişken olması dışında **aio_suspend** işlevine benzer.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **aio_suspend** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

10.4. Eşzamansız G/Ç İşlemlerinin İptal Edilmesi

Bir ya da daha fazla istek eşzamansız olarak işlenirken, yazılması istenen bir verinin artık geçersiz hale gelmesi ve dolayısıyla bu verinin daha sonra düzeltilmesinin gerekmesi gibi bazı durumlarda seçilen bir işlemin iptal edilmesi gerekebilir. Bir örnek olarak, gelen veriyi dosyalara yazan bir uygulamanın, bir dosyaya yazılmak üzere gelmiş bir verinin kuyruğa alınacak bir başka istek ile güncellenmek istemesi verilebilir. POSIX eşzamansız G/Ç gerçeklemesi böyle bir işlev içerir, fakat bu işlevin bir isteği iptale zorlama yeteneği yoktur. Sadece bir isteğin iptalinin mümkün olup olmadığına karar vermeye yardımcı olur.

```
int aio_cancel(int           dosyatanıtıcı,                               işlev
               struct aiocb *aiocbp)
```

aio_cancel işlevi askıdaki bir ya da daha fazla isteği iptal etmek için kullanılabilir. *aiocbp* parametresi **NULL** ise *dosyatanıtıcı* ile ilgili askıdaki tüm işlemleri iptal etmeye çalışır. *aiocbp* parametresi **NULL** değilse, *aiocbp* ile gösterilen belli bir isteği iptal etmeye çalışır.

Başarıyla iptal edilen istekler için isteğin sonlanması ile ilgili normal uyarı yapılmalıdır. Bunu denetleyen **struct sigevent** nesnesine bağlı olarak hiçbir şey yapılmaz, bir sinyal gönderilir ya da evre başlatılır. İstek iptal edilemezse, işlem, gerçekleştikten sonra normal yolla sonlanır.

Bir istek başarıyla iptal edildikten sonra, bu istek için yapılan bir **aio_error** çağrısı **ECANCELED** ile, **aio_return** ise –1 ile döner. Eğer istek iptal edilmemişse ve hala işlenmekteyse hata durumu **EINPROGRESS** olur.

Sonlanmadan iptal edilmiş bir istek varsa, işlevin dönüş değeri **AIO_CANCELED** olur. İptal edilmemiş istek ya da istekler varsa, dömüş değeri **AIO_NOTCANCELED** olur. Bu durumda (*aiocbp*'nin **NULL** olarak belirtildiği durum) iptal edilemeyen istekleri bulmak için **aio_error** kullanılmalıdır. Tüm istekler iptal edilmişse **aio_cancel** işlevinin dönüş değeri **AIO_ALLDONE** olacaktır.

aio_cancel çağrısı sırasında bir hata oluşmuşsa işlev –1 ile döner ve **errno** değişkenine aşağıdaki durumlardan biri atanır:

EBADF
: *dosyatanıtıcı* tanıtıcısı geçersiz

ENOSYS
: **aio_cancel** desteklenmiyor

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işlev aslında **aio_cancel64** işlevine karşılıktır. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

`int` **aio_cancel64**`(int` *dosyatanıtıcı*`, struct aiocb64 *`*aiocbp*`)` işlev

Bu işlev, argümanının **struct aiocb64** türünde bir değişken olması dışında **aio_cancel** işlevine benzer.

Kaynak dosyası **_FILE_OFFSET_BITS == 64** ile derlenmişse 32 bitlik sistemlerde bu işleve **aio_cancel** ismiyle erişilir. Yani 32 bitlik arayüz 64 bitlik olanıyla değiştirilir.

10.5. Eşzamansız G/Ç İşlemlerinin Yapılandırılması

POSIX standardı eşzamansız G/Ç işlevlerinin nasıl gerçekleneceğini belirtmez. Bunlar sistem çağrıları olabileceği gibi ayrıca kullanıcı seviyesinde de gerçekleştirilmiş olabilirler.

Bu kılavuzun yazılması sırasında geçerli olan gerçekleme, kuyruğa alınmış işlemlerin gerçekleştirilmesinde evrelerin kullanıldığı bir kullanıcı seviyesi gerçekleşmeydi. Bu gerçekleme sınırlamalarla ilgili bazı kararlar vermeyi gerektirirken, değiştirilemeyen sınırlamaların bazılarından GNU C kütüphanesinde en iyi biçimde kaçınılmıştır. Bununla birlikte, GNU C kütüphanesi bireysel kullanımla ilgili olarak eşzamansız G/Ç gerçeklemesinde bazı ayarlamalar yapılabilmesini sağlamıştır.

`struct` **aioinit** veri türü

Bu veri türü yapılandırmayı ya da ayarlanabilir parametreleri gerçeklemeye aktarmakta kullanılır. Bir yazılım bu yapının üyelerini ilklendirdikten sonra bunu gerçeklemeye **aio_init** işlevini kullanarak aktarmalıdır.

`int` **aio_threads**
: Herhangi bir anda kullanılabilen azami evre sayısını belirler.

`int` **aio_num**
: Aynı anda kuyruğa alınabilecek isteklerin azami sayısına bir yaklaşıklık sağlar.

`int` **aio_locks**
: Kullanılmadı.

`int` **aio_usedba**
: Kullanılmadı.

`int` **aio_debug**
: Kullanılmadı.

`int` **aio_numusers**
: Kullanılmadı.

`int` **aio_reserved[2]**
: Kullanılmadı.

`void` **aio_init**`(const struct aioinit *`*init*`)` işlev

Bu işlev harhangi bir eşzamansız G/Ç işlevinden önce çağrılmalıdır. Çağrılması tamamen isteğe bağlıdır, sadece eşzamansız G/Ç işlemleri gerçeklemesinin daha iyi yapılmasına yardımcı olma anlamında kullanılabilir.

aio_init çağrısı yapılmadan önce, **struct aioinit** yapısının üyeleri ilklendirilmelidir. Bu yapıldıktan sonra yapı işleve bir argüman olarak aktarılır.

İşlev bir dönüş değerine sahip olmadığı gibi işlevle ilgili atanmış bir hata da yoktur. Irix 6'daki SGI gerçeklemesindeki bir öneriden kaynaklanan bir oluşumdur. POSIX.1b veya Unix98 standartlarının kapsamında değildir.

11. Dosyalar Üzerindeki Denetim İşlemleri

Bu kısımda dosya tanıtıcıları üzerinde, dosya tanıtıcılarla ilgili seçeneklerin ayarlanması ve sorgulanması, kayıt kilitleri gibi diğer işlemlerin nasıl uygulanacağından bahsedilecektir. Bu işlemlerin hepsi **fcntl** işleviyle uygulanır.

fcntl işlevinin ikinci argümanı hangi işlemin uygulanacağını belirleyen bir komuttur. Bu işlemlerle ilgili çeşitli seçenekleri isimlendiren makrolar ile işlevler `fcntl.h` başlık dosyasında bildirilmiştir. Bu seçeneklerin çoğu **open** işlevi tarafından da kullanılır; bkz. *Dosyaların Açılması ve Kapatılması* (sayfa: 306).

```
int fcntl(int dosyatanıtıcı,                                    işlev
          int komut,
          ...)
```

fcntl işlevi *komut* ile belirtilen işlemi *dosyatanıtıcı* tanıtıcısına uygular. Bazı komutlar ek argümanlar gerektirebilir. Ek argümanlar gerektiren komutlar, bunların dönüş değerleri ve hata durumları her komutun açıklamasında ayrı ayrı belirtilmiştir.

Komutların özet listesi:

`F_DUPFD`
: Dosya tanıtıcısının kopyasını yapar (aynı açık dosyaya başka bir dosya tanıtıcı döndürür). Bkz. *Tanıtıcıların Çoğullanması* (sayfa: 339).

`F_GETFD`
: Dosya tanıtıcı ile ilgili seçenekleri döndürür. Bkz. *Dosya Tanıtıcı Seçenekleri* (sayfa: 340).

`F_SETFD`
: Dosya tanıtıcı ile ilgili seçenekleri ayarlar. Bkz. *Dosya Tanıtıcı Seçenekleri* (sayfa: 340).

`F_GETFL`
: Açık dosya ile ilgili seçenekleri döndürür. Bkz. *Dosya Durum Seçenekleri* (sayfa: 341).

`F_SETFL`
: Açık dosya ile ilgili seçenekleri ayarlar. Bkz. *Dosya Durum Seçenekleri* (sayfa: 341).

`F_GETLK`
: Bir dosya kilidi ile döner. Bkz. *Dosya Kilitleri* (sayfa: 346).

`F_SETLK`
: Bir dosya kilidini oluşturur ya da kaldırır. Bkz. *Dosya Kilitleri* (sayfa: 346).

`F_SETLKW`
: Tamamlanmasının beklenmesi dışında **F_SETLK** ile aynıdır. Bkz. *Dosya Kilitleri* (sayfa: 346).

`F_GETOWN`
: **SIGIO** sinyallarini alacak süreç ya da süreç grup kimliği ile döner. Bkz. *Sinyallerle Sürülen Girdi* (sayfa: 349).

`F_SETOWN`
: **SIGIO** sinyallarini alacak süreç ya da süreç grup kimliğini ayarlar. Bkz. *Sinyallerle Sürülen Girdi* (sayfa: 349).

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **fcntl** çağrısı sırasında evre bazı özkaynakları (bellek, dosya tanıtıcı, semafor, vb.) ayırdığında bu bir sorun olur. Evre tam bu anda bir iptal alırsa ayrılan özkaynaklar yazılım sonlanana kadar ayrılmış olarak kalır. Bu tür **fcntl** çağrılarından kaçınmak için iptal eylemcileri kullanılarak korunulmalıdır.

12. Tanıtıcıların Çoğullanması

Bir dosya tanıtıcısını ***çoğullayabilir*** ya da aynı dosya için başka bir dosya tanıtıcı edinebilirsiniz. Çoğullanan dosya tanıtıcılar aynı dosya konumunu ve aynı *dosya durum seçeneklerini* (sayfa: 341) paylaşırlarken, her biri kendi *dosya tanıtıcı seçeneklerine* (sayfa: 340) sahiptir.

Bir dosya tanıtıcısının çoğullanmasının ana amacı girdi ve çıktı ***yönlendirme***dir: bu, bir dosya ya da boruyu kendisi ile ilgili başka bir dosya tanıtıcısı ile değiştirmek anlamındadır.

Çoğullama işlemini **fcntl** işlevini **F_DUPFD** komutu ile kullanarak yapabileceğiniz gibi dosya tanıtıcıları çoğullayan **dup** ve **dup2** işlevleri ile de yapabilirsiniz.

fcntl işlevi ve seçenekleri `fcntl.h` başlık dosyasında bildirilmişken, **dup** ve **dup2** işlevlerinin prototipleri `unistd.h` başlık dosyasında bulunur.

```
int dup(int eski)                                                    işlev
```

Bu işlev *eski* dosya tanıtıcısını kullanılabilir (o an açık olmayan) ilk dosya tanıtıcı numarasına kopyalar. Bu işlem
fcntl (*eski***, F_DUPFD, 0)** çağrısına eşdeğerdir.

```
int dup2(int eski,                                                   işlev
         int yeni)
```

Bu işlev *eski* tanıtıcıyı *yeni* tanıtıcıya kopyalar.

eski geçersiz bir tanıtıcı ise **dup2** hiçbir şey yapmaz; *yeni*'yi de kapatmaz. Aksi takdirde, *eski* tanıtıcısı, zaten bir tanıtıcı olan *yeni* tanıtıcısı kapatıldıktan sonra *yeni* tanıtıcısına kopyalanır.

eski ve *yeni* farklı numaralarsa ve *eski* geçerli bir tanıtıcı numarası ise, **dup2** şu koda eşdeğerdir:

```
close (yeni);
fcntl (eski, F_DUPFD, yeni)
```

Bununla birlikte, **dup2** bunu atomik olarak yapar; **dup2** çağrısının ortasında *yeni*'nin kapatıldığı ama henüz *eski*'nin kopyası yapılmadığı bir an yoktur.

```
int F_DUPFD                                                          makro
```

Bu makro **fcntl** işlevinin *komut* argümanında kullanıldığında işlevin ilk argümanında belirtilen dosya tanıtıcıyı kopyalar.

Bu çağrı şöyle yapılır:

```
fcntl (eski, F_DUPFD, sonraki_tanıtıcı)
```

sonraki_tanıtıcı argümanı **int** türünde olmalı ve döndürülecek dosya tanıtıcısı bu değerde ya da bu değerden büyük kullanılabilir ilk dosya tanıtıcı olacak şekilde belirtilmelidir.

Böyle bir **fcntl** çağrısının normal dönüş değeri yeni dosya tanıtıcısının değeridir. Dönüş değeri –1 ise bu bir hata oluştuğunu gösterir. Aşağıdaki **errno** hata durumları bu komut için tanımlanmıştır:

EBADF

eski argümanı geçersiz

EINVAL
sonraki_tanıtıcı argümanı geçersiz

EMFILE
Kullanılabilir başka tanıtıcı yok—yazılımınız olanı zaten kullanmış. BSD ve GNU'da bu değerin alabileceği azami değeri belirten özkaynak sınırı değiştirilebilir; **RLIMIT_NOFILE** sınırı hakkında daha ayrıntılı bilgi almak için *Özkaynak Kullanımın Sınırlanması* (sayfa: 575) bölümüne bakınız.

EMFILE hata durumu **dup2** işlevinde oluşmaz çünkü **dup2** tanıtıcıyı yeni bir dosya açılışı olarak oluşturmaz, mevcut birini kapattıktan sonra ona kopyalama yapar.

Aşağıdaki örnekte yönlendirme için **dup2** işlevinin nasıl kullanıldığı gösterilmiştir. Genellikle, standart akımların (**stdin** gibi) yönlendirilmesi kabuk tarafından ya da kabuk benzeri bir yazılım tarafından yeni bir yazılımın ya da bir alt sürecin *çalıştırılması* (sayfa: 688) için bir **exec** çağrısı yapılmadan önce yapılır. Yeni bir yazılım çalıştırıldığında, **main** işlevi çalıştırılmadan önce, standart akımlar kendileriyle ilgili dosya tanıtıcılarla oluşturulur ve ilklendirilir.

Standart girdinin bir dosyaya yönlendirilmesi kabuk tarafından şöyle yapılabilirdi:

```
pid = fork ();
if (pid == 0)
  {
    char *dosyaismi;
    char *program;
    int dosya;
    …
    dosya = TEMP_FAILURE_RETRY (open (dosyaismi, O_RDONLY));
    dup2 (dosya, STDIN_FILENO);
    TEMP_FAILURE_RETRY (close (dosya));
    execv (program, NULL);
  }
```

İşlerin Başlatılması (sayfa: 721) bölümünde, süreçlerin bir boruhattının bağlamı içinde yönlendirilmesinin gerçekleştirildiği daha ayrıntılı bir örnek vardır.

13. Dosya Tanıtıcı Seçenekleri

Dosya tanıtıcı seçenekleri bir dosya tanıtıcısının çeşitli özniteliklerdir. Bu seçenekler her dosya tanıtıcısı için özeldir, yani tek bir dosya için bir dosya tanıtıcısını kopyalayarak çoğaltsanız bile yeni dosya tanıtıcısının seçeneklerinde yapacağınız değişiklik özgün tanıtıcıyı etkilemeyecektir.

Şimdilik sadece bir dosya tanıtıcı seçeneği vardır:**FD_CLOEXEC**. Bu seçenek **exec…** *işlevlerini* (sayfa: 688) kullandığınızda tanıtıcının kapanmasına sebep olur.

Bu kısımdaki semboller `fcntl.h` başlık dosyasında tanımlıdır.

`int` **F_GETFD** makro

Bu makro **fcntl** işlevinin *komut* argümanında kullanıldığında *dosyatanıtıcı* ile ilişkili dosya tanıtıcı seçeneklerini döndürür.

Bu komutun **fcntl** işlevinden döndürdüğü değer tek tek seçeneklerin bit seviyesinde VEYA'lanarak yorumlandığı negatif olmayan bir sayıdır (tek tek seçenekler dense de şimdilik sadece bir seçenek var).

Bir hata oluşmuşsa işlev –1 ile döner. Aşağıdaki **errno** hata durumu bu komut için tanımlıdır:

EBADF
: *dosyatanıtıcı* argümanı geçersiz

`int` **F_SETFD** makro

Bu makro **fcntl** işlevinin *komut* argümanında *dosyatanıtıcı* ile ilişkili dosya tanıtıcı seçeneklerini belirlemek için kullanılır. Bu komut yeni seçeneklerin belirtildiği **int** türünde üçüncü bir argüman gerektirir. Böyle bir çağrı şöyle yapılabilir:

```
fcntl (dosyatanıtıcı, F_SETFD, yeni_seçenekler)
```

Bu komutun **fcntl** işlevinden döndürdüğü değer hata durumunu belirten –1 değeri dışında belirsizdir. Seçenekler ve hata durumları **F_GETFD** komutundakilerle aynıdır.

Aşağıdaki makro **fcntl** işlevinde bir dosya tanıtıcı seçeneği olarak kullanılmak üzere tanımlanmıştır. Değeri bir bitmaskesi değeri olarak kullanılabilecek bir tamsayı sabittir.

`int` **FD_CLOEXEC** makro

Bu seçenek, bir **exec** *işlevi çağrıldığında* (sayfa: 688) dosya tanıtıcının kapatılacağını belirtmek için kullanılır. Bir dosya tanıtıcı bir **open** veya **dup** işlevi ile ayrıldığında, bu seçenek temizlenir. Böylece **exec** işlevi ile oluşturulan yeni sürecin dosya tanıtıcısını miras alabilmesi sağlanmış olur.

Dosya tanıtıcı seçeneklerini değiştirmek isterseniz, mevcut seçenekleri **F_GETFD** ile alıp değeri değiştirmelisiniz. Sadece burada açıklanan seçeneklerin varolduğu gibi bir kabul yapmamalısınız; yazılımınız yıllar sonra daha fazla seçeneğin varolduğu bir sistem üzerinde de çalışabilmelidir. Örneğin, aşağıdaki işlev diğer seçeneklere dokunmadan sadece **FD_CLOEXEC** seçeneği ile çalışır:

```
/* desc tanıtıcısına FD_CLOEXEC seçeneği,
   value sıfırdan farklıysa atanır, değilse temizlenir.
   Dönüş değeri hata yoksa 0, varsa -1 olur ve hata errno'ya
   atanır. */

int
set_cloexec_flag (int desc, int value)
{
  int oldflags = fcntl (desc, F_GETFD, 0);
  /* seçeneklerin okunması başarısız olursa,
     hata durmunun belirtip hemen dönelim. */
  if (oldflags < 0)
    return oldflags;
  /* Seçeneğin durumunu belirleyebiliriz. */
  if (value != 0)
    oldflags |= FD_CLOEXEC;
  else
    oldflags &= ~FD_CLOEXEC;
  /* Değiştirilen seçeneği dosya tanıtıcısına atayalım. */
  return fcntl (desc, F_SETFD, oldflags);
}
```

14. Dosya Durum Seçenekleri

Dosya durum seçenekleri bir açık dosyanın özniteliklerini belirlemekte kullanılır. *Dosya Tanıtıcı Seçenekleri* (sayfa: 340) bölümünde açıklanan dosya tanıtıcı seçeneklerinin aksine, dosya durum seçenekleri tek bir dosya için çoğullanan dosya tanıtıcıları arasında paylaşılır. Dosya durum seçenekleri **open** işlevinin *seçenekler* argümanında belirtilir; bkz. *Dosyaların Açılması ve Kapatılması* (sayfa: 306).

Dosya durum seçenekleri, her biri ayrı bir bölümde üç kategori halinde incelenmiştir.

- *Dosya Erişim Kipleri* (sayfa: 342) bölümü dosya için izin verilen erişim türü ile ilgilidir: okuma, yazma, ikisi de. **open** ile belirtilebilir, **fcntl** ile öğrenilebilir ama değiştirilemezler.
- *Açış Anı Seçenekleri* (sayfa: 343) bölümü **open** işlevinin yaptıklarının denetimi ile ilgili ayrıntıları içerir. Bu seçenekler **open** çağrısından sonra saklanmaz.
- *G/Ç İşlem Kipleri* (sayfa: 344) bölümünde, **read** ve **write** işlevlerinin işlemlerini etkileyen seçenekler bulunur. **open** ile belirtilebilir, **fcntl** ile öğrenilebilir veya değiştirilebilirler.

Bu kısımdaki semboller `fcntl.h` başlık dosyasında tanımlıdır.

14.1. Dosya Erişim Kipleri

Erişim kipleri bir dosya tanıtıcısına okuma izni, yazma izni veya her iki izni vermek için kullanılır (GNU sisteminde bunların hiçbirine de izin vermemek mümkün olduğu gibi, bir dosyanın bir çalıştırılabilir olarak çalıştırılmasına izin vermek de mümkündür). Erişim kipleri dosya açılırken belirlenir ve bir daha değiştirilemez.

`int` **O_RDONLY** makro

Dosya okuma erişimi için açılır.

`int` **O_WRONLY** makro

Dosya yazma erişimi için açılır.

`int` **O_RDWR** makro

Dosya hem okuma hem de yazma erişimi için açılır.

GNU sisteminde (başka sistemlerde böyle değildir), **O_RDONLY** ve **O_WRONLY** bit seviyesinde VEYA'lanabilen bağımsız değerlerdir, yani **O_RDWR** aslında **O_RDONLY|O_WRONLY**'dir. Sıfır erişim kipine de izin verilmiştir; bu kipte dosyaya girdi ve çıktı için bir işleme izin verlimez ama **fchmod** gibi başka işlemlere izin verilir. GNU sisteminde, "salt–okunur" ve "salt–yazılır" yanlış isimlendirme olarak kabul edildiğinden `fcntl.h` dosyasında dosya erişim kipleri için ek isimler tanımlanmıştır. Bu isimler GNU'ya özel kod yazarken tercih edilir. Ama POSIX.1 uyumuluğu istenen yazılımlarda aşağıdaki isimler yerine yukarıdaki POSIX.1 isimleri kullanılmalıdır.

`int` **O_READ** makro

Dosya okuma erişimi için açılır. Yalnız GNU'da tanımlıdır. **O_RDONLY** ile aynıdır.

`int` **O_WRITE** makro

Dosya yazma erişimi için açılır. Yalnız GNU'da tanımlıdır. **O_WRONLY** ile aynıdır.

`int` **O_EXEC** makro

Dosya çalıştırmak için açılır. Yalnız GNU'da tanımlıdır.

Dosya erişim kipini **fcntl** ile öğrenmek isterseniz, dosya durum seçeneklerinden erişim kipi bitlerini elde etmeniz gerekir. GNU sisteminde **O_READ** ve **O_WRITE** bitlerini seçenekler arasından kolayca elde edebilmenize karşın, POSIX.1 sistemlerde okuma ve yazma erişim kipleri ayrı ayrı elde edilebilen bitsel seçenekler değildir. Dosya erişim kiplerine taşınabilir anlamda erişmek için en iyi yöntem **O_ACCMODE** makrosunu kullanmaktır.

`int` **O_ACCMODE** makro

Bu makro dosya erişim kiplerinin dosya durum seçeneklerinden bit seviyesinde VE'lenebilen bir değer olarak elde edilmesini sağlar. Erişim kipleri **O_RDONLY**, **O_WRONLY** veya **O_RDWR** olabilir. (GNU sisteminde sıfır da olabilir ve asla **O_EXEC** bitini içermez.)

14.2. Açış Anı Seçenekleri

Açış anı seçenekleri **open** işlevinin davranışlarını etkileyen seçeneklerdir. Bu seçenekler dosya açıldıktan sonra saklanmaz. Buna bir istisna, bir G/Ç işlem kipi de olan **O_NONBLOCK** seçeneğidir ki, bu seçenek kaydedilir. **open** çağrılarının nasıl yapıldığı *Dosyaların Açılması ve Kapatılması* (sayfa: 306) bölümünde anlatılmıştır.

Açış anı seçenekleri iki alt gruba ayrılır.

- *Dosya ismi dönüşüm seçenekleri* **open** işlevinin dosyayı konumlamada dosya ismini nasıl ele alacağını ve dosyanın oluşturulabilir olup olmadığını etkiler.
- *Açış anı eylem seçenekleri* **open** işlevinin dosyanın açılışı anında yapacağı ek işlemleri belirler.

Dosya ismi dönüşüm seçenekleri şunlardır:

`int` **O_CREAT** makro

Bu bit varsa ve dosya mevcut değilse oluşturulur.

`int` **O_EXCL** makro

Hem **O_CREAT** hem de **O_EXCL** bitleri varsa ve belirtilen dosya mevcutsa **open** başarısız olur. Bu mevcut bir dosyanın fütursuzca üzerine yazılmasını engellemeyi garanti eder.

`int` **O_NONBLOCK** makro

open işlevinin dosyanın açılışı sırasında uzun süre beklememesini sağlar. Bu sadece bazı dosya çeşitlerinde anlamlıdır, genelde seri portlar gibi aygıtlarda kullanışlıdır; bu seçeneğin anlamlı olmadığı dosyalarda zararı olmaz ve yoksayılır. Çoğunlukla bir modeme bir port açılması modem taşıyıcıyı saptayana kadar engellenir; **O_NONBLOCK** etkin olduğunda **open** taşıyıcıyı beklemeden dönecektir.

O_NONBLOCK seçeneğinin hem G/Ç işlem kipi hem de bir dosya ismi dönüşüm seçeneği olarak belirtilmesi durumunda **open** işlevi ayrıca beklemeyen G/Ç kipini de etkin kılacaktır. **open** işlevinin dosyayı açarken beklemesi ama G/Ç işlemleri için beklemeyen kipi etkinleştirmesini istiyorsanız **open** işlevini **O_NONBLOCK** etkin olarak çağırmalı ve ardından bir **fcntl** çağrısı ile bu biti temizlemelisiniz.

`int` **O_NOCTTY** makro

İsimli dosya bir uçbirim aygıtı ise, aygıt sürecin denetim uçbirimi yapılmaz. Denetim uçbiriminin ne anlama geldiği *İş Denetimi* (sayfa: 716) bölümünde açıklanmıştır.

GNU sisteminde ve 4.4 BSD'de, **O_NOCTTY** sıfırdır ve bir dosya açılışı aygıtı hiçbir zaman denetim uçbirimi yapmaz; taşınabilirlik önemliyse bundan kaçınmak için **O_NOCTTY** seçeneğini kullanın.

Aşağıdaki dosya ismi dönüşüm seçenekleri sadece GNU sisteminde geçerlidir.

`int` **O_IGNORE_CTTY** makro

Aygıt sürecin denetim uçbirimi olarak belirlenmiş olsa bile isimli dosya sürecin denetim uçbirimi olarak tanınmaz. Yeni dosya tanıtıcı hiçbir zaman iş denetim sinyallerini içermeyecektir. Bkz. *İş Denetimi* (sayfa: 716).

`int` **O_NOLINK** makro

İsimli dosya bir sembolik bağ ise, bağın hedefindeki dosya değil bağın kendisi açılır. (Yeni dosya tanıtıcı ile yapılan **fstat** çağrısı bağın ismiyle yapılacak bir **lstat** çağrısından dönecek bilgileri döndürecekektir.)

`int` **O_NOTRANS** makro

İsimli dosya özellikle dönüştürülürse, dönüştürücü çağrılmaz. Dönüştürücü onu göreceğinden dosyayı sadece açar.

Açış anı eylem seçenekleri **open** işlevine aslında dosyanın açılışı ile ilgili olmayan ek işlemleri belirtmek için kullanılır. Bunun ayrı çağrılarla değil de **open** işlevinin bir parçası olarak yapılmasının sebebi **open** işlevinin bu işlemleri atomik olarak yapabilmesidir.

`int` **O_TRUNC** makro

Dosyanın uzunluğunu sıfırlar. Bu seçenek dizin veya FIFO gibi özel dosyalarda değil sadece sıradan dosyalarda kullanışlıdır. POSIX.1, bir dosyanın yazma amacıyla açılması için **O_TRUNC** kullanılmasını gerekli kılar. GNU ve BSD'de dosyayı sıfırlayabilmek için yazma izni olması gerekir ama bunu yapmak için dosyanın yazma amacıyla açılması gerekmez.

Bu POSIX.1 tarafından belirtilmiş tek açış anı eylem seçeneğidir. Dosyanın sıfırlanmasının **open** tarafından yapılmasının iyi bir sebebi yoksa bunu yapmak yerine **open** çağrısının hemen ardından **ftruncate** çağrısı yapın. **O_TRUNC** seçeneği **ftruncate** tasarlanmadan önce Unix'de vardı ama artık geriye uyumluluk adına var.

Kalan işlem kipleri BSD oluşumlarıdır. Sadece bazı sistemlerde vardır, diğerlerinde bu makrolar tanımlanmamıştır.

`int` **O_SHLOCK** makro

flock'un yaptığı gibi dosya üzerinde bir paylaşımlı kilit edinilir. Bkz. *Dosya Kilitleri* (sayfa: 346).

O_CREAT belirtilmişse,dosya oluşturulurken kilitleme atomik olarak yapılır. Yeni dosyada başka bir sürecin kilitleme yapması mümkün olmaz.

`int` **O_EXLOCK** makro

flock'un yaptığı gibi dosya üzerinde bir ayrıcalıklı kilit edinilir. Bkz. *Dosya Kilitleri* (sayfa: 346). Bu, **O_SHLOCK**'daki gibi atomik yapılır.

14.3. G/Ç İşlem Kipleri

İşlem kipleri dosya tanıtıcının girdi ve çıktı işlemlerinde nasıl kullanılacağını belirler. Bu seçenekler **open** ile belirtilir, **fcntl** ile okunur ve değiştirilir.

`int` **O_APPEND** makro

Bu bit dosyaya ekleme kipini etkin kılar. Bu bit varsa, tüm **write** işlemleri o anki dosya konumundan bağımsız olarak veriyi dosyanın sonuna ekleyecektir. Bu bir dosyaya ek yapmanın en güvenilir yoludur. Ekleme kipi, dosyaya yazan başka süreçlerin yaptıklarından etkilenmeksizin daima dosya sonuna veri eklemeyi garanti eder. Bu kipi kullanmadan kendiniz dosya konumunu dosya sonuna ayarlayıp yazmaya çalışırsanız, hemen öncesinde başka bir sürecin dosyaya yaptığı bir yazma işlemi sonucunda veriniz dosyanın sonuna değil dosyanın içinde bir yere yazılmış olabilir.

`int` **O_NONBLOCK** makro

Bu bit dosya için beklememe kipini etkin kılar. Bu bit varsa, **read** işlemlerinde o an bir girdinin mevcut olmaması durumunda işlev veriyi beklemeksizin bir hata durumu ile döner. Benzer şekilde **write** işlemlerinde çıktı hemen yazılamıyorsa işlev beklemeksizin bir hata durumu ile döner.

O_NONBLOCK seçeneğinin hem G/Ç işlem kipi olarak hem de dosya ismi dönüşüm seçeneği olarak etkin olabileceğini unutmayın; bkz. *Açış Anı Seçenekleri* (sayfa: 343).

`int` **O_NDELAY** makro

Bu, BSD ile uyumluluk adına bulunan **O_NONBLOCK** seçeneğinin atıl takma adıdır. POSIX.1 standardında tanımlanmamıştır.

`int` **O_ASYNC** makro

Bu bit eşzamansız girdi kipini etkin kılar. Bu bit varsa, girdi olduğunda **SIGIO** sinyalleri üretilecektir. Bkz. *Sinyallerle Sürülen Girdi* (sayfa: 349).

Eşzamansız girdi kipi bir BSD özelliğidir.

`int` **O_FSYNC** makro

Bu bit eşzamanlı yazma kipini etkin kılar. Bu bit varsa, **write** çağrısı veri diske yazılmadan dönmeyecektir. Eşzamanlı yazma kipi bir BSD özelliğidir.

`int` **O_SYNC** makro

O_FSYNC ile aynıdır.

`int` **O_NOATIME** makro

Bu bit varsa, **read** çağrısı *dosyanın erişim zamanını* (sayfa: 383) güncellemeyecektir. Bu kip yedekleme uygulamalarınca kullanılır, böylece yedeklenen dosya işlem sırasında okunmuş sayılmaz. Bu kipi sadece dosyanın sahibi veya süper kullanıcı etkin kılabilir.

Bu kip bir GNU oluşumudur.

14.4. Dosya Durum Seçeneklerinin Saptanması

Dosya durum seçeneklerini öğrenmek ve belirlemek için **fcntl** işlevi kullanılır.

`int` **F_GETFL** makro

Bu makro **fcntl** işlevinin *komut* argümanında kullanıldığında *dosyatanıtıcı* ile ilgili açık dosyanın dosya durum seçeneklerini okur.

Bu komutun **fcntl** işlevinden döndürdüğü değer normalde tek tek seçeneklerin bit seviyesinde VEYA'lanmasıyla elde edilen negatif olmayan bir değerdir. Dosya erişim kipleri GNU sistemi dışında eşsiz bit değerler olmadıklarından bu bitlere bu makro ile değil, **O_ACCMODE** makrosu ile erişebilirsiniz.

Bir hata durumunda **fcntl** –1 ile döner. Aşağıdaki **errno** hata durumu bu komut için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçersiz.

`int` **F_SETFL** makro

Bu makro **fcntl** işlevinin *komut* argümanında kullanılarak *dosyatanıtıcı* ile ilgili açık dosyanın dosya durum seçenekleri belirlenebilir. Bu komut, yeni seçeneklerin belirtildiği üçüncü bir argümanın varlığını gerektirir:

```
fcntl (dosyatanıtıcı, F_SETFL, yeni_seçenekler)
```

Bir dosya tanıtıcının erişim kipini bu komutla değiştiremezsiniz.

Bu komutun **fcntl** işlevinden döndürdüğü değer hata durumunu belirten –1 değeri dışında belirsizdir. Hata durumları **F_GETFL** komutu ile aynıdır.

Dosya durum seçeneklerini değiştirmek isterseniz önce o anki seçenekleri **F_GETFL** ile almalı sonra bu değeri değiştirmelisiniz. Sadece burada açıklanan seçeneklerin varolduğu gibi bir kabul yapmamalısınız; yazılımınız yıllar sonra daha fazla seçeneğin varolduğu bir sistem üzerinde de çalışabilmelidir. Örneğin, aşağıdaki işlev diğer seçeneklere dokunmadan sadece **O_NONBLOCK** seçeneğini değiştirir:

```
/* desc tanıtıcısına O_NONBLOCK seçeneği,
  value sıfırdan farklıysa atanır, değilse temizlenir.
  Dönüş değeri hata yoksa 0, varsa -1 olur ve hata errno'ya
  atanır. */

int
set_nonblock_flag (int desc, int value)
{
  int oldflags = fcntl (desc, F_GETFL, 0);
  /* seçeneklerin okunması başarısız olursa,
     hata durmunun belirtip hemen dönelim. */
  if (oldflags == -1)
    return -1;
  /* Seçeneğin durumunu belirleyebiliriz. */
  if (value != 0)
    oldflags |= O_NONBLOCK;
  else
    oldflags &= ~O_NONBLOCK;
  /* Değiştirilen seçeneği dosya tanıtıcısına atayalım. */
  return fcntl (desc, F_SETFL, oldflags);
}
```

15. Dosya Kilitleri

Kalan **fcntl** komutları, aynı anda çok sayıda süreci çalışabilen uygulamalarda bir sürecin aynı dosyaya hataya eğilimli bir yolla aynı anda erişimini engelleyen ***dosya kilitleme*** desteği içindir.

Bir *ayrıcalıklı* ya da *yazma* kilidi sürecin dosyanın belli bir parçasına yazma amacıyla ayrıcalıklı erişimini mümkün kılar. Bir yazma kilidi etkinken başka bir süreç dosyanın kilitli bölümünü kilitleyemez.

Bir *paylaşımlı* veya *okuma* kilidi başka bir sürecin dosyanın okuma kilitli parçasında bir yazma kilidi isteği yapmasına engel olur, ancak bir okuma kilidi isteğini engellemez.

Aslında **read** ve **write** işlevleri dosyanın bir yerinde bir kilit var mı, yok mu diye bakmaz. Çok sayıda süreç arasında dosya kilitlemesini etkili olarak kullanmak istiyorsanız, kilitlerin durumunu doğrudan **fcntl** çağrıları ile saptadıktan sonra işlemi yapıp yapmamaya karar vermelisiniz.

Kilitler süreçlerle ilişkilendirilir. Bir süreç belirtilen bir dosyanın her baytı için sadece bir çeşit kilitleme yapabilir. Bir dosya ile ilişkili dosya tanıtıcılardan herhangi biri kapatıldığında, diğer dosya tanıtıcılar açık bile olsa, süreçte bu dosya ile ilişkilendirilmiş tüm kilitler iptal edilir. Benzer şekilde bir süreç sonlandığında da kilitler iptal edilir, ayrıca **fork** ile *oluşturulan alt süreçler* (sayfa: 687) bu kilitleri miras almazlar.

Bir kilit oluşturulurken ne çeşit kilidin nerede oluşturulacağını belirtmek için **struct flock** yapısı kullanılır. Bu veri türü ve kilitlerle ilgili **fcntl** makroları `fcntl.h` başlık dosyasında bildirilmiştir.

`struct` **flock** veri türü

Bu yapı **fcntl** işlevi ile bir dosya kilidini yapılandırmada kullanılır. Şu üyelere sahiptir:

`short int` **l_type**
Kilidin türünü belirtir; **F_RDLCK**, **F_WRLCK** veya **F_UNLCK** makrolarından biri olabilir:

`F_RDLCK`
: Bir okuma kilidi (ya da paylaşımlı kilit) belirtir.

`F_WRLCK`
: Bir yazma kilidi (ya da ayrıcalıklı kilit) belirtir.

`F_UNLCK`
: Bölgeden kilidin kaldırılmasını belirtir.

`short int` **`l_whence`**
: **`fseek`** veya **`lseek`** işlevinin *nereye* veya *nereden* argümanında kullanıldığı gibi dosya konumunun nereye göre belirlendiğini belirtir. Değeri **`SEEK_SET`**, **`SEEK_CUR`** veya **`SEEK_END`** olabilir.

`off_t` **`l_start`**
: Kilidin uygulanacağı bölümün başlangıcından uzaklığını belirler ve yapının **`l_whence`** üyesinde belirtilen noktaya göre bayt cinsinden ifade edilir.

`off_t` **`l_len`**
: Kilitlenecek bölgenin uzunluğunu belirler. **`0`** değerinin özel bir anlamı vardır, bölgenin dosyanın sonuna kadar genişleyebileceğini belirtir.

`pid_t` **`l_pid`**
: Kilidi tutan sürecin *süreç kimliğidir* (sayfa: 686). Bu alan kilidi oluştururken yoksayılır, sadece **`fcntl`** işlevinin **`F_GETLK`** komutu ile yapılan çağrısı ile doldurulur.

`int` **`F_GETLK`** makro

Bu makro **`fcntl`** işlevinin *komut* argümanında kullanılır ve bir kilit hakkında bilgi istendiğini belirtir. Bu komut **`fcntl`** işlevinde **`struct flock *`** türünde üçüncü bir argüman gerektirir:

```
fcntl (dosyatanıtıcı, F_GETLK, kilit_gst)
```

kilit_gst argümanı ile belirtilen yerde bir kilit varsa, kilit ile ilgili bilgi **kilit_gst*'ye yazılır. Belirtilen yeni kilitle uyumluysa mevcut kilitler raporlanmaz. Bu bakımdan, hem okuma hem de yazma ile ilgili kilitleri bulmak için **`F_WRLCK`** türünde bir kilit, sadece yazma ile ilgili kilitleri bulmak için ise, **`F_RDLCK`** türünde bir kilit belirtmelisiniz.

kilit_gst tarafından belirtilen bölgeyi etkileyen birden fazla kilit varsa bunlardan sadece biri raporlanır. *kilit_gst* yapısının **`l_whence`** üyesine **`SEEK_SET`** atanır, **`l_start`** ve **`l_len`** üyelerine de kilitli bölgeyi tanımlayan değerler atanır.

Bir kilit yoksa, *kilit_gst* yapısının sadece **`l_type`** üyesine **`F_UNLCK`** atanır.

`fcntl` işlevinin bu komut ile ilgili dönüş değeri hata oluştuğunu belirten –1 değeri dışında belirsizdir. Aşağıdaki **`errno`** hata durumları bu komut için tanımlanmıştır:

`EBADF`
: *dosyatanıtıcı* argümanı geçersiz

`EINVAL`
: Ya *kilit_gst* argümanı geçerli bir kilit belirtmiyor ya da *dosyatanıtıcı* kilitleri desteklemiyor

`int` **`F_SETLK`** makro

Bu makro **`fcntl`** işlevinin *komut* argümanında bir kilit oluşturmak ya da kaldırmak amacıyla kullanır. Bu komut **`fcntl`** işlevinde **`struct flock *`** türünde üçüncü bir argüman gerektirir:

```
fcntl (dosyatanıtıcı, F_SETLK, kilit_gst)
```

Süreç dosyanın belirtilen bölgesinde bir kilide sahipse eski kilit yenisiyle değiştirilir. Mevcut bir kilidi **`F_UNLCK`** türünde bir kilit belirterek kaldırabilirsiniz.

Kilit oluşturulamazsa **`fcntl`** –1 değeriyle döner. Bu işlev başka bir sürecin kilidi bırakması için beklemez. **`fcntl`** işlevi başarılı olduğundan –1'den farklı bir değerle döner.

Aşağıdaki **`errno`** hata durumları bu komut için tanımlanmıştır:

`EAGAIN`
`EACCES`
: Dosya üzerindeki başka bir kilit tarafından engellendiğinden kilit oluşturulamıyor. Bazı sistemler bu durumda **`EAGAIN`** kullanırken başkaları **`EACCES`** kullanır; yazılımınızda her ikisini de beklemelisiniz. (GNU sistemi daima **`EAGAIN`** kullanır.)

`EBADF`
: *dosyatanıtıcı* argümanı geçersiz; ya okuma erişimi için açılmamış *dosyatanıtıcı* için bir okuma kilidi istemişsinizdir ya da yazma erişimi için açılmamış bir *dosyatanıtıcı* için yazma kilidi istemişsinizdir.

`EINVAL`
: Ya *kilit_gst* argümanı geçerli kilit bilgisi içermiyor ya da *dosyatanıtıcı* ile ilişkili dosya kilitleri desteklemiyor.

`ENOLCK`
: Sistemde dosya kilidi özkaynakları tükendi; dosya kilidi istenen yerde zaten fazlasıyla dosya kilidi var

  İyi tasarlanmış dosya sistemleri bu hatayı hiç raporlamaz, çünkü dosya kilitleri ile ilgili bir sınır yoktur. Yine de, bir dosya sisteminin ağ üzerinden eriştiği başka bir dosya sistemi bu hatayı verebileceğinden bu hatayı hesaba katmanız gerekir.

`int` **`F_SETLKW`** makro

Bu makro **`fcntl`** işlevinin *komut* argümanında bir kilit oluşturmak ya da kaldırmak amacıyla kullanır. **`F_SETLK`** komutu gibi olmakla birlikte farklı olarak kilidi ayırana ya da kilit serbest kalıncaya kadar süreci bekletir.

Bu komut da **`F_SETLK`** komutu gibi **`struct flock *`** türünde bir argüman gerektirir.

`F_SETLK` komutu için **`fcntl`** işlevindeki hata durumlarına ek olarak aşağıdaki hata durumları bu komut için tanımlanmıştır:

`EINTR`
: İşlev beklerken bir sinyal ile durduruldu. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

`EDEADLK`
: Belirtilen bölge başka bir süreç tarafından kilitlenmiş. Ama süreç, başka sürecin kilitlediği bölgeyi kendisi kilitleyene dek bekler, bu da kilit isteğinin sonsuza kadar beklenmesi anlamına gelebilir. Sistem tüm durumlarda bu hatanın saptanmasını garanti etmez ama eğer bu konuda uyarılmışsanız, şanslısınız demektir.

Dosya kilitlemenin faydalı olduğu bir duruma örnek olarak, bir yazılımın çok sayıda kullanıcı tarafından aynı anda çalıştırıldığını ve bu süreçlerin durum bilgisini ortak bir dosyaya yazdıklarını varsayabiliriz. Bu tür bir örnek, oyuncuların aldıkları puanları bir dosyaya kaydeden oyunlar olabilir. Buna başka bir örnek de hesap bilgilerini ve kullanımlarını kaydeden bir yazılım olabilir.

Bir yazılımın çok sayıda kopyasının bir dosyaya aynı anda yazması dosya içeriğinin karışmasına sebep olur. Bu çeşit sorunların oluşması, dosyaya yapılacak bir yazma işleminden önce bir yazma kilidi oluşturularak önlenebilir.

Yazılımın dosyayı tutarlı bir durumda iken okuması önemliyse bir okuma kilidi kullanılabilir. Bir okuma kilidinin varlığı başka bir sürecin dosyanın bir bölümünü yazmak için kilitlemesini önler.

Dosya kilitlerinin bir dosyaya erişimi denetlemek için "isteğe bağlı" bir protokol olduğunu unutmayın. Kilitleme protokolünü kullanmayan başka süreçlerin dosyaya erişimi hala mümkün olacaktır.

16. Sinyallerle Sürülen Girdi

Bir dosya tanıtıcıda **O_ASYNC** *durum seçeneği* (sayfa: 341) etkinse, bu dosya tanıtıcı ile ilgili bir girdi ya da çıktı olasılığı varsa bir **SIGIO** sinyali gönderilir. Sinyali alacak süreç veya süreç grubu **F_SETOWN** komutu **fcntl** işlevinde kullanılarak seçilebilir. Dosya tanıtıcısı bir soket ise bu ayrıca, sokete bir *banddışı veri* (sayfa: 431) geldiğinde alınan **SIGURG** sinyallerinin alıcılarını seçmekte de kullanılır. **SIGURG** sinyali **select** işlevinin bir "olağandışı durumu" raporladığı durumlarda gönderilir. Bkz. *Girdi ve Çıktının Beklenmesi* (sayfa: 323).)

Dosya tanıtıcı bir uçbirim aygıtı ile ilgiliyse **SIGIO** sinyalleri uçbirimin önalan süreç grubuna gönderilir. Bkz. *İş Denetimi* (sayfa: 716).

Bu kısımdaki semboller `fcntl.h` başlık dosyasında tanımlanmıştır.

`int` **F_GETOWN** makro

Bu makro **fcntl** işlevinin *komut* argümanında kullanıldığında **SIGIO** sinyallerinin gönderildiği süreç veya süreç grubu ile ilgili bilgilerin alınmasını sağlar. (Bir uçbirim için bu bilgi, **tcgetpgrp** kullanılarak alınabilen önalan süreç grubunun kimliğidir. Bkz. *Denetim Uçbirimine Erişim İşlevleri* (sayfa: 731).)

Dönüş değeri bir süreç kimliği olarak yorumlanır; negatifse, dönüş değerinin mutlak değeri süreç grup kimliğidir.

Aşağıdaki **errno** hata durumu bu komut için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçersiz

`int` **F_SETOWN** makro

Bu makro **fcntl** işlevinin *komut* argümanında kullanılarak **SIGIO** sinyallerinin gönderildiği süreç veya süreç grubunu belirtilebilir. Bu komut, **fcntl** işlevinde **pid_t** türünde üçüncü bir agüman kullanılmasını gerektirir:

```
fcntl (dosyatanıtıcı, F_SETOWN, pid)
```

pid argümanı bir süreç kimliği olmalıdır. Ayrıca mutlak değeri bir süreç grup kimliği olan negatif sayı da belirtilebilir.

Bu komutun **fcntl** işlevinden döndürdüğü değer bir hata oluşmuşsa –1'dir, aksi takdirde farklı bir değer döner. Aşağıdaki **errno** hata durumları bu komut için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçersiz.

ESRCH
: *pid*'e karşılık bir süreç ya da süreç grubu yok.

17. Soysal G/Ç Denetim İşlemleri

GNU sistem bir çok farklı aygıt ve nesne üzerindeki çoğu girdi/çıktı işlemini **read**, **write** ve **lseek** işlevinden oluşan bir kaç ilkelle gerçekleştirir. Buna karşın çoğu aygıt bu modelle karşılanamayan bir kaç tuhaf işleme ihtiyaç gösterir. Örneğin:

- Bir uçbirimde kullanılan yazıtipinin değişirilmesi.
- Bir manyetik teybe geri ya da ileri sarmasının söylenmesi. (Bayt artışları ile hareket ettirilemediklerinden **lseek** uygulanamaz.)
- Bir diskin sürücüsünden çıkarılması.
- Bir CD–ROM aygıtındaki ses kaydının çalınması.
- Bir ağın yönlendirme tablolarının bakımı.

Bu tür nesnelerin yanında soketler ve uçbirimler[6] gibi nesneler de kendilerine özgü özel işlevlere sahiptir, tüm bu durumlar için işlevler oluşturmak pratik olmazdı.

Bu küçük işlemler yerine *IOCTL*'ler olarak bilinen kod numaraları atanır ve `sys/ioctl.h` başlık dosyasında tanımlı olan **ioctl** işlevi üzerinden bunlar çoğullanır. Kod numaraları farklı başlık dosyalarında tanımlıdır.

```
int ioctl(int dosyatanıtıcı,                                    işlev
          int komut,
          …)
```

ioctl işlevi *dosyatanıtıcı* tanıtıcı üzerine *komut* soysal G/Ç işlemini uygular.

Genellikle, tek bir sayı ya da bir yapıya gösterici olarak üçüncü bir argüman daha olur. Bu argümanın anlamı, dönüş değeri ve kullanılan komut ile ilgili hata durumudur. Başarısızlık halinde çoğunlukla –1 döner.

Bazı sistemlerde farklı aygıtlar tarafından kullanılan IOCTL'ler aynı numaraları paylaşırlar. Bu durumda, böyle bir IOCTL hep aynı hatayı üretir. Aygıta özel IOCTL'leri bilinmeyen bir aygıt üzerinde kullanmaya çalışmamalısınız.

Çoğu IOCTL işletim sistemine özeldir ve/veya sadece özel sistem araçlarında kullanılır ve bu nedenle bu belgenin kapsamı dışında kalırlar. Bir IOCTL kullanım örneğini *Bantdışı Veri Aktarımı* (sayfa: 431) bölümünde bulabilirsiniz.

XIV. Dosya Sistemi Arayüzü

İçindekiler

Bu oylumda dosyalarla çalışmak için kullanılan GNU C kütüphanesi işlevleri açıklanmıştır. Girdi ve çıktı işlevlerinin tersine (*Akımlar Üzerinde Giriş/Çıkış* (sayfa: 236); *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305)), bu işlevler dosyaların içerikleri ile değil dosyaların kendileriyle ilgili işlemleri yaparlar.

Bu oylumda açıklanan oluşumlar arasında, dizinleri değiştiren veya inceleyen işlevler, dosyaları silen ya da ismini değiştiren işlevler ve erişim yetkileri ve değişiklik zamanları gibi dosya özniteliklerini incelemek ve değiştirmek için kullanılan işlevler sayılabilir.

1. Çalışma dizini

Her sürecin kendisiyle ilişkili bir dizini vardır. Bu dizine ***çalışma dizini*** denir ve göreli dosya isimlerinin çözümlenmesinde kullanılır (bkz. *Dosya İsmi Çözümlemesi* (sayfa: 233)).

Sisteme oturum açtığınızda, sistem veritabanındaki kullanıcı hesabınızla ilişkili ev dizininiz çalışma dizininiz yapılır. Bir kullanıcının ev dizinini `getpwuid` veya `getpwnam` işleviyle bulabilirsiniz; bkz. *Kullanıcı Veritabanı* (sayfa: 760).

Kullanıcılar çalışma dizinlerini **cd** gibi kabuk komutlarını kullanarak değiştirebilirler. Bu bölümde açıklanan işlevler bu komutlarda ve çalışma dizinini değiştirmek ve incelemek için başka yazılımlarda kullanılan ilkellerdir. Bu işlevlerin prototipleri `unistd.h` başlık dosyasında bildirilmiştir.

`char *`**`getcwd`**`(char *`*tampon*`, size_t `*boyut*`)` işlev

getcwd işlevi sizin tarafınızdan sağlanan *tampon* karakter dizisine o anki çalışma dizinini ifade eden mutlak dosya ismini kaydederek döner. *boyut* argümanı ile sisteme *tampon* için ayrılan boyut bildirilir.

Bu işlevin GNU kütüphanesindeki sürümü *tampon* argümanı olarak bir boş gösterici belirtebilmenizi mümkün kılar. Bu durumda **getcwd** işlevi **malloc** ile tamponu kendisi ayırır (bkz. *Özgür Bellek Ayırma* (sayfa: 50)). Eğer *boyut* sıfırdan büyükse, tampon bu kadar daha büyük olur, aksi takdirde tampon sonucu tutmaya yetecek büyüklükte olur.

İşlevin normal dönüş değeri *tampon*'dur. Başarısızlık halinde boş gösterici döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EINVAL`
: *boyut* argümanı sıfır ve *tampon* boş gösterici değil.

`ERANGE`
: *boyut* argümanı ile belirtilen uzunluk çalışma dizini isminden daha kısa. Daha büyük dizi ayırarak yeniden deneyin.

`EACCES`
: Dosya isminin bir elemanını aramak ya da okumak için izin verilmedi.

GNU'nun **getcwd (NULL, 0)** davranışını sadece **getcwd** işlevinin standart davranışını kullanarak gerçekleştirebilirdiniz:

```
char *
gnu_getcwd ()
{
  size_t size = 100;

  while (1)
    {
      char *buffer = (char *) xmalloc (size);
      if (getcwd (buffer, size) == buffer)
        return buffer;
      free (buffer);
      if (errno != ERANGE)
        return 0;
      size *= 2;
    }
}
```

Bir kütüphane işlevi olmayan ama çoğu GNU yazılımında kullanılan bir özelleştirilmiş isim olan **xmalloc** hakkında daha fazla bilgi için *malloc Örnekleri* (sayfa: 51) bölümüne bakınız.

`char *`**`getwd`**`(char *`*tampon*`)` önerilmeyen işlev

Tampon için boyut belirtilmemesi dışında **getcwd** işlevinin benzeridir. **getwd** işlevi sadece BSD uyumluluğu için GNU kütüphanesine dahil edilmiştir.

tampon argümanı en azından **PATH_MAX** bayt uzunluktaki bir tampona gösterici olmalıdır (bkz. *Dosya Sistemi Kapasite Sınırları* (sayfa: 795)). GNU sisteminde dosya ismi için bir sınırlama yoktur, yani dizin ismini içerecek yeterli alan için bu gereksizdir. Bu, bu işlevin kullanılmasının önerilmeme sebebidir.

```
char *get_current_dir_name(void)                                    işlev
```

get_current_dir_name işlevi aslında **getcwd (NULL, 0)** çağrısının eşdeğeridir. Tek fark, **PWD** değişkeninin değerinin (eğer doğruysa) dönmesidir. Bu anlaşılması zor bir farktır. Eğer **PWD** değeri ile açıklanan dosya yolu bir ya da daha fazla sembolik bağ kullanıyorsa, değerin **getcwd** tarafından döndürülmesi durumunda sembolik bağlar çözümlenebilir ve bu bakımdan farklı bir sonuç ortaya çıkabilir.

Bu işlev bir GNU oluşumudur.

```
int chdir(const char *dosyaismi)                                    işlev
```

Bu işlev sürecin çalışma dizinini *dosyaismi* yapar.

İşlevin normal dönüş değeri **0**'dır. **-1** değeri dönmüşse bir hata oluşmuş demektir. Bu işlev için tanımlanmış **errno** hata durumları *dosya ismi sözdizimi hataları* (sayfa: 234) ile *dosyaismi* ismi bir dizin değilse **ENOTDIR**'dır.

```
int fchdir(int dosyatanıtıcı)                                       işlev
```

Bu işlev sürecin çalışma dizinini *dosyatanıtıcı* ile ilişkili dizin yapar.

İşlevin normal dönüş değeri **0**'dır. **-1** değeri dönmüşse bir hata oluşmuş demektir. Bu işlev için tanımlanmış **errno** hata durumları:

EACCES
: **dirname** tarafından döndürülen dizin için okuma izni verilmedi.

EBADF
: *dosyatanıtıcı* geçerli bir dosya tanıtıcı değil.

ENOTDIR
: *dosyatanıtıcı* bir dizin ile ilişkili değil.

EINTR
: İşlev çağrısı bir sinyal ile engellendi.

EIO
: Bir G/Ç hatası oluştu.

2. Dizinlere Erişim

Bu kısımda açıklanan oluşumlar bir dizin dosyasının içeriğini okumanızı sağlar. Eğer yazılımınızın bir dizindeki tüm dosyaları listelemesini isterseniz bu kullanışlıdır.

opendir işlevi elemanları dizin girdileri olan bir ***dizin akımı*** açar. Yazılımın okumak için açılmış dizin üzerinde daha fazla denetim sağlayabilmesi gibi getirileri için bunun yerine **fdopendir** işlevi de kullanılabilir. Bu, örneğin, **open** işlevine **O_NOATIME** seçeneğinin aktarılmasını mümkün kılar.

Bu girdileri **struct dirent** nesneleri olarak almak için dizin akımı üzerinde **readdir** işlevini kullanabilirsiniz. Her girdinin dosya ismi bu yapının **d_name** üyesinde saklanır. Burada sıradan dosyaların *akım oluşumları* (sayfa: 236) ile açıkça paralellikler vardır.

2.1. Dizin Girdileri

Bu bölümde bir dizin akımından sağlanan tek bir dizin girdisinde bulacaklarınız açıklanacaktır. Bu bölümdeki tüm semboller `dirent.h` başlık dosyasında bildirilmiştir.

`struct` **`dirent`** veri türü

Dizin girdileri hakkında bilgi döndürmekte kullanılan yapıdır. Şu üyelere sahiptir:

`char` **`d_name`**`[]`
: Boş karakter sonlandırmalı dosya ismi elemanıdır. Bu yapının üyesi olarak tüm POSIX sistemlerinde bulabileceğiniz tek üyedir.

`ino_t` **`d_fileno`**
: Dosyanın seri numarasıdır. BSD uyumluluğu için bu üyeye **`d_ino`** ismiyle de erişebilirsiniz. GNU siteminde ve çoğu POSIX sisteminde, bu üye, çoğu dosya için **`stat`** çağrısından dönen **`st_ino`** üyesi ile aynıdır. Bkz. *Dosya Öznitelikleri* (sayfa: 371).

`unsigned char` **`d_namlen`**
: Dosya isminin sonlandırıcı boş karakter içermeyen **`unsigned char`** türünden uzunluğudur.

`unsigned char` **`d_type`**
: Dosyanın türüdür. Değeri için tanımlanmış sabitler:

  `DT_UNKNOWN`
  : Dosya türü bilinmiyor. Bazı sistemlerde dönen tek değerdir.

  `DT_REG`
  : Normal bir dosya.

  `DT_DIR`
  : Bir dizin.

  `DT_FIFO`
  : Bir isimli boru ya da FIFO. Bkz. *FIFO Özel Dosyaları* (sayfa: 396).

  `DT_SOCK`
  : Bir yerel alan soketi.

  `DT_CHR`
  : Bir karakter aygıtı.

  `DT_BLK`
  : Bir blok aygıtı.

  Bu üye bir GNU oluşumudur. Eğer bu üye varsa, sembolü **`_DIRENT_HAVE_D_TYPE`** tanımlıdır. Kullanıldığı sistemlerde **`struct statbuf`** yapısının **`st_mode`** üyesindeki dosya türü bitlerine karşı düşer. Eğer değer saptanamazsa üyenin değeri **`DT_UNKNOWN`** olur. Bu iki makro **`d_type`** ile **`st_mode`** değerleri arasında dönüşüm yapar:

`int` **`IFTODT`** `(mode_t` *kip*`)` makro

kip'e karşı düşen **`d_type`** değeri ile döner.

`mode_t` **`DTTOIF`** `(int` *dtürü*`)` makro

dtürü'ne karşı düşen **st_mode** değeri ile döner.

Bu yapı ileride ek üyeler içerebilir. Yeni üyeler olduğunda derleme ortamında daima **_DIRENT_HAVE_D_***xxx* biçiminde bir makro ismiyle görünecektir. Burada *xxx* yeni üyenin ismidir. Örneğin, bazı sistemlerde varolan **d_reclen** üyesi **_DIRENT_HAVE_D_RECLEN** makrosu ile görünür.

Bir dosyanın çok sayıda ismi varsa, her ismin kendi dizin girdisi olur. Böyle dosya isimlerinin tek bir dosyaya ait olup olmadığı **d_fileno** alanındaki değere bakarak anlaşılır. Bu üyenin değeri bu tür girdilerde aynıdır.

Boyut, değişiklik zamanı gibi dosya öznitelikleri dosyanın kendisinde bulunur, bir dizin girdisinin elemenları değillerdir. Bkz. *Dosya Öznitelikleri* (sayfa: 371).

2.2. Bir Dizin Akımının Açılması

Bu bölümde bir dizin akımının nasıl açılacağından bahsedilecektir. Bu bölümdeki tüm semboller `dirent.h` başlık dosyasında bildirilmiştir.

DIR veri türü

DIR bir dizin akımını ifade eden veri türüdür.

Dizin erişim işlevleri sizin yerinize bu işi yaptığından, **struct dirent** veya **DIR** türünde nesneler ayırmamalısınız. Bu nesnelere gösterici döndüren işlevleri kullanarak bu nesnelere erişebilirsiniz.

`DIR *`**opendir**`(const char *`*dizinismi*`)` işlev

opendir işlevi ismi *dizinismi* olan dizini okumak için bir dizin akımı döndürür. Akım **DIR *** türündedir.

İşlev başarısız olursa boş gösterici döndürür. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EACCES
: **dirname** ile döndürülen dizini okuma izni verilmedi.

EMFILE
: Süreç çok fazla dosya açmış.

ENFILE
: Dizini içeren sistem ya da dosya sistemi şu anda daha fazla dosya açılmasını desteklemiyor. (GNU sisteminde böyle bir sorun asla olmaz.)

ENOMEM
: Yeterli bellek yok.

DIR türü genellikle bir dosya tanıtıcı kullanılarak gerçeklenir ve **opendir** işlevi de **open** ilkelini kullanır. Bkz. *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305). Dizin akımları ve daha alt seviyede dosya tanıtıcıları bir **exec** (*Bir Dosyanın Çalıştırılması* (sayfa: 688)) çağrısı yapıldığında kapatılırlar.

opendir tarafından okumak için açılan dizin ismiyle tanınır. Bazı durumlarda bu kafi değildir. Ya da **opendir** yoluyla dizin için örtük olarak oluşturulan bir dosya tanıtıcı yazılımda istenen yol değildir. Bu gibi durumlarda başka bir arayüz kullanılabilir.

`DIR *`**fopendir**`(int` *dosyatanıtıcı*`)` işlev

fdopendir işlevi bir dosya ismi almak ve dizin için bir dosya tanıtıcı açmak yerine dosya tanıtıcının çağrı sırasında belirtilmesini gerektirmesi dışında **opendir** gibi çalışır. Bu dosya tanıtıcı dönen dizin akımı nesnesinin daha sonraki kullanımlarında kullanılır.

İşlev çağrılırken kullanılan dosya tanıtıcının bir dizin ile ilişkili olduğundan ve okumaya izin verdiğinden emin olunmalıdır.

fdopendir çağrısı başarılı olursa dosya tanıtıcı sitemin denetimi altına girer. **opendir** tarafından örtük olarak oluşturulan tanıtıcının kullanıldığı gibi kullanılabilir fakat yazılım tanıtıcıyı kapatmamalıdır.

İşlev başarısız olduğu durumlarda bir boş gösterici döndürür ve dosya tanıtıcı yazılım tarafından kullanılabilir olarak kalır. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: Dosya tanıtıcı geçersiz.

ENOTDIR
: Dosya tanıtıcı dizinle ilgili değil

EINVAL
: Dosya tanıtıcı dizin içeriğinin okunmasına izin vermiyor.

ENOMEM
: Yeterli bellek yok.

Bazı durumlarda **opendir** çağrısı ile oluşturulan dosya tanıtıcıya erişmek istenebilir. Örneğin, çalışma dizinini bir dosya tanıtıcı kullanarak değiştiren **fchdir** işlevi için bu gerekli olabilirdi. Tarihsel olarak **DIR** türü ile fazla uğraşıldığından yazılımlar alanlarına erişemezler. Ancak GNU kütüphanesinde bu böyle değildir. Erişimi sağlamak için ayrı bir işlev vardır.

```
int dirfd(DIR *dizinakımı)                                    işlev
```

dirfd işlevi *dizinakımı* ile ilişkili dosya tanıtıcı ile döner. Dizin tanıtıcı dizin akımı **closedir** ile kapatılıncaya kadar geçerli kalır. Eğer dizin akımı gerçeklemesi dosya tanıtıcılarını kullanmıyorsa işlev **-1** ile döner.

2.3. Dizin Akımlarının Okunması ve Kapatılması

Bu bölümde bir dizin akımından girdilerin nasıl okunacağı ve akımla işiniz bittiğinde onu nasıl kapatacağınız anlatılacaktır. Bu bölümdeki tüm semboller `dirent.h` başlık dosyasında bildirilmiştir.

```
struct dirent *readdir(DIR *dizinakımı)                       işlev
```

Bu işlev dizindeki sonraki girdiyi okur. Normalde dosya hakkında bilgi içeren bir yapıya gösterici ile döner. Bu yapı durağan olarak ayrıldığından işlevin sonraki çağrıları üzerine yazabilir.

Taşınabilirlik Bilgisi

Bazı sistemlerde **readdir** işlevi **.** ve **..** için, bunlar her dizinde daima geçerli dosya isimleri oldukları halde, girdi döndürmez. Bkz. *Dosya İsmi Çözümlemesi* (sayfa: 233).

Eğer dizinde başka girdi kalmamışsa ya da bir hata saptanmışsa işlev bir boş gösterici ile döner. Aşağıdaki **errno** hata durumu bu işlev için tanımlanmıştır:

EBADF

dizinakımı argümanı geçersiz.

readdir işlevi evresel değildir. Çok evreli yazılımlarda aynı *dizinakımı* ile **readdir** kullanımı dönüş değerinin üzerine yazar. Bu durum sorun oluyorsa bu işlev yerine **readdir_r** işlevini kullanın.

```
int readdir_r(DIR             *dizinakımı,                                  işlev
              struct dirent   *girdi,
              struct dirent **sonuç)
```

Bu işlev **readdir** işlevinin evresel sürümüdür. **readdir** işlevi gibi dizindeki sonraki girdiyi okur. Fakat sonuç durağan ayrılmış bellekte saklanmadığından aynı anda çalışan evreler arasında bir soruna yol açmaz. Sonuç *girdi* ile gösterilen nesne içinde döner.

Normalde, **readdir_r** işlevi sıfırla döner ve *girdi *sonuç*'a atanır. Okunacak başka girdi kalmamışsa ya da bir hata saptanmışsa **readdir_r** **sonuç*'a bir boş gösterici atar ve sıfırdan farklı bir hata kodu ile döner. Ayrıca bu hata kodunu **errno** değişkenine atar. Bu işlev için tanımlanmış hata kodları **readdir** ile aynıdır.

Taşınabilirlik Bilgisi

Bazı sistemlerde **struct dirent** yapısının **d_reclen** diye bir üyesi olmasa ve dosya isminin izin verilen en büyük boyutta olabilmesine izin verilse bile **readdir_r** dosya ismi için boş karakter sonlandırmalı dizge döndürmeyebilir. Günümüzdeki sistemlerin tamamı **d_reclen** alanına sahiptir ve eski sistemlerde çok evrelilik pek sorun oluşturmaz. Her durumda **readdir** ile ilgili böyle bir sorun yoktur, yani **d_reclen** üyesine sahip olmayan sistemlerde bile harici kilitleme ile çok evrelilik mümkün olur.

Ayrıca, **struct dirent** türünün tanımlanması da önem kazanır. **readdir_r** işlevinin ikinci parametresinde bu türde bir nesneye basitçe gösterici atanması yeterli olmayabilir. Bazı sistemler **d_name** elemanını yeterli uzunlukta tanımlamazlar. Bu durumda kullanıcının ek alan sağlaması gerekir. **d_name** dizisinde en azından **NAME_MAX + 1** karakterlik alan bulunmalıdır. **readdir_r** çağrısı şöyle yapılabilir:

```
union
{
  struct dirent d;
  char b[offsetof (struct dirent, d_name) + NAME_MAX + 1];
} u;

if (readdir_r (dir, &u.d, &res) == 0)
  …
```

32 bitlik makinalar üzerinde büyük dosyasistemlerini desteklemek için son iki işlevin LFS sürümleri de vardır.

```
struct dirent64 *readdir64(DIR *dizinakımı)                                 işlev
```

readdir64 işlevi **struct dirent64** türünde bir gösterici döndürmesi dışında **readdir** işlevi gibidir. Bu veri türünün bazı üyeleri (özellikle **d_ino** üyesi) büyük dosyasistemlerini desteklemek üzere farklı boyutta olabilir.

Diğer bakımlardan bu işlev **readdir** işleviyle eşdeğerdedir.

```
int readdir64_r(DIR               *dizinakımı,                              işlev
                struct dirent64   *girdi,
                struct dirent64 **sonuç)
```

readdir64_r işlevi 2. ve 3. parametrelerini **struct dirent64** türünde alması dışında **readdir_r** işlevi gibidir. **readdir_r** işlevinin açıklamalarında dikkat çekilen hususlar bu işlev için de geçerlidir.

`int` **closedir**`(DIR *`*dizinakımı*`)` işlev

Bu işlev *dizinakımı* ile belirtilen dizin akımını kapatır. Başarılıysa **0**, değilse **-1** döndürür.

Aşağıdaki **errno** hata durumu bu işlev için tanımlanmıştır:

EBADF
dizinakımı argümanı geçersiz.

2.4. Bir Dizinin İçeriğini Listeleyen Bir Örnek

Burada verilen örnek kod çalışma dizini içindeki dosyaların isimlerini listelemektedir:

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

int
main (void)
{
  DIR *dp;
  struct dirent *ep;

  dp = opendir ("./");
  if (dp != NULL)
    {
      while (ep = readdir (dp))
        puts (ep->d_name);
      (void) closedir (dp);
    }
  else
    perror ("Dizin açılamadı");

  return 0;
}
```

Çıktılanan listede dosyaların sırası hemen hemen rasgeledir. Daha kullanışlı bir yazılım girdileri basmadan önce sıraya sokardı; bkz. *Dizin İçeriğinin Taranması* (sayfa: 359) ve *Dizi Sıralama İşlevi* (sayfa: 204).

2.5. Dizin Akımında Rasgele Erişim

Bu bölümde açık bir dizin akımından zaten okunmuş olan bir dizin parçasının yeniden nasıl okunacağından bahsedilecektir. Bu bölümdeki tüm semboller `dirent.h` başlık dosyasında bildirilmiştir.

`void` **rewinddir**`(DIR *`*dizinakımı*`)` işlev

rewinddir işlevi *dizinakımı* ile belirtilen dizin akımını yeniden ilklendirmekte kullanılır. Böylece bir **readdir** çağrısı ile dizindeki ilk girdi ile ilgili bilgiyi tekrar döndürebilirsiniz. Dizin daha önce **opendir** ile açılmış olduğundan bu arada silinmiş ya da eklenmiş dosyalar varsa onları farketmenizi de sağlar. Yani son **opendir** veya **rewinddir** çağrısından sonra yapılan **readdir** çağrılarıyla eklenmiş dosyalar varsa bunlarla ilgili bilgiler dönerken silinmiş dosyalarla ilgili bilgi dönmeyecekir.

`long int` **telldir**`(DIR *`*dizinakımı*`)` işlev

telldir işlevi *dizinakımı* ile belirtilen dizin akımının dosya konumunu döndürür. Dizin akımının dosya konumunu tekrar bu noktaya getirmek için **seekdir** işlevini bu değerle kullanabilirsiniz.

```
void seekdir(DIR     *dizinakımı,                                    işlev
             long int konum)
```

seekdir işlevi *dizinakımı* ile belirtilen dizin akımının dosya konumunu *konum* konumuna ayarlar. *konum* değeri bu akım için daha önceki bir **telldir** çağrısından dönen değer olmalıdır. Ancak arada akım kapatılıp yeniden açılmışsa bu değer geçersiz olmuş olabilir.

2.6. Dizin İçeriğinin Taranması

Dizinlerle çalışan işlevlerin daha yüksek seviyedeki bir arayüzü de **scandir** işlevidir. Onun yardımıyla, bir dizin içindeki girdilerin bir kısmı seçilebilir, sıralanabilir ve bir isim listesi alınabilir.

```
int scandir(const char      *dizin,                                  işlev
            struct dirent ***isimlistesi,
            int            (*seçici) (const struct dirent *),
            int            (*sıralayıcı) (const void *, const void *))
```

scandir işlevi *dizin* ile belirtilen dizinin içeriğini tarar. Sonucu döndüren **isimlistesi* seçilen dizin girdilerini içeren **struct dirent** türündeki göstericilerin dizisidir ve **malloc** kullanılarak ayrılır. Dizindeki tüm girdilerin değilde sadece seçilen girdilerin döndürülmesi için işlev kullanıcı tarafından sağlanan *seçici* işlevi çağırır. Sadece *seçici* işlevin sıfırdan farklı bir değer döndürdüğü girdiler seçilir.

Son olarak, **isimlistesi* içindeki girdiler kullanıcı tarafından sağlanan *sıralayıcı* işlevi çağrılarak sıraya sokulur. *sıralayıcı* işlevine aktarılan argümanlar **struct dirent **** türündedir, bu bakımdan **strcmp** veya **strcoll** işlevleri doğrudan kullanılamaz. Kullanılabilecek işlevler için aşağıya **alphasort** ve **versionsort** işlevlerine bakınız.

İşlevin normal dönüş değeri **isimlistesi*'ne yerleştirilen girdilerin sayısıdır. **-1** dönmüşse bir hata saptanmış demektir (ya dizin okumak için açılamamıştır ya da malloc başarısız olmuştur). Bu durumda **errno** değişkenine hata durumu atanır.

Yukarıda açıklanan **scandir** işlevinin dördüncü argümanı bir sıralama işlevine bir göstericidir. Yazılımcıya kolaylık olarak GNU C kütüphanesi bu amaca uygun olarak gerçeklenmiş işlevler içerir.

```
int alphasort(const void *a,                                         işlev
              const void *b)
```

alphasort işlevi **strcoll** işlevi gibi davranır (bkz. *Dizi/Dizge Karşılaştırması* (sayfa: 104)). Aradaki fark, argümanların dizge göstericisi değil **struct dirent **** türünde olmasıdır.

alphasort işlevinin dönüş değeri a ve b girdilerinin karşılaştırılmasına bağlı olarak sıfırdan küçük bir değer, sıfır ya da sıfırdan büyük bir değer olur.

```
int versionsort(const void *a,                                       işlev
                const void *b)
```

versionsort işlevi dahili olarak **strverscmp** işlevini kullanması dışında **alphasort** işlevi gibidir.

Eğer dosya sistemi büyük dosyaları destekliyorsa, **dirent** yapısının tüm bilgiyi içermesi mümkün olmadığından **scandir** işlevi kullanılamaz. LFS sistemi için yeni bir tür, **struct dirent64**, bunu kullanmak için de yeni bir işlev vardır.

```
int scandir64(const char        *dizin,                              işlev
              struct dirent64 ***isimlistesi,
              int              (*seçici) (const struct dirent64 *),
              int              (*sıralayıcı) (const void *, const void *))
```

scandir64 işlevi dizin girdilerini **struct dirent64** türünde döndürmesi dışında **scandir** işlevi gibidir. *seçici* ile gösterilen işlev yine, istenen girdileri seçmek için kullanılır, ama işlev bu sefer **struct dirent64 *** türünde bir parametre alır.

Benzer şekilde, *sıralayıcı* işlevi de **struct dirent64 **** türünde iki parametre alır.

sıralayıcı bu sefer farklı türde argümanlar aldığından ve **alphasort** ve **versionsort** işlevleri bu türde argüman almadıklarından burada kullanılamazlar. Bu bakımdan iki işlev daha tanımlanmıştır.

```
int alphasort64(const void *a,                                          işlev
                const void *b)
```

alphasort64 işlevi **strcoll** işlevi gibi davranır (bkz. *Dizi/Dizge Karşılaştırması* (sayfa: 104)). Aradaki fark, argümanların dizge göstericisi değil **struct dirent64 **** türünde olmasıdır.

alphasort64 işlevinin dönüş değeri a ve b girdilerinin karşılaştırılmasına bağlı olarak sıfırdan küçük bir değer, sıfır ya da sıfırdan büyük bir değer olur.

```
int versionsort64(const void *a,                                        işlev
                  const void *b)
```

versionsort64 işlevi dahili olarak **strverscmp** işlevini kullanması dışında **alphasort64** işlevi gibidir.

scandir kullanırken 64 bitlik karşılaştıma işlevlerini kullanmamak (ya da tam tersi) önemlidir. Bunun çalıştığı sistemler varsa da diğerleri ümitsizce başarısız olacaktır.

2.7. Bir Dizinin İçeriğini Listeleyen İkinci Örnek

Burada, *Bir Dizinin İçeriğini Listeleyen Bir Örnek* (sayfa: 358) bölümündeki küçük yazılımın biraz daha gelişirilmiş bir sürümü vardır. Dizin içeriğini sıralamak için uğraşmaktansa **scandir** kullanmayı tercih ettik. Çağrı döndükten sonra girdiler hemen kullanıma hazır hale gelir.

```
#include <stdio.h>
#include <dirent.h>

static int
one (const struct dirent *unused)
{
  return 1;
}

int
main (void)
{
  struct dirent **eps;
  int n;

  n = scandir ("./", &eps, one, alphasort);
  if (n >= 0)
    {
      int cnt;
      for (cnt = 0; cnt < n; ++cnt)
        puts (eps[cnt]->d_name);
    }
  else
    perror ("Dizin açılamadı");
```

```
  return 0;
}
```

Bu örnekteki seçici işlevin basitliğine dikkat edin. Dizindeki tüm girdileri listelemek istediğimizden hep **1** döndürdük.

3. Dizin Ağaçlarıyla Çalışma

Buraya kadar açıklanan işlevler ya bilgiyi bit bit aldılar ya da tüm dosyaları grup halinde işleme soktular (bkz. **scandir**). Bazan alt dizinler ve içerdikleri dosyalarla çalışmak gerekir. X/Open belirtimi bunu yapmak için iki işlev tanımlamıştır. Daha basit hali System V sistemlerindeki ilk tanımından türetilmiştir ve bu bakımdan bu işlev SVID'den türetilmiş sistemlerde bulunur. Prototipler ve gerekli tanımlar `ftw.h` başlık dosyasında bulunabilir.

Bu ailenin dört işlevi vardır: **ftw**, **nftw** ile 64 bitlik olanları **ftw64** ve **nftw64**. Bu işlevlerin argümanlarından biri uygun türde bir eylemci işleve göstericidir.

```
__ftw_func_tint (*) (const char *, const struct stat *, int)
```
veri türü

ftw işlevine belirtilen eylemci işlevin türüdür. İlk parametre dosya ismine bir gösterici, ikinci parametre ise ilk parametrede ismi belirtilen dosya için doldurulan **struct stat** türünde bir nesnedir.

Son parametre o anki dosya hakkında daha fazla bilgi veren bir değerdir. Son parametre şu değerleri içerebilir:

`FTW_F`
: Öğe ya normal bir dosya ya da diğer kategorilerle eşleşmeyen (özel dosyalar, soketler gibi) bir dosyadır.

`FTW_D`
: Öğe bir dizindir.

`FTW_NS`
: **stat** çağrısı başarısız olduğundan ikinci parametrenin gösterdiği bilgi geçersizdir.

`FTW_DNR`
: Öğe okunamayan bir dizindir.

`FTW_SL`
: Öğe bir sembolik bağdır. Sembolik bağlar normalde izlendiğinden, bu değerin bir **ftw** eylemci işlevinde görünmesi sembolik bağın hedefindeki dosyanın mevcut olmadığı anlamına gelir. **nftw** işlevinde durum daha farklıdır.

  Bu değer sadece, ilk başlık dosyasından önce **_BSD_SOURCE** veya **_XOPEN_EXTENDED** tanımlanarak derlenmiş bir yazılımda kullanılabilir. Özgün SVID sistemlerinde sembolik bağlar yoktur.

Eğer kaynaklar **_FILE_OFFSET_BITS == 64** ile derlenmişse, bu tür aslında **__ftw64_func_t** türü olur. Aynı sebeple **struct stat** da **struct stat64** olur.

LFS arayüzü ve **ftw64** işlevinde kullanmak üzere **__ftw64_func_t** türü `ftw.h` dosyasında tanımlanmıştır.

```
__ftw64_func_tint (*) (const char *, const struct stat64 *, int)
```
veri türü

Bu tür **ftw64** işlevine belirtilen eylemci işlevin türü olmak dışında **__ftw_func_t** türünün benzeridir. Eylemci işlevin ikinci parametresi daha geniş değerleri mümkün kılmak için **struct stat64** türünde bir değişkene göstericidir.

`__nftw_func_t` `int (*) (const char *, const struct stat *, int, struct FTW *)` veri türü

İlk üç parametresi **__ftw_func_t** türü ile aynıdır. Ancak üçüncü argüman için daha hassas bir farklılaşmayı sağlamak için bazı ek değerler içerebilir:

`FTW_DP`
: Öğe bir dizindir ve tüm alt dizinlerine girilmiş ve raporlanmıştır. Eğer **nftw** işlevi **FTW_DEPTH** seçeneği ile çağrılmışsa **FTW_D** yerine bu değer döner (aşağıya bakınız).

`FTW_SLN`
: Öğe geçerliğini yitirmiş bir sembolik bağ. Yani gösterdiği dosya ortada yok.

Eylemci işlevin son parametresi aldığı ek değerler aşağıda açıklanan bir yapıya göstericidir.

Eğer kaynaklar **_FILE_OFFSET_BITS == 64** ile derlenmişse, bu tür aslında **__nftw64_func_t** türü olur. Aynı sebeple **struct stat** da **struct stat64** olur.

LFS arayüzü ve **ftw64** işlevinde kullanmak üzere **__nftw64_func_t** türü `ftw.h` dosyasında tanımlanmıştır.

`__nftw64_func_t` `int (*) (const char *, const struct stat64 *, int, struct FTW *)` veri türü

Bu tür **nftw64** işlevine belirtilen eylemci işlevin türü olmak dışında **__nftw_func_t** türünün benzeridir. Eylemci işlevin ikinci parametresi daha geniş değerleri mümkün kılmak için **struct stat64** türünde bir değişkene göstericidir.

`struct FTW` veri türü

Bu yapının içerdiği bilgi isim parametresinin yorumlanmasına ve dizin hiyerarşisinin zikzaklı durumu hakkında bazı bilgiler verilmesine yardımcı olur.

`int` **base**
: Değeri eylemci işlevin ilk parametresinde aktarılan dizgenin dosya isminin başlangıcına göre başlangıç konumudur. Dosya isminin başlangıcında kalan bölüm dosyanın dosya yoludur. Çalışma dizini o an bulunan öğelerden biri olduğundan, **nftw** çağrısı **FTW_CHDIR** seçeneği ile yapıldığında bu bilgi önem kazanır.

`int` **level**
: İşlem sırasında, dosyayı bulmak için kaç dizin içeri gidileceğini gösterir. İlk dizinin seviyesi 0'dır.

```
int ftw(const char  *dosyaismi,                                  işlev
        __ftw_func_t eylemci–işlev,
        int          tanıtıcı–sayısı)
```

Bu işlev, *dosyaismi* ile belirtilen dizin ve alt dizinlerinde bulunan her öğe için *eylemci–işlev* parametresi ile belirtilen işlevi çağırır. İşlev gerekirse sembolik bağları da izler ama öğeyi iki defa işleme sokmaz. Eğer *dosyaismi* ile bir dizin belirtilmemişse eylemci işlev sadece bu öğe için çağrılır.

Eylemci işleve aktarılan dosya ismi *dosyaismi* parametresinden alınarak ve aktarılan tüm dizin isimleri ve yerel dosya ismi eklenerek oluşturulur. Böylece eylemci işlev dosyaya erişmek için bu parametreyi kullanabilir. **ftw** ayrıca dosya için **stat** çağrısı da yapar ve bu bilgiyi eylemci işleve aktarır. Eğer bu **stat** çağrısı başarısız olursa bu durum eylemci işlevin üçüncü argümanına **FTW_NS** aktarılarak belirtilir. Aksi takdirde, üçüncü argümana yukarıda **__ftw_func_t** açıklamasında belirtildiği gibi aktarım yapılır.

Bir hata oluşmadığını ve işlemin devam edebileceğini belirtmek üzere eylemci işlevin 0 döndürmesi beklenir. Eğer bir hata oluşmuşsa ya da **ftw** işlevinin işlemi hemen sonlandırması isteniyorsa sıfırdan farklı bir değer döndürülmelidir. Bu işlevi sonlandırmanın tek yoludur. Eylemci işlevin içinde işleme başka bir yerde devam etmek için **setjmp** veya benzeri bir işlev kullanılmamalıdır. Bu, **ftw** işlevinin ayırdığı özkaynakların ayrılmış olarak kalmasına sebep olur.

tanıtıcı–sayısı parametresi ile **ftw** işlevinin toplam kaç dosya tanıtıcısı kullanacağı belirtilir. Ne kadar çok dosya tanıtıcısı kullanmasına izin veririrse işlev o kadar hızlı çalışır. Dizin hiyerarşisindeki her alt dizin için en fazla bir dosya tanıtıcısı kullanılır, fakat çok derinlere inildiğinde açık dosya tanıtıcılarının sayısı süreç ya da sistem için belirlenmiş sınırları aşabilir. Dahası, çok evreli yazılımlarda bu katlanarak artar. Bu bakımdan açık dosya tanıtıcılarının sayısına kabul edilebilir bir sınır belirtmek gerekir.

Eğer tüm eylemci işlev çağrıları 0 ile dönmüşse ve **ftw** tüm eylemleri uygulayabilmişse, **ftw** 0 ile döner. Eğer bir işlev çağrısı başarısız olmuşsa (**stat** çağrıları hariç) –1 ile döner. Eğer eylemci işlev çağrılarından biri sıfırdan farklı bir değerle dönmüşse **ftw** işlevi bu dönüş değerini döndürür.

Kaynakların 32 bitlik bir sistemde **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev aslında **ftw64** işlevidir, yani LFS arayüzü eski arayüzün yerine geçer.

```
int ftw64(const char     *dosyaisimi,                                    işlev
          __ftw64_func_t  eylemci–işlev,
          int             tanıtıcı–sayısı)
```

Büyük dosyalı dosya sistemleri ile çalışması dışında bu işlev **ftw** işlevinin benzeridir. Eylemci işleve dosya bilgisi aktarılırken **struct stat64** türünde bir değişken kullanılır.

32 bitlik bir sistemde, kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev **ftw** ismiyle bulunur ve eski gerçekleme tamamen LFS'ye uygun olarak değiştirilir.

```
int nftw(const char    *dosyaisimi,                                     işlev
         __nftw_func_t  eylemci–işlev,
         int            tanıtıcı–sayısı,
         int            seçenek)
```

nftw işlevi **ftw** işlevi gibi çalışır. *dosyaisimi* ile belirtilen dizin ve alt dizinlerinde ve aşağıda açıklandığı gibi bulunan her öğe için *eylemci–işlev* parametresi ile belirtilen işlevi çağırır. *tanıtıcı–sayısı* parametresi ile **nftw** işlevinin toplam kaç dosya tanıtıcısı kullanacağı belirtilir.

Birinci fark eylemci işlevin türüdür. Yukarıda açıklandığı gibi **struct FTW *** türündeki eylemci işleve ek bilgi aktarılabilir.

İkinci fark, **nftw** işlevinin dördüncü bir argüman almasıdır. Bu argümana 0 veya aşağıdaki değerlerin bit seviyesinde VEYAlanmış değeri aktarılabilir:

FTW_PHYS
: Dizin taranırken sembolik bağlar izlenmez. Bunun yerine sembolik bağlar eylemci işlevin tür parametresinde **FTW_SL** değeri kullanılarak belirtilir. Eğer sembolik bağın hedefindeki dosya mevcut değilse, bunun yerine **FTW_SLN** döner.

FTW_MOUNT
: Eylemci işlevden *dosyaismi* ile belirtilen dizinin bulunduğu dosya sistemine bağlı diğer dosya sistemlerindeki alt dizinlerle ilgili bilgi istenmez.

FTW_CHDIR

Bu seçenek verilmişse eylemci işlev çağrılmadan önce çalışma dizininden raporlanan dizine geçilir. **ntfw** işlevi eylemci işlev döndükten sonra tekrar eski çalışma dizinine geçer.

FTW_DEPTH
: Bu seçenek verilmişse ana dizin içeriği işlenmeden önce alt dizinler ve onların dosyaları üzerinde işlem yapılır (önce derinlik kipi). Bu durum ayrıca eylemci işleve **FTW_D** değil **FTW_DP** değeri aktarılarak belirtilir.

FTW_ACTIONRETVAL
: Bu seçenek belirtilmişse eylemci işlevin dönüş değeri farklı işlem görür. Eğer eylemci işlev **FTW_CONTINUE** ile dönerse işlem normal olarak devam eder. **FTW_STOP** işlemi durdur ve işlev bu değerle döner. Eğer **FTW_D** argümanı ile çağrılmış eylemci işlev **FTW_SKIP_SUBTREE** değeri ile dönerse, alt ağaç atlanır ve sonraki kardeş dizinden devam edilir; **FTW_SKIP_SIBLINGS** değeri dönerse, o anki girdinin tüm kardeş dizinleri atlanır ve işlem bir üst dizinden devam eder. Bu seçeneğin belirtildiği durumda eylemci işlev bu değerlerin dışında bir değer döndürmemelidir. Bu seçenek bir GNU oluşumudur.

İşlevin dönüş değeri **ftw** işlevindeki gibi değerlendirilir. Eğer tüm eylemci işlev çağrıları **0** ile dönmüşse ve **nftw** tüm eylemleri uygulayabilmişse, **nftw** işlevi **0** ile döner. Bellek sorunu gibi bir dahili bir hata oluşmuşsa işlev **-1** ile döner ve hata durumu *errno* değişkenine atanır. Eğer eylemci işlev çağrılarından biri sıfırdan farklı bir değerle dönmüşse **nftw** işlevi bu dönüş değerini döndürür.

Kaynakların 32 bitlik bir sistemde **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev aslında **nftw64** işlevidir, yani LFS arayüzü eski arayüzün yerine geçer.

```
int nftw64(const char      *dosyaismi,                                    işlev
           __nftw64_func_t eylemci–işlev,
           int             tanıtıcı–sayısı,
           int             seçenek)
```

Büyük dosyalı dosya sistemleri ile çalışması dışında bu işlev **nftw** işlevinin benzeridir. Eylemci işleve dosya bilgisi aktarılırken **struct stat64** türünde bir değişken kullanılır.

32 bitlik bir sistemde, kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev **nftw** ismiyle bulunur ve eski gerçekleme tamamen LFS'ye uygun olarak değiştirilir.

4. Sabit Bağlar

POSIX sistemlerinde bir dosyanın aynı anda çok sayıda ismi olabilir. İsimlerin her biri aynı değerdedir ve biri diğerine tercih edilmez.

Bir dosyaya isim eklemek için **link** işlevi kullanılır. (Bu yeni isme ayrıca dosyaya ***sabit bağ*** da denir.) Bir dosyaya yeni bir isim atanması dosya içeriğinin kopyalanmasına sebep olmaz; sadece dosyanın isimlerine yeni bir isim eklenmiş olur.

Bir dosya çeşitli dizinlerde isimlere sahip olabilir, böyle bir dosya sisteminin düzeni kesin bir hiyerarşi ya da ağaç olmaz.

Çoğu gerçeklemelerde, aynı dosyanın farklı dosya sistemlerinde sabit bağlarının olmasına izin verilmez. **link** işlevini böyle bir işlem yapmak için kullanmaya çalışırsanız, diğer dosya sistemi için bu işlemin yapılamayacağını belirten bir hata raporu alırsınız.

link işlevinin prototipi unistd.h başlık dosysında bildirilmiştir.

```
int link(const char *eski–isim,                                          işlev
         const char *yeni–isim)
```

link işlevi *eski–isim* isimli dosyaya *yeni–isim* isimli bir sabit bağ yapar.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EACCES
: Yeni bağın yazılacağı dizine yazma izni verilmedi.

EEXIST
: *yeni–isim* isminde bir dosya zaten var. Bu ismi yeni bağ ismi olarak kullanmak istiyorsanız önce mevcut ismi değiştirmelisiniz.

EMLINK
: *eski–isim* isimli dosya için zaten çok fazla ek isim var. (Bir dosyaya verilecek bağ sayısı **LINK_MAX** ile sınırlıdır; bkz. *Dosya Sistemi Kapasite Sınırları* (sayfa: 795).)

ENOENT
: *eski–isim* isminde bir dosya yok. Olmayan bir dosyaya bağ oluşturamazsınız.

ENOSPC
: Yeni bağı içerecek dizin ya da dosya sisteminde yer yok ve genişletilemiyor.

EPERM
: GNU sisteminde ve bazı sistemlerde dizinlere sabit bağ yapamazsınız. Bir çok sistem bu izni sadece ayrıcalıklı kullanıcılara verir. Bu hata sorunu raporlamakta kullanılır.

EROFS
: Yeni bağı içerecek dizine dosya sistemi salt–okunur bağlı olduğundan yazılamıyor.

EXDEV
: *yeni–isim* ile belirtilen dizin mevcut dosyadan farklı bir dosya sisteminde.

EIO
: Diske okuma ya da yazma denemesi sırasında bir donanım hatası oluştu.

5. Sembolik Bağlar

GNU sistemi ***sembolik bağları*** destekler. Bu aslında bir dosya ismine gösterici olan bir dosya çeşididir. Sabit bağların tersine, dizinlere ve diğer sosya sistemlerindeki dosyalara ve dizinlere sembolik bağlar yapılabilir. Ayrıca, olmayan bir dosyaya da bir sembolik bağ yapılabilir. Tersine sembolik bağın hedefindeki dosyanın silinmesi sembolik bağı sadece geçersiz hale getirir. Hedef dosya oluşturulana kadar bağ dosyası başarısız olarak kalır.

Sembolik bağların tercih edilmesinin bir başka sebebi de bağı açmaya çalıştığınızda bazı özel şeylerin yapılmasıdır. **open** işlevine dosya ismi olarak bir sembolik bağ ismi vererek bir bağı açmaya çalışırsanız, bağın içerdiği dosya ismini okur ve onun yerine bu dosyayı açar. **stat** işlevi ise tersine sembolik bağın kendisi ile değil, gösterdiği dosya ile çalışır.

Ayrıca, dosya silme ve isim değiştirme gibi bazı işlemler bağın kendisi üzerinde yapılır. **readlink** ve **lstat** işlevleri ayrıca sembolik bağları izlemekten kaçınır, çünkü onların amacı bağ hakkında bilgi sağlamaktır. Sabit bağ yapan **link** işlevi sembolik bağlara da sabit bağ yapar.

Bazı sistemlerde, dosyalarla çalışan bazı işlevlerin bir dosya yolu çözümlenirken kaç tane sembolik bağın izleneceğine ilişkin bir sınırlama vardır. Bu sınır sistemde eğer varsa `sys/param.h` başlık dosyasında tanımlıdır.

`int` **MAXSYMLINKS** makro

MAXSYMLINKS makrosu bazı işlevlerin **ELOOP** değerini döndürmeden kaç tane sembolik bağ izleyebileceğini belirtir. Tüm işlevler böyle davranmaz ve bu değer **sysconf** işlevinin **_SC_SYMLOOP** için döndürdüğü değerle aynı değildir. Aslında, **sysconf** böyle bir sınır olmadığını belirten bir değer döndürür.

Bu bölümdeki işlevlerin çoğu `unistd.h` başlık dosyasında bildirilmiştir.

```
int symlink(const char *eski–isim,                                    işlev
            const char *yeni–isim)
```

symlink işlevi *eski–isim* isimli dosyaya *yeni–isim* isimli bir sembolik bağ yapar.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EEXIST`
: *yeni–isim* isminde bir dosya zaten var.

`EROFS`
: Yeni bağı içerecek dizine dosya sistemi salt–okunur bağlı olduğundan yazılamıyor.

`ENOSPC`
: Yeni bağı içerecek dizin ya da dosya sisteminde yer yok ve genişletilemiyor.

`EIO`
: Diske okuma ya da yazma denemesi sırasında bir donanım hatası oluştu.

```
int readlink(const char *dosyaismi,                                   işlev
             char       *tampon,
             size_t      boyut)
```

readlink işlevi sembolik bağın ismini *dosyaismi* ile alır ve sembolik bağın hedefindeki dosyanın ismini **tampon*'a yerleştirir. Dosya ismi dizgesi boş karakter sonlandırmalı *değildir*; işlev *tampon* ile gösterilen dizgeye yerleştirilen karakterlerin sayısı ile döner, dolayısıyla *boyut* argümanı en azından bu değerde olmalıdır.

Eğer işlev *boyut* değerine eşit bir değerle dönmüşse, dosya ismi için yeterli yerin olup olmadığı hakkında bir fikir vermez. Bu durumda tamponu büyütüp tekrar denemelisiniz. Bir örnek:

```
char *
readlink_malloc (const char *filename)
{
  int size = 100;
  char *buffer = NULL;

  while (1)
    {
      buffer = (char *) xrealloc (buffer, size);
      int nchars = readlink (filename, buffer, size);
      if (nchars < 0)
        {
          free (buffer);
          return NULL;
        }
      if (nchars < size)
        return buffer;
      size *= 2;
    }
}
```

İşlevin dönüş değeri **-1** ise bir hata oluşmuş demektir.*Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EINVAL
: İsmi belirtilen dosya bir sembolik bağ değil.

EIO
: Diske okuma ya da yazma denemesi sırasında bir donanım hatası oluştu.

Bazı durumlarda sembolik bağların hedefindeki gerçek dosya isminin başka hiçbir sembolik bağ, önek ve dosya yolunda **.** veya **..** olmaksızın çözümlenmesi istenebilir. Bu isteğe bir örnek, aynı dosya düğümüne erişen farklı dosya isimlerinin karşılaştırılmasıdır.

```
char *canonicalize_file_name(const char *isim)                    işlev
```

canonicalize_file_name işlevi *isim* ile belirtilen dosya isminin ne **.**, **..** elemanları ne herhangi bir dosya yolu ayracı (**/**) ne de sembolik bağ içeren mutlak ismini döndürür. İşlev dönüş değerini **malloc** ile ayrılmış bellek bloğunda döndürür. Eğer dönen sonuç artık kullanılmayacaksa **free** çağrısı ile serbest bırakılmalıdır.

Bir dosya yolu elemanı yoksa işlev bir boş gösterici ile döner. Bu ayrıca, dosya yolunun uzunluğu **PATH_MAX** karakteri aşarsa da döndürülür. Her durumda **errno** değişkenine hata durumu atanır.

ENAMETOOLONG
: Sonuçlanan dosya yolu çok uzun. Bu hata dosya isimlerinde uzunluk sınırı olan dosya sistemlerinde oluşur.

EACCES
: Dosya yolunu oluşturan elemanlardan en az biri okunamıyor.

ENOENT
: Girdi dosya ismi boş.

ENOENT
: Dosya yolunu oluşturan elemanlardan en az biri mevcut değil.

ELOOP
: **MAXSYMLINKS**'den fazla sembolik bağ izlendi.

Bu işlev bir GNU oluşumudur ve `stdlib.h` başlık dosyasında bildirilmiştir.

Unix standardı **canonicalize_file_name** işlevinin bir benzeri olarak sonucun yerleştirileceği tamponu kullanıcının belirteceği bir işlev tanımlamıştır.

```
char *realpath(const char *restrict isim,                         işlev
               char *restrict       sonuç)
```

realpath işlevi *sonuç* parametresinde boş gösterici belirtildiğinde tamamen **canonicalize_file_name** gibi davranır. İşlev bir tampon ayırır ve ona bir gösterici ile döner. Eğer *sonuç* **NULL** değilse, sonuç onun gösterdiği tampona kopyalanır. Çağrıcıya yanıtı yeterince büyük bir tampon ayırmaktır. **PATH_MAX**'ın tanımlı olduğu sistemlerde tampon bu uzunlukta olmalıdır. Dosya yolu uzunluğu için bir sınırlama bulunmayan sistemlerde uzunluk tahmin edilemeyeceğinden **realpath** işlevi ikinci parameresinde **NULL** dışında bir değer belirterek çağrılmamalıdır.

Bir diğer fark da, işlev **NULL** ile döndüğünde *sonuç* tamponunun (sıfırdan farklıysa) mevcut olmayan ya da okunamayan dosya yolu parçaları içereceğidir. Bu durumda **errno** değişkenine **EACCES** ya da **ENOENT** atanır.

Bu işlev `stdlib.h` dosyasında bildirilmiştir.

Bu işlevi kullanmanın bir faydası da geniş çapta kullanım alanı olmasıdır. Sakıncası ise uzun dosya isimleri için sınırlama olmayan sistemlerde uzun dosya yolu başarısızlıkları raporlamasıdır.

6. Dosyaların Silinmesi

Bir dosyayı **unlink** veya **remove** ile silebilirsiniz.

Silme işleminde aslında sadece dosyanın ismi silinir. Eğer dosya sadece isimden ibaretse dosyanın kendisi de silinmiş olur. Eğer dosyanın başka isimleri de varsa onlar bu isimler altında hala erişilebilir olur (bkz. *Sabit Bağlar* (sayfa: 364)).

`int` **unlink**`(const char *`*dosyaismi*`)` işlev

unlink işlevi *dosyaismi* ile belirtilen dosya ismini siler. Eğer dosya sadece isimden ibaretse dosyanın kendisi de silinir. (Aslında, eğer herhangi bir süreç dosyayı açmışsa silme işlemi süreçler dosyayı kapatıncaya kadar ertelenir.)

unlink işlevi `unistd.h` başlık dosyasında bildirilmiştir.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EACCES`
: Silinecek dosyanın bulunduğu dizinde yazma izniniz yok ya da dizinde yapışkan bit var ve dosya size ait değil.

`EBUSY`
: Bu hata dosyanın sistem tarafından kullanıldığından silinemeyeceğini belirtir. Örneğin, dosya ismi kök dizini ya da bir dosya sisteminin bağlı olduğu dizini belirtiyorsa bu hata oluşur.

`ENOENT`
: Silinmek istenen dosya mevcut değil.

`EPERM`
: Bazı sistemlerde **unlink** bir dizini silmek için kullanılamaz ya da en azından sadece ayrıcalıklı kullanıcı bunu yapabilir. Böyle sorunlarla karşılaşmamak için dizinleri silerken **rmdir** kullanın. (GNU sisteminde **unlink** kesinlikle bir dizin ismini silemez.)

`EROFS`
: Silinecek dosya bir salt–okunur bağlı dosya sisteminde ve bu değiştirilemez.

`int` **rmdir**`(const char *`*dosyaismi*`)` işlev

rmdir işlevi bir dizini siler. Silinecek dizin boş olmalıdır; başka bir deyişle, sadece **.** ve **..** girdilerini içermelidir.

Birçok bakımdan, **rmdir** işlevi **unlink** gibi davranır. Bu işlev için iki hata durumu daha tanımlanmıştır:

`ENOTEMPTY`
`EEXIST`

Silinecek dizin boş değil.

Bu iki hata kodu bir diğerinin eşanlamlısıdır, bazı sistemlerde biri, bazılarında öbürü kullanılır. GNU sisteminde daima **ENOTEMPTY** kullanılır.

rmdir işlevi `unistd.h` başlık dosyasında bildirilmiştir.

```
int remove(const char *dosyaismi)                                    işlev
```

Bu bir dosyayı silmek için kullanılan bir ISO C işlevidir. Dosyalarla **unlink** gibi, dizinlerle **rmdir** gibi çalışır. **remove** işlevi `stdio.h` başlık dosyasında bildirilmiştir.

7. Dosya İsimlerinin Değiştirilmesi

Bir dosyanın ismini değiştirmek için **rename** işlevi kullanılır.

```
int rename(const char *eski–isim,                                    işlev
           const char *yeni–isim)
```

rename işlevi ismi *eski–isim* ile belirtilen dosyanın ismini *yeni–isim* yapar. Evvelce *eski–isim* ile erişilen dosyaya artık *yeni–isim* ile erişilebilecektir. (Eğer dosyanın başka isimleri de varsa, onlar hala geçerli olacaktır.)

yeni–isim ile belirtilen dosya, *eski–isim* ile belirtilen dosya ile aynı dosya sisteminde olmalıdır.

rename için özel bir durum, *eski–isim* ve *yeni–isim* ile belirtilen isimlerin aynı dosyanın iki ismi olmasıdır (çok isimli dosya). Bu durumda yapılacak tek şey *eski–isim* ile belirtilen dosya ismini silmektir. Ancak, POSIX bu durumda işlevin hiçbir şey yapmamasını ve başarı raporlamasını gerektirir. Sizin sisteminizin nasıl davranacağını bilemeyiz.

eski–isim bir dizin değilse ve *yeni–isim* diye bir dosya mevcutsa isim değiştirme işlemi sırasında bu dosya silinir. Bu bakımdan, *yeni–isim* ile bir dizin belirtilmişse, bu durumda **rename** başarısız olur.

eski–isim bir dizinse, *yeni–isim* mevcut olmamalı ya da boş bir dizin ismi olmalıdır. İkinci durumda *yeni–isim* isimli dizin önce silinecektir. *yeni–isim* ile ismi değiştirlecek *eski–isim* isimli dizinin bir alt dizini belirtilmemelidir.

rename işlevinin kullanışlı bir özelliği, eski isim yeni isim ile değiştirilirken, dosyaya önce yeni ismin eklenmesi, sonra eski ismin silinmesi şeklinde değiştirme işleminin atomik yapılmasıdır. Yani, eğer işlem sırasında bir sistem çökmesi yaşanırsa, her iki ismin hala mevcut olması olasıdır.

İşlevin dönüş değeri **-1** ise bir hata oluşmuş demektir. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EACCES`
: Dizinlerden biri *yeni–isim*'i içeriyor ya da *eski–isim* yazmaya izin vermiyor; veya *yeni–isim* ve *eski–isim* birer dizin ve onlardan birine yazma izniniz yok.

`EBUSY`
: *eski–isim* veya *yeni–isim* ile belirtilen dizinlerden biri sistem tarafından kullanılıyor ve sistem bunları isim değişikliğine karşı engelliyor. Bu hata bir dosya sisteminin bağlı olduğu dizin için ve onun dizinleri süreçlerin çalışma dizinlerini içeriyorsa oluşur.

`ENOTEMPTY`
`EEXIST`

yeni–isim isimli dizin boş değil. GNU sistemi bu hata için daima **ENOTEMPTY** döndürür, diğer sistemler **EEXIST** döndürebilir.

EINVAL
eski–isim, *yeni–isim* isimli dizini içeriyor.

EISDIR
yeni–isim bir dizin ama *eski–isim* değil.

EMLINK
yeni–isim'in üst dizini çok fazla bağ (girdi) içerecekti.

ENOENT
eski–isim isimli bir dosya yok.

ENOSPC
yeni–isim'i içerecek dizinde ve dosya sisteminde yer yok.

EROFS
İşlem salt–okunur bağlı bir dosya sisteminde yapılmaya çalışılıyor.

EXDEV
yeni–isim ve *eski–isim* farklı dosya sistemleri üzerinde.

8. Dizinlerin Oluşturulması

Dizinler **mkdir** işlevi ile oluşturulur. (Ayrıca, aynı şeyi yapan **mkdir** adında bir kabuk komutu vardır.)

```
int mkdir(const char *dosyaismi,                                    işlev
          mode_t      kip)
```

mkdir işlevi *dosyaismi* isimli bir yeni ve boş bir dizin oluşturur.

kip argümanı ile yeni dizin dosyasının izinleri belirtilir. Bunun hakkında daha fazla bilgi için *Erişim İzinleri için Kip Bitleri* (sayfa: 378) bölümüne bakınız.

0 dönüş değeri işlevin başarılı olduğunu, **-1** ise bir hata oluştuğunu gösterir. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EACCES
Yeni dizinin ekleneceği dizin için yazma izniniz yok.

EEXIST
dosyaismi isminde bir dosya zaten var.

EMLINK
Yeni dizinin ekleneceği dizin çok fazla girdi içeriyor.

İyi tasarlanmış dosya sistemleri bu hatayı asla döndürmez, çünkü onlar diskinizin tutabileceğinden çok daha fazla girdiye izin verirler. Ancak yine de bu hatayı alabileceğinizi hesaba katmalısınız, çünkü sonuç ağ üzerinden erişilen başka bir makinadan gelebilir.

ENOSPC
Yeni dizini oluşturmak için dosya sisteminde yer yok.

EROFS

Yeni dizinin ekleneceği dizin bir salt–okunur bağlı dosya sisteminde ve bu değiştirilemez.

Bu işlevi kullanmak için yazılımınıza `sys/stat.h` başlık dosyasını dahil etmelisiniz.

9. Dosya Öznitelikleri

Bir dosya üzerinde **`ls -l`** kabuk komutunu verirseniz, komut size dosyanın uzunluğu, kime ait olduğu, son değişiklik tarihi, vs. hakkında bilgi verir. Bunlara ***dosya öznitelikleri*** denir ve dosyanın ismiyle değil dosyanın kendisiyle ilgilidirler.

Bu kısım bir dosyanın özniteliklerinin değiştirilmesi ve sorgulanması hakkında bilgi içerir.

9.1. Dosya Özniteliklerinin Anlamları

Bir dosyanın özniteliklerini okuduğunuzda, onlar **`struct stat`** denilen bir yapıda gelirler. Bu bölümde özniteliklerin isimleri, veri türleri ve anlamları açıklanacaktır. Bir dosyanın özniteliklerini okuyan işlevleri *Bir Dosyanın Özniteliklerinin Okunması* (sayfa: 374) bölümünde bulabilirsiniz.

Bu bölümdeki bütün semboller `sys/stat.h` başlık dosyasında bildirilmiştir.

`struct` **`stat`** veri türü

`stat` yapısı bir dosyanın öznitelikleri hakkında bilgi döndürmekte kullanılır. En azından şu üyeleri içerir:

`mode_t` **`st_mode`**
: Dosya kipini belirtir. Bu *dosya türü bilgisi* (sayfa: 375) ile *dosya izinlerinin bitlerini* (sayfa: 378) içerir.

`ino_t` **`st_ino`**
: Dosyanın seri numarası. Bu dosyayı aynı aygıttaki diğer dosyalardan ayırır.

`dev_t` **`st_dev`**
: Dosyayı içeren aygıt. **`st_ino`** ve **`st_dev`** birlikte alındığında dosyayı eşsiz olarak kimliklendirir.

`nlink_t` **`st_nlink`**
: Dosyaya sabit bağların sayısı. Bu sayı bu dosya için kaç dizinin girdi içerdiğini gösterir. Sayı hep azalıyorsa, dosyayı açık tutan süreç kalmadığı anda dosya kendisini iptal eder. If the count is ever decremented to zero, then the file itself is discarded as soon as no process still holds it open. Sembolik bağlar toplama dahil değildir.

`uid_t` **`st_uid`**
: Dosyanın ait olduğu kullanıcının kullanıcı kimliği. Bkz. *Dosya İyeliği* (sayfa: 377).

`gid_t` **`st_gid`**
: Dosyanın ait olduğu grubun grup kimliği. Bkz. *Dosya İyeliği* (sayfa: 377).

`off_t` **`st_size`**
: Normal bir dosya için bayt cinsinden dosya uzunluğu. Aygıt dosyaları için bu alandaki değer anlamlı değildir. Sembolik bağlarda hedef dosya isminin uzunluğudur.

`time_t` **`st_atime`**
: Dosyaya son erişim zamanı. Bkz. *Dosya Zamanları* (sayfa: 383).

`unsigned long int` **`st_atime_usec`**
: Dosyaya son erişim zamanının ondalık kısmı. Bkz. *Dosya Zamanları* (sayfa: 383).

`time_t` **`st_mtime`**
: Dosya içeriğinin son değişiklik zamanı. Bkz. *Dosya Zamanları* (sayfa: 383).

`unsigned long int` **st_mtime_usec**
Dosya içeriğinin son değişiklik zamanının ondalık kısmı. Bkz. *Dosya Zamanları* (sayfa: 383).

`time_t` **st_ctime**
Dosya özniteliklerinin son değişiklik zamanı. Bkz. *Dosya Zamanları* (sayfa: 383).

`unsigned long int` **st_ctime_usec**
Dosya özniteliklerinin son değişiklik zamanının ondalık kısmı. Bkz. *Dosya Zamanları* (sayfa: 383).

`blkcnt_t` **st_blocks**
Dosyanın diskte kapladığı alanın 512 baytlık bloklar cinsinden miktarı.

Disk bloklarının sayısı dosya boyutu ile birebir orantılı değildir, bunun iki sebebi vardır: Dosya sistemi bazı blokları dahili kayıtlarını tutmak için kullanabilir; dosya seyrek olabilir—dosya sıfırlarla doldurulmuş "delikler" içerebilir ama bunlar aslında diskte yer kaplamaz.

Bir dosyanın seyrek olup olmadığı bu değeri **st_size** ile karşılaştırarak yaklaşık olarak söylemek mümkündür:

```
(st.st_blocks * 512 < st.st_size)
```

Bu sınama mükemmel değildir çünkü gerçekten seyrek olan bir dosyanın seyrek olduğu bu yöntemle saptanamayabilir. Ama pratik uygulamalar için bu bir sorun değildir.

`unsigned int` **st_blksize**
Dosyayı okumak ya da dosyaya yazmak için bayt cinsinden en uygun blok boyu. Bu değeri dosya ile yapacağınız okuma ve yazma işlemleri için ne kadar tampon ayıracağınızı belirlemek için kullanabilirsiniz. (Bu değerin **st_blocks** ile ilgisi yoktur.)

Dosya boyutlarının 2^63 bayta ulaştığı büyük dosya destekli (LFS) sistemler için bu yapının genişletilmesi gerekir.

`struct` **stat64** veri türü

Bu yapının üyeleri de üye isimleri de **struct stat** ile aynıdır. Tek fark **st_ino**, **st_size** ve **st_blocks** üyelerinin daha büyük değerleri tutabilmesi için faklı türde olmasıdır.

`mode_t` **st_mode**
Dosya kipini belirtir. Bu *dosya türü bilgisi* (sayfa: 375) ile *dosya izinlerinin bitlerini* (sayfa: 378) içerir.

`ino64_t` **st_ino**
Dosyanın seri numarası. Bu dosyayı aynı aygıttaki diğer dosyalardan ayırır.

`dev_t` **st_dev**
Dosyayı içeren aygıt. **st_ino** ve **st_dev** birlikte alındığında dosyayı eşsiz olarak kimliklendirir.

`nlink_t` **st_nlink**
Dosyaya sabit bağların sayısı. Bu sayı bu dosya için kaç dizinin girdi içerdiğini gösterir. Sayı hep azalıyorsa, dosyayı açık tutan süreç kalmadığı anda dosya kendisini iptal eder. If the count is ever decremented to zero, then the file itself is discarded as soon as no process still holds it open. Sembolik bağlar toplama dahil değildir.

`uid_t` **st_uid**
Dosyanın ait olduğu kullanıcının kullanıcı kimliği. Bkz. *Dosya İyeliği* (sayfa: 377).

`gid_t` **st_gid**
Dosyanın ait olduğu grubun grup kimliği. Bkz. *Dosya İyeliği* (sayfa: 377).

`off64_t` **st_size**
: Normal bir dosya için bayt cinsinden dosya uzunluğu. Aygıt dosyaları için bu alandaki değer anlamlı değildir. Sembolik bağlarda hedef dosya isminin uzunluğudur.

`time_t` **st_atime**
: Dosyaya son erişim zamanı. Bkz. *Dosya Zamanları* (sayfa: 383).

`unsigned long int` **st_atime_usec**
: Dosyaya son erişim zamanının ondalık kısmı. Bkz. *Dosya Zamanları* (sayfa: 383).

`time_t` **st_mtime**
: Dosya içeriğinin son değişiklik zamanı. Bkz. *Dosya Zamanları* (sayfa: 383).

`unsigned long int` **st_mtime_usec**
: Dosya içeriğinin son değişiklik zamanının ondalık kısmı. Bkz. *Dosya Zamanları* (sayfa: 383).

`time_t` **st_ctime**
: Dosya özniteliklerinin son değişiklik zamanı. Bkz. *Dosya Zamanları* (sayfa: 383).

`unsigned long int` **st_ctime_usec**
: Dosya özniteliklerinin son değişiklik zamanının ondalık kısmı. Bkz. *Dosya Zamanları* (sayfa: 383).

`blkcnt64_t` **st_blocks**
: Dosyanın diskte kapladığı alanın 512 baytlık bloklar cinsinden miktarı.

`unsigned int` **st_blksize**
: Dosyayı okumak ya da dosyaya yazmak için bayt cinsinden en uygun blok boyu. Bu değeri dosya ile yapacağınız okuma ve yazma işlemleri için ne kadar tampon ayıracağınızı belirlemek için kullanabilirsiniz. (Bu değerin **st_blocks** ile ilgisi yoktur.)

Bazı dosya özniteliklerinin kendilerine özel veri türleri vardır. (Aslında hepsi bildiğiniz tamsayı veri türlerinin karşılığıdır.) Bu veri türleri `sys/types.h` ve `sys/stat.h` başlık dosyalarında tanımlanmıştır. Aşağıda bunların bir listesini bulacaksınız.

mode_t veri türü

Dosya kiplerini göstermekte kullanılan tamsayı veri türü. GNU sisteminde **unsigned int**'e eşdeğerdir.

ino_t veri türü

Dosya seri numarasını (Bunlara bazan ***dosya indisi*** dendiği de olur) göstermekte kullanılan tamsayı veri türü. GNU sisteminde **unsigned long int**'e eşdeğerdir.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu tür **ino64_t** ile eşdeğerdir.

ino64_t veri türü

LFS desteği olan sistemlerde dosya seri numarasını göstermekte kullanılan tamsayı veri türü. GNU sisteminde **unsigned long long int**'e eşdeğerdir.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu tür **ino_t** ismiyle bulunur.

dev_t veri türü

Dosyayı içeren aygıtı göstermekte kullanılan tamsayı veri türü. GNU sisteminde **int**'e eşdeğerdir.

nlink_t veri türü

Dosya bağlarının sayısını göstermekte kullanılan tamsayı veri türü. GNU sisteminde **unsigned short int**'e eşdeğerdir.

blkcnt_t veri türü

Blok sayısını göstermekte kullanılan tamsayı veri türü. GNU sisteminde **unsigned long int**'e eşdeğerdir.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu tür **blkcnt64_t** ile eşdeğerdir.

blkcnt64_t veri türü

LFS desteği olan sistemlerde blok sayısını göstermekte kullanılan tamsayı veri türü. GNU sisteminde **unsigned long long int**'e eşdeğerdir.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu tür **blkcnt_t** ismiyle bulunur.

9.2. Bir Dosyanın Özniteliklerinin Okunması

Bir dosyanın özniteliklerini öğrenmek için **stat**, **fstat** ve **lstat** işlevleri kullanılır. Öznitelikleri bir **struct stat** nesnesinde döndürürler. Bu işlevler **sys/stat.h** başlık dosyasında bildirilmiştir.

```
int stat(const char  *dosyaismi,
         struct stat *tampon)
```
işlev

stat işlevi ismi *dosyaismi* ile belirtilen dosyanın özniteliklerini *tampon* ile gösterilen yapı içinde döndürür.

dosyaismi bir sembolik bağın ismiyse, bağın hedefindeki dosyanın öznitelikleri döndürülür. Sembolik bağın hedefindeki dosya mevcut değilse işlev dosyanın mevcut olmadığını bildirerek başarısız olur.

İşlem sorunsuz yerine getirilmişse **0**, aksi takdirde **-1** döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumu bu işlev için tanımlanmıştır:

ENOENT
: *dosyaismi* isminde bir dosya yok.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu işlev **stat64** ile aynıdır.

```
int stat64(const char    *dosyaismi,
           struct stat64 *tampon)
```
işlev

2^31 bayttan daha büyük dosyalarla çalışmanın mümkün olduğu 32 bitlik sistemlerde bu işlev **stat** işlevine eşdeğerdir. Bunu mümkün kılmak için sonucu döndüren *tampon*, **struct stat64** türünde bir yapıya göstericidir.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu işlev **stat** ismiyle bulunur.

```
int fstat(int          dosyatanıtıcı,
          struct stat *tampon)
```
işlev

fstat işlevi argüman olarak dosya ismi yerine bir *açık dosya tanıtıcı* (sayfa: 305) alması dışında **stat** işlevinin benzeridir.

stat gibi, **fstat** işlevi de başarı durumunda **0** ve hata oluşmuşsa **-1** ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF

dosyatanıtıcı argümanı geçerli bir dosya tanıtıcı değil.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu işlev **fstat64** ile aynıdır.

```
int fstat64(int             dosyatanıtıcı,                                  işlev
            struct stat64 *tampon)
```

Bu işlev **fstat** işlevine benzer fakat 32 bitlik platformlarda büyük dosyalarla çalışır. Büyük dosyalarla çalışmak için *dosyatanıtıcı* dosya tanıtıcısı **open64** veya **creat64** ile sağlanmış olmalıdır. *tampon* ile gösterilen değişken büyük değerleri tutabilen **struct stat64** türünde olmalıdır.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu işlev **fstat** ismiyle bulunur.

```
int lstat(const char  *dosyaismi,                                           işlev
          struct stat *tampon)
```

lstat işlevi **stat** işlevi gibidir fakat *sembolik bağlara* (sayfa: 365) izin vermez. Eğer *dosyaismi* bir sembolik bağ ismi ise, **lstat** bağın hedefi ile değil bağ dosyasının kendisi ile ilgili bilgi döndürür; bunun dışında **lstat** işlevi **stat** gibi çalışır.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu işlev **lstat64** ile aynıdır.

```
int lstat64(const char    *dosyaismi,                                       işlev
            struct stat64 *tampon)
```

2^31 bayttan daha büyük dosyalarla çalışmanın mümkün olduğu 32 bitlik sistemlerde bu işlev **lstat** işlevine eşdeğerdir. Bunu mümkün kılmak için sonucu döndüren *tampon*, **struct stat64** türünde bir yapıya göstericidir.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği sistemlerde bu işlev **lstat** ismiyle bulunur.

9.3. Bir Dosyanın Türünün Sınanması

Dosya kipi, dosya özniteliklerinin **st_mode** alanında saklanır ve iki çeşit bilgi içerebilir: dosya türü kodu ve erişim izin bitleri. Bu bölümde sadece bir dosyanın dizin mi, soket mi, sembolik bağ mı her ne haltsa belirlenmesine yarayan tür kodları açıklanacaktır. Erişim izinleri *Erişim İzinleri için Kip Bitleri* (sayfa: 378) bölümünde açıklanmıştır.

Dosya kipi içindeki dosya türü bilgisine erişmenin iki yolu vardır. İlkinde, belirtilen dosya kipini okuyup dosya türünün kendinin ifade etttiği türde olup olmadığı hakkında bilgi veren ***isnat makroları*** vardır. İkinci yolla ise, dosya kipini bir mask ile süzüp sadece dosya türü kodunu bıraktıktan sonra bunu desteklenen dosya türü sabitleriyle karşılaştırarak dosya türü saptanır.

Bu bölümdeki tüm semboller `sys/stat.h` başlık dosyasında bildirilmiştir. Aşağıdaki isnat makroları, bakılacak dosya için **stat** tarafından döndürülen **st_mode** alanındaki değer olan *m* değerine göre dosya türünü sınar:

```
int S_ISDIR(mode_t m)                                                       makro
```

Dosya bir dizin ise bu makro sıfırdan farklı bir değer döndürür.

```
int S_ISCHR(mode_t m)                                                       makro
```

Dosya bir karakter aygıtı dosyası (örn, uçbirim aygıtı) ise bu makro sıfırdan farklı bir değer döndürür.

```
int S_ISBLK(mode_t m)                                                       makro
```

Dosya bir blok aygıtı dosyası (örn, bir disk bölümü) ise bu makro sıfırdan farklı bir değer döndürür.

`int S_ISREG(mode_t m)` makro

Dosya bir normal dosya ise bu makro sıfırdan farklı bir değer döndürür.

`int S_ISFIFO(mode_t m)` makro

Dosya bir FIFO ya da boru ise bu makro sıfırdan farklı bir değer döndürür. Bkz. *Borular ve FIFOlar* (sayfa: 393).

`int S_ISLNK(mode_t m)` makro

Dosya bir sembolik bağ ise bu makro sıfırdan farklı bir değer döndürür. Bkz. *Sembolik Bağlar* (sayfa: 365).

`int S_ISSOCK(mode_t m)` makro

Dosya bir soket ise bu makro sıfırdan farklı bir değer döndürür. Bkz. *Soketler* (sayfa: 398).

BSD uyumluluğu için, desteklenen dosya türünü sınayan ve POSIX olmayan bir yöntem daha vardır. Dosya türü kodunu elde etmek için kip **S_IFMT** ile bit seviyesinde VE'lenir ve ilgili sabitle karşılaştırılır. Örneğin,

```
S_ISCHR (kip)
```

ifadesi ile

```
((kip  S_IFMT) == S_IFCHR)
```

ifadesi eşdeğerdir.

`int S_IFMT` makro

Bir kip değerinden dosya türünü çıkarmak için kullanılan bir bit maskesidir.

Dosya türü kodlarının sembolik isimleri:

`S_IFDIR`
Bir dizin dosyasının dosya türü sabitidir.

`S_IFCHR`
Bir karakter aygıtı dosyasının dosya türü sabitidir.

`S_IFBLK`
Bir blok aygıtı dosyasının dosya türü sabitidir.

`S_IFREG`
Bir normal dosyanın dosya türü sabitidir.

`S_IFLNK`
Bir sembolik bağın dosya türü sabitidir.

`S_IFSOCK`
Bir soketin dosya türü sabitidir.

`S_IFIFO`
Bir FIFO veya borunun dosya türü sabitidir.

POSIX.1b standardı, dosya sisteminde nesne olarak gerçeklenebilmesi olası bir kaç nesneden daha bahseder. Bunlar ileti kuyrukları, semaforlar ve paylaşımlı bellek nesneleridir. Bu nesnelerin diğer dosyalardan ayrımsanmasını mümkün kılmak için POSIX standardı üç yeni makrodan bahseder. Fakat diğer makroların aksine, parametre olarak **st_mode** alanının değerini almazlar. Bunun yerine **struct stat** yapısının tamamı için bir gösterici alırlar.

`int S_TYPEISMQ(struct stat *s)` makro

Eğer sistem, POSIX ileti kuyruklarını ayrı nesneler olarak gerçekleştiriyorsa ve dosya bir ileti kuyruğu nesnesi ise, bu makro sıfırdan farklı bir değerle döner. Tüm diğer durumlarda sonuç sıfırdır.

```
int S_TYPEISSEM(struct stat *s)                                    makro
```

Eğer sistem, POSIX semaforlarını ayrı nesneler olarak gerçekleştiriyorsa ve dosya bir semafor nesnesi ise, bu makro sıfırdan farklı bir değerle döner. Tüm diğer durumlarda sonuç sıfırdır.

```
int S_TYPEISSHM(struct stat *s)                                    makro
```

Eğer sistem, POSIX paylaşımlı bellek nesnelerini ayrı nesneler olarak gerçekleştiriyorsa ve dosya bir paylaşımlı bellek nesnesi ise, bu makro sıfırdan farklı bir değerle döner. Tüm diğer durumlarda sonuç sıfırdır.

9.4. Dosya İyeliği

Her dosyanın sistemde kayıtlı kullanıcı olarak tanımlı bir *sahibi* ve sistemde tanımlı gruplardan biri olarak bir *grubu* vardır. Dosya sahibi çoğunlukla bir dosyayı düzenleyebilen kullanıcı olarak ele alınırsa da asıl amaç erişim denetimidir.

Dosya sahibi ve grubu erişimi saptamakta kullanılır. Bunlar için her dosyada erişim izin bitleri tanımlanmıştır; bir bit kümesi dosyanın sahibinin yetkilerini, ikinci bir bit kümesi dosyaya erişim yetkisi olan gruba tanınan yetkileri, üçüncüsü bir bit kümesi ise diğerlerine tanınan yetkileri belirlemekte kullanılır. Bu veriye dayalı olarak erişime nasıl karar verildiği *Erişim İzinleri* (sayfa: 380) bölümünde ayrıntılı olarak açıklanmıştır.

Bir dosya oluşturulurken, sahibi, *sürecin etkin kullanıcı kimliği* (sayfa: 743) yapılarak dosya oluşturulur. Dosyanın grup kimliği dosyayı içerecek dosya sistemine bağlı olarak ya sürecin etkin grup kimliği ya da dosyayı içeren dizinin grup kimliği yapılır. Uzak bir dosya sistemine eriştiğinizde uygulanan kurallar sizin dosya sisteminizin değil uzak dosya sisteminin kuralları olacaktır. Bu bakımdan yazılımınız üzerinde çalıştığı sistemin davranış çeşidine bakmaksızın her davranış çeşidine uyum sağlamaya hazır olmalıdır.

Mevcut bir dosyanın sahibini ve/veya grubunu **chown** işlevini kullanarak değiştirebilirsiniz. Bu işlev **chown** ve **chgrp** kabuk komutlarının ilkelidir.

Bu işlevin prototipi `unistd.h` başlık dosyasında bildirilmiştir.

```
int chown(const char *dosyaismi,                                   işlev
          uid_t       kullanıcı,
          gid_t       grup)
```

chown işlevi ismi *dosyaismi* ile belirtilen dosyanın sahibini *kullanıcı* ve grubunu *grup* olarak değiştirir.

Bazı sistemlerde bir dosyanın iyeliğinin değiştirilmesi set–user–ID ve set–group–ID bitlerinin temizlenmesine yol açar (Bu bitlerin dosyanın yeni sahipleri ile ilgisi olmadığından bu böyledir.) Diğerleri ile ilişkili izin bitleri değişmez.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EPERM
: Bu sürecin yetkileri istenen değişikliği yapmak için yetersiz.

  Sadece ayrıcalıklı kullanıcı ve dosyanın sahibi dosyanın grubunu değiştirebilir. Çoğu sistemde dosyanın sahibini sadece ayrıcalıklı kullanıcı değiştirebilirken bazı sistemlerde ise dosya sahibini değiştirmenize o dosyanın sahibi olarak görünüyorsanız izin verilir. Uzak bir dosya sistemine eriştiğinizde uygulanan kurallar sizin dosya sisteminizin değil uzak dosya sisteminin kuralları olacaktır.

_POSIX_CHOWN_RESTRICTED makrosu hakkında bilgi edinmek için *Dosya Desteği Seçenekleri* (sayfa: 796) bölümüne bakınız.

EROFS
: Dosya, salt–okunur bağlı bir dosya sisteminde.

```
int fchown(int dosyatanıtıcı,                                    işlev
           int kullanıcı,
           int grup)
```

Bu işlev bir dosya ismi değil *dosyatanıtıcı* ile belirtilen bir açık dosya tanıtıcı alması dışında **chown** işlevi gibidir.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcı değil.

EINVAL
: *dosyatanıtıcı* argümanı normal bir dosyayla değil, bir boru ya da soket ile ilişkili.

EPERM
: Bu sürecin yetkileri istenen değişikliği yapmak için yetersiz. Ayrıntılar için yukarı, **chmod** işlevine bakınız.

EROFS
: Dosya, salt–okunur bağlı bir dosya sisteminde.

9.5. Erişim İzinleri için Kip Bitleri

Dosya kipi, dosya özniteliklerinin **st_mode** alanınada saklanır ve iki çeşit bilgi içerebilir: dosya türü kodu ve erişim izin bitleri. Bu bölümde sadece bir dosyayı kimlerin okuyabileceği veya yazabileceğini denetleyen erişim izni bitleri açıklanacaktır. Dosya türü kodları *Erişim İzinleri için Kip Bitleri* (sayfa: 378) bölümünde açıklanmıştır.

Bu bölümdeki sembollerin hepsi `sys/stat.h` başlık dosyasında tanımlanmıştır. Bir dosyanın erişim izinlerini denetleyen dosya kipi bitleri için tanımlanmış sabitler:

S_IRUSR
S_IREAD
: Dosyanın sahibi için okuma yetkisi biti. Bir çok sistemde bu bit 0400'dür. **S_IREAD** sabiti bu sabitin BSD uyumluluğu için sağlanmış artık atıl olmuş bir eşanlamlısıdır.

S_IWUSR
S_IWRITE
: Dosyanın sahibi için yazma yetkisi biti. Bir çok sistemde bu bit 0200'dür. **S_IWRITE** sabiti bu sabitin BSD uyumluluğu için sağlanmış artık atıl olmuş bir eşanlamlısıdır.

S_IXUSR
S_IEXEC
: Dosyanın sahibi için normal dosyalarda çalıştırma, dizinlerde arama yetkisi biti. Bir çok sistemde bu bit 0100'dür. **S_IEXEC** sabiti bu sabitin BSD uyumluluğu için sağlanmış artık atıl olmuş bir eşanlamlısıdır.

S_IRWXU

(S_IRUSR | S_IWUSR | S_IXUSR) ifadesinin eşdeğeridir.

S_IRGRP
: Dosyanın grubu için okuma yetkisi biti. Bir çok sistemde bu bit 040'tır.

S_IWGRP
: Dosyanın grubu için yazma yetkisi bitleri. Bir çok sistemde bu bit 020'dir.

S_IXGRP
: Dosyanın grubu için çalıştırma ve arama yetkisi biti. Bir çok sistemde bu bit 010'dur.

S_IRWXG
: **(S_IRGRP | S_IWGRP | S_IXGRP)** ifadesinin eşdeğeridir.

S_IROTH
: Diğer kullanıcılar için okuma yetkisi biti. Bir çok sistemde bu bit 04'tür.

S_IWOTH
: Diğer kullanıcılar için yazma yetkisi biti. Bir çok sistemde bu bit 02'dir.

S_IXOTH
: Diğer kullanıcılar için çalıştırma ve arama yetkisi biti. Bir çok sistemde bu bit 01'dir.

S_IRWXO
: **(S_IROTH | S_IWOTH | S_IXOTH)** ifadesinin eşdeğeridir.

S_ISUID
: Çalıştırma biti üzerinde etkili set–user–ID bitidir. Bir çok sistemde bu bit 04000'dir. Bkz. *Bir Sürecin Aidiyeti Nasıl Değiştirilir?* (sayfa: 743).

S_ISGID
: Çalıştırma biti üzerinde etkili set–group–ID bitidir. Bir çok sistemde bu bit 02000'dir. Bkz. *Bir Sürecin Aidiyeti Nasıl Değiştirilir?* (sayfa: 743).

S_ISVTX
: ***yapışkan bit***. Bir çok sistemde bu bit 01000'dir.

  Bu bit, bir dizin için bu dizindeki bir dosyayı silme iznini sadece dosyanın sahibine verir. Normalde bir kullanıcı ya bir dizindeki tüm dosyaları silebilir ya da hiçbirini silemez (kullanıcının dizine yazma izni olup olmamasına bağlı olarak). Bu sınırlamalar uygulandığında bir dosyayı silebilmek için hem dosya sizin dosyanız olmalı hem de onun bulunduğu dizine yazma izniniz olmalıdır. Buna bir istisna, dizinin sahibi olmaktır. Dizinin sahibi olan kullanıcı dizin içindeki dosyaların hepsini dosyaların sahiplerinin kim olduğuna bakılmaksızın silme yetkisine sahiptir. Bu bit **/tmp** dizininde faydalı bir amaç için kullanılır; bu dizinde herkes dosya oluşturabilir ama kimse diğerinin dosyasını silemez.

  Evvelce, bir çalıştırılabilir dosyada yapışkan bit etkin olduğunda sistemin takaslama kurallarında bu yazılım için değişiklik yapılırdı. Normalde, bir yazılım sonlandığında onun bellekteki sayfaları serbest bırakılır ve yeniden kullanıma hazır tutulurdu. Eğer çalıştırılabilir dosyanın yapışkan biti etkinse, yazılım sonlandığında bellekteki sayfaları serbest bırakılmaz, yazılım hala çalışıyormuş gibi bellekte tutulurdu. Bu durum aynı yazılım defalarca çalıştırıldığında bu yazılım için bir ayrıcalık oluştururdu. Bu kullanım artık günümüzde atıl olmuştur. Artık, bir yazılım sonlandığında bellekteki sayfaları bir ihtiyaç hasıl olana kadar serbest bırakılmamaktadır. Aynı yazılım tekrar çalıştırıldığında eski sayfaları hala bellekte duruyorsa onlar kullanılabilmekte, bir ihtiyaçtan dolayı kullanılmışsa yazılım tekrar belleğe yüklenmektedir.

Günümüzdeki bazı sistemlerde bir çalıştırılabilir dosya açısından yapışkan bit anlamlı değildir, böyle sistemlerde bu bit dizinler dışında etkinleştirilemez. Eğer bunu denerseniz, **chmod** işlevi **EFTYPE** hatasıyla başarısız olur; bkz. *Dosya İzinlerinin Atanması* (sayfa: 380).

Bazı sistemler (özellikle SunOS) yapışkan bit kullanımı ile ilgili olarak farklı bir uygulama yapar. Eğer yapışkan bit bir çalıştırılabilir *olmayan* dosya için etkinleştirilirse, tamamen zıt bir uygulama olarak, o dosyanın sayfaları belleğe alınmaz. Bunun kullanım alanı, bir NFS sunucusu üzerinde, disksiz istemcilerin takas alanı olarak kullanmak üzere ayrılmış dosyalardır. Bu dosyalar istemci makinanın belleğinde sayfalandığından, bunların bir de sunucu makinanın belleğinde sayfalanması anlamsız olacağından bu yönteme başvurulmuştur. Bu kullanımda yapışkan bit ayrıca dosya sisteminin disk üzerinde düzenli olarak dosyanın değişiklik zamanını kaydetmesinin başarısız olmasını sağlar (bir takas dosyasıyla nasılsa kimse ilgilenmez, denerek).

Bu bit sadece BSD sistemlerinde geçerlidir (ve ondan türetilmiş sistemlerde). Bu bakımdan bu biti kullanmak için **_BSD_SOURCE** özellik seçim makrosunu tanımlı yapmalısınız (bkz. *Özellik Sınama Makroları* (sayfa: 25)).

Yukarıda listelenen sembollerin bit değerlerini kullanarak yazılımınızda hata ayıklarken dosya kip değerlerini çözümleyebilirsiniz. Bu bit değerleri çoğu sistemde geçerlidir ama hepsinin olacağı garanti değildir.

Uyarı

Dosya izinleri için doğrudan sayıları kullanmak iyi bir uygulama olmaz. Taşınabilir olmayacağından başka, bitlerin anlamlarını hatırlamak için yazılımınızın koduna bakmak gerekir. Temiz bir yazılım sembol isimleri kullanır.

9.6. Erişim İzinleri

İşletim sistemi normalde bir dosyanın erişim izinlerine, sürecin etkin kullanıcı ve grup kimlikleri ve ek grup kimlikleri ile dosyanın sahibi, grubu ve izin bitlerine birlikte bakarak karar verir. Bu kavramlar ayrıntılı olarak *Bir Sürecin Aidiyeti* (sayfa: 743) bölümünde anlatılmıştır.

Eğer sürecin etkin kullanıcı kimliği ile dosyanın sahibinin kullanıcı kimliği aynı ise, bu kullanıcının okuma, yazma ve çalıştırma/arama izinleri geçerli olur. Benzer şekilde, eğer, sürecin etkin veya ek grup kimliklerinden biri dosyanın grup kimliği ile aynıysa, bu grubun izinleri geçerli olur. Aksi takdirde, diğerlerinin izinlerine bakılır.

root gibi ayrıcalıklı kullanıcılar, izin bitlerine bakılmaksızın her dosyaya erişebilirler. Özel bir durum olarak, çalıştırılabilir bir dosyayı ayrıcalıklı kullanıcının dahi çalıştırabilmesi için dosyanın çalıştırma biti etkin olmalıdır.

9.7. Dosya İzinlerinin Atanması

Dosyaları oluşturmakta kullanılan **open** veya **mkdir** gibi ilkel işlevler yeni oluşturulacak dosyaya atanacak dosya izinlerini belirleyen bir *kip* argümanı alırlar. Bu kip kullanılmadan önce sürecin *dosya oluşturma maskesi* ya da *umask* ile değişikliğe uğratılır.

Dosya oluşturma maskesinde izinleri ifade eden bitler yeni oluşturulan dosyalar için iptal edilecek izinleri belirtir. Örneğin, maskede diğerlerine bütün erişim izinleri verilmişse, diğer kategorisindeki hiçbir süreç bu dosyaya erişemeyecektir; dosyayı oluşturan işlevin *kip* argümanında diğerlerine tüm erişim izinleri verilimiş olsa bile! Başka bir deyişle dosya oluşturma maskesi vermek istediğiniz erişim izinlerinin tümleyenidir.

Dosya oluşturan yazılımlar genellikle *kip* argümanında herkese tüm izinleri veren bir değer belirtirler. Normal bir dosya için bu herkese okuma ve yazma izinleri vermek şeklindedir. Daha sonra dosya oluşturulurken bu izinler kullanıcının dosya oluşturma maskesi kullanılarak sınırlanır.

İsmini belirterek mevcut bir dosyanın izinlerini değiştirmek için **chmod** işlevi kullanılır. Bu işlev belirtilen izinleri kullanırken dosya oluşturma maskesini yoksayar.

Normal kullanımda, dosya oluşturma maskesi kullanıcının oturum açma kabuğu tarafından (**umask** kabuk komutu ile) ilklendirilir ve tüm alt süreçler tarafından miras alınır. Uygulama yazılımları dosya oluşturma maskesi için normalde endişelenmezler. Onun özdevinimli oluşturulduğu kabul edilir.

Yazılımınızın, bir dosyanın, dosya oluşturma maskesini yoksayarak erişim izinlerini belirlemesini istiyorsanız bunun en kolay yolu dosyayı açtıktan sonra dosya oluşturma maskesini değiştirmek yerine, **fchmod** işlevini kullanmaktır. Aslında, dosya oluşturma maskesini değiştirme işlemi sadece kabuk tarafından yapılır. Kabuk bunu **umask** işlevini kullanarak yapar.

Bu bölümdeki işlevler `sys/stat.h` başlık dosyasında bildirilmiştir.

`mode_t` **umask**`(mode_t` *maske*`)` işlev

umask işlevi çağrıldığı sürecin dosya oluşturma maskesini *maske* yapar ve önceki dosya oluşturma maskesi ile döner.

Bu örnekte, dosya oluşturma maskesinin kalıcı olarak değiştirilmeksizin **umask** ile nasıl okunacağı gösterilmiştir:

```
mode_t
read_umask (void)
{
  mode_t mask = umask (0);
  umask (mask);
  return mask;
}
```

Ancak, maske değerini sadece okumak istiyorsanız **getumask** işlevini kullanmak daha iyidir, çünkü bu işlev evreseldir (en azından GNU sisteminde).

`mode_t` **getumask**`(void)` işlev

Çağrıldığı sürecin dosya oluşturma maskesi ile döner. Bu işlev bir GNU oluşumudur

`int` **chmod**`(const char *`*dosyaismi*`,`
`mode_t` *kip*`)` işlev

chmod işlevi, ismi *dosyaismi* ile belirtilen dosyanın erişim izinlerini *kip* ile belirtilen değere ayarlar.

dosyaismi bir sembolik bağ ise, **chmod** bağın değil, bağın hedefindeki dosyanın izinlerini değiştirir.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`ENOENT`
: Belirtilen dosya yok.

`EPERM`
: Bu sürecin, bu dosyanın erişim izinlerini değiştirme izni yok. Sadece dosyanın sahibi (sürecin etkin kullanıcı kimliğinden saptanır) ya da ayrıcalıklı kullanıcı onları değiştirebilir.

`EROFS`
: Dosya salt–okunur bağlı bir dosya sistemi üzerinde.

`EFTYPE`

kip argümanı **S_ISVTX** bitini ("yapışkan bit") içeriyor ama ismi belirtilen dosya bir dizin değil. Bazı dosya sistemlerinde yapışkan bitin dosyalara verilmesine izin verilirken bazılarında da verilmez (sadece dizinlerde izin verilir).

EFTYPE hatasını sadece yapışkan bitin dizinler dışında anlamlı olmadığı dosya sistemlerinde alırsınız. Bu olduğu takdirde, yapışkan biti içermeyen bir *kip* değeri ile tekrar **chmod** çağrısı yapın. Yapışkan bit ile ilgili daha ayrıntılı bilgi edinmek için *Erişim İzinleri için Kip Bitleri* (sayfa: 378) bölümüne bakınız.

```
int fchmod(int dosyatanıtıcı,                                          işlev
           int kip)
```

Bu işlev argüman olarak dosya ismi yerine bir *açık dosya tanıtıcı* (sayfa: 305) alması dışında **chmod** işlevinin benzeridir.

İşlev başarılıysa **0** ile değilse **-1** ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcı değil.

EINVAL
: *dosyatanıtıcı* argümanı bir boru veya soket ya da erişim izinlerine konu olmayan bir şeye ait.

EPERM
: Bu sürecin dosyanın izinlerini değiştirme yetkisi yok. Sadece dosyanın sahibi (sürecin etkin kullanıcı kimliğinden saptanır) ya da ayrıcalıklı kullanıcı izinleri değiştirebilir.

EROFS
: Dosya salt–okunur bağlı bir dosya sisteminde bulunuyor.

9.8. Dosya Erişim İzinlerinin Sınanması

Bazı durumlarda kullanıcının erişim yetkisi olmayan bazı dosyalara ya da aygıtlara yazılım üzerinden erişebilmesi istenir. Olası bir çözüm yazılımın setuid bitini etkinleştirmektir. Böyle bir yazılım çalıştırılırsa, sürecin etkin kullanıcı kimliği yazılım dosyasının sahibi olarak değiştirilir. Böylece, yazılımın sahibi **root** yapılıp setuid biti de etkinleştirilerek, normalde sadece ayrıcalıklı kullanıcı tarafından erişilebilen `/etc/passwd` gibi dosyalara, yazma erişimi sağlanabilir.

Bunun yanında bir kullanıcının erişim izni olmayan dosyalara erişmesine izin vermeyecek bir düzenleme de düşünülebilir. Bu durumda yazılım, bir dosyayı okumadan ya da ona yazmadan önce *kullanıcının* gerekli erişim izinlerine sahip olup olmadığını sınamalıdır.

Bunu yapmak için, sürecin etkin kullanıcı kimliğine değil gerçek kullanıcı kimliğine dayalı erişim izinlerini sınayan **access** işlevi kullanılır. (Setuid özelliği gerçek kullanıcı kimliği değiştirmez, böylece yazılımı gerçekte kimin çalıştırdığı saptanır.)

Bu erişimi sınamanın daha kolay açıklanabilen bir yolu daha vardır, ama onun da kullanımı zordur. Bu yöntemde işlem dosya kip bitlerini öğrenerek ve sistemin kendi erişim hesaplaması taklit edilerek yapılır. Bu yöntemin kullanılması pek tercih edilmez, çünkü bir çok sistem ek erişim denetim özelliklerine sahiptir ve yazılımınız farklı sistemlerin farklı erişim özelliklerini taşınabilir olarak taklit etmek zorunda kalacaktır. **access** işlevi bu işlemleri sizin yerinize yapar.

access işlevi *sadece* ve sadece setuid yazılımlarda kullanmak için değildir. Setuid olmayan bir yazılım daima gerçek kimlik yerine etkin kimliği kullanır.

Bu bölümdeki semboller `unistd.h` başlık dosyasında bildirilmiştir.

```
int access(const char *dosyaismi,                                    işlev
           int         nasıl)
```

access işlevi ismi *dosyaismi* ile belirtilen dosyaya *nasıl* ile belirtilen yolla erişilip erişilemeyeceğini sınar. *nasıl* argümanında belirtilebilecek değer ya **R_OK**, **W_OK** ve **X_OK** seçeneklerinin bit seviyesinde VEYA'lanmışı ya da **F_OK** varlık sınaması olabilir.

Bu işlev erişim izinlerini sınamak için sürecin etkin kullanıcı ve grup kimliklerini değil, gerçek kullanıcı ve grup kimliklerini kullanır. Sonuç olarak, bu işlevi bir **setuid** veya **setgid** yazılımda (*Bir Sürecin Aidiyeti Nasıl Değiştirilir?* (sayfa: 743)) kullanıyorsanız, işlev yazılımı gerçekte hangi kullanıcı çalıştırıyorsa o kullanıcıya göre bilgi verir.

Erişime izin verilmişse işlev **0** ile, aksi takdirde **-1** ile döner. (Başka bir deyişle, eğer istenen erişime *izin verilmezse*, **access** işlevi, bir isnat işlevi gibi düşünülerek doğru ile döner.)

Dosya ismi hatalarına (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EACCES`
: *nasıl* ile belirtilen erişime izin verilmiyor.

`ENOENT`
: Dosya mevcut değil.

`EROFS`
: Salt–okunur bağlı bir dosya sistemi üzerindeki bir dosya için yazma izni istendi.

access işlevinin *nasıl* argümanında kullanmak için tasarlanmış olan bu makrolar tamsayı sabitler olarak `unistd.h` başlık dosyasında tanımlanmıştır:

```
int R_OK                                                             makro
```

Okuma izni için sınama seçeneği.

```
int W_OK                                                             makro
```

Yazma izni için sınama seçeneği.

```
int X_OK                                                             makro
```

çalıştırma/arama izni için sınama seçeneği.

```
int F_OK                                                             makro
```

Dosyanın mevcut olup olmadığını sınama seçeneği.

9.9. Dosya Zamanları

Her dosyanın kendisiyle ilgili üç zaman damgası vardır: erişim zamanı, değişiklik zamanı ve öznitelik değişiklik zamanı; bkz. *Dosya Öznitelikleri* (sayfa: 371).

Bu zamanların hepsi mutlak zaman biçiminde gösterilen **time_t** türünde nesnelerdir. Bu veri türü **time.h** başlık dosyasında tanımlanmıştır. Zaman değerlerinin gösterilmesi ve değiştirilmesi hakkında daha fazla bilgi için *Mutlak Zaman* (sayfa: 542) bölümüne bakınız.

Bir dosyanın okunması erişim zamanını güncellerken, yazılması değişiklik zamanını günceller. Dosya oluşturulduğu zaman, üç zaman damgasına da dosyanın oluşturulduğu zaman değeri atanır. Ek olarak, yeni girdiyi içeren dizinin erişim ve değişiklik zamanı da güncellenir.

link ile bir dosyaya yeni isim eklenmesi, isim eklenen dosyanın öznitelik değişiklik zamanını günceller ve bu yeni ismi içeren dizinin öznitelik ve içerik değişiklik zamanları da güncellenir. **unlink**, **remove** veya **rmdir** ile dosya isminin silinmesi de aynı alanları etkiler. Bir dosyanın isminin **rename** ile değiştirilmesi sadece bu değişiklikten etkilenen iki dizinin içerik ve öznitelik değişiklik zamanlarını günceller, ismi değiştirilen dosyada zaman güncellemesi yapılmaz.

Bir dosyanın özniteliklerinin değiştirilmesi (örn, **chmod** ile), öznitelik değişiklik zamanını günceller.

Öznitelik değişiklik zamanı dışında bir dosyanın değişiklik zamanlarını **utime** işleviyle doğrudan değiştirebilirsiniz. Bu oluşumu yazılımınızda kullanmak için yazılımınıza `utime.h` başlık dosyasını dahil etmelisiniz.

`struct` **utimbuf** veri türü

utimbuf yapısı bir dosyaya yeni erişim ve değişiklik zamanlarını belirtmek için **utime** işlevi ile kullanılır. Şu üyeleri içerir:

`time_t` **actime**
: Dosyanın erişim zamanı.

`time_t` **modtime**
: Dosyanın (içerik) değişiklik zamanı.

```
int utime(const char           *dosyaismi,                                  işlev
          const struct utimbuf *zamanlar)
```

Bu işlev ismi *dosyaismi* ile belirtilen dosyanın dosya zamanlarını değiştirir.

Eğer *zamanlar* bir boş gösterici ise, dosyanın erişim ve değişiklik zamanları güncellenir. Aksi takdirde, zaman damgalarına *zamanlar* ile gösterilen **utimbuf** yapısının **actime** ve **modtime** üyelerindeki değerler atanır.

Her durumda dosyanın öznitelik değişiklik zamanı güncellenir (çünkü dosyanın zaman ile ilgili öznitelikleri değişmiştir).

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EACCES`
: *zamanlar* argümanında boş gösterici aktarıldığı durumda bir izin sorunu var. Dosya zaman damgalarını güncelleyebilmek için ya dosyanın sahibi ya da ayrıcalıklı kullanıcı olmalısınız.

`ENOENT`
: Dosya mevcut değil.

`EPERM`
: Eğer *zamanlar* argümanı boş gösterci değilse ya dosyanın sahibi ya da ayrıcalıklı kullanıcı olmalısınız.

`EROFS`
: Dosya salt–okunur bağlı bir dosya sisteminde bulunuyor.

Her üç zaman damgasınında çözünürlüğü arttıran mikrosaniyelik bir parçası vardır. Bu alanlar 0 ile 999,999 mikrosaniye arasında değer alabilen üç alanla ifade edilir: **st_atime_usec**, **st_mtime_usec** ve **st_ctime_usec**. Bu alanlar bir **timeval** yapısının **tv_usec** üyesine karşılıktır; bkz. *Yüksek Çözünürlüklü Zaman* (sayfa: 543).

utimes işlevi **utime** işlevi gibidir, ancak dosya zamanlarının ondalık kısımlarını da belirtebilmenizi sağlar. Bu işlevin prototipi `sys/time.h` başlık dosyasında bulunur.

```
int utimes(const char     *dosyaismi,                                    işlev
           struct timeval zaman[2])
```

Bu işlev ismi *dosyaismi* ile belirtilen dosyanın erişim ve değişiklik zamanlarını değiştirir. Yeni dosya erişim zamanı *zaman*`[0]` ile ve yeni değişiklik zamanı *zaman*`[1]` ile belirtilir. *zaman* olarak bir boş gösterici belirtilirse **utime** işlevi gibi dosyanın erişim ve değişiklik zamanlarını günceller. Bu işlev BSD'den gelmektedir.

İşlevin dönüş değerleri ve hata durumları **utime** işlevininkilerle aynıdır.

```
int lutimes(const char     *dosyaismi,                                    işlev
            struct timeval zaman[2])
```

Bu işlev **utimes** gibi olmakla birlikte sembolik bağları izlemez. **utimes** işlevi bir sembolik bağın hedefindeki dosyanın erişim ve değişiklik zamanlarını değiştirdiği halde, **lutimes** işlevi (**lstat** gibi; *Sembolik Bağlar* (sayfa: 365)) sembolik bağ dosyasının kendi erişim ve değişiklik zamanlarını değiştirir. Bu işlev BSD'den gelmektedir ve tüm platformlarca desteklenmemektedir (desteklenmiyorsa, işlev **ENOSYS** hatasıyla başarısız olur).

İşlevin dönüş değerleri ve hata durumları **utime** işlevininkilerle aynıdır.

```
int futimes(int            dosyatanıtıcı,                                 işlev
            struct timeval zaman[2])
```

Bu işlev **utimes** gibidir, ancak argüman olarak dosya ismi yerine bir *açık dosya tanıtıcısı* (sayfa: 305) alır. Bu işlev BSD'den gelmektedir ve tüm platformlarca desteklenmemektedir (desteklenmiyorsa, işlev **ENOSYS** hatasıyla başarısız olur).

utimes gibi, **futimes** işlevi de başarılı olduğunda **0** ile aksi takdirde **-1** ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EACCES`
: *zamanlar* argümanındaa boş gösterici aktarıldığı durumda bir izin sorunu var. Dosya zaman damgalarını güncelleyebilmek için ya dosyanın sahibi ya da ayrıcalıklı kullanıcı olmalısınız.

`EBADF`
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcı değil.

`EPERM`
: Eğer *zamanlar* argümanı boş gösterci değilse ya dosyanın sahibi ya da ayrıcalıklı kullanıcı olmalısınız.

`EROFS`
: Dosya salt–okunur bağlı bir dosya sisteminde bulunuyor.

9.10. Dosya Boyu

Normalde dosya boyu özdevinimli olarak belirlenir. Bir dosya 0 boyda başlar ve veri yazıldıkça özdevinimli olarak uzar. Ayrıca, bir **open** ya da **fopen** çağrısı ile bir dosyanın içeriğini silmek ve boş duruma getirmek mümkündür.

Yine de, bazan bir dosyanın boyunu küçültmek gerekebilir. Bu işlem **truncate** ve **ftruncate** işlevleri ile yapılır. Bunlar BSD Unix'den gelir. **ftruncate** işlevi daha sonra POSIX.1'e eklenmiştir.

Bazı sistemler bu işlevlerle bir dosyayı uzatmaya da (delikler oluşturarak) izin verir. Dosyalar özdevinimli uzatılamadığında *bellek eşlemli G/Ç* (sayfa: 319) kullanılırken yararlı olur. Ancak, bu taşınabilir olmadığından, dosyaların eşlenmesine izin veriyorsa **mmap** ile gerçeklenmelidir (**_POSIX_MAPPED_FILES** tanımlıysa izin verir).

Bu işlevlerin normal dosyalar dışında kullanılması *tanımlanmamış* sonuçlara yol açabilir. Çoğu sistemde böyle bir çağrı, aslında hiçbir işlem yapmakszın başarılı görünecektir.

```
int truncate(const char *dosyaismi,                                    işlev
             off_t      uzunluk)
```

truncate işlevi ismi *dosyaismi* ile belirtilen dosyanın uzunluğunu *uzunluk* yapar. Eğer *uzunluk* önceki uzunluktan küçükse dosyanın sonundaki veri kaybedilecektir. Bu işlemin gerçekleşebilmesi için kullanıcının dosyaya yazma izni olmalıdır.

Eğer *uzunluk* önceki uzunluktan daha büyükse dosyanın sonuna delikler eklenir. Ancak, bazı sistemler bu işlemi desteklemez ve dosya değişmeden kalır.

Kaynakların 32 bitlik bir sistemde **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev aslında **truncate64** işlevidir ve **off_t** türü 2^63 bayt uzunluğa kadar dosyaları mümkün kılan 64 bitlik bir türdür.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EACCES
: Dosya ya bir dizin ya da yazılabilir değil.

EINVAL
: *uzunluk* negatif.

EFBIG
: Dosya boyu, işletim sisteminin sınırlarından fazlasına genişletiliyor.

EIO
: Bir donamım G/Ç hatası oluştu.

EPERM
: Dosya ya sona eklemeli ya da değiştirilemez türde.

EINTR
: İşlem bir sinyal ile engellendi.

```
int truncate64(const char *isim,                                       işlev
               off64_t     uzunluk)
```

Bu işlev **truncate** işlevinin benzeridir. Farkı, *uzunluk* argümanının 32 bitlik makinalarda bile 64 bitlik genişlikte olmasıdır. Böylece 2^63 bayta kadar dosya uzunlukları belirtilebilir.

32 bitlik bir sistemde, kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev **truncate** ismiyle bulunur ve eski gerçekleme tamamen LFS'ye uygun olarak değiştirilir.

```
int ftruncate(int    dosyatanıtıcı,                                    işlev
              off_t uzunluk)
```

Bu işlev **truncate** gibidir, ancak bir açık dosya tanıtıcı ile çalışır. Dosya yazma amacıyla açılmış olmalıdır.

POSIX standardı, dosyanın yeni *uzunluk* değerinin özgün dosya boyundan daha büyük olduğu durumda ne yapılacağını gerçeklemeye bırakmıştır. **ftruncate** işlevi ya dosyayı hiçbir şey yapmadan bırakır ya da istenen boya arttırır. İkinci durumda uzatılan bölge sıfırlarla doldurulur. **ftruncate** işlevi ile dosya boyunun arttırılması pek güvenilir olmasa da eğer uzatilabiliyorsa bu olası en hızlı yöntemdir. Bu işlev ayrıca eğer sistem tarafından gerçeklenmişse POSIX paylaşımlı bellek bölütleri üzerinde de çalışır.

ftruncate işlevi özellikle **mmap** ile birlikte kullanıldığında yararlıdır. Dosya ile eşlenen bellek bölgeleri sabit uzunlukta olduğundan son eşlenen sayfaya daha fazla bilgi yazarak dosya boyu büyütülemez. Dosya boyunu büyütmek için dosya yeni boyutla yeniden belleğe eşlenmelidir. Bunun nasıl yapıldığı aşağıda bir örnekle gösterilmiştir.

Kaynakların 32 bitlik bir sistemde **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev aslında **ftruncate64** işlevidir ve **off_t** türü 2^63 bayt uzunluğa kadar dosyaları mümkün kılan 64 bitlik bir türdür.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* bir açık dosya ile ilgili değil.

EACCES
: *dosyatanıtıcı* ya bir dizin ya da yazmak için açılmamış.

EINVAL
: *uzunluk* negatif.

EFBIG
: Dosya boyu, işletim sisteminin sınırlarından fazlasına genişletiliyor.

EIO
: Bir donamım G/Ç hatası oluştu.

EPERM
: Dosya ya sona eklemeli ya da değiştirilemez türde.

EINTR
: İşlem bir sinyal ile engellendi.

```
int ftruncate64(int      id,                                                    işlev
                off64_t  uzunluk)
```

Bu işlev **ftruncate** işlevinin benzeridir. Farkı, *uzunluk* argümanının 32 bitlik makinalarda bile 64 bitlik genişlikte olmasıdır. Böylece 2^63 bayta kadar dosya uzunlukları belirtilebilir.

32 bitlik bir sistemde, kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev **ftruncate** ismiyle bulunur ve eski gerçekleme tamamen LFS'ye uygun olarak değiştirilir.

Yukarıda bahsedildiği gibi burada **ftruncate** işlevinin **mmap** ile birlikte kullanılışına küçük bir örnek vardır::

```
int fd;
void *start;
size_t len;

int
add (off_t at, void *block, size_t size)
```

```
{
  if (at + size > len)
    {
      /* Dosyanın boyunu değiştir ve belleğe eşle.  */
      size_t ps = sysconf (_SC_PAGESIZE);
      size_t ns = (at + size + ps - 1) & ~(ps - 1);
      void *np;
      if (ftruncate (fd, ns) < 0)
        return -1;
      np = mmap (NULL, ns, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
      if (np == MAP_FAILED)
        return -1;
      start = np;
      len = ns;
    }
  memcpy ((char *) start + at, block, size);
  return 0;
}
```

add işlevi dosyada keyfi bir konuma bir bellek bloğunu yazar. Eğer dosyanın mevcut uzunluğu yetersizse dosya uzatılır. Uzatmanın sayfa sayısına yuvarlandığına dikkat edin. Bu **mmap**'in bir gereksinimidir. Yazılım gerçek boyutu daima izler ve işlem bittiğinde son bir **ftruncate** çağrısıyla dosyanın gerçek boyunu belirler.

10. Özel Dosyaların Oluşturulması

mknod işlevi aygıt dosyaları gibi özel dosyaları oluşturmakta kullanılan bir ilkeldir. GNU kütüphanesi bu işlevi BSD uyumluluğu adına içerir.

mknod işlevi `sys/stat.h` başlık dosyasında bildirilmiştir.

```
int mknod(const char *dosyaismi,                                    işlev
          int         kip,
          int         aygıt)
```

mknod işlevi ismi *dosyaismi* ile belirtilen özel dosyayı oluşturur. *kip* argümanı ile özel dosyalarla ilgili çeşitli bitleri içeren kip belirtilir. Örneğin karakter aygıtı dosyaları için **S_IFCHR** veya blok aygıtı dosyaları için **S_IFBLK**. Bkz. *Bir Dosyanın Türünün Sınanması* (sayfa: 375).

aygıt argümanı ile dosyanın hangi aygıt ile ilişkilendirileceği belirtilir. En doğru yorumu, oluşturulan özel dosyanın çeşidine bağlıdır.

İşlev başarılı olduğunda **0**, aksi takdirde **-1** ile döner. *Dosya ismi hatalarına* (sayfa: 234) ek olarak aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EPERM
: İşlevi çağıran süreç ayrıcalıklı değil. Sadece ayrıcalıklı kullanıcı özel dosyaları oluşturabilir.

ENOSPC
: Yeni dosyayı içerecek dizin ya da dosya sistemi dolu ve genişletilemiyor.

EROFS
: Yeni dosyayı içerecek dizin salt–okunur bağlı bir dosya sistemi üzerinde.

EEXIST
: Zaten *dosyaismi* isminde bir dosya var. Bu dosyayı değiştirmek istiyorsanız önce eskisini silmelisiniz.

11. Geçici Dosyalar

Yazılımınızda bir geçici dosya kullanmanız gerekiyorsa, onu açmak için **tmpfile** işlevini ya da geçici dosyaya bir isim vermek ve onu sonradan **fopen** ile açmak istiyorsanız **tmpnam** (daha iyisi: **tmpnam_r**) işlevini kullanabilirsiniz.

tempnam işlevi **tmpnam** gibidir, ancak geçici dosyaların gideceği dizini belirtebilirsiniz, bunun dışında dosya isimlendirmesi aynı yöntemle yapılır. Çok evreli yazılımlar açısından **tempnam** işlevinin evresel olması ama **tmpnam** işlevinin bir durağan tampona gösterici döndürmesiyle evresel olmaması önemli bir farktır.

Bu oluşumlar `stdio.h` balık dosyasında bildirilmiştir.

`FILE *tmpfile(void)` işlev

Bu işlev, **"wb+"** kipinde (güncelleme kipi) **fopen** çağrısı ile açılmış gibi bir geçici ikilik dosya oluşturur. Bu dosya kapatıldığında ya da yazılım sonlandığında dosya özdevinimli olarak silinir. (Bazı diğer ISO C sistemlerinde eğer yazılım anormal şekilde sonlanırsa dosya silinmeyebilir.)

Bu işlev evreseldir.

Kaynakların 32 bitlik bir sistemde **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev aslında **tmpfile64** işlevidir, yani LFS arayüzü eski arayüzün yerine geçer.

`FILE *tmpfile64(void)` işlev

Bu işlev **tmpfile** işlevine benzer, ancak dönen akım 32 bitlik makinalarda 2^31 bayttan daha büyük dosyalar için kullanılabilir.

Dönüş türünün hala **FILE *** olduğunu, LFS arayüzüne özel bir **FILE** türü olmadığına lütfen dikkat edin.

32 bitlik bir sistemde, kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği durumda bu işlev **tmpfile** ismiyle bulunur ve eski gerçekleme tamamen LFS'ye uygun olarak değiştirilir.

`char *tmpnam(char *sonuç)` işlev

Bu işlev herhangi bir mevcut dosyaya ait olmayan bir geçerli dosya ismi oluşturur ve bunu döndürür. Eğer *sonuç* argümanı bir boş gösterici ise dönüş değeri bir dahili durağan dizgeye bir gösterici olup, işlevin daha sonraki çağrıları ile üzerine yazılabilir, dolayısıyla bu durumda işlev evresel olmayacaktır. Aksi takdirde, *sonuç* argümanı en az **L_tmpnam** karakterlik bir diziye gösterici olmalıdır. Bu durumda sonuç bu diziye yazılacak ve işlev evresel olacaktır.

Önceden oluşturulmuş dosyaları silmeden defalarca **tmpnam** çağrısı yaparsanız işlevin başarısız olma olasılığı vardır. Bu, geçici dosya isimlerine ayrılan uzunluğun sınırlı olması nedeniyle işlevin sadece sonlu sayıda farklı isme imkan vermesindendir. Eğer işlev başarısız olursa bir boş gösterici döndürür.

Uyarı

Dosya isminin oluşturulması sırasında, başka bir süreç daha **tmpnam** kullanarak aynı isimde bir dosya oluştururs a, bu bir güvenlik açığına yol açabilir. Gerçekleme tahmini zor isimler üretir, fakat yine de dosyayı açarken **O_EXCL** seçeneğini kullanmalısınız. Bu sorunla karşılaşmamak için en iyi yöntem **tmpfile** veya **mkstemp** kullanmaktır.

`char *tmpnam_r(char *sonuç)` işlev

Bu işlev, eğer *sonuç* bir boş gösterici ise boş gösterici döndürmesi dışında **tmpnam** işlevininin hemen hemen aynısıdır.

tmpnam işlevinin evresel olmayan kullanımına karşı bir önlem içermesiyle bu işlevin evresel olması garanti edilmiştir.

Uyarı

Bu işlev de **tmpnam** işlevinin oluşturabildiği güvenlik açığı sorunundan muzdariptir.

`int` **L_tmpnam** makro

Bu makronun değeri **tmpnam** işlevi ile üretilen dosya ismini tutacak yeterli büyüklükteki dizge için en küçük uzunluğu veren bir tamsayı sabit ifadesidir.

`int` **TMP_MAX** makro

TMP_MAX makrosunun değeri **tmpnam** ile oluşturulabilecek geçici dosya isimlerinin sayısının alt sınırıdır. **tmpnam** işlevini, çok fazla geçici dosyaya sahip olduğunuzu belirterek başarısız olmadan en azından bu kadar defa çağırabilirsiniz.

GNU kütüphanesi ile çok büyük sayıda geçici dosya ismi oluşturabilirsiniz. Eğer bu isimlerle gerçekten dosya oluşturmaya çalışırsanız daha isimler tükenmeden disk üzerindeki yeriniz tükenebilir. Diğer sistemlerde sabit ve daha az sayıda geçici dosya ismi oluşturulabilir ve bu sınır asla **25**'i aşmaz.

```
char *tempnam(const char *dizin,
              const char *önek)
```
işlev

Bu işlev tamamen eşsiz bir geçici dosya ismi oluşturur. Eğer *önek* bir boş gösterici değilse, bu dizgenin ilk beş karakteri dosya isminde önek olarak kullanılır. İşlevin dönüş değeri **malloc** ile ayrılmış bir dizgedir. Dolayısıyla bu alanla işiniz bittiğinde, artık kullanmayacaksanız **free** ile serbest bırakmalısınız.

Dönen dizge özdevimli olarak ayrıldığından bu işlev evreseldir.

Geçici dosya isminin dizin öneki aşağıdaki listedeki maddeler sırayla uygulanarak saptanır. Dizin mevcut ve yazılabilir olmalıdır.

- Eğer tanımlıysa, **TMPDIR** ortam değişkeni. Güvenlik kaygılarıyla bu sadece yazılım SUID ya da SGID etkin değilse uygulanır.
- Bir boş gösterici değilse, *dizin* argümanı.
- **P_tmpdir** makrosunun değeri.
- **/tmp** dizini.

Bu işlev SVID uyumluluğu için tanımlanmıştır.

Uyarı

Dosya isminin oluşturulması sırasında, başka bir süreç daha **tmpnam** kullanarak aynı isimde bir dosya oluşturursa, bu bir güvenlik açığına yol açabilir. Gerçekleme tahmini zor isimler üretir, fakat yine de dosyayı açarken **O_EXCL** seçeneğini kullanmalısınız. Bu sorunla karşılaşmamak için en iyi yöntem **tmpfile** veya **mkstemp** kullanmaktır.

`char *`**P_tmpdir** SVID makrosu

Bu makro geçici dosyalar için öntanımlı dizin ismidir.

Daha eski Unix sistemleri burya kadar bahsedilen işlevlere sahip değildi. Bunların yerine **mktemp** ve **mkstemp** işlevleri kullanılırdı. Bu işlevlerin her ikisi de belirttiğiniz bir dosya ismi şablon dizgesini değiştirerek çalışır. Bu dizgenin son altı karakteri XXXXXX olmalıdır. Bu altı X dizgenin eşsiz olmasını sağlamak üzere altı karakterle değiştirilir. Kullanılan şablon dizgesi şuna benzer:

/tmp/*önek*XXXXXX

Burada *önek* yazılım tarafından belirlenen eşsiz bir dizgedir.

Bilgi

mktemp ve **mkstemp** şablon dizgesini değiştirdiklerinde dolayı, dizgeyi bir sabit olarak aktarmamalısınız. Dizge sabitler normalde salt–okunur saklama alanına sahiptir. Bu bakımdan **mktemp** veya **mkstemp** dizge sabitini değiştirmeye çalışırsa yazılımınız çökebilir.

Bu işlevler stdlib.h başlık dosyasında bildirilmiştir.

`char *mktemp(char *şablon)` işlev

mktemp işlevi, yukarıda açıklandığı gibi *şablon* dizgesini değiştirerek eşsiz bir dosya ismi üretir. İşlev başarılı olduğunda değiştirilen *şablon* ile döner. Eğer işlev eşsiz bir isim bulamazsa *şablon*'u boş dizge haline getirip döner. Eğer *şablon*'un son altı karakteri XXXXXX değilse işlev bir boş gösterici döndürür.

Uyarı

Dosya isminin oluşturulması sırasında, başka bir süreç daha **mktemp** kullanarak aynı isimde bir dosya oluştururursa, bu bir güvenlik açığına yol açabilir. Gerçekleme tahmini zor isimler üretir, fakat yine de dosyayı açarken **O_EXCL** seçeneğini kullanmalısınız. Bu sorunla karşılaşmamak için en iyi yöntem **mkstemp** kullanmaktır.

`int mkstemp(char *şablon)` işlev

mkstemp işlevi **mktemp**'in yaptığı gibi bir eşsiz dosya ismi oluşturur, fakat ayrıca dosyayı sizin için **open** (*Dosyaların Açılması ve Kapatılması* (sayfa: 306)) ile açar. İşlev başarılı olursa, *şablon* dizgesini yerinde değiştirir ve dosyayı okuma ve yazma için açarak bir dosya tanıtıcı ile döner. Eğer işlev bir eşsiz dosya ismi oluşturamazsa, **-1** ile döner. Eğer *şablon* dizgesi XXXXXX ile bitmiyorsa, işlev **-1** ile döner ve *şablon* dizgesini değiştirmez.

Dosya **0600** kipi ile açılır. Eğer dosyaya diğer kullanıcılarında erişebilmesi isteniyorsa bu kip ayrıca değiştirilmelidir.

mktemp'in aksine, **mkstemp** işlevi bir geçici dosya oluşturmaya çalışan başka yazılımlarla çatışmadan eşsiz bir dosya oluşturmayı garanti eder. Bu, **open** işlevinin **O_EXCL** seçeneği ile kullanmasından dolayıdır (bu seçenek sayesinde, dosyayı oluşturmak isterseniz ve böyle bir dosya mevcutsa bir hata alırsınız).

`char *mkdtemp(char *şablon)` işlev

mkdtemp işlevi, ismi eşsiz bir dizin oluşturur. Başarılı olursa dizinin ismini *şablon*'a yazar ve *şablon* ile döner. **mktemp** ve **mkstemp** gibi *şablon* dizgesi XXXXXX ile bitmelidir.

Eğer **mkdtemp** bir eşsiz isimli dizin oluşturamazsa, boş gösterici ile döner ve *errno* değişkenine ilgili hata durumını atar. Eğer *şablon* dizgesi XXXXXX ile bitmiyorsa, **mkdtemp** işlevi **NULL** ile döner ve *şablon*'u değiştirmez. Bu durumda *errno* değişkenine **EINVAL** atanır.

Dizin **0700** kipiyle oluşturulur.

mkdtemp işlevi bir geçici dizin oluşturmaya çalışan başka yazılımlarla çatışmadan eşsiz bir dizin oluşturmayı garanti eder. Bu, **open** işlevini **O_EXCL** seçeneği ile kullanmasından dolayıdır. Bkz. *Dizinlerin Oluşturulması* (sayfa: 370).

mkdtemp işlevi OpenBSD'den gelir.

XV. Borular ve FIFOlar

İçindekiler

Bir ***boru*** süreçler arası haberleşme mekanizmasıdır; bir süreç tarafından boruya yazılan veri başka bir süreç tarafından okunabilir. Veri ilk giren, ilk çıkar (FIFO) sırasıyla ele alınır. Borunun adı yoktur; bir kullanımlık oluşturulur ve her iki uç, boruyu oluşturan süreç tarafından erişilebilir olmalıdır.

Bir ***FIFO özel dosyası*** boru ile aynıdır, fakat anonim, geçici bağlantı olmak yerine, bir FIFO'nun bir adı veya diğer dosyalar gibi isimleri vardır. Süreçler FIFO'yu üzerinden haberleşmek için açarlar.

Boru veya FIFOnun her iki ucu aynı anda açılmalıdır. Eğer herhangi bir sürecin üzerine yazmadığı bir boru veya FIFO dosyasından okuma yapıyorsanız (belki hepsi dosyayı kapatmış veya çıkmış olabilir), okuma sonucunda dosya–sonu (EOF) döner. Üzerinde okuma işlemi olmayan bir boru veya FIFO'ya yazmak hata durumu olarak karşılanır; bir **SIGPIPE** sinyali üretir ve eğer sinyal yakalanıyor ya da bloklanıyorsa **EPIPE** hata koduyla sonlanır.

Ne borular ne de FIFO özel dosyaları dosya içinde konumlamaya izin vermez. Okuma ve yazma işlemleri sırayla gerçekleşir; dosyanın başından okunur ve sonuna yazılır.

1. Bir Borunun Oluşturulması

Boru oluşturmak için en ilkel işlev **pipe** işlevidir. Bu borunun okuma ve yazma uçlarının her ikisini de oluşturur. Tek bir sürecin kendisiyle konuşması için boru kullanımı pek kullanışlı değildir. Tipik kullanım şekli, bir işlemin bir veya daha fazla *alt süreci oluşturmadan* (sayfa: 687) önce boruyu oluşturmasıdır. Bundan sonra boru üst ve alt süreç arasında veya iki alt süreç arasında haberleşme için kullanılır.

pipe işlevi `unistd.h` başlık dosyası içinde tanımlıdır.

`int` **pipe**`(int` *dosyatnm*`[2])` işlev

> **pipe** işlevi boruyu oluşturur ve borunun okuma ve yazma uçları için dosya tanımlayıcıları (sırasıyla) *dosyatnm*`[0]` ve *dosyatnm*`[1]` içine koyar.
>
> Girdi ucunun önce geldiğini hatırlamanın kolay bir yolu dosya tanımlayıcı **0**'ın standart girdi ve dosya tanımlayıcı **1**'in standart çıktı olmasıdır.
>
> Başarı halinde **pipe**, **0** değerini döndürür. Başarısızlık halinde ise **−1** döndürülür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:
>
> `EMFILE`
> Sürecin çok sayıda açık dosyası var.
>
> `ENFILE`
> Sistemde çok sayıda açık dosya var. **ENFILE** hakkında daha fazla bilgi için *Hata Kodları* (sayfa: 32) bölümüne bakınız. Bu hata GNU sisteminde hiçbir zaman oluşmaz.

Burada basit bir boru oluşturma yazılımı görüyoruz. Bu yazılım **fork** işlevini (*Bir Sürecin Oluşturulması* (sayfa: 687)) alt süreç oluşturmak için kullanmıştır. Üst süreç veriyi boruya yazar, alt süreç okur.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

/* Borudan karakterleri oku ve stdout'a yaz. */

void
read_from_pipe (int dosya)
{
  FILE *akim;
  int c;
  akim = fdopen (dosya, "r");
  while ((c = fgetc (akim)) != EOF)
    putchar (c);
  fclose (akim);
}

/* Boruya rastgele birşeyler yaz. */

void
write_to_pipe (int dosya)
{
  FILE *akim;
  akim = fdopen (dosya, "w");
  fprintf (akim, "Merhaba!\n");
  fprintf (akim, "Elveda!\n");
  fclose (akim);
}

int
main (void)
{
  pid_t pid;
  int boru[2];

  /* Boruyu yarat. */
  if (pipe (boru))
    {
      fprintf (stderr, "Boruda hata oluştu.\n");
      return EXIT_FAILURE;
    }

  /* Alt süreci oluşturalım. */
  pid = fork ();
  if (pid == (pid_t) 0)
    {
      /* Bu bir alt süreç.
         Önce diğer ucu kapatalım. */
      close (boru[1]);
      read_from_pipe (boru[0]);
      return EXIT_SUCCESS;
    }
  else if (pid < (pid_t) 0)
```

```
    {
      /* Alt süreç oluşturulamadı. */
      fprintf (stderr, "Alt süreç oluşturulamadı.\n");
      return EXIT_FAILURE;
    }
  else
    {
      /* Bu bir üst süreç.
         Önce diğer ucu kapatalım. */
      close (boru[0]);
      write_to_pipe (boru[1]);
      return EXIT_SUCCESS;
    }
}
```

2. Bir Alt Sürece Boru Hattı

Boruların genel kullanım şekli alt süreç olarak çalışan bir yazılım ile veri alışverişidir. Bunu yapmanın bir yolu **pipe** (boru oluşturmak için), **fork** (alt süreç oluşturmak için), **dup2** (bir alt süreci bir boruyu standart girdi veya çıktı kanalı olarak kullanmaya zorlamak için) ve **exec** (yeni bir yazılımı çalıştırmak için) birleşimini kullanmaktır. Ya da, **popen** ve **pclose** işlevlerini kullanabilirsiniz..

popen ve **pclose** kullanmanın yararı arayüzünün daha basit ve kullanımının kolay oluşudur. Ancak düşük seviyeli işlevleri doğrudan kullanma esnekliği yoktur.

```
FILE *popen(const char *komut,                                    işlev
            const char *kip)
```

popen işlevi **system** işlevleriyle yakından ilgilidir; bkz. *Bir Komutun Çalıştırması* (sayfa: 685). *komut* kabuk komutunu alt süreç olarak çalıştırır. Fakat, komutun bitmesini beklemek yerine, bir alt süreç borusu oluşturur ve boruyla ilişkili bir akım döndürür.

Eğer *kip* argümanı için **"r"** belirtirseniz, alt sürecin standart çıktı kanalındaki veriyi almak için akımdan okuma yapabilirsiniz. Alt süreç, standart girdi kanalının özelliklerini üst süreçten (miras) alır.

Benzer bir şekilde, *kip* argümanı için **"w"** belirtirseniz, alt sürecin standart girdi kanalına veri göndermek için akıma yazabilirsiniz. Alt süreç, standart çıktı kanalının özelliklerini üst süreçten alır.

Hata durumunda **popen** boş gösterici ile döner. Bu boru veya akım oluturulamadığında veya yazılım çalıştırılamadığında olabilir.

```
int pclose(FILE *akım)                                            işlev
```

pclose işlevi **popen** tarafından oluşturulan *akım*ı kapatmak için kullanılır. **system** işlevinin yaptığı gibi, alt sürecin bitmesi için bekler ve onun durum değerini döndürür.

Burada **popen** ve **pclose** işlevlerinin, çıktının **more** sayfalama yazılımı gibi başka bir yazılımla filtrelenmesi için nasıl kullanıldığını gösteren bir örnek görüyoruz.

```
#include <stdio.h>
#include <stdlib.h>

void
write_data (FILE * akim)
{
  int i;
```

```
  for (i = 0; i < 100; i++)
    fprintf (akim, "%d\n", i);
  if (ferror (akim))
    {
      fprintf (stderr, "Akıma yazılamadı.\n");
      exit (EXIT_FAILURE);
    }
}

int
main (void)
{
  FILE *cikti;

  cikti = popen ("more", "w");
  if (!cikti)
    {
      fprintf (stderr,
               "ya parametre yanlis ya da dosya sayisi fazla.\n");
      return EXIT_FAILURE;
    }
  write_data (cikti);
  if (pclose (cikti) != 0)
    {
      fprintf (stderr,
               "Artik calistirilamiyor veya baska bir hata var.\n");
    }
  return EXIT_SUCCESS;
}
```

3. FIFO Özel Dosyaları

Bir FIFO özel dosyası boruya benzer, ancak oluşturulma şekli farklıdır. Anonim bir haberleşme kanalı olmaktansa, FIFO özel dosyası dosya sistemine **mkfifo** işlevi çağrılarak girer.

FIFO özel dosyasını bir kere bu şekilde oluşturduktan sonra, herhangi bir süreç, sıradan bir dosyaya yapıldığı gibi, onu okuma veya yazma için açabilir. Fakat, üzerinde her hangi bir girdi veya çıktı işlemi sürdürmeden önce her iki ucun da eş zamanlı açılması gerekir. FIFO'yu okumak için açmak, normalde başka bir süreç aynı FIFO'yu yazmak için açana kadar bloke eder (Tam tersi de geçerlidir).

mkfifo işlevi `sys/stat.h` başlık dosyası içinde tanımlıdır.

```
int mkfifo(const char *dosyaismi,                                    işlev
           mode_t      kip)
```

mkfifo işlevi *dosyaismi* adında bir FIFO özel dosyası oluşturur. *kip* argümanı dosyanın izinlerini belirlemek için kullanılır; bkz. *Dosya İzinlerinin Atanması* (sayfa: 380).

mkfifo'nun normal hatasız dönüş değeri **0**'dır. Hata durumunda, **-1** döndürülür. Bildik *dosya ismi hatalarına* (sayfa: 234) ek olarak, aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EEXIST`
: Bu isimde dosya zaten var.

`ENOSPC`
: Dizin veya dosya sistemi genişletilemez.

EROFS
: Dosyayı içeren dizin salt–okunur bir dosya sisteminde.

4. Borunun G/Ç Bütünlüğü

Yazılan verinin miktarı **PIPE_BUF** değerinden büyük olmadığı sürece borudan okuma ve yazma işlemi ***atomik*** bir işlemdir. Bu veri aktarımının anlık bir birim olarak göründüğü anlamına gelir, bu nedenle sistemdeki hiçbir şey tamamlanmış halini gözlemleyemez. Atomik G/Ç hemen başlayamayabilir (tampon alanı veya veri için beklemesi gerekebilir), fakat başladı mı hemen biter.

Büyük miktarda veri okumak veya yazmak atomik olmayabilir; örneğin, dosya tanımlayıcısını paylaşan diğer süreçlerin çıktı verisi araya serpiştirilmiş olabilir. Aynı zamanda, bir kere **PIPE_BUF**'a karakterler yazıldığında, okuma yapılıncaya kadar başka yazımlar durdurulur.

PIPE_BUF parametresi hakkında daha fazla bilgi için *Dosya Sistemi Kapasite Sınırları* (sayfa: 795) bölümüne bakınız.

XVI. Soketler

İçindekiler

Bu kısımda soketleri kullanarak süreçlerarası iletişim (IPC– InterProcess Communication) için GNU oluşumlarından bahsedilmiştir.

Bir ***soket*** genelleştirilmiş süreçlerarası iletişim kanalıdır. Boru (pipe) gibi soket de bir dosya tanımlayıcı olarak temsil edilir. Borulardan farklı olarak soketler birbiriyle ilişkisi olmayan süreçler arasındaki iletişimi ve hatta ağ üzerindeki farklı makinalar üzerinde çalışan süreçler arası iletişimi de destekler. Soketlerin birincil kullanım alanları farklı makinalarla iletişimdir; **`telnet`**, **`rlogin`**, **`ftp`**, **`talk`** ve diğer bildik ağ yazılımları soketleri kullanır.

Bütün işletim sistemleri soketleri desteklememektedir. GNU kütüphanesinde, işletim sisteminden bağımsız olarak `sys/socket.h` başlık dosyası ve beraberinde soket işlevleri daima bulunur, fakat eğer sistem soketleri desteklemiyorsa bu işlevler daima başarısız olur.

Eksik

Yayın iletileri veya Internet arayüzünün ayarlanması ile ilgili oluşumlar henüz belgelenmemiştir. IPv6 ile ilgili bazı yeni işlevler ve evresel (reentrant) işlevler de henüz belgelenmemiştir.

1. Soket Kavramları

Bir soket oluşturduğunuzda, kullanmak istediğiniz iletişim tarzını ve bunu uygulayacak protokol türünü de belirtmeniz gerekir. Bir soketin ***iletişim tarzı***, soket üzerinde veri gönderimi ve alımının kullanıcı–seviyesindeki anlamını tanımlar. İletişim tarzının seçimi aşağıdaki sorulara cevap olur:

Veri iletimi birimleri nelerdir?
Bazı iletişim tarzları veriye büyük bir yapısı olmayan bir dizi bayt olarak bakar; diğerleri ise bu baytları gruplayarak birer kayıt olarak ele alır (bu bağlamda ***paket*** olarak bilinirler).

Normal bir işlem sırasında veri kaybı olur mu?
Bazı iletişim tarzları gönderilen her verinin ulaştığını garanti eder (sistem veya ağ göçmediği takdirde); diğer tarzlar ara sıra gerçekleşen veri kaybını normal karşılayıp, bazı paketleri mükerrer veya yanlış sırada gönderebilirler.

Güvenilmez bir iletişim tarzını kullanan bir yazılım tasarımı genellikle kayıp veya yanlış sırada gönderilen paketleri tespit edecek, gerektiğinde veriyi tekrar gönderecek tedbirler içerir.

İletişim sadece tek eşle mi gerçekleştirilir?
Bazı iletişim tarzları telefon konuşmasına benzer— karşı soket ile bir ***bağlantı*** kurarsınız ve veri alışverişini gerçekleştirirsiniz. Diğer tarzlar ise mektuba benzer—gönderilecek her ileti için adres belirtmek gerekir.

Ayrıca, soketi isimlendirmek için bir ***isim alanı*** seçilmelidir. Soket adı ("adresi") sadece belirli bir isim alanı içerisinde anlamlıdır. Aslında, soket ismi için kullanılacak veri türü bile isim alanına bağlı olabilir. İsim alanları "Etki Alanı" (domain) olarak da adlandırılır, ancak kavram karmaşası yaratmamak için bu kullanımdan kaçınıyoruz.

Her isim alanının **PF_** ile başlayan sembolik bir ismi vardır. Buna ilişkin **AF_** ile başlayan sembolik isim bu isim alanının adres biçimini gösterir.

Son olarak i sağlayacak protokolün seçilmesi gerekir. ***Protokol*** veri gönderim ve alımında hangi alt seviye mekanizmanın kullanılacağını belirler. Her protokol belirli bir isim alanı ve iletişim tarzı için geçerlidir; bir isim alanı zaman zaman ***protokol ailesi*** olarak adlandırıldığı için, isim alanının sembolik ismi **PF_** (Protocol Family) ile başlar.

Protokol kuralları iki yazılım, belki de iki ayrı bilgisayar arasında geçen veriye uygulanır; bu kuralların birçoğu işletim sistemi tarafından halledilir ve sizin hakkında bilgi sahibi olmanız gerekmez. Protokoller ile ilgili sizin bilmeniz gerekenler:

- İki soket arasında iletişimin gerçekleşmesi için, soketler *aynı* protokolü kullanmalıdır.
- Her protokol belirli bir tarz ve isim alanının bir birleşimi olarak anlam kazanır ve uygun olmayan bileşimlerle kullanılamaz. Örneğin TCP protokolü sadece bayt akımı tarzında iletişim ile İnternet isim alanına uyar.
- Her tarz ve isim alanı bileşimi için bir ***öntanımlı protokol*** vardır, protokol numarasına 0 verilerek istenebilir. Normalde yapmanız gereken de budur—öntanımlı olanı kullanın.

Bu kısımdaki açıklamaların başından sonuna çeşitli yerlerde değişken/parametre boyutunu göstermek gerekmektedir. Ve işte burada sorun başlar. İlk uygulamalarda bu değişkenlerin değişken türü basitçe **int** idi. Zamanımızda bir çok makinada **int** 32 bit genişliğindedir ve *fiilen* 32 bitlik bir değişken standardı yaratmıştır. Bu tür değişkenlerin referansları, çekirdeğe aktarıldığı için önemlidir.

Ardından Posixçiler, "bütün büyüklük değerleri **size_t** türündedir" sözleriyle arayüzü birleştirmiştir. 64 bitlik makinalarda **size_t** 64 bit genişliğindedir, böylece değişkenlere göstericiler ortadan kalkmıştır.

Unix98 belirtimi **socklen_t** türü ile bir çözüm üretmiştir. Bu tür, POSIX'in **size_t** olarak değiştirdiği tüm hallerde kullanılır. Bu türün tek gereksinimi işaretsiz en az 32 bittir. Bu nedenle 32 bitlik değişkenlere aktarılacak göstericiler, 64 bitlik değerler kullanan uygulamalarla kolayca aktarılabilir.

2. İletişim Tarzları

GNU kütüphanesi her biri farklı özellikte, çeşitli türlerde soketleri destekler. Bu bölüm desteklenen soket türlerini anlatmaktadır. Burada listelenen sembolik sabitler `sys/socket.h` içerisinde tanımlanmıştır.

`int` **SOCK_STREAM** makro

SOCK_STREAM tarzı bir *boru* (sayfa: 393) gibidir. Belirli bir karşı soket bağlantısıyla çalışır ve veriyi bir bayt akımı halinde güvenle iletir.

Bu tarz ayrıntılı olarak *Soketlerin Bağlantılarla Kullanılması* (sayfa: 422) konu başlığı altında açıklanmıştır.

`int` **SOCK_DGRAM** makro

SOCK_DGRAM tarzı, tek tek adreslenen paketlerin güvensiz bir şekilde iletiminde kullanılır. Bu, **SOCK_STREAM** tarzının tam karşıtıdır.

Sokete bu türde her veri yazımında, veri bir paket haline gelir. **SOCK_DGRAM** soketlerinin bağlantıları olmadığı için her paketle birlikte alıcı adresinin belirtilmesi gerekir.

Sistemin sizin veri iletimi ile ilgili isteklerinizde verdiği tek garanti gönderilen her paketin teslimi için elinden gelenin en iyisini deneyeceğidir. Dördüncü ve beşinci pakette başarısızlığa uğradıktan sonra altıncı pakette başarıya ulaşabilir, yedinci paket altıncıdan önce ulaşabilir ve altıncı paketten sonra ikinci defa ulaştırılabilir.

SOCK_DGRAM'ın tipik kullanım şekli, makul bir süre içerisinde karşı taraftan yanıt gelmemesi halinde paketin tekrar gönderilmesinin kabul edilebildiği durumlardır.

Datagram Soket İşlemleri (sayfa: 433), konu başlığı altında datagram soketlerinin kullanımı hakkında ayrıntılı bilgi bulunmaktadır.

`int` **`SOCK_RAW`** makro

Bu arz alt–seviye ağ protokolleri ve arayüzlerine erişimi desteklemektedir. Sıradan kullanıcı yazılımları genellikle bu tarzı kullanma ihtiyacı duymazlar.

3. Soket Adresleri

Soket ismi genelde ***adres*** olarak kullanılır. Soket adresleriyle ilgili işlev ve sembollerin isimlendirmesinde tutarsızlıklar vardır, bazen "isim" terimi bazen "adres" terimi kullanılmıştır. Soket konusu içinde bu terimleri eşanlamlı kabul edebilirsiniz.

`socket` işlevi ile yeni oluşturulan bir soketin adresi yoktur. Diğer süreçlerin onunla iletişim kurması için adres vermeniz gereklidir. Biz buna ***adresin sokete bağlanması*** diyoruz ve bunu yapmak için **`bind`** işlevini kullanıyoruz.

Diğer süreçlerin soketi bulup iletişime başlayabilmesi için soket adresine ihtiyaç duyulacaktır. Siz diğer soketleri kullanacaksanız onlara bir adres belirtebilirsiniz, fakat bu genelde anlamsızdır; soketten ilk veri gönderiminde veya onunla bir bağlantı başlattığınızda, siz belirtmediyseniz sistem otomatik olarak bir adres atayacaktır.

Bazen istemcinin adres belirtmesi gerekir çünkü sunucu adrese göre ayrım yapmaktadır; örneğin, **`rsh`** ve **`rlogin`** protokolleri istemcinin soket adresine bakar ve sadece geçiş parolasının `IPPORT_RESERVED` (sayfa: 415) değerinden daha küçük olup olmadığını kontrol eder.

Soket adresleriyle ilgili ayrıntılar kullandığınız isim alanına bağlı olarak değişir. Bu konu hakkında daha ayrıntılı bilgi *Yerel İsim Alanı* (sayfa: 404) veya *İnternet İsim Alanı* (sayfa: 406) konu başlıkları altında bulunabilir.

Soket adresini belirtmek ve sınamak için **`bind`** and **`getsockname`** işlevleri isim alanına bakılmaksızın kullanılabilir. Bu işlevler adresi kabul etmek için sahte bir veri türü olan **`struct sockaddr *`** türünü kullanırlar. Pratikte adres sizin kullandığınız biçime uygun başka bir veri türündeki yapıda bulunur, fakat **`bind`** işlevine aktarırken adresi **`struct sockaddr *`** biçimine çevirmelisiniz.

3.1. Adres Biçimleri

`bind` ve **`getsockname`** işlevleri soket adresine gösterici olarak genel bir veri türü olan **`struct sockaddr *`** türünü kullanırlar. Bu veri türünün bir adresi yorumlamak veya oluşturmak için kullanılması verimli değildir; bunun için soketin isim alanı ile uyumlu bir veri türünün kullanılması gerekir.

Bu nedenle genel kullanımda, uygun isim alanına özgü bir adres oluşturulur, ardından da **`bind`** veya **`getsockname`** çağrılarak **`struct sockaddr *`** türünde bir göstericiye dönüştürülür.

`struct sockaddr` veri türünden alabileceğiniz bir bilgi parçası da ***adres biçim tasarımcısı***dır. Bu, adresin tamamını anlamak için hangi veri türünü kullanacağınızı belirtir.

Bu bölümdeki semboller `sys/socket.h` başlık dosyasında tanımlanmıştır.

`struct` **`sockaddr`** veri türü

`struct sockaddr` türü aşağıdaki üyelere sahiptir:

`short int` **`sa_family`**
Bu, adresin adres biçim kodudur. Takip eden verinin biçimini tanımlar.

`char` **`sa_data`**`[14]`
Biçime bağımlı olan asıl soket adresi verisidir. Uzunluğu da biçime bağlıdır ve 14 den de büyük olabilir. **`sa_data`**'nın 14 olan uzunluğu temelde isteğe bağlıdır.

Her adres biçimi **AF_** ile başlayan bir sembolik isme sahiptir. Her biri ilgili isim alanını tasarlayan bir **PF_**sembolü ile eşleşir. Aşağıda adres biçim isimlerinin listesi görülmektedir:

AF_LOCAL
: Bu, yerel isim alanıyla giden adres biçimini tasarlar (PF_LOCAL isim alanının adıdır). Bu adres biçimi hakkında *Yerel İsim Alanı ile İlgili Ayrıntılar* (sayfa: 405) bölümünde daha ayrıntılı bilgi bulabilirsiniz.

AF_UNIX
: Bu **AF_LOCAL** ile eşanlamlıdır. Gerçi **AF_LOCAL** POSIX.1g tarafından önerilmişse de, **AF_UNIX** sistemler arasında daha taşınabilirdir. **AF_UNIX**, BSD kaynaklı geleneksel ismidir, hatta birçok POSIX sistemleri de onu desteklemektedir. Bu aynı zamanda Unix98 belirtimi için de seçilen isimdir. (Aynı şey **PF_UNIX** ve dolayısıyla **PF_LOCAL** için de geçerlidir.)

AF_FILE
: Bu da uyumluluk için konulmuştur ve **AF_LOCAL** ile aynıdır (Keza **PF_FILE** da **PF_LOCAL** ile aynıdır).

AF_INET
: Bu İnternet isim alanıyla giden adres biçimini tasarlar (**PF_INET** isim alanının adıdır). Bkz. *İnternet Soket Adreslerinin Biçimleri* (sayfa: 407).

AF_INET6
: Bu, **AF_INET** ile benzerdir, fakat IPv6 protokolüyle ilgilidir (**PF_INET6** IPv6'ya ilişkin isim alanının adıdır).

AF_UNSPEC
: Bu belirli bir adres biçimi tasarlamaz. Kullanımı çok nadirdir, örneğin "bağlı" bir datagram soketinin varsayılan hedef adresini silmek için kullanılır. Bkz. *Datagramların Gönderilmesi* (sayfa: 433).

  İlişkin isim alanı tasarlayıcı sembolü **PF_UNSPEC** tamamlayıcı olarak bulunmaktadır, fakat bir yazılım içinde kullanmanın bir anlamı yoktur.

`sys/socket.h` birçok farklı ağ türü için **AF_** ile başlayan sembolleri tanımlar, birçoğu veya tamamı aslında gerçekleştirilmemiştir. Bunlar düzgün çalışmaya başladığı zaman ve nasıl kullanıldığı hakkında bilgi edindiğimizde bunları da belgelendireceğiz.

3.2. Adreslerin Atanması

Bir sokete adres atamak için **bind** işlevi kullanılır. **bind** işlevinin prototipi `sys/socket.h` başlık dosyasındadır. Örnek kullanımlar için *Soketlerde Yerel İsim Alanı Örneği* (sayfa: 406) veya *İnternet Soketi Örneği* (sayfa: 419)'ne bakınız.

```
int bind(int                soket,                                  işlev
         struct sockaddr *adres,
         socklen_t          uzunluk)
```

bind işlevi *soket* soketine bir adres atar. *adres* ve *uzunluk* argümanları adresi belirtir; adresin ayrıntılı biçimi isim alanına bağlıdır. Adresin ilk kısmı daima, isim alanını belirten ve adresin o isim alanı biçiminde olduğunu söyleyen, biçim tasarlayıcısıdır.

Hata olduğunda **-1** olmadığında **0** değeri döndürür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *soket* argümanı geçerli bir dosya tanımlayıcı değil.

ENOTSOCK

socket tanımlayıcı bir soket değil.

EADDRNOTAVAIL
: Belirtilen adres bu makinada bulunmamaktadır.

EADDRINUSE
: Başka bir soket zaten belirtilen adresi kullanmaktadır.

EINVAL
: *soket* soketi zaten bir adrese sahiptir.

EACCES
: İstenilen adrese erişim için izniniz yok. (İnternet etki alanı içinde, sadece süper kullanıcı 0 ile **IPPORT_RESERVED - 1** aralığında port numarası tanımlayabilir; bkz. *İnternet Portları* (sayfa: 415).

Soketin özel isim alanına bağlı olarak ek koşullar mümkün olabilir.

3.3. Adresin Okunması

getsockname işlevi bir İnternet soketinin adresini almak için kullanılır. Bu işlevin prototipi `sys/socket.h` başlık dosyasındadır.

```
int getsockname(int                soket,                                  işlev
                struct sockaddr *adres,
                socklen_t       *uzunluk–gstr)
```

getsockname işlevi *soket* soketinin *adres* ve *uzunluk–gstr* argümanlarıyla belirlenen adresi ile ilgili bilgiyi döndürür. *uzunluk–gstr* argümanının bir gösterici olması nedeniyle *adres* boyutu kadar yer ayrılacak şekilde ilklendirilmelidir, böylece değer döndürüldüğünde adres verisinin gerçek boyutunu içerecektir.

Adres verisinin biçimi soket isim alanına bağlıdır. Belirtilen bir isim alanının bilgi uzunluğu genelde sabittir, böylece normalde ne kadar alan gerektiğini tam olarak bilirsiniz. Genel uygulama soketin isim alanı için uygun bir veri türü kullanarak değer için yer ayırmak, ardından adresi **getsockname**'e aktarmak için **struct sockaddr *** türüne dönüştürmektir.

Hata olduğunda **-1** olmadığında **0** değeri döndürür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *soket* argümanı geçerli bir dosya tanımlayıcı değil.

ENOTSOCK
: *socket* tanımlayıcı bir soket değil.

ENOBUFS
: Bu işlem için yeterli dahili tampon yok.

Dosya isim alanındaki bir soketin adresini okuyamazsınız. Bu sistemin geri kalanı için de geçerlidir, bir dosya tanımlayıcıdan dosyanın ismini bulmak mümkün değildir.

4. Arayüz İsimlendirmesi

Her ağ arayüzünün bir ismi vardır. Bu isim genellikle arayüz türüyle ilişkili birkaç harften oluşur, buna ek olarak eğer aynı türde birden fazla arayüz varsa sonuna bir numara eklenir. Örneğin **lo** (geridönüş arayüzü – loopback interface) ve **eth0** (birinci Ethernet arabirimi).

Her ne kadar bunun gibi isimler insanlar için uygun olsa da bir yazılımın arayüz bilgisine her ihtiyaç olduğunda bu kullanım hantal kalabilir. Bu gibi durumlarda arayüze küçük pozitif tamsayı bir değer olan ***indis*** ile erişilir.

Sözü edilen işlevler, sabitler ve veri türleri `net/if.h` başlık dosyasında bildirilmiştir.

`size_t` **`IFNAMSIZ`** — sabit

Bu sabit, arayüz ismini ve sonlandırıcı boş karakteri tutmak için gerekli azami tampon boyutunu belirtir.

`unsigned int` **`if_nametoindex`**`(const char *`*arayüz_ismi*`)` — işlev

Bu işlev belirli bir isme karşılık gelen arayüz indisini verir. Belirtilen isimde bir arayüz yoksa 0 döndürür.

`char *`**`if_indextoname`**`(unsigned int` *arayüz_indisi*`,`
`char *`*arayüz_ismi*`)` — işlev

Bu işlev bir arayüz indisini karşılık gelen arayüz ismine eşler. Döndürülen isim *arayüz_ismi* ile gösterilen tampona yerleştirilir (*arayüz_ismi* en az **`IFNAMSIZ`** bayt uzunluğunda olmalıdır). İndis geçersizse işlevin dönüş değeri bir boş göstericidir, aksi takdirde *arayüz_ismi* 'dir.

`struct` **`if_nameindex`** — veri türü

Bu veri türü bir arayüz ile ilgili bilgiyi tutmak için kullanılır. Aşağıdaki üyelere sahiptir:

`unsigned int` **`if_index`**
Arayüz indisidir.

`char *`**`if_name`**
Boş karakter sonlandırmalı arayüz ismidir.

`struct if_nameindex *`**`if_nameindex`**`(void)` — işlev

Bu işlev her biri mevcut arayüzlerin **`if_nameindex`** yapılarından oluşan bir dizi döndürür. Listenin sonu 0 arayüzü ve bir boş isim göstericisi içeren bir yapı ile belirtilmiştir.

Döndürülen yapı kullanımdan sonra **`if_freenameindex`** ile serbest bırakılmalıdır.

`void` **`if_freenameindex`**`(struct if_nameindex *`*gstr*`)` — işlev

Bu işlev evvelce yapılmış bir **`if_nameindex`** çağrısından dönen yapıyı serbest bırakır.

5. Yerel İsim Alanı

Bu bölüm sembolik ismi (soket oluşturulurken ihtiyaç duyulur) **`PF_LOCAL`** olan yerel isim alanı ile ilgili ayrıntıları açıklar. Yerel isim alanı "Unix etki alanı soketleri" olarak da bilinir. Diğer bir ismi de dosya isim alanıdır, çünkü soket adresleri genelde dosya isimleri olarak gerçeklenmiştir.

5.1. Yerel İsim Alanı Kavramları

Yerel isim alanında soket adresleri dosya isimleridir. Soket adresi olarak istediğiniz herhangi bir dosya ismini verebilirsiniz, fakat onu içeren dizine yazma hakkı vermeniz gereklidir. Bu dosyalar zaten genellikle **`/tmp`** dizini içine konulur.

Yerel isim alanının tuhaf bir özelliği de ismin sadece bağlantı açılırken kullanılmasıdır; bir kere adres açıldıktan sonra anlamı yoktur ve bulunmasa da olur.

Diğer bir tuhaflık ise böyle bir sokete, diğer makina soketin ismini içeren dosya sistemini paylaşsa da bağlanılamamasıdır. Soketi dizin listesinde görebilirsiniz ancak bağlanamazsınız. Bazı yazılımlar bunun avantajını kullanırlar, örneğin istemciye kendi süreç kimliğini (PID) sorar ve farklı istemcileri ayırt etmek için süreç kimliğini kullanır. Fakat, biz size tasarladığınız protokollerde bu yöntemi kullanmanızı tavsiye etmeyiz, belki bir gün aynı dosya sistemini kullanan diğer makinalardan da bağlantılara izin verilebilir. Bunun yerine, her yeni istemciye onu belirleyici bir numara gönderebilirsiniz.

Yerel isim alanındaki soketi kapattıktan sonra, dosya ismini silmeniz gerekir. **unlink** veya **remove**'u bunun için kullanınız. Bilgi için *Dosyaların Silinmesi* (sayfa: 368) bölümüne bakınız.

Yerel isim alanı herhangi bir iletişim tarzı için sadece bir protokolü destekler; bunun protokol numarası **0**'dır.

5.2. Yerel İsim Alanı ile İlgili Ayrıntılar

Yerel isim alanında bir soket oluşturmak için, **socket** veya **socketpair** işlevinin *isimalanı* argümanında **PF_LOCAL** sabitini kullanın. Bu sabit `sys/socket.h` dosyası içerisinde tanımlıdır.

`int` **PF_LOCAL** makro

Bu içerisindeki soket adresleri yerel adlar olan yerel isim alanını ve onula ilgili protokol ailesini gösterir. **PF_Local** ise Posix.1g tarafından kullanılan makrodur.

`int` **PF_UNIX** makro

PF_LOCAL ile aynıdır ve uyumluluk için konmuştur.

`int` **PF_FILE** makro

PF_LOCAL ile aynıdır ve uyumluluk için konmuştur.

Yerel isim alanları içindeki soket isimlerini tanımlayan yapı `sys/un.h` başlık dosyası içinde tanımlıdır:

`struct` **sockaddr_un** veri türü

Bu yapı yerel isim alanının soket adreslerini tanımlamak için kullanılır. Aşağıdaki üyelere sahiptir:

`short int` **sun_family**
Bu *soket adresinin* (sayfa: 401) adres ailesini veya biçimini belirtir. Yerel isim alanını belirtmek için AF_LOCAL değerini saklamalısınız.

`char` **sun_path**`[108]`
Bu kullanılacak dosyanın ismidir.

Bitmedi!

108 neden sihirli bir numaradır? RMS bu sıfır uzunluğundaki dizinin oluşturulması ve aşağıdaki örnekteki gibi dosya isminin uzunluğuna göre uygun miktarda saklama alanı ayrılması için **alloca** işlevinin kullanılmasını önermektedir.

Yerel isim alanındaki bir soket adresi için *uzunluk* parametresini, dosya isminin dizge uzunluğu (dizgeye ayrılan alan değil) ve **sun_family** bileşeninin toplamı olarak hesaplamalısınız. Bu **SUN_LEN** makrosu kullanılarak yapılabilir:

`int` **SUN_LEN**`(struct sockaddr_un *`*göstersi*`)` makro

Bu makro yerel isim alanındaki soket adresinin uzunluğunu hesaplar.

5.3. Soketlerde Yerel İsim Alanı Örneği

Buradaki örnekte yerel isim alanında bir soketin nasıl oluşturulduğu ve isimlendirildiği gösterilmiştir.

```
#include <stddef.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>

int
make_named_socket (const char *dosyaismi)
{
  struct sockaddr_un isim;
  int soket;
  size_t boyut;

  /* Soketi oluşturalım. */
  soket = socket (PF_LOCAL, SOCK_DGRAM, 0);
  if (soket < 0)
    {
      perror ("socket");
      exit (EXIT_FAILURE);
    }

  /* Sokete bir isim verelim. */
  isim.sun_family = AF_LOCAL;
  strncpy (isim.sun_path, dosyaismi, sizeof (isim.sun_path));
  isim.sun_path[sizeof (isim.sun_path) - 1] = '\0';

  /* Adresin büyüklüğü,
     dosya isminin başlangıç konumu
     artı uzunluğu
     artı bir boş karakterdir.
     Bir seçenek olarak bunu kullanabilirsiniz:
      boyut = SUN_LEN (&isim);
  */
  boyut = (offsetof (struct sockaddr_un, sun_path)
           + strlen (isim.sun_path) + 1);

  if (bind (soket, (struct sockaddr *) &isim, boyut) < 0)
    {
      perror ("bind");
      exit (EXIT_FAILURE);
    }

  return soket;
}
```

6. İnternet İsim Alanı

Bu bölümde protokollerle ilgili ayrıntılar ve İnternet isim alanında kullanılan soket isimlendirme eğilimlerini açıklanmıştır.

Başlangıçta İnternet isim alanı sadece IP sürüm 4'ü (IPv4) kullanırdı. İnternetteki konak sayısının artmasıyla,

daha büyük bir adres alanına sahip yeni bir protokol gerekti; IP sürüm 6 (IPv6). IPv6 128 bitlik adresleri ortaya koydu (IPv4 32 bitliktir) ve diğer özellikleriyle de sonunda IPv4'ün yerine geçecektir.

IPv4 İnternet isim alanında soket oluştururken **socket** veya **socketpair** işlevinde argüman olarak **PF_INET** sembolik ismini kullanın. IPv6 adresleri için **PF_INET6** makrosu gerekir. Bu makrolar `sys/socket.h` başlık dosyasında tanımlanmıştır.

`int` **PF_INET** makro

IPv4 İnternet isim alanını ve onunla ilişkili protokol ailesini belirtir.

`int` **PF_INET6** makro

IPv6 İnternet isim alanını ve onunla ilişkili protokol ailesini belirtir.

Bir İnternet isim alanı soket adresi aşağıdaki bileşenleri içerir:

- Bağlanmak istediğiniz makinanın adresi. İnternet adresi çeşitli yollarla belirtilebilir; bunlar *İnternet Soket Adreslerinin Biçimleri* (sayfa: 407), *Konak Adresleri* (sayfa: 408) ve *Konak İsimleri* (sayfa: 412) konularında açıklanmıştır.
- Bağlanılacak makinanın port numarası. Bkz. *İnternet Portları* (sayfa: 415).

Adres ve port numarasının ***ağ bayt sırası*** denilen meşru biçimde gösterildiğinden emin olmalısınız. Bilgi için *Bayt Sırası Dönüşümü* (sayfa: 417) bölümüne bakınız.

6.1. İnternet Soket Adreslerinin Biçimleri

İnternet isim alanında, hem IPv4 (**AF_INET**) hem de IPv6 (**AF_INET6**) protokolleri için, bir soket adresi bir konak adresiyle onun bir portunun adresinden oluşur. Ek olarak, seçtiğiniz protokol de adresin bir parçası olur, çünkü yerel port numaraları sadece belli bir protokol içinde anlam kazanır.

İnternet isim alanında soket adreslerinin gösteriminde kullanılan veri türleri `netinet/in.h` başlık dosyasında tanımlıdır.

`struct` **sockaddr_in** veri türü

İnternet isim alanındaki soket adreslerinin gösteriminde kullanılan veri türüdür. Aşağıdaki üyelere sahiptir:

`sa_family_t` **sin_family**
: Soket adresinin adres ailesini veya biçimini tanımlar. **AF_INET** değerini bu üye içinde saklamanız gerekir. Bkz. *Soket Adresleri* (sayfa: 401).

`struct in_addr` **sin_addr**
: Konak makinanın İnternet adresidir. Bir değerin nasıl alınacağı veya buraya kaydedeceği *Konak Adresleri* (sayfa: 408) ve *Konak İsimleri* (sayfa: 412) bölümlerinde anlatılmıştır.

`unsigned short int` **sin_port**
: Port numarasıdır. Bkz. *İnternet Portları* (sayfa: 415).

bind veya **getsockname** işlevlerini çağırdığınızda, eğer IPv4 İnternet isim alanı soket adreslerini kullanıyorsanız *uzunluk* parametresi olarak **sizeof (struct sockaddr_in)** değerini belirtmelisiniz.

`struct` **sockaddr_in6** veri türü

Bu veri türü IPv6 isim alanındaki soket adreslerinin gösteriminde kullanılır. Aşağıdaki üyelere sahiptir:

`sa_family_t` **sin6_family**

Soket adresinin adres ailesini veya biçimini tanımlar. **AF_INET6** değerini bu üye içinde saklamanız gerekir. Bkz. *Soket Adresleri* (sayfa: 401).

`struct in6_addr` **sin6_addr**
Konak makinanın IPv6 adresidir. Bir değerin nasıl alınacağı veya buraya kaydedeceği *Konak Adresleri* (sayfa: 408) ve *Konak İsimleri* (sayfa: 412) bölümlerinde anlatılmıştır.

`uint32_t` **sin6_flowinfo**
Bu alan henüz gerçeklenmemiştir.

`uint16_t` **sin6_port**
Port numarasıdır. Bkz. *İnternet Portları* (sayfa: 415).

6.2. Konak Adresleri

İnternetteki her bilgisayarın, o bilgisayarı internetteki diğer bilgisayarlardan ayıran ve tanımlayan numaralardan oluşan bir veya daha fazla ***internet adresi*** vardır. Kullanıcılar genellikle IPv4 numaralı konak adreslerini, **128.52.46.32** gibi noktalarla ayrılmış dört numaralı bir dizi olarak yazarlar. IPv6 numaralı konak adreslerini ise **5f03:1200:836f:c100::1** gibi iki nokta üst üste ile ayrılmış en fazla sekiz bölümden oluşan onaltılık numaralarla yazarlar.

Her bilgisayarın ayrıca, noktalarla ayrılmış kelimelerden oluşan **mescaline.gnu.org** gibi bir veya birden fazla ***konak ismi*** vardır.

Kullanıcıların konak adreslerini belirtmelerine olanak sağlayan yazılımlar genellikle hem numaralı adresi hem de konak ismini kabul eder. Bir bağlantı açmak için yazılım numaralı bir adrese ihtiyaç duyar ve bu nedenle konak ismini karşılık gelen numaralı adrese çevirmek zorundadır.

6.2.1. Kısaca Konak Adresleri

IPv4 İnternet konak adresi dört baytlık veri tutan bir numaradır. Tarihsel olarak bakıldığında bunlar iki kısma ayrılır, bir ***ağ numarası*** ve o ağ içerisindeki bir ***yerel ağ adresi***. 1990'ların ortasında ortaya çıkan sınıfsız adresler bu yaklaşımı değiştirdi. Bazı işlevler içlerinde eski tanımları aradıkları için, biz öncelikle sınıf tabanlı ağları, sonra da sınıfsız adresleri anlatacağız. IPv6 sadece sınıfsız adresleri kullanır, bu nedenle aşağıdaki paragraflar IPv6 adreslerine uygulanmaz.

Sınıf tabanlı IPv4 ağ numaraları ilk bir, iki veya üç bayttan oluşur; geriye kalan baytlar yerel adreslerdir.

IPv4 ağ numaraları Ağ Bilgi Merkezi (NIC – Network Information Center) denilen merkeze kayıtlıdır ve A, B, C adında üç sınıfa bölünmüşlerdir. Yerel ağdaki makinaların yerel ağ adres numaraları o bu merkezce değil, ağın yöneticisi tarafından kaydedilir.

A sınıfı ağların 0 ile 127 arasında değişen tek baytlık bir ağ numarası vardır. Az sayıda A sınıfı ağ vardır, fakat herbiri devasa boyutlarda konak sayısına sahiptir. Orta ölçekli B sınıfı ağların ilk baytı 128'den 191'e kadar olan iki baytlık ağ numaraları vardır. C sınıfı ağlar en küçükleridir; ilk baytı 192 ile 255 arasında olan üç baytlık ağ numaraları vardır. Bu nedenle İnternet adresinin ilk bir, iki veya üç baytı ağı belirtir. İnternet adresinin kalan baytları da o ağ içerisindeki adresini belirtir.

A sınıfı ağdaki 0 bütün ağlara yayın için ayrılmıştır. Ek olarak, her ağdaki 0 numaralı konak adresi ağ içerisindeki bütün konaklara yayın için ayrılmıştır. Bu kullanımlar günümüzde kullanılmamaktadır fakat uyumluluk nedeniyle 0 numaralı ağı ve 0 numaralı konak numarasını kullanmamalısınız.

A sınıfı ağdaki 127 geridönüş (loopback) arabirimi için ayrılmıştır; 127.0.0.1 İnternet adresini konak makina için kullanabilirsiniz.

Bir makina birden fazla ağın üyesi olabileceği için birden fazla İnternet konak adresine sahip olabilir. Fakat, bir adres hiç bir zaman birden fazla makinayı belirtemez.

İnternet adresleri için standart noktalı gösterimde dört numaralama biçimi vardır:

a.b.c.d
: Bu adresin dört baytlık kısmının tamamını tanımlar ve genelde kullanılan gösterim şeklidir.

a.b.c
: Adresin son kısmı, *c*, 2 baytlık bir büyüklük olarak yorumlanır. Bu *a.b* ağ adres numaralı B sınıfı bir ağdaki konak adreslerinin belirtilmesi için kullanışlıdır.

a.b
: Adresin son kısmı, *b*, 3 baytlık bir büyüklük olarak yorumlanır. Bu *a* ağ adres numaralı A sınıfı bir ağdaki konak adreslerinin belirtilmesi için kullanışlıdır.

a
: Eğer sadece tek parça verilirse, bu doğrudan konak adres numarasına karşılıktır.

Adresin her kısmında, tabanı tanımlamak için alışıldık C kabulleri uygulanır. Diğer bir deyişle, sayının, başında **0x** veya **0X** varsa onaltılık tabanda, **0** varsa sekizlik tabanda; diğer durumlarda onluk tabanda olduğu kabul edilir.

6.2.2. Sınıfsız Adresler

IPv4 adreslerini (ve IPv6 adreslerini) şimdi sınıfsız olarak düşünürsek; A, B ve C sınıfları arasındaki farkları görmezden gelebiliriz. Bir IPv4 konak adresi 32 bitlik adres ve 32 bitlik maskeden oluşmaktadır. Maske ağ kısmı için birler ve konak kısmı için sıfırlar içerir. Ağ kısmı soldan itibaren birlerle, kalanı da konağı gösteren sıfırlardan oluşur. Sonuç olarak, ağ maskesi (netmask) sadece bir bitleriyle belirtilebilir. A, B ve C sınıfları bu kuralın sadece özel durumlarıdır. Örneğin, A sınıfı adreslerinin ağ maskesi **255.0.0.0**'dır veya 8 uzunluğunda önekleri vardır.

Sınıfsız IPv4 ağ adresleri, noktalı gösterimin sonuna bir bölü ayracıyla eklenmiş önek uzunluğu biçiminde yazılır. Örneğin 10 numaralı A sınıfı ağı **10.0.0.0/8** olarak yazılır.

6.2.3. IPv6 Adresleri

IPv6 adresleri 128 bitlik veri içerir (IPv4 32 bit içerir). Bir konak adresi genelde iki nokta üst üste (:) ile ayrılmış sekiz adet 16 bitlik onaltılık sayı olarak yazılır. Ardarda gelen sıfırları kısaltmak için iki adet iki nokta üst üste (:) yan yana konulur. Örneğin, IPv6 geridönüş adresi **0:0:0:0:0:0:0:1**, sadece **::1** şeklinde yazılabilir.

6.2.4. Konak Adresinin Veri Türü

IPv4 İnternet konak adresleri bazı yaklaşımlarda (**uint32_t** türünde) tamsayı değerler olarak gösterilir. Diğer yaklaşımlarda tamsayı **struct in_addr** türünde bir yapı içerisine paketlenir. Kullanımın tutarlı olması daha iyi olurdu ancak tamsayı değerini yapıdan çıkarmak veya yapı içerisine sokmak zordur.

IPv4 İnternet konak adresleri için **uint32_t** veya **struct in_addr** kullanmak yerine **unsigned long int** kullanan eski kodlar bulabilirsiniz. Eskiden **unsigned long int** 32 bitlik numaraydı fakat 64 bitlik makinalarla birlikte bu değişti. **unsigned long int** türünün 32 bit olmadığı makinalarda bu veri türü kodun çalışmamasına neden olabilir. **uint32_t** türü Unix98 tarafından tanımlanmış ve 32 bit olması garanti edilmiştir.

IPv6 İnternet konak adresleri 128 bittir ve **struct in6_addr** yapısı içerisinde paketlenmişlerdir.

Aşağıdaki İnternet adreslerinin temel tanımları `netinet/in.h` başlık dosyasında tanımlanmıştır:

`struct` **in_addr**	veri türü

Bu veri türü, IPv4 İnternet konak adresini tutmak için kullanılmaktadır. Konak adres numarasını **uint32_t** olarak kaydeden **s_addr** adında sadece bir alana sahiptir.

`uint32_t` **INADDR_LOOPBACK** makro

Makinanın gerçek adresini bulmak yerine bu sabiti "makinanın adresi" olarak kullanabilirsiniz. Bu genellikle **localhost** olarak çağrılır ve IPv4 İnternet adresi **127.0.0.1**'dir. Bu özel sabit kendi makinanızın adresini bulma çilesinden sizi kurtarır. Ayrıca, özellikle, makinanın kendisiyle konuşması durumunda herhangi bir ağ trafiğinden kaçınmak için sistem **INADDR_LOOPBACK**'i kullanır.

`uint32_t` **INADDR_ANY** makro

Bu sabiti adres verilmesi sırasında "gelen herhangi bir adres" yerine kullanabilirsiniz. Bkz. *Adreslerin Atanması* (sayfa: 402). Bu, İnternet bağlantılarını kabul etmek istediğinizde **struct sockaddr_in** yapısının **sin_addr** üyesine vereceğiniz adres olarak kullanışlıdır.

`uint32_t` **INADDR_BROADCAST** makro

Bu sabit bir yayın iletisi göndermek için kullanacağınız adrestir.

`uint32_t` **INADDR_NONE** makro

Bu sabit bazı işlevler tarafından hata olarak döndürülür.

`struct` **in6_addr** veri türü

Bu veri türü bir IPv6 adresi tutmak için kullanılır. Çeşitli yollarla (bir birleşik yapı üzerinden) erişilebilir 128 bitlik veri tutar.

`struct in6_addr` **in6addr_loopback** sabit

Bu sabit IPv6 **::1** geridönüş adresidir. Bunun ne anlama geldiğini öğrenmek için yukarıya bakınız. **IN6ADDR_LOOPBACK_INIT** makrosu kendi değişkenlerinize bu değeri verebilmeniz için sağlanmıştır.

`struct in6_addr`**in6addr_any** sabit

Bu sabit IPv6 **::** belirtilmemiş adresidir. Bunun ne anlama geldiğini öğrenmek için yukarıya bakınız. **IN6ADDR_ANY_INIT** makrosu kendi değişkenlerinize bu değeri verebilmeniz için sağlanmıştır.

6.2.5. Konak Adresi İşlevleri

İnternet adreslerini işlemek için kullanılan bu ek işlevler `arpa/inet.h` başlık dosyasında tanımlıdır. Bunlar İnternet adreslerini ağ bayt sırasında, ağ numaraları ve ağ içi yerel adres numaralarını da konak bayt sırasında göstermektedir. Ağ ve konak bayt sırası ile ilgili açıklamayı *Bayt Sırası Dönüşümü* (sayfa: 417) bölümünde bulabilirsiniz.

```
int inet_aton(const char      *isim,
              struct in_addr *adres)
```
işlev

Bu işlev *isim* IPv4 İnternet konak adresini standart noktalı gösterimden ikilik veriye dönüştürür ve *adres*'in gösterdiği **struct in_addr** içinde saklar. Eğer adres geçerli ise **inet_aton** sıfırdan farklı bir değer, aksi takdirde sıfır döndürür.

```
uint32_t inet_addr(const char *isim)
```
işlev

Bu işlev *isim* isimli IPv4 İnternet konak adresini standart noktalı gösterim şeklinden ikilik veriye dönüştürür. Eğer girdi geçerli değilse, **inet_addr** işlevi **INADDR_NONE** döndürür. Bu yukarıda anlatıldığı gibi **inet_aton** için atıl bir arayüzdür. Atıl olmasının nedeni **INADDR_NONE** adresinin (255.255.255.255) geçerli bir adres olmasıdır ve **inet_aton** işlevinin hata döndürmek için daha temiz bir yol sunmasıdır.

```
uint32_t inet_network(const char *isim)                                    işlev
```

Bu işlev, standart noktalı gösterim şeklinde verilen *isim* adresinden ağ numarasını elde eder. Döndürülen adres konak bayt sırasındadır. Eğer girdi geçerli değilse, **inet_network** işlevi **-1** değerini döndürür.

İşlev sadece geleneksel IPv4 A, B ve C sınıfı ağ türleri ile çalışır. Sınıfsız adreslerle çalışmaz ve bu şekilde kullanılmamalıdır.

```
char *inet_ntoa(struct in_addr adres)                                      işlev
```

Bu işlev *adres* IPv4 İnternet konak adresini standart noktalı gösterim şeklinde bir dizgeye dönüştürür. Dönüş değeri durağan olarak ayrılmış tampona bir göstericidir. İşlevin sonraki çağrılarında aynı tampon üzerine yazılacağından, yazılan değer kaydedileceğinden değerin kopyalanması gerekir.

Çok evreli (multi–threaded) yazılımlarda her evrenin kendi durağan olarak ayrılmış tamponu vardır. Fakat hala **inet_ntoa**'nın sonraki çağrılarında aynı evre son çağrıdaki sonucun üzerine yazar.

inet_ntoa yerine yeni bir işlev olan **inet_ntop** kullanılmalıdır, çünkü bu hem IPv4 hem de IPv6 adreslerini desteklemektedir.

```
struct in_addr inet_makeaddr(uint32_t ağ,                                  işlev
                             uint32_t yerel)
```

Bu işlev *ağ* ağ numarası ile ağ içindeki *yerel* yerel adres numarası değerlerini birleştirerek bir IPv4 İnternet konak adresi yapar.

```
uint32_t inet_lnaof(struct in_addr adres)                                  işlev
```

Bu işlev *adres* İnternet konak adresinin ağ içindeki yerel adres kısmını döndür.

İşlev sadece geleneksel IPv4 A, B ve C sınıfı ağ türleri ile çalışır. Sınıfsız adreslerle çalışmaz ve bu şekilde kullanılmamalıdır.

```
uint32_t inet_netof(struct in_addr adres)                                  işlev
```

Bu işlev *adres* İnternet konak adresinin ağ numarası kısmını döndürür.

İşlev sadece geleneksel IPv4 A, B ve C sınıfı ağ türleri ile çalışır. Sınıfsız adreslerle çalışmaz ve bu şekilde kullanılmamalıdır.

```
int inet_pton(int          biçim,                                          işlev
              const char *adres,
              void       *tampon)
```

Bu işlev bir İnternet adresini metin gösteriminden ağ biçimine (ikilik) dönüştürür. *biçim* dönüştürüleceği adres türünü belirtmek üzere **AF_INET** veya **AF_INET6** olmalıdır. *adres* girilen dizgeye bir göstericidir ve *tampon* sonuç için kullanılan tampona göstericidir. Tamponun yeterli büyüklükte olmasını işlevi çağıran sağlamalıdır.

```
const char *inet_ntop(int          biçim,                                  işlev
                      const void *adres,
                      char       *tampon,
                      size_t      uzunluk)
```

Bu işlev bir İnternet adresini ağ biçiminden (ikilik), metin gösterimine dönüştürür. *biçim* dönüştürülecek adres türünü belirtmek üzere **AF_INET** veya **AF_INET6** olmalıdır. *adres* çevrilecek adrese bir göstericidir. *tampon* sonuç için kullanılan tampona göstericidir ve *uzunluk* bu tamponun uzunluğudur. İşlevden dönecek değer tamponun adresi olacaktır.

6.2.6. Konak İsimleri

İnternet adresleri için standart noktalı gösterime ek olarak bir konağa ulaşmak için sembolik isim de kullanılabilir. Sembolik ismin yararı akılda kalmasının kolay oluşudur. Örneğin, İnternet adresi **158.121.106.19** olan bir makina **alpha.gnu.org** şeklinde de bilinir; ve **gnu.org** etki alanındaki diğer makinalar ona sadece **alpha** ile erişebilir.

Arka planda, sistem, konak isimlerini konak numaralarına eşleyerek bunların kayıtlarını tutabileceği bir veritabanı kullanır. Bu veritabanı genellikle `/etc/hosts` dosyası veya bir isim sunucusunun sunduğu eşdeğeridir. Veritabanına erişim için kullanılan işlev ve sembol tanımları `netdb.h` içerisindedir. `netdb.h` dosya olarak içerilirse, BSD'nin özellikleri de tümüyle gelir.

`struct` **hostent** — veri türü

Bu veri türü konak veritabanındaki bir girdinin gösterimi için kullanılmıştır. Aşağıdaki üyelere sahiptir:

`char *`**h_name**
: Bu konağın "resmi" adıdır.

`char **`**h_aliases**
: Bunlar konağın diğer adlarıdır, boş karakter sonlandırmalı dizgeler dizisi olarak gösterilmişlerdir.

`int` **h_addrtype**
: Bu konak adres tiürüdür; pratikte, değeri her zaman **AF_INET** veya IPv6 konakları için kullanıldıklarında **AF_INET6**'dır. Prensipte diğer türde adresler veritabanında İnternet adresleri olarak gösterilebilirler; eğer bu yapılırsa, bu alanda **AF_INET** veya **AF_INET6**'dan farklı bir değer görebilirsiniz. Bkz. *Soket Adresleri* (sayfa: 401).

`int` **h_length**
: Bu her adresin bayt olarak uzunluğudur.

`char **`**h_addr_list**
: Bu konak adres dizgeleri dizisidir. (Hatırlatma: bir konak birden fazla ağa bağlı olabilir ve her biri için farklı adrese sahip olabilir.) Dizi bir boş gösterici ile sonlandırılır.

`char *`**h_addr**
: Bu **h_addr_list[0]** ile aynıdır; diğer bir deyişle, ilk konak adresidir.

Konak veritabanı düşünülüğünde, her adres sadece **h_length** bayt uzunluğunda bir bellek bloğundan ibarettir. Fakat diğer yaklaşımlarda IPv4 adreslerin bir **struct in_addr** veya bir **uint32_t** şekline dönüştürülebileceği şeklinde bir iç varsayım vardır. **struct hostent** yapısındaki konak adresleri her zaman ağ bayt sırasında verilmiştir; bilgi için *Bayt Sırası Dönüşümü* (sayfa: 417) bölümüne bakınız.

Konak veritabanında arama yaparak belirli bir konak hakkında bilgi almak için **gethostbyname**, **gethostbyname2** veya **gethostbyaddr** işlevlerini kullanabilirsiniz. Bilgi durağan olarak ayrılmış bir yapı içinde döndürülür; çağrılar arasında kaydetmek için bilgiyi kopyalamanız gerekir. Ayrıca, **getaddrinfo** ve **getnameinfo** işlevlerini de bu bilgiye ulaşmak için kullanabilirsiniz.

`struct hostent *`**gethostbyname**`(const char *`*isim*`)` — işlev

gethostbyname işlevi *isim* ile isimlendirilmiş konak hakkında bilgi döndürür. Eğer bulamazsa, bir boş gösterici döndürür.

`struct hostent *`**gethostbyname2**`(const char *`*isim*`,`
`int` *biçim*`)` — işlev

gethostbyname2 işlevi **gethostbyname** gibidir, fakat çağırıcıya sonuç için istediği adres ailesini (örneğin **AF_INET** veya **AF_INET6**) belirtme imkanı sunar.

```
struct hostent *gethostbyaddr(const char *adres,                    işlev
                              size_t      uzunluk,
                              int         biçim)
```

gethostbyaddr işlevi *adres* İnternet adresine sahip konak hakkında bilgi döndürür. *adres* parametresi aslında bir karaktere gösterici değildir – bu IPv4 veya IPv6 adresine gösterici olabilir. *uzunluk* argümanı *adres* adresinin (bayt cinsinden) boyutudur. *biçim* adres biçimini belirtir; bir IPv4 İnternet adresi için, **AF_INET** değerini belirtiniz; bir IPv6 İnternet adresi için, **AF_INET6** kullanınız.

Eğer bulamazsa, **gethostbyaddr** boş gösterici döndürür.

Eğer **gethostbyname** veya **gethostbyaddr** ile isim sorgulama yapılamazsa, sebebini **h_errno** değişkeninin değerine bakarak bulabilirsiniz. (Bu işlevler için **errno** değerini değiştirmek daha temiz bir çözüm olurdu, ancak **h_errno** kullanımı diğer sistemlerle de uyumludur.)

Burada **h_errno** için karşılaşabileceğiniz hata kodlarını görüyoruz:

HOST_NOT_FOUND
Veritabanında böyle bir konak yok.

TRY_AGAIN
Bu durum isim sunucusu ile bağlantı kurulamadığında gerçekleşir. Eğer tekrar denerseniz, başarabilirsiniz.

NO_RECOVERY
Geri dönülemez bir hata oluştu.

NO_ADDRESS
Konak veritabanı isim için bir girdi içeriyor, fakat buna ilişkin bir İnternet adresi yok.

Yukarıdaki arama işlevlerinin ortak özellikleri: hiçbiri evresel (reentrant) işlevler değildir ve çok evreli uygulamalarda kullanılamazlar. Bu nedenle GNU C kütüphanesi bu bağlamda bir grup yeni işlev sunmaktadır.

```
int gethostbyname_r(const char *restrict       isim,                 işlev
                    struct hostent *restrict   sonuç_tamponu,
                    char *restrict             tampon,
                    size_t                     tampon_uzunluğu,
                    struct hostent **restrict  sonuç,
                    int *restrict              hatanum)
```

gethostbyname_r işlevi *isim* adındaki konak hakında bilgi döndürür. Çağrı sırasında işleve *sonuç_tamponu* parametresi ile **struct hostent** türünde bir nesneye gösterici aktarılmalıdır. Ek olarak işlev fazladan bir tampon alanına ihtiyaç duyabildiğinden çağrı sırasında işleve tampona bir gösterici *tampon* ile ve tamponun uzunluğu da *tampon_uzunluğu* ile aktarılmalıdır.

Sonucun tutulduğu tampona gösterici, başarılı bir işlev çağrısından sonra döndürülen **sonuç* içinde bulunmaktadır. Eğer bir hata oluşur veya girdi bulunamazsa **sonuç* göstericisi bir boş göstericidir. Başarı sıfır dönüş değeri ile belirtilir. Eğer işlev çalışmazsa dönüş değeri bir hata numarasıdır. **gethostbyname** için tanımlanan hatalara ek olarak bu **ERANGE**de olabilir. Bu durumda çağrı daha büyük bir tampon ile tekrarlanmalıdır. İlave hata bilgisi global değişken **h_errno**'da değil *hatanum* ile gösterilen nesnede saklanır.

Burada küçük bir örnek görüyoruz:

```
struct hostent *
gethostname (char *host)
{
  struct hostent hostbuf, *hp;
  size_t hstbuflen;
  char *tmphstbuf;
  int res;
  int herr;

  hstbuflen = 1024;
  /* Tamponu ayıralım, ama daha sonra bellek kaçağına
     neden olmamak için serbest bırakmayı unutmayalım.  */
  tmphstbuf = malloc (hstbuflen);

  while ((res = gethostbyname_r (host, &hostbuf, tmphstbuf, hstbuflen,
                                 &hp, &herr)) == ERANGE)
    {
      /* Tamponu büyütelim.  */
      hstbuflen *= 2;
      tmphstbuf = realloc (tmphstbuf, hstbuflen);
    }
  /*  Hata var mı, bakalım.  */
  if (res || hp == NULL)
    return NULL;
  return hp;
}
```

```
int gethostbyname2_r(const char                *isim,                 işlev
                     int                        biçim,
                     struct hostent *restrict   sonuç_tamponu,
                     char *restrict             tampon,
                     size_t                     tampon_uzunluğu,
                     struct hostent **restrict  sonuç,
                     int *restrict              hatanum)
```

gethostbyname2_r işlevi **gethostbyname_r** gibidir, fakat çağrı sırasında sonuç için istenen adres *biçim*ini (örneğin **AF_INET** veya **AF_INET6**) belirtme imkanı sunar.

```
int gethostbyaddr_r(const char                *adres,                işlev
                    size_t                     uzunluk,
                    int                        biçim,
                    struct hostent *restrict   sonuç_tamponu,
                    char *restrict             tampon,
                    size_t                     tampon_uzunluğu,
                    struct hostent **restrict  sonuç,
                    int *restrict              hatanum)
```

gethostbyaddr_r işlevi *adres* İnternet adresine sahip konak hakkında bilgi döndürür. *adres* parametresi aslında bir karaktere gösterici değildir – bu IPv4 veya IPv6 adresine gösterici olabilir. *uzunluk* argümanı *adres* adresinin (bayt cinsinden) boyutudur. *biçim* adres biçimini belirtir; bir IPv4 İnternet adresi için, bir **AF_INET** değeri belirtiniz; bir IPv6 İnternet adresi için, **AF_INET6** kullanınız.

gethostbyname_r işlevine benzer olarak, çağırıcı, sonuç için gerekli tampon bölgeyi ve iç kullanım için gerekli belleği ayarlamak zorundadır. Başarı halinde işlev sıfır döndürür. Aksi takdirde değer bir hata numarasıdır ve burada **ERANGE** çağırıcının–sunduğu tamponun yeterli olmadığını belirten özel bir anlama sahiptir.

sethostent, **gethostent** ve **endhostent** kullanarak bütün konak veritabanını bir girdi için taraya-bilirsiniz. Bu işlevleri kullanırken dikkatli olunuz, çünkü bunlar evresel (reentrant) işlevler değildirler.

`void` **sethostent**`(int` *açıkkal*`)` işlev

Bu işlev konak veritabanını tarama yapmak için açar. Bu işlev çağrısının ardından girdileri okumak için **gethostent** çağırabilirsiniz.

Eğer *açıkkal* argümanı sıfır değise, bu bir bayrağı kaldırarak **gethostbyname** veya **gethostbyaddr** işlevlerine yapılan çağrılarda veritabanının kapanmamasını sağlar (normalde olması gerektiği gibi). Bu yaklaşım işlevlerin sık çağırılması durumunda veritabanının her çağrıda tekrardan açılmasından kurtararak verimi artırır.

`struct hostent *`**gethostent**`(void)` işlev

Bu işlev konak veritabanındaki sıradaki girdiyi döndürür. Eğer başka girdi yoksa boş gösterici döndürür.

`void` **endhostent**`(void)` işlev

Bu işlev konak veritabanını kapatır.

6.3. İnternet Portları

İnternet isim alanındaki bir soket adresi bir makinanın İnternet adresi ve makina üzerindeki soketleri birbirinden ayıran (belirtilen protokole göre) ***port numarası***ndan oluşur. Port numaraları 0 ile 65,535 arasında değer alır.

IPPORT_RESERVED değerinin altındaki port numaraları **finger** ve **telnet** gibi standart sunucular için ayrılmıştır. Bunların kayıtlarını tutan bir veritabanı vardır ve bir servis ismini bir port numarasına eşleştirmek için **getservbyname** işlevini kullanabilirsiniz. Bilgi için *Servis Veritabanı* (sayfa: 415) bölümüne bakınız.

Eğer veritabanında tanımlanmış standart sunucular dışında bir sunucu yazarsanız onun için bir port numarası seçmeniz gerekir. Bunun için **IPPORT_USERRESERVED** değerinden büyük bir numara kullanın; bu numaralar sunucular için ayrılmıştır ancak sistem tarafından özdevinimli olarak üretilmemiştir. Diğer kullanıcılar tarafından çalıştırılan sunucularla karışıklıktan kaçınmak sizin sorumluluğunuzdadır.

Bir soketi adres belirtmeden kullanırsanız, sistem onun için bir port numarası üretir. Bu numara **IPPORT_RESERVED** ve **IPPORT_USERRESERVED** arasındadır.

Aslında internette iki farklı soketin bir port numarasını kullanması, her ikisi de aynı soket adresiyle (konak adresi artı port numarası) e çalışmadığı sürece olasıdır. Port numarasını üst düzey protokollerin bunu gerektirmesi gibi özel durumlar haricinde tekrar kullanmamalısınız. Normalde sistem bunu yapmanıza izin vermez; **bind** normalde farklı port numaraları vermenizde ısrar eder. Port numarasını tekrar kullanmak için, **SO_REUSEADDR** *soket seçeneğini* (sayfa: 439) etkinleştirmeniz gerekir.

Bu makrolar `netinet/in.h` başlık dosyasında tanımlıdır.

`int` **IPPORT_RESERVED** makro

IPPORT_RESERVED değerinden düşük port numaraları süper kullanıcının kullanımı için ayrılmıştır.

`int` **IPPORT_USERRESERVED** makro

IPPORT_USERRESERVED değerinden büyük ya da eşit port numaraları doğrudan kullanıma ayrılmıştır; bunlar özdevinimli ayrılmazlar.

6.4. Servis Veritabanı

Bilinen servislerin kayıtlarını tutan veritabanı genellikle `/etc/services` dosyası veya bir isim sunucusundaki eşdeğeridir. Sizler bu araçları, ki `netdb.h` içinde tanımlıdır, servis veritabanına erişim için kullanabilirsiniz.

`struct` **servent** veri türü

Bu veri türü servis veritabanındaki girdilerle ilgili bilgiyi tutar. Aşağıdaki üyelere sahiptir:

`char *s_name`
: Bu servisin "resmi" adıdır.

`char **s_aliases`
: Bunlar servisin diğer adlarıdır ve dizgelerden oluşan bir dizi olarak gösterilir. Bir boş gösterici diziyi sonlandırır.

`int s_port`
: Bu servisin port numarasıdır. Port numaraları ağ bayt sırasında verilmiştir; bilgi için *Bayt Sırası Dönüşümü* (sayfa: 417) bölümüne bakınız.

`char *s_proto`
: Bu servisle beraber kullanılan protokolün adıdır. Bkz. *Protokol Veritabanı* (sayfa: 417).

Belirli bir servis hakkında bilgi almak için, **getservbyname** veya **getservbyport** işlevlerini kullanınız. Bilgi durağan olarak ayrılmış bir yapı içinde döndürülür; çağrılar sırasında, kaydetme ihtiyacı duyarsanız bilgiyi kopyalamanız gerekir.

```
struct servent *getservbyname(const char *isim,                    işlev
                              const char *protokol)
```

getservbyname işlevi *isim* ile isimlendirilmiş *protokol* protokolünü kullanan servis hakkında bilgi döndürür. Eğer böyle bir servis bulamazsa boş gösterici döndürür.

Bu işlev sunucular için olduğu kadar istemciler için de kullanışlıdır; sunucular hangi portu *dinlemeleri* (sayfa: 423) gerektiğini bu işlevle belirlerler.

```
struct servent *getservbyport(int           port,                  işlev
                              const char *protokol)
```

getservbyport işlevi *port* portunda *protokol* protokolünü kullanan servis hakkında bilgi döndürür. Eğer böyle bir servis bulamazsa boş gösterici döndürür.

setservent, **getservent** ve **endservent** işlevlerini kullanarak da servis veritabanını tarayabilirsiniz. Bu işlevleri kullanırken dikkat ediniz çünkü bunlar evresel (reentrant) işlevler değildirler.

```
void setservent(int açıkkal)                                       işlev
```

Bu işlev servis veritabanını taramaya başlamak için açar.

Eğer *açıkkal* argümanı sıfır değise, bu bir bayrağı kaldırarak **getservbyname** veya **getservbyport** işlevlerine yapılan çağrılarda veritabanının kapanmamasını sağlar (normalde olması gerektiği gibi). Bu yaklaşım işlevlerin sık çağırılması durumunda veritabanının her çağrıda tekrardan açılmasından kurtararak verimi artırır.

```
struct servent *getservent(void)                                   işlev
```

Bu işlev servis veritabanındaki bir sonraki girdiyi geri döndürür. Eğer başka girdi yoksa boş gösterici döndürür.

```
void endservent(void)                                              işlev
```

Bu işlev servis veritabanını kapatır.

6.5. Bayt Sırası Dönüşümü

Bir kelime (word) içindeki bayt sırası dönüşümü için farklı bilgisayarlar farklı yaklaşımlar kullanır. Bazı bilgisayarlar bir kelimenin en anlamlı baytını başa (bu "big–endian" sıralama olarak adlandırılır) ve diğerleri de sona ("little–endian" sıralama) koyar.

İnternet protokolleri ağ üzerinde aktarılan veri için genlde geçerli bir bayt sırası yaklaşımı belirlediklerinden dolayı farklı bayt sıralama yöntemleri kullanan makinalar haberleşebilir. Bu ***ağ bayt sırası*** olarak bilinir.

Bir İnternet soket bağlantısı kurulacağında **sockaddr_in** yapısının üyeleri olan **sin_port** ve **sin_addr** verisinin ağ bayt sırasında gösterildiğinden emin olunması gerekir. Eğer soketten gönderilen iletilerde bir tamsayı veriyi kodluyorsanız, bunu da ağ bayt sırasına göre kodlamalısınız. Eğer bunu yapmıyorsanız, yazılımınız çalışırken veya diğer makinalarla konuşurken çökebilir.

Eğer port numarası ve konak adresini almak için **getservbyname**, **gethostbyname** veya **inet_addr** kullanıyorsanız, değerler zaten ağ bayt sırasındadır ve bunları doğrudan **sockaddr_in** yapısına kopyalayabilirsiniz.

Aksi takdirde, değerleri kendiniz dönüştürmek zorunda kalırsınız. **sin_port** üyesine atanacak değerleri dönüştürmek için **htons** ve **ntohs** kullanınız. **sin_addr** üyesine atanacak IPv4 adreslerini dönüştürmek için de **htonl** ve **ntohl** kullanınız. (**struct in_addr** ile **uint32_t** veri türlerinin bir diğerine eşdeğer olduğunu hatırlayalım.) Bu işlevler `netinet/in.h` başlık dosyası içinde tanımlıdır.

`uint16_t` **htons**`(uint16_t` *short_konak*`)` işlev

Bu işlev **uint16_t** türündeki *short_konak* tamsayısını konak bayt sırasından ağ bayt sırasına dönüştürür.

`uint16_t` **ntohs**`uint16_t` *short_ağ*`)` işlev

Bu işlev **uint16_t** türündeki *short_ağ* tamsayısını ağ bayt sırasından konak bayt sırasına dönüştürür.

`uint32_t` **htonl**`(uint32_t` *long_konak*`)` işlev

Bu işlev **uint32_t** türündeki *long_konak* tamsayısını konak bayt sırasından ağ bayt sırasına dönüştürür.

Bu IPv4 İnternet adresleri için kullanılır.

`uint32_t` **ntohl**`(uint32_t` *long_ağ*`)` işlev

Bu işlev **uint32_t** türündeki *long_ağ* tamsayısını ağ bayt sırasından konak bayt sırasına dönüştürür.

Bu IPv4 İnternet adresleri için kullanılır.

6.6. Protokol Veritabanı

Soketlerle kullanılan iletişim protokolleri verinin nasıl değiş/tokuş edildiği konusundaki alt seviye ayrıntılarla ilgilidir. Örneğin, protokol veri aktarımındaki ve ileti yönlendirme komutlarındaki hataları bulmak için sağlama toplamı (checksum) uygular. Normal kullanıcı yazılımlarının bu ayrıntılarla ilgilenmeleri gerekmez.

İnternet isim alanındaki öntanımlı iletişim protokolü iletişim tarzına bağlıdır. Akım (stream) iletişimi için öntanımlı iletişim tarzı TCP'dir (Transmission Control Protocol – Denetimli Aktarım Protokolü). Datagram iletişimi için öntanımlı iletişim tarzı UDP'dir (User Datagram Protocol – Kullanıcı Datagram Protokolü). Güvenilir datagram iletişimi için için öntanımlı iletişim tarzı RDP'dir (Reliable Datagram Protocol – Güvenilir Datagram Protokolü). Sizin hemen hemen her zaman öntanımlı iletişim tarzını kullanmanız gerekir.

İnternet protokolleri genelde numara yerine isimle belirtilirler. Bir konak tarafından bilinen ağ protokolleri bir veritabanında saklanmaktadır. Bu genellikle ya `/etc/protocols` dosyasından veya bir isim sunucusunun

sunduğu eşdeğerinden elde edilir. Protokol numarası ile ilişkili bir protokol ismini veritabanında aramak için **getprotobyname** işlevini kullanabilirsiniz.

Burada protokol veritabanına erişim için gerekli araçların ayrıntılı açıklamasını bulabilirsiniz. Bunlar`netdb.h` başlık dosyası içerisinde tanımlıdır.

`struct` **protoent** veri türü

Bu veri türü ağ protokolleri veritabanındaki girdileri göstermek için kullanılır. Aşağıdaki üyelere sahiptir:

`char *`**p_name**
Bu protokolün resmi adıdır.

`char **`**p_aliases**
Bunlar protokolün diğer adlarıdır, bir dizgeler dizisi olarak belirtilmişlerdir. Dizinin son elemanı bir boş göstericidir.

`int` **p_proto**
Bu protokol numarasıdır (konak bayt sırasında); bu üyeyi **socket** (sayfa: 420) işlevinin *protokol* argümanında kullanın.

Belirli bir protokolü protokol veritabanında aramak için **getprotobyname** ve **getprotobynumber** işlevlerini kullanabilirsiniz. Bilgi durağan olarak ayrılmış bir yapı içinde döner; eğer çağrılar arasında bilgiyi kullanmak isterseniz kopyalamalısınız.

`struct protoent *`**getprotobyname**`(const char *`*isim*`)` işlev

getprotobyname işlevi *isim* isimli ağ protokolü hakkında bilgi döndürür. Eğer böyle bir protokol yoksa, boş gösterici döndürür.

`struct protoent *`**getprotobynumber**`(int` *protocol*`)` işlev

getprotobynumber işlevi *protocol* numaralı ağ protokolü hakkında bilgi döndürür. Eğer böyle bir protokol yoksa, boş gösterici döndürür.

setprotoent, **getprotoent** ve **endprotoent** işlevlerini kullanarak bütün protokol veritabanını bir protokol için tarayabilirsiniz. Bu işlevler evresel (reentrant) işlevler olmadığı için kullanırken dikkatli olunuz.

`void` **setprotoent**`(int` *açıkkal*`)` işlev

Bu işlev protokol veritabanını taramaya başlamak için açar.

Eğer *açıkkal* argümanı sıfır değise, bu bir bayrağı kaldırarak **getprotobyname** veya **getprotobynumber** işlevlerine yapılan çağrılarda veritabanının kapanmamasını sağlar (normalde olması gerektiği gibi). Bu yaklaşım işlevlerin sık çağırılması durumunda veritabanının her çağrıda tekrardan açılmasından kurtararak verimi artırır.

`struct protoent *`**getprotoent**`(void)` işlev

Bu işlev protokol veritabanında sıradaki girdiyi döndürür. Başka girdi yoksa boş gösterici döndürür.

`void` **endprotoent**`(void)` işlev

Bu işlev protokol veritabanını kapatır.

6.7. İnternet Soketi Örneği

Burada İnternet isim alanında bir soketin nasıl oluşturulacağını ve isimlendirileceğini gösteren bir örnek görüyoruz. Yeni oluşturulan soket yazılımın çalıştığı makinada bulunur. Makinanın İnternet adresini bulup kullanmak yerine, bu örnekte konak adres, yerine **INADDR_ANY** kullanılmıştır; sistem bunu makinanın gerçek adresi ile değiştirir.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

int
make_socket (uint16_t port)
{
  int sock;
  struct sockaddr_in name;

  /* Soketi oluşturalım. */
  sock = socket (PF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    {
      perror ("socket");
      exit (EXIT_FAILURE);
    }

  /* Sokete bir isim verelim. */
  name.sin_family = AF_INET;
  name.sin_port = htons (port);
  name.sin_addr.s_addr = htonl (INADDR_ANY);
  if (bind (sock, (struct sockaddr *) &name, sizeof (name)) < 0)
    {
      perror ("bind");
      exit (EXIT_FAILURE);
    }

  return sock;
}
```

Burada konak ismi ve port numarası verildiğinde **sockaddr_in** yapısını nasıl dolduracağınızı gösteren bir örnek görüyoruz:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void
init_sockaddr (struct sockaddr_in *isim,
               const char *konakismi,
               uint16_t port)
{
  struct hostent *konakbilgisi;

  isim->sin_family = AF_INET;
  isim->sin_port = htons (port);
```

```
  konakbilgisi = gethostbyname (konakismi);
  if (konakbilgisi == NULL)
    {
      fprintf (stderr, "%s diye bir konak yok.\n", konakismi);
      exit (EXIT_FAILURE);
    }
  isim->sin_addr = *(struct in_addr *) konakbilgisi->h_addr;
}
```

7. Diğer İsim Alanları

Diğer bazı isim alanları ve ilişkili protokol aileleri de desteklenmektedir fakat henüz belgelendirilmemiştir, çünkü sık kullanılmamaktadır. **PF_NS** Xerox Ağ Yazılımı (Xerox Network Software) protokollerini belirtir. **PF_ISO** Open Systems Interconnect'i belirtir. **PF_CCITT**, CCITT protokollerini belirtir. `socket.h` bu sembolleri ve henüz gerçeklenmemiş diğerlerinin isimlendirme protokollerini tanımlar.

PF_IMPLINK konaklar ve İnternet İleti İşlemcileri arası iletişim için kullanılır. Bunun ve sıklıkla kullanılan yerel ağ yönlendirme protokolü **PF_ROUTE** hakkında bilgi için GNU Hurd Kılavuzuna bakınız (Bir gün olacak İnşallah...).

8. Soketlerin Açılması ve Kapatılması

Bu bölüm soket açma ve kapatma için kullanılan asıl kütüphane işlevlerini anlatır. Aynı işlevler tüm isim alanları ve bağlantı tarzları için çalışır.

8.1. Bir Soketin Oluşturulması

Soket oluşturmanın en ilkel yöntemi `sys/socket.h` içerisinde tanımlı **socket** işlevini kullanmaktır.

```
int socket(int isimalanı,                                                    işlev
           int tarz,
           int protokol)
```

Bı işlev bir soket oluşturur. *tarz* iletişim tarzını belirler, ki bu *İletişim Tarzları* (sayfa: 400) bölümünde listelenen soket tarzlarından biri olmalıdır. *isimalanı* argümanı isim alanını belirtir; **PF_LOCAL** (Bkz. *Yerel İsim Alanı* (sayfa: 404)) veya **PF_INET** (Bkz. *İnternet İsim Alanı* (sayfa: 406)) olmak zorundadır. *protokol* belirli bir protokolü gösterir (Bkz. *Soket Kavramları* (sayfa: 399)); *protokol* için sıfır genellikle doğru değerdir.

socket işlevinin dönüş değeri yeni soket için bir dosya tanımlayıcıdır. Hata halinde **-1** değeri döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EPROTONOSUPPORT`
: *protokol* veya *tarz* belirtilen *isimalanı* tarafından desteklenmiyor.

`EMFILE`
: Süreç zaten çok sayıda açık dosya tanımlayıcısına sahip.

`ENFILE`
: Sistem zaten çok sayıda açık dosya tanımlayıcısına sahip.

`EACCES`
: Süreç belirtilen *tarz* ya da *protokol* ile soket açma yetkisine sahip değil.

`ENOBUFS`
: Sistem dahili tampon alanını tüketti.

socket işlevinin döndürdüğü dosya tanımlayıcı hem okuma hem de yazma işlemlerini desteklemektedir. Fakat, borular gibi, soketler de dosya içi konumlama işlemlerini desteklememektedir.

socket işlevinin kullanımına ilişkin örnekler için *Soketlerde Yerel İsim Alanı Örneği* (sayfa: 406) ya da *İnternet Soketi Örneği* (sayfa: 419) bölümüne bakınız.

8.2. Bir Soketin Kapatılması

Soketin kullanımı sona erdiğinde, basitçe onun dosya tanımlayıcısını **close** ile kapatabilirsiniz; bkz. *Dosyaların Açılması ve Kapatılması* (sayfa: 306). Bağlantı üzerinde hala aktarılmayı bekleyen veri varsa, normalde **close** aktarımın tamamlanmasına çalışır. Bu davranışı **SO_LINGER** soket seçeneğini bir zamanaşımı değeri belirtmek için kullanarak kontrol edebilirsiniz; bkz. *Socket Seçenekleri* (sayfa: 438).

Ayrıca, sadece aktarım veya alımı durdurmak isterseniz, `sys/socket.h` içerisinde tanımlı **shutdown** işlevini çağırabilirsiniz.

```
int shutdown(int soket,                                                    işlev
             int nasıl)
```

shutdown işlevi *soket* soketinin bağlantısını kapatır. *nasıl* argümanı nasıl bir eylem yapılacağını belirler:

`0`
: Bu soketin veri alımını durdur. Eğer veri hala geliyorsa, reddedilir.

`1`
: Bu soketten veri gönderimini durdur. Aktarım için bekleyen veri iptal edilir. Gönderilmiş veri için ulaştı bilgisi beklenmez, eğer veri kaybolduysa tekrar gönderilmez.

`2`
: Hem alımı hem de gönderimi durdur.

Başarı halinde dönüş değeri **0**, başarısızlıkta **-1**'dir. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *soket* geçerli bir dosya tanımlayıcı değil.

ENOTSOCK
: *soket* bir soket değil.

ENOTCONN
: *soket* bağlı değil.

8.3. Soket Çiftleri

Bir ***soket çifti*** bir çift bağlı (ancak isimsiz) soketten oluşur. Bu yapı boruya çok benzer ve çoğunlukla da böyle kullanılırlar. Soket çifti `sys/socket.h` başlık dosyası içerisinde tanımlı **socketpair** işleviyle oluşturulur. Soket çifti bir boruya çok benzer; ana fark soket çifti çift yönlü oluduğu halde; borunun bir sadece–girdi ve bir sadece–çıktı ucu olmasıdır (Bkz. *Borular ve FIFOlar* (sayfa: 393)).

```
int socketpair(int isimalanı,                                              işlev
               int tarz,
               int protokol,
               int dosya_tanımlayıcı[2])
```

Bu işlev *dosya_tanımlayıcı*`[0]` ve *dosya_tanımlayıcı*`[1]` içine dosya tanımlayıcılarını yerleştirerek bir soket çifti oluşturur. Soket çifti aynı anda iki yönlü (full–duplex) iletişim yapılabilen bir iletişim kanalıdır, böylece hem okuma hem yazma her iki ucundan da gerçekleşebilmektedir.

isimalanı, *tarz* ve *protokol* argümanları **socket** işlevindeki gibi kullanılmaktadır. *tarz İletişim Tarzları* (sayfa: 400) bölümünde listelenen iletişim tarzalarından biri olmalıdır. *isimalanı* argümanı isim alanını belirler ve **AF_LOCAL** (sayfa: 404) olmalıdır; *protokol* iletişim protokolünü belirler, fakat tek anlamlı değer sıfırdır.

Eğer *tarz* bağlantısız bir iletişim tarzını belirtiyorsa, elde ettiğiniz iki soket bağlı değildir, fakat her ikisi de birbirini öntanımlı hedef adres olarak bilir, böylece birbirlerine paket gönderebilirler.

socketpair işlevi başarı durumunda **0**, başarısızlıkta **-1** döndürür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EMFILE`
Süreç çok fazla açık dosya tanımlayıcısına sahip.

`EAFNOSUPPORT`
Belirtilen isim alanı desteklenmiyor.

`EPROTONOSUPPORT`
Belirtilen protokol desteklenmiyor.

`EOPNOTSUPP`
Belirtilen protokol soket çifti oluşturmayı desteklemiyor.

9. Soketlerin Bağlantılarla Kullanılması

En sık kullanılan iletişim tarzları belirli bir sokete bağlantı yapımını ve onunla veri değişimini içerir. Bağlantı yapılması bakışımsızdır; bir taraf (***istemci***) bağlantı isteğinde bulunurken diğer taraf (***sunucu***) soket oluşturur ve bağlantı isteği için bekler.

9.1. Bir Bağlantının Oluşturulması

Bir bağlantının kurulmasında, sunucu bağlantı için beklerken istemci bağlanır ve sunucu kabul eder. Burada bizim tartıştığımız şey istemci yazılımın `sys/socket.h` içinde tanımlı **connect** işleviyle ne yapması gerektiğidir.

```
int connect(int                soket,                                  işlev
            struct sockaddr *adres,
            socklen_t          uzunluk)
```

connect işlevi *soket* dosya tanımlayıcısına sahip soketten *adres* ve *uzunluk* argümanları ile adresi belirtilen sokete bir bağlantı başlatır. (Bu soket tipik olarak başka makinadadır ve bir sunucu olarak kurulmuş olması gerekir.) Bu argümanların nasıl yorumlandıkları konusunda bilgi için *Soket Adresleri* (sayfa: 401) bölümüne bakınız.

Normalde, **connect** sunucu isteğe yanıt verene kadar bekler. **connect** işlevinin yanıt beklemeden hemen dönmesini sağlamak için *soket* soketinde *bloklanmayan kipi* (sayfa: 341) seçebilirsiniz.

connect işlevi başarı durumunda **0**, başarısızlıkta **-1** döndürür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EBADF`

soket soketi geçerli bir dosya tanımlayıcı değil.

`ENOTSOCK`
soket dosya tanımlayıcısı bir soket değil.

`EADDRNOTAVAIL`
Belirtilen adres uzaktaki makinada yok.

`EAFNOSUPPORT`
adres'in isim alanı bu soket tarafından desteklenmiyor.

`EISCONN`
soket soketi zaten bağlı.

`ETIMEDOUT`
Bağlantı girişimi zaman aşımına uğradı.

`ECONNREFUSED`
Sunucu bağlantı isteğini açıkça reddetti.

`ENETUNREACH`
adres ile belirtilen ağa bu konak erişemez.

`EADDRINUSE`
adres ile belirtilen soket adresi zaten kullanımda.

`EINPROGRESS`
*soket*soketi baskılanamayan kipte olduğundan bağlantı hemen kurulamıyor. **`select`** ile bağlantının tam olarak ne zaman kurulabileceğini tespit edebilirsiniz; Bkz. *Girdi ve Çıktının Beklenmesi* (sayfa: 323). Bağlantı tam kurulamadan aynı sokete tekrar bir **`connect`** çağrısı yapılırsa, çağrı **`EALREADY`** hatası ile sonlanır.

`EALREADY`
soket soketi baskılanamayan kipte ve askıdaki bağlantısı devam ediyor (üstteki **`EINPROGRESS`**'a bakınız).

Bu işlev çok evreli yazılımlar için iptal noktası olarak tanımlanmıştır, bu nedenle ayrılan özkaynakların (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) evre iptal edilse dahi serbest bırakılmasının sağlanması şarttır.

9.2. Bağlantıların Dinlenmesi

Şimdi sunucu sürecin soket üzerinden bağlantıları kabul etmek için ne yapması gerektiğini inceleyelim. Öncelikle sokete gelen bağlantı isteklerini alabilmesi için **`isten`** işlevini kullanması gerekli, ardında da gelen her bağlantıyı kabul etmek için **`accept`** işlevini çağırmalıdır(Bkz. *Bağlantıların Kabul Edilmesi* (sayfa: 424)). Sunucu soketi üzerinde bağlantı isteği olduğunda, **`select`** işlevi soketin bağlantı kabul etmek için ne zaman hazır olacağını bildirir (Bkz. *Girdi ve Çıktının Beklenmesi* (sayfa: 323)).

`listen` işlevi bağlantısız iletişim tarzı kullanılan soketler için kullanılamaz.

Bir bağlantı isteği gelene kadar çalışmaya başlamayan ağ sunucusu bile yazabilirsiniz. Bkz. *inetd Sunucuları* (sayfa: 437).

Internet isim alanında, port erişimini kontrol için özel bir koruma mekanizması yoktur, herhangi bir makinadaki bir süreç sunucunuza bağlanabilir. Sunucunuza erişimi kısıtlamak istiyorsanız ya sunucunuzun bağlantı isteğinde bulunan adresi incelemesini sağlayın ya da başka bir uzlaşma veya kimlik doğrulama protokolü uygulayın.

Yerel isim alanında ise, bildiğimiz dosya koruma bitleri sokete bağlanmak için kimin erişim hakkı var kontrol eder.

```
int listen(int           soket,                                    işlev
           unsigned int n)
```

listen işlevi bağlantıları kabul edecek *soket* soketini etkinleştirir, böylece sunucu soketi olur.

n argümanı bekleyen bağlantılar için kuyruk uzunluğunu belirler. Kuyruk dolunca, bağlanmak için teşebbüs eden yeni istemciler, sunucu kuyrukta bekleyen bir bağlantı için **accept** işlevini çağırana kadar **ECONNREFUSED** hatası ile sonlanırlar.

listen işlevi başarı durumunda **0**, başarısızlıkta **-1** döndürür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *soket* soketi geçerli bir dosya tanımlayıcı değil.

ENOTSOCK
: *soket* dosya tanımlayıcısı bir soket değil.

EOPNOTSUPP
: *soket* soketi bu işlemi desteklemiyor.

9.3. Bağlantıların Kabul Edilmesi

Bir sunucu bağlantı isteği aldığında, bağlantıyı isteği kabul ederek tamamlayabilir. Bunun için **accept** işlevi kullanılır.

Sunucu olarak kurulmuş olan bir soket çok sayıda istemciden gelen bağlantı isteklerini kabul edebilir. Sunucunun dinlediği soket bağlantının bir parçası olmaz; bunun yerine, **accept** bağlantıyla ilgil olarak yeni bir soket oluşturur. **accept** bu yeni soketin dosya tanımlayıcısını döndürür. Sunucu soketi diğer bağlantı isteklerini dinlemek için kalır.

Sunucu soketindeki bekleyen bağlantı isteği sayısı sınırlıdır. İstemcilerden gelen bağlantı istekleri sunucunun karşılık verebileceğinden hızlı gelirse, kuyruk dolabilir ve ilave istekler **ECONNREFUSED** hatası ile reddedilebilir. Azami kuyruk uzunluğunu **listen** işlevinin bir argümanı olarak belirtebilirsiniz; yine de sistem kuyruk uzunluğu için kendi iç sınırını baskın kılabilir.

```
int accept(int               soket,                                 işlev
           struct sockaddr *adres,
           socklen_t       *uzunluk_gstr)
```

Bu işlev *soket* sunucu soketindeki bir bağlantı isteğini kabul etmek için kullanılır.

accept işlevi, *soket* soketi için baskılanamayan kip seçilmediği sürece, süren bir bağlantı yoksa bekler. (baskılanamayan soketlerde de **select** işlevini kullanarak süren bağlantıların bitmesini beklemek mümkündür.) *Dosya Durum Seçenekleri* (sayfa: 341), baskılanamayan kip hakkında bilgi içerir.

adres ve *uzunluk_gstr* argümanları bağlantıyı başlatan istemci soketinin ismi hakkında bilgi döndürür. Bilginin biçimi hakkında bilgi edinmek için *Soket Adresleri* (sayfa: 401) bölümüne bakınız.

Bağlantının kabul edilmesi ile bağlantı *soket* soketinden yapılmaz. Bunun yerine, bağlantı yapılacak yeni bir soket oluşturulur. **accept** işlevinin normal dönüş değeri yeni soketin dosya tanımlayıcısıdır.

accept işlevinin ardından, *soket* soketi bağlantısız ve açık olarak kalır ve kapatılıncaya kadar dinlemeye devam eder. *soket* ile accept işlevini tekrar çağırarak başka bağlantılar kabul edebilirsiniz.

Hata oluşursa accept **-1** döndürür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *soket* soketi geçerli bir dosya tanımlayıcı değil.

ENOTSOCK
: *soket* dosya tanımlayıcısı bir soket değil.

EOPNOTSUPP
: *soket* dosya tanımlayıcısı bu işlemi desteklemiyor.

EWOULDBLOCK
: *soket* soketi baskılanamayan kipte ve bekleyen bir bağlantı yok.

Bu işlev çok evreli yazılımlar için iptal noktası olarak tanımlanmıştır, bu nedenle ayrılan özkaynakların (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) evre iptal edilse dahi serbest bırakılmasının sağlanması şarttır.

accept işlevi bağlantısız iletişim tarzlarını kullanan soketler için kullanılamaz.

9.4. Bana Kim Bağlı?

```
int getpeername(int                soket,                                  işlev
                struct sockaddr *adres,
                socklen_t       *uzunluk_gstr)
```

getpeername işlevi *soket*'in bağlı olduğu soket adresini döndürür; adresi *adres* ve *uzunluk_gstr* ile belirtilen bellek alanında saklar. Adresin uzunluğunu **uzunluk_gstr* içinde saklar.

Adresin biçimi hakkında bilgi edinmek için *Soket Adresleri* (sayfa: 401) bölümüne bakınız. Bazı işletim sistemlerinde, **getpeername** sadece İnternet etki alanında çalışır.

getpeername işlevi başarı durumunda **0**, başarısızlıkta **-1** döndürür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *soket* soketi geçerli bir dosya tanımlayıcı değil.

ENOTSOCK
: *soket* dosya tanımlayıcısı bir soket değil.

ENOTCONN
: *soket* soketi bağlı değil.

ENOBUFS
: Dahili tamponlar yeterli değil.

9.5. Veri Aktarımı

Soket bir kez karşıya bağlandığında, veri aktarımı için sıradan ***okuma*** ve ***yazma*** işlemleri (*Girdi ve Çıktı İlkelleri* (sayfa: 308)) yapabilirsiniz. Bir soket iki yönlü haberleşme kanalıdır, böylece okuma ve yazma işlemleri her iki uçta da gerçekleştirilebilir.

Soket işlemlerine özgü bazı G/Ç kipleri de vardır. Bu kipleri belirtmek için, **recv** ve **send** işlevlerini daha genel olan **read** ve **write** işlevleri yerine kullanmanız gerekir. **recv** ve **send**işlevleri, özel G/Ç kiplerini kontrol etmek için çeşitli bayrakları belirtebileceğiniz, ek bir argüman alırlar. Örneğin, **MSG_OOB** bayrağını belirterek

sırasız veri okuyup yazabilirsiniz, **MSG_PEEK** bayrağını girdiyi gözetlemek için, **MSG_DONTROUTE** bayrağını yönlendirme bilgisinin çıktıda içerilmesini kontrol için kullanabilirsiniz.

9.5.1. Veri Gönderimi

send işlevi sys/socket.h başlık dosyası içinde tanımlıdır. Eğer *bayraklar* argümanınız sıfır ise, **send** yerine **write** kullanabilirsiniz; bkz. *Girdi ve Çıktı İlkelleri* (sayfa: 308). Eğer soket bağlıyken bağlantısı koptuysa, **send** veya **write**'ın her hangi bir kullanımı için **SIGPIPE** (sayfa: 610) sinyalini alırsınız.

```
int send(int     soket,                                                    işlev
         void   *tampon,
         size_t  boyut,
         int     bayraklar)
```

send işlevi **write** (sayfa: 310) gibidir, ancak fazladan *bayraklar* argümanına sahiptir. Olası bayrak değerleri *Soket Verisi Seçenekleri* (sayfa: 427) bölümünde anlatılmıştır.

Bu işlev aktarılan bayt miktarı ile veya hata durumunda **-1** ile döner. Soket baskılanamayan kipteyse **send** (**write** gibi) verinin henüz bir kısmını gönderdikten sonra dönebilir. Baskılanamayan kip hakkında daha fazla bilgi için *Dosya Durum Seçenekleri* (sayfa: 341) bölümüne bakınız.

Unutmayalım ki, başarılı bir dönüş değeri her ne kadar verinin hatasız bir şekilde gönderildiğini belirtse de, hatasız bir şekilde alındığını belirtmez.

Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *soket* soketi geçerli bir dosya tanımlayıcı değil.

EINTR
: Veri gönderilmeden önce işlem bir sinyal tarafından kesildi. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

ENOTSOCK
: *soket* dosya tanımlayıcısı bir soket değil.

EMSGSIZE
: Soket türü verinin bütün olarak gönderilmesini gerektiriyor, fakat veri bunun olması için çok büyük.

EWOULDBLOCK
: Soket baskılanamayan kipte ve yazma işlemi soketi baskılar. (Normalde **send** işlem bitinceye kadar soketi baskılar.)

ENOBUFS
: Yeterli dahili tampon alanı yok.

ENOTCONN
: Bu sokete hiç bağlanmadınız.

EPIPE
: Bu soket bağlıydı ancak bağlantı koptu. Bu durumda, **send** öncelikle bir **SIGPIPE** sinyali üretir; sinyal ihmal edilir veya baskılanırsa ya da bu sinyalin yakalayıcısı dönerse **send** işlevi **EPIPE** hatası ile sonlanır.

Bu işlev çok evreli yazılımlar için iptal noktası olarak tanımlanmıştır, bu nedenle ayrılan özkaynakların (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) evre iptal edilse dahi serbest bırakılmasının sağlanması şarttır.

9.5.2. Veri Alımı

recv işlevi sys/socket.h başlık dosyası içinde tanımlıdır. Eğer *bayraklar* argümanlarınız sıfırsa, **recv** yerine **read** kullanabilirsiniz; bkz. *Girdi ve Çıktı İlkelleri* (sayfa: 308).

```
int recv(int     soket,                                                    işlev
         void   *tampon,
         size_t  boyut,
         int     bayraklar)
```

recv işlevi **read** (sayfa: 308) gibidir, ancak fazladan *bayraklar* argümanına sahiptir. Olası bayrak değerleri *Soket Verisi Seçenekleri* (sayfa: 427) bölümünde anlatılmıştır.

Eğer *soket* için baskılanamayan kip seçildiyse ve okunacak veri yoksa, **recv** beklemeden hemen sonlanır. Baskılanamayan kip hakkında daha fazla bilgi için *Dosya Durum Seçenekleri* (sayfa: 341) bölümüne bakınız.

Bu işlev aktarılan bayt miktarı ile veya hata durumunda **-1** ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *soket* soketi geçerli bir dosya tanımlayıcı değil.

ENOTSOCK
: *soket* dosya tanımlayıcısı bir soket değil.

EWOULDBLOCK
: Soket baskılanamayan kipte ve okuma işlemi soketi baskılar. (Normalde **recv** okunacak girdi oluncaya kadar soketi baskılar.)

EINTR
: Veri okunmadan önce işlem bir sinyal tarafından kesildi. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

ENOTCONN
: Bu sokete hiç bağlanmadınız.

Bu işlev çok evreli yazılımlar için iptal noktası olarak tanımlanmıştır, bu nedenle ayrılan özkaynakların (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) evre iptal edilse dahi serbest bırakılmasının sağlanması şarttır.

9.5.3. Soket Verisi Seçenekleri

send ve **recv** işlevlerindeki *bayraklar* argümanı bir bit maskesidir. Aşağıdaki makroların değerlerini bit bit VEYAlayarak bu argüman için değer elde edebilirsiniz. Hepsi sys/socket.h başlık dosyası içinde tanımlıdır.

```
int MSG_OOB                                                               makro
```

Sırasız veri gönderilir ve alınır; bkz. *Bantdışı Veri Aktarımı* (sayfa: 431).

```
int MSG_PEEK                                                              makro
```

Veriye bakılır ancak girdi kuyruğundan çıkarılmaz. Bu sadece **recv** gibi girdi işlevleri için anlamlıdır (**send** ile anlamlı değildir).

`int` **`MSG_DONTROUTE`** makro

Yönlendirme bilgisi iletinin içinde bulunmaz. Bu sadece çıktı işlevlerinde anlamlıdır ve genellikle sadece tanı veya yönlendirme amaçlı yazılımlarda kullanılır. Bunu burada anlatmaya çalışmayacağız.

9.6. Bayt Akımlı Soket Örneği

Burada Internet isim alanında bayt akımlı bir soket için bağlantı yapan örnek bir istemci yazılım görüyoruz; sunucuya bağlandıktan sonra sunucuya sadece bir dizge gönderip çıkmaktadır.

Bu yazılım soket adresini ayarlamak için **`init_sockaddr`** kullanmaktadır; bkz. *İnternet Soketi Örneği* (sayfa: 419).

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT            5555
#define MESSAGE         "Alooo!! Hala egleniyor musun?!?"
#define SERVERHOST      "mescaline.gnu.org"

void
write_to_server (int filedes)
{
  int nbytes;

  nbytes = write (filedes, MESSAGE, strlen (MESSAGE) + 1);
  if (nbytes < 0)
    {
      perror ("write");
      exit (EXIT_FAILURE);
    }
}


int
main (void)
{
  extern void init_sockaddr (struct sockaddr_in *name,
                             const char *hostname,
                             uint16_t port);
  int sock;
  struct sockaddr_in servername;

  /* Soketi oluşturalım. */
  sock = socket (PF_INET, SOCK_STREAM, 0);
  if (sock < 0)
    {
      perror ("socket (client)");
      exit (EXIT_FAILURE);
    }

  /* Sunucuya bağlanalım. */
```

```
  init_sockaddr (&servername, SERVERHOST, PORT);
  if (0 > connect (sock,
                   (struct sockaddr *) &servername,
                   sizeof (servername)))
    {
      perror ("connect (client)");
      exit (EXIT_FAILURE);
    }

  /* Sunucuya veriyi gönderelim. */
  write_to_server (sock);
  close (sock);
  exit (EXIT_SUCCESS);
}
```

9.7. Bayt Akımlı Bağlantı Sunucusu Örneği

Sunucu tarafı daha karmaşıktır. Aynı anda ço sayıda istemcinin sunucuya bağlı kalmasını istediğimiz için, basitçe **read** veya **recv** işlevini çağırarak tek bir istemciden girdi beklemek doğru olmaz. Yapılması gereken şey **select** işlevini (*Girdi ve Çıktının Beklenmesi* (sayfa: 323)) kullanarak açık olan bütün soketlerden girdi beklemektir. Bu sunucuya ilave bağlantı istekleriyle ilgilenme imkanı da verir.

Bu sunucu istemciden bir ileti aldığında ilgi çekecek hiç bir şey yapmaz. Dosya sonu durumunu algıladığında o istemci için soketi kapatır (bu aynı zamanda istemcinin de soketi kapatmasına neden olur).

Bu yazılım soket adresini ayarlamak için **make_socket**'i kullanır; bkz. *İnternet Soketi Örneği* (sayfa: 419).

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT    5555
#define MAXMSG  512

int
read_from_client (int filedes)
{
  char buffer[MAXMSG];
  int nbytes;

  nbytes = read (filedes, buffer, MAXMSG);
  if (nbytes < 0)
    {
      /* Okuma hatası. */
      perror ("read");
      exit (EXIT_FAILURE);
    }
  else if (nbytes == 0)
    /* Sosya sonu. */
    return -1;
  else
    {
```

```
      /* Veri okundu. */
      fprintf (stderr, "Sunucu: gelen ileti: `%s'\n", buffer);
      return 0;
    }
}

int
main (void)
{
  extern int make_socket (uint16_t port);
  int sock;
  fd_set active_fd_set, read_fd_set;
  int i;
  struct sockaddr_in clientname;
  size_t size;

  /* Soketi oluşturalım ve bağlantıları kabul etmesi için ayarlayalım. */
  sock = make_socket (PORT);
  if (listen (sock, 1) < 0)
    {
      perror ("listen");
      exit (EXIT_FAILURE);
    }

  /* Etkin soketleri ilklendirelim. */
  FD_ZERO (&active_fd_set);
  FD_SET (sock, &active_fd_set);

  while (1)
    {
      /* Etkin soketlerden bilgi gelene kadar baskılayalım. */
      read_fd_set = active_fd_set;
      if (select (FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0)
        {
          perror ("select");
          exit (EXIT_FAILURE);
        }

      /* Girdi bekleyerek bütün soketleri hizmete sokalım. */
      for (i = 0; i < FD_SETSIZE; ++i)
        if (FD_ISSET (i, &read_fd_set))
          {
            if (i == sock)
              {
                /* Dinlenen sokette bağlantı isteği var. */
                int new;
                size = sizeof (clientname);
                new = accept (sock,
                              (struct sockaddr *) &clientname,
                              &size);
                if (new < 0)
                  {
                    perror ("accept");
                    exit (EXIT_FAILURE);
                  }
                fprintf (stderr,
                         "Sunucu: Konak %s, %hd. portundan baglaniyor\n",
```

```
                          inet_ntoa (clientname.sin_addr),
                          ntohs (clientname.sin_port));
                FD_SET (new, &active_fd_set);
              }
            else
              {
                /* Bağlı olan sokete veri ulaşıyor. */
                if (read_from_client (i) < 0)
                  {
                    close (i);
                    FD_CLR (i, &active_fd_set);
                  }
              }
          }
      }
}
```

9.8. Bantdışı Veri Aktarımı

Bağlantılı akımlar, sıradan veriye nazaran teslim edilme önceliğine sahip ***bantdışı veri*** aktarımına izin vermektedir. Bantdışı veri gönderiminin asıl kullanılma nedeni, özel durumlarda uyarı gönderme ihtiyacıdır. Bantdışı veri göndermek için **send** işlevini **MSG_OOB** bayrağıyla kullanınız (Bkz. *Veri Gönderimi* (sayfa: 426)).

Bantdışı veriler yüksek öncelikle alınırlar çünkü alıcı işlem onu sırayla almak zorunda değildir; bir sonraki bantdışı veriyi okumak için, **recv** işlevini **MSG_OOB** bayrağıyla kullanınız (Bkz. *Veri Alımı* (sayfa: 427)). Sıradan okuma işlemleri bantdışı veriyi okumazlar; sadece sıradan veriyi okurlar.

Soket, bantdışı verinin geldiğini görünce, kendi sürecine ya da süreç grubuna **SIGURG** sinyalini gönderir. Soket sahibini **fcntl** işlevinde **F_SETOWN** komutunu kullanarak belirtebilirsiniz; bkz. *Sinyallerle Sürülen Girdi* (sayfa: 349). Bantdışı veriyi okuma gereksinimi gibi bir durumda uygun hareketi yapmak için bu sinyalin yakalayıcısını kurmanız gerekir, bkz. *Sinyal İşleme* (sayfa: 601).

Diğer bir seçenek olarak, sokette özel durum oluşması için bekleyebilen **select** işlevini kullanarak bantdışı veri olana kadar bekleyebilir ya da bekleyen bantdışı veri var mı diye bakabilirsiniz. **select** işlevi hakkında daha fazla bilgi için *Girdi ve Çıktının Beklenmesi* (sayfa: 323) bölümüne bakınız.

Bantdışı verinin bildirilmesi (**SIGURG** veya **select** ile) bantdışı verinin gelmek üzere olduğunu gösterir; veri daha sonra ulaşabilir. Eğer bantdışı veriyi daha gelmeden okumaya çalışırsanız, **recv** işlevi **EWOULDBLOCK** hatası ile sonlanır.

Bantdışı veri gönderilince, akımdaki sıradan veri otomatik olarak "im"lenir, ki bu da bant–dışı verinin "ne durumda olabileceğini" gösterir. Bu bantdışı verinin anlamı "şimdiye kadar gönderdiklerimi iptal et" ise kullanışlıdır. Buradaki alıcı işlemde, imlenmeden önce sıradan verinin gönderilip gönderilmediğini sınayabilirsiniz:

```
success = ioctl (socket, SIOCATMARK, &imgeldi);
```

Eğer soketin okuma göstericisine "im" ulaştıysa bir tamsayı değişken olan *imgeldi* sıfır olmayan bir değer yapılır.

Burada bantdışı iminden önce gelen sıradan veriyi iptal eden bir işlev görüyoruz:

```
int
discard_until_mark (int soket)
{
  while (1)
    {
      /* Bu isteğe bağlı bir sınır değildir; herhangi bir büyüklük olabilir.  */
      char tampon[1024];
```

```
      int imgeldi, tamam;

      /* İm geldiyse işlev dönsün.  */
      tamam = ioctl (soket, SIOCATMARK, &imgeldi);
      if (tamam < 0)
        perror ("ioctl");
      if (imgeldi)
        return;

      /* Aksi takdirde, bir miktar sıradan veriyi oku ve iptal et.
         Bu imden sonrasını okumamayı garantiler
         tabii ki imden önce başlıyorsa.  */
      tamam = read (soket, tampon, sizeof(tampon));
      if (tamam < 0)
        perror ("read");
    }
}
```

Eğer imden önceki veriyi iptal etmek istemiyorsanız, belki de bantdışı veriye sistem içi tampon bölgesinde yer açmak için bir kısmını okumak istersiniz. Eğer bantdışı veriyi okumaya çalışır ve **EWOULDBLOCK** hatası alırsanız, bir miktar sıradan veriyi okumaya çalışın (kaydederek daha sonra istediğinizde kullanabilirsiniz) ve yer açıldığını görün. Örnek:

```
struct tampon
{
  char *tmp;
  int boyut;
  struct tampon *sonraki;
};

/* Bantdışı veriyi SOKET'ten oku ve verinin adresini ve
   büyüklüğünü 'struct tampon' içinde döndür.

   Bantdışı veriye yer açmak için bir miktar sıradan veriyi okumak gerekebilir.
   Bu durumda, sıradan veri 'sonraki' alanında bulunan
   bir tamponlar zincirine kaydedilir.  */

struct tampon *
read_oob (int soket)
{
  struct tampon *son = 0;
  struct tampon *liste = 0;

  while (1)
    {
      /* Bu keyfi bir sınırdır.
         Bunu sınırsız yapmayı bilen birileri var mı?  */
#define BOYUT 1024
      char *tamp = (char *) xmalloc (BOYUT);
      int tamam;
      int imgeldi;

      /* Bantdışı veriyi bir daha okumaya çalışalım.  */
      tamam = recv (soket, tamp, BOYUT, MSG_OOB);
      if (tamam >= 0)
        {
          /* Artık elimizde, döndürüyoruz.  */
```

```
          struct tampon *veri
            = (struct buffer *) xmalloc (sizeof (struct tampon));
          veri->tmp = tamp;
          veri->boyut = tamam;
          veri->sonraki = liste;
          return veri;
        }

      /* Yoksa, imin gelip gelmediğine bakalım.  */
      tamam = ioctl (soket, SIOCATMARK, &imgeldi);
      if (tamam < 0)
        perror ("ioctl");
      if (imgeldi)
        {
          /* İm gelmiş; geriye kalan sıradan verinin faydası olmaz.
             Bir süre bekleyelim.   */
          sleep (1);
          continue;
        }

      /* Aksi takdirde, bir miktar sıradan veriyi okuyup kaydedelim.
         Bu imden sonrasını okumamayı garantiler
         tabii ki imden önce başlıyorsa.  */
      tamam = read (soket, tamp, BOYUT);
      if (tamam < 0)
        perror ("read");

      /* Bu veriyi tampon listesine kaydedelim.  */
      {
        struct tampon *veri
          = (struct tampon *) xmalloc (sizeof (struct tampon));
        veri->tmp = tamp;
        veri->boyut = tamam;

        /* Yeni veriyi listenin sonuna ekleyelim.  */
        if (son)
          son->sonraki = veri;
        else
          liste = veri;

        son = veri;
      }
    }
}
```

10. Datagram Soket İşlemleri

Bu bölüm bağlantı kullanmayan iletişim tarzlarının (**SOCK_DGRAM** ve **SOCK_RDM** tarzları) nasıl kullanıldığını anlatır. Bu tarzların kullanımında veri paketler içerisinde gruplanır ve her paket bağımsız birer iletişim sağlar. Her paket için hedef adresi belitmeniz gerekir.

Datagram paketleri mektuplara benzer: her birini bağımsız olarak kendi hedef adresleriyle gönderirsiniz ve onlar yanlış sırayla ulaşabilirler veya ulaşamazlar.

listen ve **accept** işlevleri soketlerde bağlantısız iletişim tarzlarında kullanılamazlar.

10.1. Datagramların Gönderilmesi

Datagram soketinden veri gönderiminin normal yolu `sys/socket.h` içinde tanımlı **sendto** işlevini kullanmaktır.

Bir datagram soketi üzerinde **connect** işlevini çağırabilirsiniz, fakat bu sadece soket üzerinden veri aktarımları için öntanımlı hedefi belirler. Bir soketin bir öntanımlı adresi olduğunda oraya **send** (*Veri Gönderimi* (sayfa: 426)) veya **write** (*Girdi ve Çıktı İlkelleri* (sayfa: 308)) kullanarak paket gönderebilirsiniz. Öntanımlı hedefi, *adres* argümanında **AF_UNSPEC** adres biçimiyle **connect** işlevini çağırarak iptal edebilirsiniz. **connect** işlevi hakkında daha fazla bilgi için *Bir Bağlantının Oluşturulması* (sayfa: 422) bölümüne bakınız.

```
int sendto(int                soket,                                    işlev
           void              *tampon,
           size_t             boyut,
           int                bayraklar,
           struct sockaddr   *adres,
           socklen_t          uzunluk)
```

sendto işlevi *tampon* içindeki veriyi, *soket* soketi üzerinden, *adres* ve *uzunluk* argümanlarıyla belirtilmiş hedef adresine aktarır. *boyut* argümanı aktarılacak bayt sayısını belirtir.

bayraklar argümanı **send** işlevindeki gibi yorumlanır; bkz. *Soket Verisi Seçenekleri* (sayfa: 427).

Dönüş değeri ve hata durumları da **send** işlevindeki gibidir, fakat hataların algılanması ve raporlanması için sisteme güvenemezsiniz; en sık rastlanan hata paket kaybolması veya belirtilen adreste onu alıcının bulunmamasıdır ve makinanızdaki işletim sisteminin genelde bundan haberi yoktur.

sendto işlevinden bir önceki çağrıya ilişkin sorunları raporlaması istenebilir.

Bu işlev çok evreli yazılımlar için iptal noktası olarak tanımlanmıştır, bu nedenle ayrılan özkaynakların (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) evre iptal edilse dahi serbest bırakılmasının sağlanması şarttır.

10.2. Datagramların Alınması

recvfrom işlevi datagram soketinden bir paket okur ve ek olarak nereden gönderilmiş olduğunu söyler. Bu işlev `sys/socket.h` içinde tanımlıdır.

```
int recvfrom(int                soket,                                  işlev
             void              *tampon,
             size_t             boyut,
             int                bayraklar,
             struct sockaddr   *adres,
             socklen_t         *uzunluk_gstr)
```

recvfrom işlevi *soket* soketinden bir paketi *tampon* alanına okur. *boyut* argümanı okunacak azami bayt sayısını belirtir.

Eğer paket *boyut* bayttan uzunsa, paketin ilk *boyut* baytı alınır ve paketin geri kalanı kaybolur. Paketin gerisini okumanın hiç bir yolu yoktur. Bu nedenle, bir paket protokolü kullandığınızda, paketin ne uzunlukta olacağını her zaman bilmeniz gerekir.

adres ve *uzunluk_gstr* argümanları paketin geldiği yerin adresini döndürmek için kullanılır. Bkz. *Soket Adresleri* (sayfa: 401). Yerel etki alanındaki bir soket için adres bilgisi anlamlı değildir, çünkü böyle bir soketin adresini okuyamazsınız (Bkz. *Yerel İsim Alanı* (sayfa: 404)). Bu bilgiyle ilgilenmiyorsanız *adres* argümanına boş gösterici belirtebilirsiniz.

bayraklar argümanı **recv** işlevindeki gibi yorumlanır (bkz. *Soket Verisi Seçenekleri* (sayfa: 427)). Dönüş değeri ve hata durumları da **recv** işlevi ile aynıdır.

Bu işlev çok evreli yazılımlar için iptal noktası olarak tanımlanmıştır, bu nedenle ayrılan özkaynakların (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) evre iptal edilse dahi serbest bırakılmasının sağlanması şarttır.

Paketi kimin gönderdiğini bulmak istemiyorsanız **recvfrom** yerine sadece **recv** işlevini de (bkz. *Veri Alımı* (sayfa: 427)) kullanabilirsiniz (çünkü paketin nereden gelmesi gerektiğini biliyor olabilirsiniz). Hatta *bayraklar* argümanını belirtmek istemezseniz **read** işlevi bile kullanılabilir (bkz. *Girdi ve Çıktı İlkelleri* (sayfa: 308)).

10.3. Datagram Soket Örneği

Burada yerel isim alanındaki datagram akımı üzerinden veri gönderen bir grup örnek bulunmaktadır. Hem istemci hem de sunucu yazılımları *Soketlerde Yerel İsim Alanı Örneği* (sayfa: 406) içinde gösterilen**make_named_socket** işlevini kendi soketlerini oluşturmak ve isimlendirmek için kullanmaktadır.

Öncelikle sunucu yazılımını görüyoruz. Gelecek veriler için bir döngü içerisinde beklemekte ve gelen veri gerisin geriye göndericiye iletilmektedir. Tabii ki bu işe yarar bir yazılım değil, fakat ana fikri vermektedir.

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SERVER  "/tmp/serversocket"
#define MAXMSG  512

int
main (void)
{
  int sock;
  char message[MAXMSG];
  struct sockaddr_un name;
  size_t size;
  int nbytes;

  /* Öncelikle dosya ismini kaldıralım,
     eğer isim yoksa bir sorun yok */
  unlink (SERVER);

  /* Soketi oluşturup sonsuz döngüye girelim. */
  sock = make_named_socket (SERVER);
  while (1)
    {
      /* Bir datagram bekliyoruz */
      size = sizeof (name);
      nbytes = recvfrom (sock, message, MAXMSG, 0,
                         (struct sockaddr *) & name, &size);
      if (nbytes < 0)
        {
          perror ("recfrom (server)");
          exit (EXIT_FAILURE);
        }

      /* Bir tanı iletisi verelim */
```

```
      fprintf (stderr, "Sunucu: alınan ileti: %s\n", message);

      /* İletiyi göndericiye geri gönderelim. */
      nbytes = sendto (sock, message, nbytes, 0,
                       (struct sockaddr *) & name, size);
      if (nbytes < 0)
        {
          perror ("sendto (server)");
          exit (EXIT_FAILURE);
        }
    }
}
```

10.4. Datagramların Okunmasıyla İlgili Örnek

Bu da önceki sunucuyla ilişkili istemci yazılımıdır.

It sends a datagram to the server and then waits for a reply. Notice that the socket for the client (as well as for the server) in this example has to be given a name. This is so that the server can direct a message back to the client. Since the socket has no associated connection state, the only way the server can do this is by referencing the name of the client.

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SERVER  "/tmp/serversocket"
#define CLIENT  "/tmp/mysocket"
#define MAXMSG  512
#define MESSAGE "Aloo!!! Adanaaa, Adana mi orasi?!?"

int
main (void)
{
  extern int make_named_socket (const char *name);
  int sock;
  char message[MAXMSG];
  struct sockaddr_un name;
  size_t size;
  int nbytes;

  /* Soketi oluşturalim. */
  sock = make_named_socket (CLIENT);

  /* Sunucu soket adresini ilklendirelim. */
  name.sun_family = AF_LOCAL;
  strcpy (name.sun_path, SERVER);
  size = strlen (name.sun_path) + sizeof (name.sun_family);

  /* Datagramı gönderelim. */
  nbytes = sendto (sock, MESSAGE, strlen (MESSAGE) + 1, 0,
                   (struct sockaddr *) & name, size);
  if (nbytes < 0)
    {
```

```
      perror ("sendto (client)");
      exit (EXIT_FAILURE);
    }

  /* Yanıt bekleyelim. */
  nbytes = recvfrom (sock, message, MAXMSG, 0, NULL, 0);
  if (nbytes < 0)
    {
      perror ("recfrom (client)");
      exit (EXIT_FAILURE);
    }

  /* Durumu bildirelim. */
  fprintf (stderr, "İstemci: gelen ileti: %s\n", message);

  /* Ortalığı temizleyelim. */
  remove (CLIENT);
  close (sock);
}
```

Datagram soket haberleşmesinin güvenilir olmadığını aklınızdan çıkarmayınız. Bu örnekte, istemci yazılım, ileti sunucuya ulaşmazsa veya sunucunun cevabı gelmezse sonsuza kadar bekler. Yazılımı sonlandırmak veya yeniden başlatmak çalıştıran kişiye kalmıştır. Daha özdevinimli bir çözüm ise **select** (*Girdi ve Çıktının Beklenmesi* (sayfa: 323)) ile cevap için bir zaman aşımı belirtmek ve bu süre sonunda iletiyi tekrar göndermek veya soketi kapatarak çıkmaktır.

11. inetd Artalan Süreci

Önceki bölümde kendi dinleme işlemini yapan bir sunucu yazılımınının nasıl yazıldığını anlattık. Bu tür bir sunucu bağlanacaklar için çalışıyor durumda olmalıdır.

Bir Internet portunda bir hizmet sunmanın diğer bir yolu ise bir artalan süreç yazılımı olan **inetd**'nin dinlemesidir. **inetd**, sürekli çalışan ve belirli portlardan gelecek iletiler için bekleyen bir yazılımdır. Bir ileti aldığında, bağlantıyı kabul eder (eğer soket tarzı bağlantı kabul ediyorsa) ve ilgili sunucu yazılımını çalıştırmak için **fork** ile bir alt süreç oluşturur. Port ve ilgili programları `/etc/inetd.conf` dosyasında tanımlamalısınız.

11.1. inetd Sunucuları

inetd ile çalışacak bir sunucu yazmak oldukça basittir. Her seferinde birileri uygun porttan bağlantı isteğinde bulunur ve yeni bir sunucu süreci başlar. Bu durumda bağlantı zaten vardır; soket, sunucu sürecinde standart girdi ve çıktı tanımlayıcısı (0 ve 1) olarak bulunmaktadır. Böylece sunucu yazılımı okuma ve yazma işlemlerine başlayabilir. Genelde yazılım sadece sıradan G/Ç imkanlarına ihtiyaç duyar; aslında, genel kullanıma yönelik, soketten birşey anlamayan bir filtreleme yazılımı **inetd** ile bayt akımlı sunucu olarak çalışabilir.

inetd'yi aynı zamanda bağlantısız iletişim tarzlarını kullanan sunucular için de kullanabilirsiniz. Bu sunucular için, **inetd** bağlantı kabul etmeye çalışmaz çünkü bağlantı imkanı yoktur. Sadece, 0 tanımlayıcısından gelen datagram paketini okuyabilen sunucu yazılımını başlatır. Sunucu yazılımı bir isteği ele alıp sonra çıkabilir veya daha fazla istek gelmeden, yazmasını sağlayabilirsiniz. **inetd**'yi ayarlarken sunucunun kullandığı bu iki teknikten birini belirtmelisiniz.

11.2. inetd Yapılandırması

`/etc/inetd.conf` dosyası **inetd**'ye hangi portları dinleyeceğini ve bunlarda hangi sunucu yazılımlarını çalıştıracağını söyler. Normalde dosya içindeki her girdi bir satırdır, bunları çoklu satırlara bölmek isterseniz girdinin ilk satırını boşlukla başlatmanız gerekir. **#** ile başlayan satırlar açıklama satırlarıdır.

Aşağıda `/etc/inetd.conf` dosyasından iki standart girdi görüyoruz:

```
ftp     stream  tcp     nowait  root    /libexec/ftpd   ftpd
talk    dgram   udp     wait    root    /libexec/talkd  talkd
```

Bir girdi aşağıdaki biçimdedir:

servis tarz protokol bekleme kullanıcı yazılım seçenekler

servis alanı bu programın hangi servisi desteklediğini söyler. Bu **`/etc/services`** içinde tanımlı servis isimlerinden biri olmalıdır. **`inetd`**, *servis*i bu girdi için hangi portun dinleneceğine karar vermek için kullanır.

tarz ve *protokol* alanları soketi dinlemek için kullanılan iletişim tarzını ve protokolünü belirler. *tarz*, küçük harflere çevrilmiş ve başındaki **`SOCK_`** kaldırılmış, **`stream`** veya **`dgram`** gibi bir iletişim tarzı adı olmalıdır. *protokol*, `/etc/protocols` içinde listelenen protokollerden biri olmalıdır. Tipik protokol isimleri bayt akımlı bağlantılar için **`tcp`** ve güvensiz datagramlar için **`udp`**'dir.

bekleme alanı ya **`wait`** ya da **`nowait`** olmalıdır. Eğer bağlantısız iletişim tarzı kullanıyorsanız ve sunucu başladığında gelen çoklu istekler ele alınıyorsa **`wait`** kullanın. **`inetd`**'nin gelen her ileti veya istek için yeni bir süreç başlatması gerekiyorsa **`nowait`** kullanın. *tarz* bağlantıları kullanıyorsa, *bekleme* alanı **`nowait`** olmalıdır.

kullanıcı sunucunun hangi kullanıcı altında çalışması gerektiğidir. **`inetd root`** olarak çalışır, böylece çocuklarına istediği kullanıcı kkimliğini verebilir. Yapabilirseniz, *kullanıcı* için **`root`** kullanmamak en iyisidir; fakat bazı sunucular, örneğin Telnet ve FTP, kendileri için kullanıcı adı ve parola okurlar. Bu sunucuların ilk başta **`root`** olmaları gerekir böylece ağ üzerinden gelen verinin yönlendirdiği şekilde giriş yapabilirler.

seçenekler ile birlikte *yazılım* sunucuyu başlatacak komutu belirtir. *yazılım* çalıştırılabilir bir dosyanın tam dosya ismi olmalıdır. *seçenekler* boşluklarla ayrılmış her hangi sayıda sözcükten oluşur, ki bunlar *yazılım*'ın komut satırı seçenekleri olur. *seçenekler* içindeki ilk sözcüğün argüman numarası sıfırdır ve bu yazılım isminin kendisidir (dizinsiz olarak).

`/etc/inetd.conf` dosyasını değiştirirseniz, **`inetd`**'ye **`SIGHUP`** sinyali göndererek dosyayı tekrar okumasını ve yeni içeriğe uymasını söyleyebilirsiniz. **`ps`** ile **`inetd`**'nin süreç kimliğini (PID) saptayabilirsiniz.

12. Socket Seçenekleri

Bu bölümde soketlerin davranışlarını değiştiren çeşitli özelliklerin nasıl okunduğu veya ayarlandığı ve altındaki haberleşme protokolleri anlatılmıştır.

Bir soket seçeneğini değiştirirken seçeneğin hangi ***seviye***ye ait olduğunu belirtmeniz gerekir. Bu, özelliğin soket arayüzüne mi yoksa alt–seviye haberleşme protokol arayüzüne mi etki edeceğini belirler.

12.1. Soket Seçenek İşlevleri

Burada soket seçeneklerini incelemek ve değiştirmek için kullanılan işlevleri göreceğiz. Bunlar `sys/socket.h` içinde bildirilmiştir.

```
int getsockopt(int         soket,                                    işlev
               int         seviye,
               int         sçnismi,
               void       *sçndeğeri,
               socklen_t  *sçnuzunluk_gstr)
```

`getsockopt` işlevi *soket* soketi için *seviye* seviyesindeki *sçnismi* seçeneğinin değeriyle ilgili bilgi döndürür.

Seçenek değeri *sçndeğeri*'nin gösterdiği tampon içinde saklanır. İşlevi çağırmadan önce, tampon boyutunu **sçnuzunluk_gstr* içinde vermeniz gerekir; dönüşte, tamponda gerçekte saklanan bilginin bayt sayısını içerecektir.

Çoğu seçenek *sçndeğeri* tamponunu tek bir **int** değer olarak yorumlar.

getsockopt işlevinin başarı halinde döndürdüğü değer **0**, başarısızlık halinde **-1** dir. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
soket argümanı geçerli bir dosya tanımlayıcısı değil.

ENOTSOCK
soket tanımlayıcısı bir soket değil.

ENOPROTOOPT
sçnismi belirtilen *seviye*ye duyarsız.

```
int setsockopt(int        soket,                                    işlev
               int        seviye,
               int        sçnismi,
               void      *sçndeğeri,
               socklen_t  sçnuzunluğu)
```

Bu işlev *soket* soketi için *seviye* seviyesinde *sçnismi* soket seçeneğini etkinleştirmek kullanılır. Seçenek değeri *sçnuzunluğu* uzunluğundaki *sçndeğeri* tamponuna aktarılır.

12.2. Soket Seviye Seçenekleri

```
int SOL_SOCKET                                                        sabit
```

Bu sabit, bu bölümde açıklanan soket seviye seçeneklerini değiştirmek için **getsockopt** veya **setsockopt** işlevinde *seviye* argümanı olarak kullanınılır.

Aşağıda listesi verilen soket seviye seçeneklerinin isimleri `sys/socket.h` başlık dosyası içinde tanımlıdır.

SO_DEBUG
Bu seçenek ilgili protokol modülleri içindeki hata ayıklama bilgileri kaydının etkinleştirilmesi ile ilgilidir. **int** türünde değer alır; sıfırdan farklı ise "evet" anlamına gelir.

SO_REUSEADDR
Bu seçenek **bind** (*Adreslerin Atanması* (sayfa: 402)) işlevinin soket için yerel adreslerin tekrar kullanımına izin verip vermeyeceğini kontrol eder. Bu özellik etkinleştirilirse, aynı Internet port numarasına sahip iki sokete sahip olursunuz; fakat sistem bu iki aynı–isimli soketi Internet ortamında karışıklık yaratacak şekilde kullanmanıza izin vermez. Bu özelliğin varoluş nedeni bazı yüksek seviyeli Internet protokollerine dayanmaktadır, örneğin FTP, aynı port numarasının tekrar kullanımını gerektirmektedir.

int türünde değer alır; sıfırdan farklı ise "evet" anlamına gelir.

SO_KEEPALIVE
Bu seçenek ilgili protokolün bağlantılı bir soketten iletinin belirli aralıklarla gönderilip gönderilmeyeceğini belirler. Eğer karşıdaki soket bu iletilere yanıt veremezse, bağlantı kesilmiş olarak kabul edilir. **int** türünde değer alır; sıfırdan farklı ise "evet" anlamına gelir.

SO_DONTROUTE
Bu seçenek giden iletilerin normal ileti yönlendirme imkanlarını devre dışı bırakıp bırakmayacağını kontrol eder. Eğer belirtilirse, iletiler doğrudan ağ arayüzüne gönderilir. **int** türünde değer alır; sıfırdan farklı ise "evet" anlamına gelir.

SO_LINGER
Bu seçenek güvenli teslim garantisi veren bir soketin bağlantısı kesilirse ve hala aktarılmamış ileriler varsa ne olacağını belirtir; bkz. *Bir Soketin Kapatılması* (sayfa: 421). **struct linger** türünde değer alır.

`struct` **linger** veri türü

Bu yapı aşağıdaki üyelere sahiptir:

`int` **l_onoff**
Bu alan mantıksal değer olarak yorumlanır. Sıfırdan farklıysa, **close** veri aktarılıncaya kadar veya zaman aşımına uğrayıncaya kadar baskılanır.

`int` **l_linger**
Bu zaman aşımını saniye cinsinden belirtir.

SO_BROADCAST
Bu seçenek datagramların soketten yayınlanabilmesi ile iilgilidir. **int** türünde değer alır; sıfırdan farklı ise "evet" anlamına gelir.

SO_OOBINLINE
Eğer bu seçenek belirtilirse, soketten alınan bantdışı veri normal girdi sırasına konulur. Bu **MSG_OOB** bayrağını belirtmeden **read** veya **recv** ile okuma yapmaya izin verir. Bkz. *Bantdışı Veri Aktarımı* (sayfa: 431). **int** türünde değer alır; sıfırdan farklı ise "evet" anlamına gelir.

SO_SNDBUF
Bu seçenek çıktı için kullanılan tamponun boyutunu getirir veya belirtir. Bayt cinsinden olan **size_t** türündedir.

SO_RCVBUF
Bu seçenek girdi için kullanılan tamponun boyutunu getirir veya belirtir. Bayt cinsinden olan **size_t** türündedir.

SO_STYLE
SO_TYPE
Bu seçenek sadece **getsockopt** ile kullanılabilir. Soketin iletişim tarzını getirmek için kullanılır. **SO_TYPE** eski ismidir ve **SO_STYLE** GNU içinde tercih edilen ismidir. **int** türünde değer alır ve değeri bir iletişim tarzını gösterir; bkz. *İletişim Tarzları* (sayfa: 400).

SO_ERROR
Bu seçenek sadece **getsockopt** ile kullanılabilir. Soketin hata durumunu sıfırlamak için kullanılır. Önceki hata durumunu gösteren **int** türünde bir değerdir.

13. Ağ İsimleri Veritabanı

Birçok sistem bilinen ağ isimlerinin listesini kaydederek sistem geliştiricisine sunan bir veritabanı ile beraber gelir. Bu bilgi genellikle **/etc/networks** dosyası içerisinde veya eşdeğer bir isim sunucusunda tutulur. Bu

veritabanı **route** gibi yönlendirme yazılımları için kullanışlıdır ancak ağ üzerinde basitçe haberleşen yazılımlar için değildir. Bu veritabanına erişmek için gerekli işlevler `netdb.h` içinde bildirilmiştir.

`struct` **netent** veri türü

Bu veri türü ağ veritabanındaki girdi bilgilerininin gösterimi için kullanılır. Aşağıdaki üyelere sahiptir:

`char *`**n_name**
"Resmi" ağ ismidir.

`char **`**n_aliases**
Bunlar bir dizgeler dizisi olarak alternatif ağ isimleridir. Diziyi bir boş gösterici sonlandırır.

`int` **n_addrtype**
Bu ağ numarasının türüdür; Internet ağları için herzaman **AF_INET**'e eşittir.

`unsigned long int` **n_net**
Bu ağ numarasıdır. Ağ numaraları *konak bayt sırası* (sayfa: 417) ile döndürülür.

Ağ veritabanında belirli bir ağ hakkında arama yapıp bilgi edinmek için **getnetbyname** veya **getnetbyaddr** işlevi kullanılır. Bilgi durağan olarak ayrılmış bir yapıda geri döner; saklamak gerekiyorsa bilginin kopyalanması gerekir.

`struct netent *`**getnetbyname**`(const char *`*isim*`)` işlev

getnetbyname işlevi *isim* ile isimlendirilmiş ağ hakkında bilgi döndürür. Öyle bir ağ yoksa boş gösterici döndürür.

`struct netent *`**getnetbyaddr**`(unsigned long int` *ağ*`, int` *tür*`)` işlev

getnetbyaddr işlevi *tür* türünde ve *ağ* numarasındaki ağ hakkında bilgi döndürür. Internet ağları için *tür* argümanına bir **AF_INET** değeri belirtmelisiniz.

getnetbyaddr öyle bir ağ yoksa boş gösterici döndürür.

Ayrıca **setnetent**, **getnetent** ve **endnetent** kullanarak ağ veritabanını tarayabilirsiniz. Bu işlevleri kullanırken dikkatli olun çünkü bunlar evresel (reentrant) işlevler değildir.

`void` **setnetent**`(int` *açıkkal*`)` işlev

Bu işlev ağ veritabanını açar ve sayacı başa getirir.

açıkkal argümanı sıfır ise bu bir bayrağa değer verir böylece sonraki **getnetbyname** veya **getnetbyaddr** işlev çağrıları veritabanını kapatmaz (normalde olması gerektiği gibi). Eğer bu işlevleri çok kez çağırıyorsanız, her çağrıda veritabanını tekrar açmakdan kurtulur ve daha verimli bir sonuç elde edersiniz.

`struct netent *`**getnetent**`(void)` işlev

Bu işlev ağ veritabanındaki sonraki girdiyi döndürür. Eğer sonraki girdi yoksa boş gösterici döndürür.

`void` **endnetent**`(void)` işlev

Bu işlev ağ veritabanını kapatır.

XVII. Düşük Seviyeli Uçbirim Arayüzü

İçindekiler

Bu oylumda açıklanan işlevler uçbirim aygıtlarına özeldir. Bu işlevleri kullanarak girdilerin yansılanmasını engelleyebilir, hat hızı ve akış denetimi gibi seri hat karakteristiklerini değiştirebilir, dosyasonu karakteri olarak kullanılacak karakteri, komut satırı düzenlemeyi, sinyal gönderimini, v.s. değiştirebilirsiniz.

Bu oylumdaki işlevlerin çoğu dosya tanıtıcılarla çalışır. Dosya tanıtıcıların ne olduğu ve bir dosya tanıtıcısının bir uçbirim olarak nasıl açıldığı gibi konuları *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305) bölümünde bulabilirsiniz.

1. Uçbirimlerin Tanımlanması

Bu kısımda bahsedilen işlevler sadece uçbirim aygıtlarına karşılık olan dosyalarla çalışır. Bir dosyanın bir uçbirimle ilişkili olup olmadığını **isatty** işlevini kullanarak öğrenebilirsiniz.

Bu kısımdaki işlevlerin prototipleri `unistd.h` başlık dosyasında bildirilmiştir.

`int` **isatty**`(int` *dosyatanıtıcı*`)` işlev

Bu işlev, *dosyatanıtıcı* bir uçbirimle ilişkili bir dosya tanıtıcı ise **1** ile değilse **0** ile döner.

Bir dosya tanıtıcı bir uçbirimle ilişkili ise ilişkili dosya ismini **ttyname** işlevini kullanarak öğrenebilirsiniz. Ayrıca *Denetim Uçbiriminin İsimlendirilmesi* (sayfa: 729) bölümünde açıklanan **ctermid** işlevine de bakınız.

`char *`**ttyname**`(int` *dosyatanıtıcı*`)` işlev

dosyatanıtıcı bir uçbirimle ilişkili bir dosya tanıtıcı ise, **ttyname** işlevi uçbirim dosyasının ismini içeren durağan ayrılmış boş karakter sonlandırmalı bir dizgeye bir gösterici ile döner. Dosya tanıtıcısı bir uçbirime karşılık değilse ya da dosya ismi saptanamamışsa işlev boş gösterici ile döner.

```
int ttyname_r(int     dosyatanıtıcı,                                    işlev
              char   *tampon,
              size_t uzunluk)
```

Sonucun kullanıcı tarafından belirtilen *uzunluk* uzunluktaki *tampon* tamponu ile döndürülmesi dışında **ttyname** işlevi gibidir.

ttyname_r işlevinin normal dönüş değeri sıfırdır. Aksi takdirde hatayı belirten bir hata numarası ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcısı değil

ENOTTY
: *dosyatanıtıcı* bir uçbirimle ilişkili değil

ERANGE
: Tampon uzunluğu olarak *uzunluk*, döndürülecek dizgeyi saklamak için çok küçük

2. G/Ç Kuyrukları

Bu kısımdaki işlevler bir uçbirim aygıtının girdi ve çıktı kuyrukları ile ilgilidir. Bu kuyruklar, *G/Ç akımlarına* (sayfa: 236) göre gerçeklenmiş tamponlamadan bağımsız olarak *çekirdek içindeki* bir tamponlama şeklinde gerçeklenmiştir.

Uçbirim girdi kuyruğu çoğunlukla kullandığı tamponun ismiyle *sürekli yazma (typeahead) tamponu* olarak da anılır. Bir uçbirim tarafından alınmış ancak henüz bir süreç tarafından okunmamış karakterleri içerir.

Girdi kuyruğunun uzunluğunu **MAX_INPUT** ve **_POSIX_MAX_INPUT** parametreleri belirler; bkz. *Dosya Sistemi Kapasite Sınırları* (sayfa: 795). Kuyruk uzunluğunun en azından **MAX_INPUT** karakterlik olacağını düşünebilirsiniz, ancak kuyruk daha büyük olabilir ve hatta uzunluğu özdevimli olarak değişebilir. Eğer **IXOFF** girdi kipi biti (bkz. *Girdi Kipleri* (sayfa: 447)) atanarak akış denetimi etkinleştirilmişse, kuyruğu taşmadan korumak gerektiğinde uçbirim sürücüsü uçbirime STOP ve START karakterlerini aktarır. Aksi takdirde, girdi uçbirimin kabul edebildiğinden hızlı gelirse bir kısım girdi kaybolabilir. Kurallı kipte, bir satırsonu karakteri alınıncaya kadar girdi kuyrukta kalır, dolayısıyla çok uzun bir satır yazdığınızda uçbirim girdi kuyruğu dolabilir. Bakınız: *İki Girdi Tarzı: Kurallı veya Kuralsız* (sayfa: 443).

Uçbirim çıktı kuyruğu girdi kuyruğu gibidir, ancak çıktı içindir. Süreçler tarafından yazılmış fakat henüz uçbirime aktarılmamış karakterleri içerir. Eğer **IXON** girdi kipi biti (bkz. *Girdi Kipleri* (sayfa: 447)) atanarak akış denetimi etkinleştirilmişse, uçbirim sürücüsü, çıktı aktarımını durdurmak ve yeniden başlatmak için uçbirim tarafından gönderilen STOP ve START karakterlerine uymaya çalışır.

Temizleme uçbirim girdi kuyruğundaki alınmış ancak henüz okunmamış karakterlerin iptal edilmesi anlamına gelir. Benzer olarak, uçbirim çıktı kuyruğuna yazılmış ancak henüz aktarılmamış karakterlerin iptal edilmesi anlamına da gelir.

3. İki Girdi Tarzı: Kurallı veya Kuralsız

POSIX sistemleri iki temel girdi kipini destekler: kurallı ve kuralsız.

Kurallı girdi işleme kipinde girdi, satırsonu (`'\n'`), EOF veya EOL karakterleri ile sonlandırılmış satırlar halinde işlenir. Kullanıcı tarafından satırın tamamı yazılana kadar hiçbir girdi okunmaz. Girdi alındıktan sonra, kaç bayt istendiğine bakılmaksızın **read** (*Girdi ve Çıktı İlkelleri* (sayfa: 308)) işlevi tek satırlık bir girdi ile döner.

Kurallı girdi kipinde, girdi düzenleme oluşumlarını işletim sistemi sağlar: o anki metin satırı içindeki bazı karakterler metin düzenleme işlemlerini gerçekleştiren ERASE ve KILL gibi özel karakterler olarak yorumlanır. Bkz. *Girdi Düzenleme Karakterleri* (sayfa: 454).

_POSIX_MAX_CANON ve **MAX_CANON** sabitleri, kurallı kipte tek bir satırda bulunabilecek karakterlerin sayısının üst sınırını belirleyen parametrelerdir. Bkz. *Dosya Sistemi Kapasite Sınırları* (sayfa: 795). Satır uzunluğunun en azından **MAX_INPUT** karakterlik olacağını düşünebilirsiniz, ancak satır daha uzun olabilir ve hatta uzunluğu özdevimli olarak değişebilir.

Kuralsız girdi işleme kipinde karakterler satırlar halinde gruplanmaz ve ERASE veya KILL gibi metin düzenleme karakterleri dikkate alınmaz. Girdinin baytlar halinde okunduğu kuralsız girdi kipi MIN ve TIME ayarları ile denetlenir. Bkz. *Kuralsız Girdi* (sayfa: 458).

Çoğu uygulama kurallı girdi kipini kullanır, çünkü bu kip kullanıcıya metni satır satır düzenleyebilme imkanı verir. Uygulama tek karakterlik komutlar kabul edecekse ve kendi metin düzenleme oluşumları olacaksa kuralsız kip tercih edilir.

Kurallı ve kuralsız girdi seçimi **struct termios** yapısının **c_lflag** üyesinde **ICANON** seçeneğinin kullanılmasına bağlıdır. Bkz. *Yerel Kipler* (sayfa: 451).

4. Uçbirim Kipleri

Bu kısımda girdi ve çıktının denetiminde kullanılan çeşitli uçbirim öznitelikleri açıklanmıştır. Bu kısımdaki işlevler, veri yapıları ve sembolik sabitler `termios.h` başlık dosyasında bildirilmiştir.

Uçbirim öznitelikleri ile dosya özniteliklerini birbirine karıştırmayın. Bir uçbirimle ilişkilendirilmiş bir aygıt özel dosyası *Dosya Öznitelikleri* (sayfa: 371) bölümünde anlatılan dosya özniteliklerine sahiptir ve bunlar uçbirim aygıtının bu bölümde bahsedilecek öznitelikleri ile ilgili değildir.

4.1. Uçbirim Kipi Veri Türleri

Bir uçbirimin özniteliklerinin tamamı **struct termios** türünde bir yapı içinde saklanır. Bu yapı, öznitelikleri okumak ve değiştirmek için **tcgetattr** ve **tcsetattr** işlevleri ile kullanılır.

`struct` **termios** — veri türü

Bir uçbirimin G/Ç özniteliklerinin kaydedildiği yapıdır. Yapı en azından şu üyeleri içermelidir:

`tcflag_t` **c_iflag**
: Girdi kipleri ile ilgili seçenekleri belirleyen bit maskesi; bkz. *Girdi Kipleri* (sayfa: 447).

`tcflag_t` **c_oflag**
: Çıktı kipleri ile ilgili seçenekleri belirleyen bit maskesi; bkz. *Çıktı Kipleri* (sayfa: 449).

`tcflag_t` **c_cflag**
: Denetim kipleri ile ilgili seçenekleri belirleyen bit maskesi; bkz. *Denetim Kipleri* (sayfa: 449).

`tcflag_t` **c_lflag**
: Yerel kipler ile ilgili seçenekleri belirleyen bit maskesi; bkz. *Yerel Kipler* (sayfa: 451).

`cc_t` **c_cc**`[NCCS]`
: Çeşitli denetim işlevleri ile ilişkili karakterlerin belirtildiği bir dizi; bkz. *Özel Karakterler* (sayfa: 454).

struct termios yapısı, ayrıca gidi ve çıktı iletim hızlarını kodlayan üyeler de içerir, fakat gösterim belirlenmemiştir. Hız değerlerinin nasıl öğrenileceği ve belirtileceği *Hat Hızı* (sayfa: 453) bölümünde açıklanmıştır.

struct termios yapısının üyeleri bundan sonraki bölümlerde ayrı ayrı ele alınıp açıklanacaktır.

tcflag_t veri türü

Çeşitli uçbirim seçenekleri ile ilgili bit maskeleri için kullanılan bir işaretsiz tamsayı türüdür.

cc_t veri türü

Çeşitli uçbirim denetim işlevleri ile ilgili karakterler için kullanılan bir işaretsiz tamsayı türüdür.

`int` **NCCS** makro

c_cc dizisinin eleman sayısını belirleyen makro.

4.2. Uçbirim Kipi İşlevleri

```
int tcgetattr(int                dosyatanıtıcı,                    işlev
              struct termios *termios–p)
```

Bu işlev *dosyatanıtıcı* dosya tanıtıcısı ile ilişkili uçbirim aygıtının özniteliklerini öğrenmek için kullanılır. Öznitelikler *termios–p* ile gösterilen yapı içinde döner.

tcgetattr başarılıysa 0 ile, değilse bir hata oluştuğunu belirtmek üzere –1 ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcı değil

ENOTTY
: *dosyatanıtıcı* bir uçbirim ile ilişkili değil

```
int tcsetattr(int                      dosyatanıtıcı,                  işlev
              int                      nezaman,
              const struct termios *termios–p)
```

Bu işlev *dosyatanıtıcı* dosya tanıtıcısı ile ilişkili uçbirim aygıtının özniteliklerini ayarlamak için kullanılır. Yeni öznitelikler *termios–p* ile gösterilen yapı içinde alınır.

nezaman argümanı kuyruklanmış girdi ve çıktının ne zaman işleme sokulacağını belirtmek için kullanılır. Şu değerlerden birini içerebilir:

TCSANOW
: Değişiklik hemen yapılır.

TCSADRAIN
: Kuyruktaki tüm çıktı yazıldıktan sonra değişiklik yapılır. Bu seçeneği çıktıyı etkileyen parametreleri değiştirirken kullanmalısınız.

TCSAFLUSH
: **TCSADRAIN** gibidir, ayrıca kuyruktaki tüm girdi iptal edilir.

TCSASOFT

Yukarıdaki seçeneklerin herbirini ekleyebileceğiniz bir seçenek bitidir. Uçbirim donanımının durum değiştirmesinin yasaklanması anlamına gelir. Bir BSD oluşumudur ve sadece BSD sistemleri ile GNU sisteminde desteklenir.

TCSASOFT kullanımı, *termios–p* ile gösterilen yapının **c_cflag** üyesine **CIGNORE** bitinin atanması ile tamamen aynı ayarı yapar. **CIGNORE** ile ilgili daha fazla bilgi için *Denetim Kipleri* (sayfa: 449) bölümüne bakınız.

Eğer bu işlev kendi denetim uçbirimi ile ilgili olarak bir artalan sürecinden çağrılmışsa, kendi süreç grubundaki tüm süreçler, sürecin yazmaya çalıştığı yolla bir **SIGTTOU** sinyali gönderir. Ancak, işlevi çağıran sürecin **SIGTTOU** sinyallerini engellemesi ya da yoksayması durumunda işlem yine yapılır ama sinyal gönderilmez. Bkz. *İş Denetimi* (sayfa: 716).

tcsetattr başarılıysa 0 ile döner. Aksi halde –1 ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcı değil

ENOTTY
: *dosyatanıtıcı* bir uçbirimle ilişkili değil

EINVAL
: Ya *nezaman* argümanı geçersiz ya da *termios–p* argümanındaki veride yanlış birşeyler var.

tcgetattr ve **tcsetattr** işlevleri uçbirim aygıtını bir dosya tanıtıcısı ile belirttiği halde, öznitelikler dosya tanıtıcısının değil uçbirim aygıtının kendisine aittir. Bu, uçbirim aygıtının özniteliklerindeki değişikliklerin kalıcı olduğu anlamına gelir; eğer başka bir süreç daha sonra bu uçbirimi açarsa, süreç dosya tanıtıcısını açarken sizin özniteliklerde belirttiğiniz değişikliklere ilişkin hiçbir şey yapmadığı halde değişmiş öznitelikleri görecektir.

Benzer şekilde, tek bir sürecin aynı uçbirim için çok sayıda veya yinelenmiş dosya tanıtıcıları varsa, uçbirim özniteliklerindeki değişiklikler tüm dosya tanıtıcılarının girdi ve çıktılarını etkileyecektir. Yani, bir uçbirimi tek karakter okunan, yansılama yapılmayan kipte kullanıyorsanız aynı uçbirimi farklı bir dosya tanıtıcı kullanarak satır tamponlu, yansılanan kipte açamazsınız. Ama uçbirimi önce istediğiniz kipe sokup açabilir işiniz bittikten sonra diğer kipe geçip uçbirimi bırakabilirsiniz.

4.3. Uçbirim Kiplerinin Doğru Dürüst Belirtilmesi

Bir uçbirimin kipini değiştireceğiniz zaman, önce **tcgetattr** çağrısı ile uçbirim aygıtının o anki kipini öğrenmeli, bu kipi istediğiniz özniteliklerle değiştirdikten sonra sonucu **tcsetattr** çağrısı ile uçbirime göndermelisiniz.

struct termios yapısını basitçe ilklendirip öznitelikleri istediğiniz gibi değiştirip sonra bunu **tcsetattr** işleviyle aktarmanız hiç iyi bir yöntem değildir. Yazılımınızın yıllar sonra bu kılavuzda belgelenmemiş üyelerin desteklendiği sistemlerde çalışabileceğini varsaymalısınız. İlgilenmediğiniz yapı üyelerini değiştirmekten kaçınmak en iyi yöntemdir.

Dahası, farklı uçbirim aygıtları farklı kip seçimleri gerektirebilir. Bu bakımdan öznitelikleri bir uçbirimden diğerine körü körüne kopyalamaktan da kaçınmalısınız.

c_iflag, **c_oflag** ve **c_cflag** üyelerinde olduğu gibi bir üye bağımsız seçeneklerin bir kolleksiyonunu içeriyorsa üyenin değerini tamamen değiştirmek de kötü olacaktır. Bunu yapmak yerine üyeyi mevcut değerlerle başlatmalı ve ilgisiz seçeneklere dokunmadan sadece yazılımınız için gerekli seçenekleri değiştirmelisiniz.

Bu örnekte **struct termios** yapısındaki diğer verilere dokunmadan sadece bir seçeneğin (**ISTRIP**) değiştirilmesi gösterilmiştir:

```
int
set_istrip (int desc, int value)
{
  struct termios settings;
  int result;

  result = tcgetattr (desc, &settings);
  if (result < 0)
    {
      perror ("tcgetattr'de hata");
      return 0;
    }
  settings.c_iflag &= ~ISTRIP;
  if (value)
    settings.c_iflag |= ISTRIP;
  result = tcsetattr (desc, TCSANOW, &settings);
  if (result < 0)
    {
      perror ("tcsetattr'de hata");
      return 0;
    }
  return 1;
}
```

4.4. Girdi Kipleri

Bu bölümde girdi işleme ile ilgili düşük seviyeli özellikleri denetlemeye yarayan uçbirim özniteliklerinden bahsedilmiştir: eşlik hatalarının yakalanması, geçici kesme sinyalleri, akış denetimi, <RET> ve <LFD> karakterleri.

Bu seçeneklerin hepsi **struct termios** yapısının **c_iflag** üyesindeki bitlerdir. Üye bir tamsayıdır ve bu seçenekleri **&**, **|** ve ^ işleçleri ile değiştirebilirsiniz. **c_iflag** üyesinin değerini toptan değiştirmeyi denemeyin; ilgisiz seçeneklere dokunmadan sadece sizi ilgilendiren seçenekleri değiştirin (bkz. *Uçbirim Kiplerinin Doğru Dürüst Belirtilmesi* (sayfa: 446)).

`tcflag_t` **INPCK** — makro

Bu bit varsa, girdi eşlik sınaması etkindir. Yoksa, girdideki eşlik hataları ile ilgili hiçbir şey yapılmaz; karakterler basitçe uygulamaya aktarılır.

Girdi işlemede eşik sınaması, ilgili uçbirim donanımında eşlik üretiminin ve eşlik saptanmasının etkin olup olmamasından bağımsızdır; bkz. *Denetim Kipleri* (sayfa: 449). Örneğin, **INPCK** girdi kipi seçeneğini temizleyip, **PARENB** denetim kipi seçeneğini etkileştirip girdideki eşlik hatalarını yoksayarken çıktıda hala eşlik üretebilirsiniz.

Bu bit varsa, bir eşlik hatası saptanırken **IGNPAR** veya **PARMRK** bitlerinin varlığı veya yokluğu önem kazanır. Bu bitlerin hiçbiri yoksa, eşlik hatası olan bir bayt uygulamaya **'\0'** karakteri olarak aktarılır.

`tcflag_t` **IGNPAR** — makro

Bu bit varsa, tertip veya eşlik hataları olan baytlar yoksayılır. Bu seçenek **INPCK** seçeneği de etkinse işe yarar.

`tcflag_t` **PARMRK** — makro

Bu bit varsa, tertip veya eşlik hataları olan baytlar yazılıma aktarılırken imlenir. Bu bit **INPCK**'ın varlığı ve **IGNPAR**'ın yokluğu durumunda anlamlıdır.

Hatalı baytlar, baytlardan önce gönderilen iki baytla **377** ve **0** baytları ile imlenir. Dolaysıyla, yazılım, hatalı baytı uçbirimden alırken aslında üç bayt okur.

Bir geçerli bayt **0377** değerine sahipse ve **ISTRIP** seçeneği etkinse yazılım bu baytı bir eşlik hatasının imi olarak ele alabilir. Bunu önlemek, yani **0377** karakterinin kendisi olarak ele alınmasını sağlamak için bu bayt yazılıma 2 bayt olarak, **0377 0377** olarak aktarılmalıdır.

`tcflag_t` **ISTRIP** makro

Bu bit varsa, geçerli girdi baytları yedi bit uzunlukta kabul edilir; aksi takdirde, hepsi sekiz bit kabul edilir.

`tcflag_t` **IGNBRK** makro

Bu bit varsa, geçici kesme (break) durumları yosayılır.

Geçici kesme durumu, eşzamansız seri veri iletişimi bağlamında tek bir bayttan daha uzun sıfır değerli bitler olarak tanınır.

`tcflag_t` **BRKINT** makro

Bu bit varsa ve **IGNBRK** yoksa bir geçici kesme durumu saptandığında uçbirim girdi ve çıktı kuyruklarını temizlenir ve uçbirimle ilişkili önalan süreç grubuna bir **SIGINT** sinyali gönderilir.

Ne **BRKINT** ne de **IGNBRK** varsa ve bir geçici kesme durumu saptandığında eğer **PARMRK** yoksa uygulamaya tek bir **'\0'** karakteri, **PARMRK** varsa üç karakterlik bir dizi, **'\377'**, **'\0'**, **'\0'** gönderilir.

`tcflag_t` **IGNCR** makro

Bu bit varsa girdideki satırbaşı (**'\r'**) karakterleri iptal edilir. Satırbaşı karakterinin iptal edilmesi <**RET**> tuşuna basıldığında hem satırbaşı hem de satırsonu karakteri gönderen uçbirimlerde yararlıdır.

`tcflag_t` **ICRNL** makro

Bu bit varsa ve **IGNCR** yoksa, girdiden alınan satırbaşı (**'\r'**) karakterleri uygulamaya satırsonu (**'\n'**) karakterleri olarak aktarılır.

`tcflag_t` **INLCR** makro

Bu bit varsa, girdiden alınan satırsonu (**'\n'**) karakterleri uygulamaya satırbaşı (**'\r'**) karakterleri olarak aktarılır.

`tcflag_t` **IXOFF** makro

Bu bit varsa, girdi üzerinde başlat/durdur denetimi etkinleştirilir. Başka bir deyişle, girdi uygulamanın okuyabileceğinden daha hızlı geliyorsa bilgisayar gerektiği zaman STOP ve START karakterleri gönderir. Bu fikir, gerçek uçbirim donanımının iletişimi STOP karakteri geldiğinde beklemeye alması, START karakterinde ise iletişime kaldığı yerden devam ettirmesi esasına dayanır. Bkz. *Akış Denetimi için Özel Karakterler* (sayfa: 457).

`tcflag_t` **IXON** makro

Bu bit varsa, çıktı üzerinde başlat/durdur denetimi etkinleştirilir. Başka bir deyişle, bilgisayar bir karakteri aldığında çıktıyı bir START karakteri alıncaya kadar bekletir. Bu durumda uygulamaya STOP ve START karakterleri kesinlikle aktarılmaz. Bu bit yoksa, STOP ve START karakterleri sıradan karakterler olarak okunabilir. Bkz. *Akış Denetimi için Özel Karakterler* (sayfa: 457).

`tcflag_t` **IXANY** makro

Bu bit varsa, STOP karakteri ile bekletilen çıktıyı herhangi bir girdi karakteri başlatır. Aksi takdirde bekletilen çıktılama sadece START karakteri ile sürdürülür.

Bu bir BSD oluşumudur; sadece BSD sistemlerinde ve GNU sisteminde desteklenir.

`tcflag_t` **IMAXBEL** makro

Bu bit varsa, uçbirim girdi tamponu dolduğunda uçbirime çanı çaldırmak için bir BELL karakteri (**007**) gönderilir.

Bu bir BSD oluşumudur.

4.5. Çıktı Kipleri

Bu bölümde çıktı karakterlerinin nasıl dönüştürüleceği ve gösterilirken nasıl biçimleneceği ile ilgili seçenekler ve alanlar açıklanmıştır. Bunların tamamı **struct termios** yapısının **c_oflag** üyesindeki değerin içinde yer alır.

Üye bir tamsayıdır ve bu seçenekleri **&**, **|** ve ^ işleçleri ile değiştirebilirsiniz. **c_oflag** üyesinin değerini toptan değiştirmeyi denemeyin; ilgisiz seçeneklere dokunmadan sadece sizi ilgilendiren seçenekleri değiştirin (bkz. *Uçbirim Kiplerinin Doğru Dürüst Belirtilmesi* (sayfa: 446)).

`tcflag_t` **OPOST** makro

Bu bit varsa, çıktı verisi ilgili uçbirim aygıtında gösterimek üzere bir takım yollarla işlenir. Bu işlem genellikle satırsonu ve satırbaşı karakterlerinin satırsonu (**'\n'**) karakteri ile değiştirilmesi şeklinde olur.

Bu bit yoksa, karakterler oldukları gibi aktarılırlar.

Aşağıdaki üç bit birer BSD oluşumudur ve sadece BSD sistemleri ile GNU sisteminde vardır. Bunlar sadece **OPOST** bitinin varlığında anlamlıdır.

`tcflag_t` **ONLCR** makro

Bu bit varsa, satırsonu karakterlerinin önüne birer satırbaşı karakteri yerleştirilir.

`tcflag_t` **OXTABS** makro

Bu bit varsa, sekme karakterleri sekizlik sütunlar oluşturacak biçimde boşluk karakteri ile değiştirilir.

`tcflag_t` **ONOEOT** makro

Bu bit varsa, çıktıdaki **C-d** karakterleri iptal edilir. Bu karakterler çoğu çevirmeli ağ uçbiriminde hattın kesilmesine yol açar.

4.6. Denetim Kipleri

Bu bölümde eşzamansız seri veri iletimi ile ilgili denetim parametreleri olan uçbirim seçenekleri ve alanlarından bahsedilecektir. Bu seçenekler diğer uçbirim port çeşitlerinde etkisiz olabilir (örneğin, bir ağ bağlantısı olarak uçbirimsiler). Bu seçeneklerin hepsi **struct termios** yapısının **c_cflag** üyesinin bitleridir.

c_cflag bir tamsayıdır ve bu seçenekleri **&**, **|** ve ^ işleçleri ile değiştirebilirsiniz. **c_cflag** üyesinin değerini toptan değiştirmeyi denemeyin; ilgisiz seçeneklere dokunmadan sadece sizi ilgilendiren seçenekleri değiştirin (bkz. *Uçbirim Kiplerinin Doğru Dürüst Belirtilmesi* (sayfa: 446)).

`tcflag_t` **CLOCAL** makro

Bu bit varsa, uçbirimin "yerel olarak" bağlı olduğu ve modem durum satırlarının (örneğin, taşıyıcının saptanması) yoksayıldığı anlaşılır. Bu bit yoksa, **open** işlevini **O_NONBLOCK** seçeneğinin yokluğunda çağırırsanız, çoğu sistemde **open** işlevi bir modem bağlantısı sağlanıncaya kadar engellenir.

Bu bit yoksa ve bir modem bağlantısı saptanamazsa, (eğer varsa) uçbirim için denetim süreç grubuna bir **SIGHUP** sinyali gönderilir. Bir bağlantı kesilmesinden sonraki okuma işlemleri bir dosyasonu durumuna, yazma işlemleri ise bir **EIO** hatasının dönmesine sebep olur. Bu durumu ortadan kaldırmak için uçbirim aygıtı kapatılıp yeniden açılmalıdır.

`tcflag_t` **HUPCL** makro

Bu bit varsa, açık uçbirim aygıtı olan tüm süreçler çıktığında ya da dosyalarını kapattıklarında bir modem bağlantı kesmesi üretilir.

`tcflag_t` **CREAD** makro

Bu bit varsa, girdi uçbirimden okunabilir. Aksi takdirde, girdi geldiği anda iptal edilir.

`tcflag_t` **CSTOPB** makro

Bu bit varsa, iki durdurma biti kullanılır. Aksi takdirde, sadece bir durdurma biti kullanılır.

`tcflag_t` **PARENB** makro

Bu bit varsa, eşlik bitinin üretimi ve saptanması etkinleştirilir. Girdideki eşlik hatalarının nasıl ele alındığı *Girdi Kipleri* (sayfa: 447) bölümünde anlatılmıştır.

Bu bit yoksa, çıktı karakterlerinde eşlik biti eklenmez ve girdi karakterlerinde eşlik bitlerinin doğruluğuna bakılmaz.

`tcflag_t` **PARODD** makro

Bu bit sadece **PARENB** biti varsa anlamlıdır. **PARODD** biti varsa tek eşlik kullanılır, aksi takdirde çift eşlik kullanılır.

Denetim kipi seçenekleri ayrıca karakter başına bit sayısı için bir alan içerir. Bu değeri çıkarmak için **CSIZE** makrosunu bir maske olarak kullanabilirsiniz: **settings.c_cflag & CSIZE**

`tcflag_t` **CSIZE** makro

Karakter başına bit sayısı için bir maskedir.

`tcflag_t` **CS5** makro

Her baytın beş bit olduğunu belirtir.

`tcflag_t` **CS6** makro

Her baytın altı bit olduğunu belirtir.

`tcflag_t` **CS7** makro

Her baytın yedi bit olduğunu belirtir.

`tcflag_t` **CS8** makro

Her baytın sekiz bit olduğunu belirtir.

Aşağıdaki dört bit birer BSD oluşumudur; sadece BSD sistemlerinde ve GNU sisteminde vardır.

`tcflag_t` **CCTS_OFLOW** makro

Bu bit varsa, CTS teline (RS232 protokolü) göre çıktı akış denetimi etkin olur.

`tcflag_t` **CRTS_IFLOW** makro

Bu bit varsa, RTS teline (RS232 protokolü) göre çıktı akış denetimi etkin olur.

`tcflag_t` **MDMBUF** makro

Bu bit varsa, taşıyıcı bazlı çıktı akış denetimi etkin olur.

`tcflag_t` **CIGNORE** makro

Bu bit varsa, denetim kiplerinin ve hat hızı değerlerinin tamamen yoksayıldığı anlamına gelir. Bu sadece bir **tcsetattr** çağrısı ile anlam kazanır.

cfgetispeed ve **cfgetospeed** çağrılarından dönen hız değerleri ile **c_cflag** üyesi çağrı ile etkisiz olacaktır. **c_cflag** içindeki donanımla ilgili ayrıntılara dokunmadan diğer üyelerdeki yazılımsal kiplerin tamamını değiştirmek isterseniz **CIGNORE** faydalıdır. (Bu, **tcsettattr** ile **TCSASOFT** seçeneğinin atanması ile ilgilidir.)

Bu bit **tcgetattr** tarafından döndürülen yapıda asla bulunmaz.

4.7. Yerel Kipler

Bu bölümde **struct termios** yapısının **c_lflag** üyesindeki seçeneklerden bahsedilecektir. Bu seçenekler genelde *Girdi Kipleri* (sayfa: 447) bölümünde bahsedilen girdi kipleri seçeneklerinden yansılama, sinyaller, kurallı ve kuralsız girdi seçimi gibi girdi işlemenin daha yüksek seviyeli işlemlerini denetler.

c_lflag bir tamsayıdır ve bu seçenekleri **&**, **|** ve **^** işleçleri ile değiştirebilirsiniz. **c_lflag** üyesinin değerini toptan değiştirmeyi denemeyin; ilgisiz seçeneklere dokunmadan sadece sizi ilgilendiren seçenekleri değiştirin (bkz. *Uçbirim Kiplerinin Doğru Dürüst Belirtilmesi* (sayfa: 446)).

`tcflag_t` **ICANON** makro

Bu bit varsa, kurallı girdi işleme kipi etkin olur. Aksi takdirde, girdi kuralsız kipte işlenir. Bkz. *İki Girdi Tarzı: Kurallı veya Kuralsız* (sayfa: 443).

`tcflag_t` **ECHO** makro

Bu bit varsa, girdi karakterlerinin uçbirime yansılanması etkin olur.

`tcflag_t` **ECHOE** makro

Bu bit varsa, yansılama sırasında girdi ekrandaki son satırın son karakteri ERASE karakteri tarafından silinmiş olarak gösterilir. Aksi takdirde silinen karakter yansılanıp silinerek ne yapılmış olduğu gösterilir (girdinin gösterildiği uçbirimlerde yararlıdır).

Bu bit sadece gösterim sırasındaki davranışı denetler; ERASE karakterinin davranışını ve girdinin silinmesini, tamamen ilgisiz olan **ECHOE**'nin ne olduğuna bakılmaksızın, **ICANON** biti denetler.

`tcflag_t` **ECHOPRT** makro

Bu bit **ECHOE** gibidir, ERASE karakteri mekanik bir uçbirimdekine benzer bir yolla gösterilir. ERASE karakterini tuşladığınızda silinen ilk karakterin öncesine bir **** karakteri basılır. Tekrar ERASE karakterini tuşlarsanız sonraki karakter silinir. Bunun ardından bir normal karakteri tuşlarsanız karakter basılmadan önce bir **/** karakteri basılır.

Bu bir BSD oluşumudur ve sadece BSD sistemleri ile GNU sisteminde vardır.

`tcflag_t` **ECHOK** makro

Bu bit, KILL karakteri normal olarak yansılandıktan sonra yeni satıra geçilmesini sağlayarak, KILL karakterinin özel bir gösterimini etkinleştirir. **ECHOKE** (aşağıda) davranışı bundan daha hoş görünür.

Bu bit yoksa, KILL karakteri yokmuşçasına yansılanır. Kullanıcı sadece KILL karakterinin önceki girdiyi sildiğini hatırlayacak, ekranda bunun belirtisini göstermeyecektir.

Bu bit sadece ekrandaki davranışı denetler; KILL karakterinin tanınması ve girdinin silinmesi sadece **ICANON** bitinin varlığına bağlıdır, **ECHOK** bu bakımdan etkili değildir.

`tcflag_t` **ECHOKE** makro

Bu bit **ECHOK** bitine benzer. Ekrandan satırın tamamının kesilerek silinmesi ile kendini gösteren KILL karakterinin özel bir gösterimini etkinleştirir. Bu bir BSD oluşumudur ve sadece BSD sistemleri ile GNU sisteminde vardır.

`tcflag_t` **ECHONL** makro

Bu bit ve **ICANON** biti varsa, satırsonu (**'\n'**) karakteri **ECHO** biti yoksa bile yansılanır.

`tcflag_t` **ECHOCTL** makro

Bu bit ve **ECHO** biti varsa, denetim karakterlerine karşı düşen karakterler **^** ile öncelenerek yansılanır. Örneğin ctrl+A, **^A** olarak yansılanır. Uçbirimde denetim karakterlerinin istenmeyen etkiler oluşturmaması için bazan etkileşimli girdi kipinde tercih edilir.

Bu bir BSD oluşumudur ve sadece BSD sistemleri ile GNU sisteminde vardır.

`tcflag_t` **ISIG** makro

Bu bit INTR, QUIT ve SUSP karakterlerinin tanınması ile ilgilidir. Bu karakterlerle ilgili işlevler sadece bu bitin varlığında ilgili işlemleri yaparlar. Girdi kipinin kurallı veya kuralsız olması bu karakterlerin yorumlanışını etkilemez.

Bu karakterlerin tanınmasını iptal ederken dikkatli olmalısınız. Yoksa uygulamalar kullanıcılar tarafından kolayca durdurulamazlar. Bu biti kaldırırsanız, yazılımınızda bu karakterlerin gönderdiği sinyalleri gönderecek ya da çıkmayı sağlayacak bir arayüz oluşturmanız gerekir. Bkz. *Sinyal Gönderen Karakterler* (sayfa: 456).

`tcflag_t` **IEXTEN** makro

POSIX.1, **IEXTEN**'i gerçekleme ile tanımlanmış manada verir, dolayısıyla bu bitin tüm sistemlerde böyle yorumlanacağından emin olamazsınız.

BSD sistemlerinde ve GNU sisteminde bu bit LNEXT ve DISCARD karakterlerini etkinleştirir. Bkz. *Diğer Özel Karakterler* (sayfa: 458).

`tcflag_t` **NOFLSH** makro

Normalde, INTR, QUIT ve SUSP karakterleri uçbirimin girdi ve çıktı kuyruklarının temizlenmesine sebep olur. Bu bit varsa, kuyruklar temizlenmez.

`tcflag_t` **TOSTOP** makro

Bu bit varsa ve sistem iş denetimini destekliyorsa, **SIGTTOU** sinyalleri uçbirime yazmaya çalışan artalan süreçleri tarafından üretilir. Bkz. *Denetim Uçbirimine Erişim* (sayfa: 717).

Aşağıdaki bitler birer BSD oluşumudur ve bunlar sadece BSD sistemleri ile GNU sisteminde vardır.

`tcflag_t` **ALTWERASE** makro

Bu bit WERASE karakterinin silme işlemini nasıl yapacağını belirler. WERASE karakteri bir sözcüğü başlangıcına kadar geriye doğru siler. Burada sorun bu başlangıcın nasıl belirleneceğidir.

Bu bit yoksa, sözcüğün başlangıcı bir boşluk karakterinden sonra gelen ilk boşluk olmayan karakterdir. Bu bit varsa, sözcüğün başlangıcı, varsa bir alfanümerik karakter, yoksa hemen ardından bir karakter gelen bir alt çizgi karakteridir.

WERASE karakteri ile ilgili daha fazla bilgi için *Girdi Düzenleme Karakterleri* (sayfa: 454) bölümüne bakınız.

`tcflag_t` **FLUSHO** makro

Bu bit DISCARD karakteri tuşlandığında konum değiştirir. Bu bit birlenirken tüm çıktı iptal edilir. Bkz. *Diğer Özel Karakterler* (sayfa: 458).

`tcflag_t` **NOKERNINFO** makro

Bu bitin varlığ STATUS karakterinin işlenmesini iptal eder. Bkz. *Diğer Özel Karakterler* (sayfa: 458).

`tcflag_t` **PENDIN** makro

Bu bit varsa, yeniden basılacak bir girdi satırı var demektir. REPRINT karakterinin tuşlanması bu bitin birlenmesine ve yeniden basma işinin bitimine kadar bir olarak kalmasına sebep olur. Bkz. *Girdi Düzenleme Karakterleri* (sayfa: 454).

4.8. Hat Hızı

Uçbirim hat hızı, bilgisayara uçbirime ne kadar hızlı okuma ve yazma yapacağını belirtmek için kullanılır.

Uçbirim gerçek bir seri hatta bağlıysa, belirttiğiniz uçbirim hızı aslında hat hızını denetler (Eğer hat hızı ile uçbirimin kendi hızı eşleşmezse iletişim gerçekleşmez). Gerçek seri portlar sadece belirli standart hızları kabul ederler. Ayrıca, bazı donanımlar standart hızların tamamını desteklemeyebilir. Hızın sıfır olarak belirtilmesi bir çevirmeli ağ bağlantısının kesilmesine ve modem denetim sinyallerinin durdurulmasına sebep olur.

Eğer uçbirim gerçek bir seri hat değilse (örneğin, bir ağ bağlantısı), hat hızı gerçekte iletim hızını etkilemez ama bazı uygulamalar boşluk doldurma miktarını saptayabilmek için bu değeri kullanabilecektir. En iyisi bir hattın hızını, asıl uçbirimin asıl hızı ile eşleşen bir değer olarak belirtmektir, ancak boşluk doldurma miktarını değiştiren farklı değerler hakkında deneyimli olmanız gerekir.

Esas olarak her uçbirim için iki hat hızı vardır: girdi ve çıktı hızı. Bunları birbirinden bağımsız olarak ayarlamak mümkünse de çoğunlukla her iki yönde de aynı hızlar kullanılır.

Hız değerleri **struct termios** yapısında saklanır ama onlara doğrudan yapı üzerinden erişmeye kalkmayın. Bu değerleri yapı içinde değiştirmek ya da okumak için aşağıdaki işlevleri kullanın:

`speed_t` **cfgetospeed**`(const struct termios *`*termios–p*`)` işlev

**termios–p* yapısında saklanan çıktı hat hızı ile döner.

`speed_t` **cfgetispeed**`(const struct termios *`*termios–p*`)` işlev

**termios–p* yapısında saklanan girdi hat hızı ile döner.

```
int cfsetospeed(struct termios *termios–p,                    işlev
                speed_t         hız)
```

Bu işlev *hız* değerini **termios–p* yapısında çıktı hızı olarak saklar. Normal dönüş değeri sıfırdır, –1 değeri bir hata oluştuğunu gösterir. Eğer *hız* bir hız değeri belirtmiyorsa işlev –1 ile döner.

```
int cfsetispeed(struct termios *termios–p,                    işlev
                speed_t         hız)
```

Bu işlev *hız* değerini **termios–p* yapısında girdi hızı olarak saklar. Normal dönüş değeri sıfırdır, –1 değeri bir hata oluştuğunu gösterir. Eğer *hız* bir hız değeri belirtmiyorsa işlev –1 ile döner.

```
int cfsetspeed(struct termios *termios–p,                                    işlev
               speed_t         hız)
```

Bu işlev *hız* değerini **termios–p* yapısında hem girdi hem de çıktı hızı olarak saklar. Normal dönüş değeri sıfırdır, –1 değeri bir hata oluştuğunu gösterir. Eğer *hız* bir hız değeri belirtmiyorsa işlev –1 ile döner. Bu işlev bir BSD oluşumudur.

```
speed_t                                                                      veri türü
```

speed_t hat hızını göstermek için kullanılan bir işaretsiz tamsayı veri türüdür.

cfsetospeed ve **cfsetispeed** işlevleri hataları sadece hat hızları sistem tarafından desteklenmediğinde raporlar. Eğer hat hızlarını temel olarak kabul edilebilir değerlerde belirtirseniz bu işlevler başarılı olur. Fakat işlevler belli bir donanım aygıtının bu hat hızlarını destekleyip desteklemediğine bakmaz, aslında hangi aygıtın kullanılacağını da bilmezler. Eğer **tcsetattr** işlevi ile belli bir aygıta bir hız belirtirseniz ve bu aygıt bu hızı desteklemiyorsa işlev –1 ile dönecektir.

Taşınabilirlik Bilgisi

GNU kütüphanesinde, yukarıda bahsedilen işlevler hızları saniyede bit sayısı olarak kabul eder ve saniyede bit sayısı olarak döndürürler. Diğer kütüphanelerde hızların belirli kodlarla belirtilmesi gerekir. POSIX.1 taşınabilirliği için, hızı aşağıdaki sembolleri kullanarak belirtmelisiniz, onların sayısal değerleri sisteme bağımlıdır fakat her isim belli bir anlama gelir: **B110**=> 100 bps, **B300**=> 300 bps, ... böyle gider. Bu hızları taşınabilir olarak belirtmenin başka bir yolu yoktur. Bunlar sadece seri hatların desteklediği, sıklıkla kullanılan hızları belirtirler.

```
B0  B50  B75  B110  B134  B150  B200
B300  B600  B1200  B1800  B2400  B4800
B9600  B19200  B38400  B57600  B115200
B230400  B460800
```

BSD bunlara ek olarak iki hız sembolünü takma ad olarak tanımlar: **EXTA** sembolü **B19200** için, **EXTB** sembolü de **B38400** için bir takma addır. Bu semboller atıl olmuştur.

4.9. Özel Karakterler

Kurallı girdi kipinde uçbirim sürücüsü çeşitli denetim işlevlerini yerine getiren bir miktar özel karakter tanır. Bunlar ERASE (<**DEL**> tuşu) karakterinin de dahil olduğu metin düzenleme karakterleridir. **SIGINT** sinyalini gönderen INTR karakteri (normalde **C-c**) ve sinyal gönderen diğer karakterler hem kurallı hem de kuralsız kipte kullanılabilirler. Bu karakterlerin hepsi bu bölümde açıklanmıştır.

Kullanılacak karakterler **struct termios** yapısının **c_cc** üyesinde belirtilir. Bu üye bir dizidir; her eleman belli bir rol için bir karakter belirtir. Her eleman için elemanın indisini belirten bir sembolik sabit tanımlanmıştır. Örneğin, **VINTR** sembolü INTR karakterini belirten elemanın indisidir; *termios*`.c_cc[VINTR]` içinde **'='** karakteri saklanmışsa INTR karakteri **'='** karakteri olur.

Bazı sistemlerde, özel bir karakterin işlevini, bu rol için **_POSIX_VDISABLE** değerini belirterek iptal edebilirsiniz. Bu değer herhangi bir karakter kodunun karşılığı değildir. Kullandığınız işletim sisteminin **_POSIX_VDISABLE** desteği olup olmadığının nasıl belirtildiği ile ilgili bilgileri *Dosya Desteği Seçenekleri* (sayfa: 796) bölümünde bulabilirsiniz.

4.9.1. Girdi Düzenleme Karakterleri

Bu özel karakterler sadece kurallı girdi kipinde etkindir. Bkz. *İki Girdi Tarzı: Kurallı veya Kuralsız* (sayfa: 443).

```
int VEOF                                                                     makro
```

Özel denetim karakterleri dizisinin dosyasonu (EOF) karakterini içeren elemanının indisidir. Karakter *termios*.c_cc[VEOF] içindedir.

Dosyasonu karakteri sadece kurallı girdi kipinde tanınır. Tıpkı satırsonu karakteri gibi bir satırı sonlandırmasına rağmen dosyasonu karakteri dosyadaki son satırın ilk ve son karakteridir. **read** ile yapılan okumada dosyasonunu belirrtmek üzere dönen karakter sayısı sıfır olur, dosyasonu karakterinin kendisi iptal edilir.

Genellikle dosyasonu karakteri **C-d** ile elde edilir.

`int` **VEOL** makro

Özel denetim karakterleri dizisinin satırsonu (EOL) karakterini içeren elemanının indisidir. Karakter *termios*.c_cc[VEOL] içindedir.

EOL karakteri sadece kurallı girdi kipinde tanınır. Tıpkı satırsonu karakteri gibi bir satırı sonlandırmakta kullanılır. EOL karakteri iptal edilmez, girdi satırındaki son karakter olarak okunur.

Satır sonunda <**RET**> tuşuna basarak EOL karakterini kullanmaya ihtiyaç yoktur. ICRNL seçeneğinin etkin olması yeterlidir. Aslında bu öntanımlı sistem işlerindendir.

`int` **VEOL2** makro

Özel denetim karakterleri dizisinin ikincil satırsonu (EOL2) karakterini içeren elemanının indisidir. Karakter *termios*.c_cc[VEOL2] içindedir.

EOL2 karakteri EOL karakter (yukarıda) gibidir, ama farklı bir karakterdir. Bu bir satırı sonlandırmak için iki karakter gerektiği durumda belirtilebilir. Birini EOL diğerini EOL2 karakteri olarak belirtebilirsiniz.

EOL2 karakteri bir BSD oluşumudur; sadece BSD sistemleri ile GNU sisteminde vardır.

`int` **VERASE** makro

Özel denetim karakterleri dizisinin silme (ERASE) karakterini içeren elemanının indisidir. Karakter *termios*.c_cc[VERASE] içindedir.

ERASE karakteri sadece kurallı girdi kipinde tanınır. Kullanıcı silme karakterini tuşladığında önceki karakter silinir. (Uçbirim çokbaytlı karakterler üretiyorsa girdide birden fazla bayt silinebilir.) Bu karakter satırın başına kadar silme yapan karakter olarak kullanılamaz. Karakterin kendisi girdide iptal edilir.

Genellikle ERASE karakteri <**DEL**> ile elde edilir.

`int` **VWERASE** makro

Özel denetim karakterleri dizisinin sözcük silme (WERASE) karakterini içeren elemanının indisidir. Karakter *termios*.c_cc[VWERASE] içindedir.

WERASE karakteri sadece kurallı girdi kipinde tanınır. Kullanıcı bu karakteri tuşladığında önceki sözcük ve sonraki boşluklar silinir; sözcükten önceki boşluklara dokunulmaz.

"Sözcük" (word) tanımı **ALTWERASE** kipine bağımlıdır. Bkz. *Yerel Kipler* (sayfa: 451).

ALTWERASE kipi etkin değilse bir sözcük boşluk ve sekmeleri içeremez.

ALTWERASE kipi etkinse, bir sözcük harfleri rakamlar ve altçizgi karakterlerinden oluşabilir ve isteğe bağlı olarak harf, rakam veya altçizgi olmayan bir karakterle bitebilir.

Genellikle WERASE karakteri **C-w** ile elde edilir.

Bu bir BSD oluşumudur.

`int` **VKILL** makro

Özel denetim karakterleri dizisinin KILL karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VKILL]` içindedir.

KILL (satır silme) karakteri sadece kurallı girdi kipinde tanınır. Kullanıcı KILL karakterini tuşladığında etkin satırın tamamı silinir. Bu arada KILL karakterinin kendisi de silinir.

Genellikle KILL karakteri **C-u** ile elde edilir.

`int` **VREPRINT** makro

Özel denetim karakterleri dizisinin REPRINT karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VREPRINT]` içindedir.

REPRINT karakteri sadece kurallı girdi kipinde tanınır. O anki girdi satırının tekrarlanmasını sağlar. Bu karakteri tuşlarken bir yandan da eşzamansız çıktı geliyorsa bu karakter yazdığınız satırı temiz olarak yeniden görmenizi sağlar.

Genellikle REPRINT karakteri **C-r** ile elde edilir.

Bu bir BSD oluşumudur.

4.9.2. Sinyal Gönderen Karakterler

Bu özel karakterler hem kurallı hem de kuralsız girdi kipinde etkin olabilir ancak sadece **ISIG** seçeneği etkinse bu mümkündür (Bkz. *Yerel Kipler* (sayfa: 451)).

`int` **VINTR** makro

Özel denetim karakterleri dizisinin INTR karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VINTR]` içindedir.

INTR (interrupt – kesme) karakteri, uçbirimle ilişkili önalan işindeki tüm süreçler için **SIGINT** sinyalinin yayınlanmasına sebep olur. Bu arada INTR karakterinin kendisi silinir. Sinyaller ile ilgili daha fazla bilgi için *Sinyal İşleme* (sayfa: 601) bölümüne bakınız.

Genellikle INTR karakteri **C-c** ile elde edilir.

`int` **VQUIT** makro

Özel denetim karakterleri dizisinin QUIT karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VQUIT]` içindedir.

QUIT karakteri, uçbirimle ilişkili önalan işindeki tüm süreçler için **SIGQUIT** sinyalinin yayınlanmasına sebep olur. Bu arada QUIT karakterinin kendisi silinir. Sinyaller ile ilgili daha fazla bilgi için *Sinyal İşleme* (sayfa: 601) bölümüne bakınız.

Genellikle QUIT karakteri **C-** ile elde edilir.

`int` **VSUSP** makro

Özel denetim karakterleri dizisinin SUSP karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VSUSP]` içindedir.

SUSP (suspend – bekletme, askıya alma, süreci artalana atma) karakteri sadece gerçekleme *iş denetimini* (sayfa: 716) destekliyorsa tanınır. SUSP karakteri, uçbirimle ilişkili önalan işindeki tüm süreçler için **SIGTSTP** sinyalinin yayınlanmasına sebep olur. Bu arada SUSP karakterinin kendisi silinir. Sinyaller ile ilgili daha fazla bilgi için *Sinyal İşleme* (sayfa: 601) bölümüne bakınız.

Genellikle SUSP karakteri **C-z** ile elde edilir.

Bazı uygulamalar SUSP karakterinin normal yorumunu iptal eder. Sizin yazılımınız da bunu yapıyorsa, kullanıcının işi durdurabilmesini sağlayacak mekanizmaları sağlamanız gerekir. Kullanıcı bu mekanizmayı devreye soktuğunda yazılım sadece kendi sürecine değil, süreç grubundaki tüm süreçlere **SIGTSTP** sinyalini göndermelidir (Bkz. *Başka Bir Sürece Sinyal Gönderme* (sayfa: 628)).

`int` **VDSUSP** makro

Özel denetim karakterleri dizisinin DSUSP karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VDSUSP]` içindedir.

DSUSP (suspend – bekletme, askıya alma, süreci artalana atma) karakteri sadece gerçekleme *iş denetimini* (sayfa: 716) destekliyorsa tanınır. SUSP karakteri gibi **SIGTSTP** sinyalini gönderir ama bunu doğrudan göndererek yapmaz, yazılım onu bir girdi olarak okumaya çalışıyorsa bu gerçekleşir. Bütün sistemlerin iş denetimleri DSUSP karakterini desteklemez; sadece BSD uyumlu sistemler ile GNU sisteminde desteklenir.

Sinyaller ile ilgili daha fazla bilgi için *Sinyal İşleme* (sayfa: 601) bölümüne bakınız.

Genellikle DSUSP karakteri **C-y** ile elde edilir.

4.9.3. Akış Denetimi için Özel Karakterler

Bu karakterler hem kurallı hem de kuralsız girdi kipinde etkili olabilir, fakat kullanımları **IXON** ve **IXOFF** seçenekleri ile denetlenir (Bkz. *Girdi Kipleri* (sayfa: 447)).

`int` **VSTART** makro

Özel denetim karakterleri dizisinin START karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VSTART]` içindedir.

START karakteri **IXON** ve **IXOFF** girdi kiplerini desteklemek için kullanılır. **IXON** varsa, START karakteri alındığında çıktı bekleme durumundan çıkarılır ve START karakterinin kendisi iptal edilir. **IXANY** varsa, herhangi bir karakter alındığında çıktı bekleme durumundan çıkarılır. **IXOFF** varsa, START karakterini sistem ayrıca uçbirime de iletebilir.

Genellikle START karakteri **C-q** ile elde edilir. Bu değeri değiştirmeniz mümkün değildir, çünkü donanım ne belirttiğinize bakmaksızın **C-q** kullanacağınızı varsayacakır.

`int` **VSTOP** makro

Özel denetim karakterleri dizisinin STOP karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VSTOP]` içindedir.

STOP karakteri **IXON** ve **IXOFF** girdi kiplerini desteklemek için kullanılır. **IXON** varsa, bir STOP karakterinin alınması çıktının bekletilmesine sebep olur ve STOP karakteri iptal edilir. **IXOFF** varsa, sistem girdi kuyruğunun taşmasını önlemek için STOP karakterini uçbirime de iletir .

Genellikle STOP karakteri **C-s** ile elde edilir. Bu değeri değiştirmeniz mümkün değildir, çünkü donanım ne belirttiğinize bakmaksızın **C-s** kullanacağınızı varsayacakır.

4.9.4. Diğer Özel Karakterler

Bu özel karakterler sadece BSD sistemlerinde ve GNU sisteminde mevcuttur.

`int` **`VLNEXT`** makro

Özel denetim karakterleri dizisinin LNEXT karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VLNEXT]` içindedir.

LNEXT karakteri hem kurallı hem de kuralsız girdi kipinde sadece **`IEXTEN`** seçeneği etkin olduğunda anlamlıdır. Kullanıcının bu karakterden sonra tuşladığı özel bir karakterin kıymeti harbiyesini yokeder. Karakter normalde bazı düzenleme işlemlerini yapma ya da sinyal gönderme yeteneğine sahip olsa bile sıradan bir karakter olarak okunur. Bu Emacs'ın **`C-q`** komutuna eşdeğerdir ve "LNEXT" "literal next" kısaltmasıdır.

Genellikle LNEXT karakteri **`C-v`** ile elde edilir.

`int` **`VDISCARD`** makro

Özel denetim karakterleri dizisinin DISCARD karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VDISCARD]` içindedir.

DISCARD karakteri hem kurallı hem de kuralsız girdi kipinde sadece **`IEXTEN`** seçeneği etkin olduğunda anlamlıdır. Çıktı iptal seçeneğinin durumunu değiştirir. Seçenek varsa, tüm yazılım çıktısı iptal edilir. Seçeneğin varlığı ayrıca o an çıktı tamponundaki tüm çıktının iptal edilmesine sebep olur. Herhangi bir başka karakterin tuşlanması seçeneği sıfırlar.

`int` **`VSTATUS`** makro

Özel denetim karakterleri dizisinin STATUS karakterini içeren elemanının indisidir. Karakter *termios*`.c_cc[VSTATUS]` içindedir.

STATUS karakterinin etkisi, çalışan süreç hakkında o anki durum iletisinin basılmasıdır.

STATUS karakteri sadece kurallı girdi kipinde ve sadece **`NOKERNINFO`** yoksa tanınır.

4.10. Kuralsız Girdi

Kuralsız girdi kipinde ERASE ve KILL gibi metin düzenleme karakterleri yoksayılır. Kullanıcının girdiyi düzenleyebildiği sistem oluşumları kuralsız kipte iptal edilir. Sinyal gönderme ve akış denetimi hariç tüm girdi karakterleri uygulamaya oldukları gibi yazıldıkları şekilde aktarılırlar. Metin düzenleme ile ilgili girdilerin alınması gerekliyse bunlar yazılım içinde gerçeklenmelidir.

Kuralsız kipte kaç girdinin ne kadar süre bekleneceğini denetleyen MIN ve TIME parametreleri vardır. Bunları kullanarak girdi yapıldıkça hiç beklemeden girdileri hemen işleme sokabilirsiniz.

MIN ve TIME **`struct termios`** yapısının üyesi olan **`c_cc`** dizisinde saklanır. Bu dizinin her elemanı belli bir role sahiptir ve her elemana rolü ifade eden bir sembolik sabit indis olarak kullanılarak erişilir. MIN ve TIME için bu indisler **`VMIN`** ve **`VMAX`**'tır.

`int` **`VMIN`** makro

`c_cc` dizisinin MIN değerini içeren elemanının indisidir. Değer *termios*`.c_cc[VMIN]` içindedir.

MIN değeri sadece kuralsız girdi kipinde anlamlıdır; girdi kuyruğundan **`read`** ile okuma yapabilmek için en az kaç karakter gerektiğini belirleyen bir sayıdır.

int **VTIME**

```
int VTIME                                                    makro
```

c_cc dizisinin TIME değerini içeren elemanının indisidir. Değer *termios*.c_cc[VTIME] içindedir.

TIME değeri sadece kuralsız girdi kipinde anlamlıdır; girdi kuyruğundan **read** ile okuma yapabilmek için ne kadar süreyle bekleme yapılacağını belirler; 0.1 saniyelik katları ifade eden bir tamsayı değerdir.

MIN ve TIME değerleri **read** işlevinin değer döndürmesi için gerekli kriterleri oluşturur. Sıfır ve sıfırdan farklı değerler ayrıca anlamlıdır. Bu bakımdan dört olası durum vardır:

- TIME ve MIN, ikisi de sıfırdan farklı olabilir.

  Bu durumda TIME, girdi yapılırken her girdi karakterinden sonra ne kadar süre bekleneceğini belirtir. İlk karakter alındıktan sonra MIN bayt alınıncaya kadar her karakter için TIME kadar süre beklenir, hangisi önce dolarsa **read** okunan karakterlerle döner.

  read ilk karakter alınıncaya kadar TIME süresi dolsa bile daima bekler, ancak kuyrukta MIN karakterden fazlası varsa işlevden istenen karakter sayısına göre MIN karakterden fazlası dönebilir.

- TIME ve MIN, ikisi de sıfır olabilir.

  Bu durumda **read** kuyrukta kaç karakter varsa hepsini istek sayısı uzunlukta anında döndürür. Bir girdi yoksa **read** sıfır değeriyle döner.

- MIN sıfır, TIME sıfırdan farklı olabilir.

  Bu durumda, **read** ilk girdi için TIME kadar süreyle bekler ve kuyrukta kaç karakter varsa hepsini istek sayısı uzunlukta anında döndürür, bir girdi yoksa **read** sıfır değeriyle döner.

- TIME sıfır, MIN sıfırdan farklı olabilir.

  Bu durumda **read** kuyruğa MIN bayt yazılıncaya kadar bekler ve istenen sayıda karakterle döner. Eğer kuyrukta MIN karakterden fazla karakter varsa MIN karakterden fazlası ile dönebilir.

MIN için 50 belirtirken işlevden 10 karakter okumasını istersek ne olacak? Normalde, **read** tamponda 50 karakter birikene kadar (ya da yukarıda açıklanan kurallar neyi gerektiriyorsa) bekler ve ilk 10 karakterle döner. Kalan 40 karakter sonraki **read** çağrıları ile okunmak üzere tamponda bırakılır.

Taşınabilirlik Bilgisi

Bazı sistemlerde MIN ve TIME alanları EOF ve EOL alanları ile aynıdır. MIN ve TIME değerleri sadece kuralsız kipte anlamlı iken EOF ve EOL sadece kurallı kipte anlamlı olduğundan bu durum çeşitli sorunlara yol açar. GNU kütüphanesinde bu alanlar ayrıdır.

```
void cfmakeraw(struct termios *termios_p)                    işlev
```

Bu işlev, uçbirimi BSD'de "temel kip" (raw mode) denilen kipe sokmak için kolay bir yol olarak kullanılır. Bu aslında bir kuralsız girdi kipidir ve uçbirimi çoğu işlem bakımından işlevsiz bir kanal haline getirir.

İşlev tam olarak şunları yapar:

```
termios_p->c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP
                        |INLCR|IGNCR|ICRNL|IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO|ECHONL|ICANON|ISIG|IEXTEN);
```

```
termios_p->c_cflag &= ~(CSIZE|PARENB);
termios_p->c_cflag |= CS8;
```

5. BSD Uçbirim Kipleri

Uçbirim kiplerini okumak ve belirlemek için genelde *Uçbirim Kipleri* (sayfa: 444) bölümünde açıklanan işlevler yararlıdır. Bununla birlikte, bazı sistemlerde bazı şeyleri yapabilmek için bu bölümde anlatılan ve BSD sisteminden alınmış işlevleri kullanabilirsiniz. Çoğu sistemde bu işlevler mevcut değildir. Hatta GNU C kütüphanesinde bu işlevler, içlerinde Linux'un da bulunduğu çoğu çekirdek ile **errno**= **ENOSYS** hatasını vererek başarısız olacaktır.

Bu bölümde bahsedilen semboller `sgtty.h` başlık dosyasında bildirilmiştir.

`struct` **sgttyb** veri türü

Bu yapı **gtty** ve **stty** için girdi ve çıktı parametreleri listesidir.

`char` **sg_ispeed**
Girdi için hat hızı

`char` **sg_ospeed**
Çıktı için hat hızı

`char` **sg_erase**
Silme karakteri

`char` **sg_kill**
Satır silme karakteri

`int` **sg_flags**
Çeşitli seçenekler

`int` **gtty**`(int` *dosyatanıtıcı*`,` `struct sgttyb` **öznitelikler*`)` işlev

Uçbirimin özniteliklerini okur.

gtty işlevi *dosyatanıtıcı* dosya tanıtıcısı ile açılan uçbirimin özniteliklerini **öznitelikler* ile gösterilen yapı içinde döndürür.

`int` **stty**`(int` *dosyatanıtıcı*`,` `struct sgttyb` **öznitelikler*`)` işlev

Uçbirimin özniteliklerini değiştirir.

stty işlevi *dosyatanıtıcı* dosya tanıtıcısı ile açılan uçbirimi **öznitelikler* ile belirtilen özniteliklerle ayarlar.

6. Hat Denetim İşlevleri

Bu işlevleri uçbirim aygıtı üzerinde çeşitli denetim eylemlerini gerçekleştirir. Uçbirim erişimi ile ilgili olarak, bunlar çıktıda şöyle birşeyler yapıyor gibi ele alınır: Bir artalan süreci kendi denetim uçbirimi üzerinde bu işlevlerden birini kullandığında, normalde süreç grubundaki tüm süreçler bir **SIGTTOU** sinyali gönderir. Çağıran süreç **SIGTTOU** sinyallerini engelliyor ya da yoksayıyorsa, bu durumda işlem yine yapılır ama sinyal gönderilmez. Bkz. *İş Denetimi* (sayfa: 716).

`int` **tcsendbreak**`(int` *dosyatanıtıcı*`,` `int` *süre*`)` işlev

Bu işlev, *dosyatanıtıcı* dosya tanıtıcısı ile ilişkili uçbirim üzerinde sıfır bitlerinden oluşan bir akımı ileterek bir geçici kesme durumu (break condition) oluşturur. Geçici kesme durumunu süresi *süre* argümanınca denetlenir. Sıfırsa süre 0.25 ile 0.5 saniye arasında olur. Bu, sıfırdan farklı değerlerin sisteme bağlı olduğu anlamına gelir.

Bu işlev, eğer uçbirim bir eşzamasız seri veri portu değilse hiçbir şey yapmaz.

Normalde dönüş değeri sıfırdır. Bir hata oluştuğunda –1 döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* geçerli bir dosya tanıtıcı değil

ENOTTY
: *dosyatanıtıcı* bir uçbirimle ilişkili değil

`int` **tcdrain**`(int` *dosyatanıtıcı*`)` işlev

tcdrain işlevi kuyruktaki çıktı *dosyatanıtıcı* uçbirimine iletilinceye kadar bekler.

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **tcdrain** çağrısı sırasında evre bazı özkaynakları (bellek, dosya tanıtıcı, semafor, vb.) ayırdığında bu bir sorun olur. Evre tam bu anda bir iptal alırsa ayrılan özkaynaklar yazılım sonlanana kadar ayrılmış olarak kalır. Bu tür **tcdrain** çağrılarından kaçınmak için iptal eylemcileri kullanılarak korunulmalıdır.

Normalde dönüş değeri sıfırdır. Bir hata oluştuğunda –1 döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* geçerli bir dosya tanıtıcı değil

ENOTTY
: *dosyatanıtıcı* bir uçbirimle ilişkili değil

EINTR
: İşlem bir sinyalle durduruldu. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

`int` **tcflush**`(int` *dosyatanıtıcı*`,`
`int` *kuyruk*`)` işlev

tcflush işlevi *dosyatanıtıcı* dosya tanıtıcısı ile ilişkili uçbirimin girdi ya da çıktı kuyruğundaki veriyi temizlemek için kullanılır. *kuyruk* argümanı temizlenecek kuyruğu belirtmek için kullanılır ve şu değerlerden biri olabilir:

TCIFLUSH
: Alınmış ama henüz okunmamış veri temizlenir.

TCOFLUSH
: Yazılmış ama henüz iletilmemiş veri temizlenir.

TCIOFLUSH
: Girdi ve çıktı kuyruklarının ikisi de temizlenir.

Normalde dönüş değeri sıfırdır. Bir hata oluştuğunda –1 döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* geçerli bir dosya tanıtıcı değil

ENOTTY
: *dosyatanıtıcı* bir uçbirimle ilişkili değil

EINVAL
: *kuyruk* argümanı olarak belirtilen değer geçersiz

Bu işlevin ismine bakarak hatırlamak zor olacaktır, çünkü "flush" (boşaltma) işlemi normalde önceki işlev için kullanılır ve girdi ve çıktının iptal edimesi ile çelişir. **tcflush** işlevi POSIX standardında belirtildiğinden maalesef ismini değiştiremiyoruz.

```
int tcflow(int dosyatanıtıcı,                                    işlev
           int eylem)
```

tcflow işlevi *dosyatanıtıcı* dosya tanıtıcısı ile ilişkili uçbirim üzerinde XON/XOFF akış denetimi ile ilgili işlemleri gerçekleştirmekte kullanılır.

eylem argümanı gerçekleştirilecek eylemi belirtmek için kullanılır ve aşağıdaki değerlerden biri olabilir:

TCOOFF
: Çıktı iletimi beklemeye alınır.

TCOON
: Çıktı iletimi yeniden başlatılır.

TCIOFF
: STOP karakteri iletilir.

TCION
: START karakteri iletilir.

STOP ve START karakterleri hakkında daha ayrıntılı bilgiyi *Özel Karakterler* (sayfa: 454) bölümünde bulabilirsiniz.

Normalde dönüş değeri sıfırdır. Bir hata oluştuğunda –1 döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* geçerli bir dosya tanıtıcı değil

ENOTTY
: *dosyatanıtıcı* bir uçbirimle ilişkili değil

EINVAL
: *eylem* argümanı ile belirtilen değer geçersiz

7. Kuralsız Kip Örneği

Bu örnekte, kuralsız kipte tek bir karakteri bir uçbirimden okumak için ne yapılması gerektiği gösterilmiştir:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
```

```
/* Uçbirim özniteliklerini bir değişkende saklayalım. */

struct termios saved_attributes;

void
reset_input_mode (void)
{
  tcsetattr (STDIN_FILENO, TCSANOW, &saved_attributes);
}

void
set_input_mode (void)
{
  struct termios tattr;
  char *name;

  /* stdin bir uçbirim mi. */
  if (!isatty (STDIN_FILENO))
    {
      fprintf (stderr, "Bir uçbirim değil.\n");
      exit (EXIT_FAILURE);
    }

  /* Uçbirim özniteliklerini daha sonra yerine koymak için saklayalım. */
  tcgetattr (STDIN_FILENO, &saved_attributes);
  atexit (reset_input_mode);

  /* Şimdi uçbirimi kuralsız kipe sokalım.
     Girdinin tekrar yansılanmaması için yansılamayı da kapatacağız. */
  tcgetattr (STDIN_FILENO, &tattr);
  tattr.c_lflag &= ~(ICANON|ECHO); /* ICANON ve ECHO temizlendi. */
  tattr.c_cc[VMIN] = 1;                /* Girdi tek karakterlik okunsun. */
  tattr.c_cc[VTIME] = 0;               /* Okuma için beklenmesin. */
  tcsetattr (STDIN_FILENO, TCSAFLUSH, &tattr);
}

int
main (void)
{
  char c;

  set_input_m
  char c;

  set_input_mode ();

  while (1)
    {
      read (STDIN_FILENO, &c, 1);
      if (c == '\004')          /* C-d */
        break;
      else
        write (STDOUT_FILENO, &c, 1);
    }

  return EXIT_SUCCESS;
```

```
}
```

Bu yazılım, bir sinyalle sonlandırıldığında ya da çıkarken özgün uçbirim kiplerini tekrar yerine koyar. **exit** ile çıkılırken **atexit** işlevi ile bunu yapar (bkz. *Çıkışta Temizlik* (sayfa: 682)).

Kabuğun süreç durdurulurken ve başlatılırken uçbirim kiplerini sıfırladığı varsayılmıştır; bkz. *İş Denetimi* (sayfa: 716). Fakat bazı kabuklar bunu yapmaz, bu durumda uçbirim kiplerini sıfırlamak için iş denetim sinyallerine eylemci oluşturmanız gerekebilir.

8. Uçbirimsiler

Bir ***uçbirimsi*** (pseudo terminal), bir uçbirim gibi davranan özel bir süreçlerarası iletişim kanalıdır. Kanalın bir ucu *ana* uç ya da *ana uçbirimsi aygıt* olarak adlandırılırken diğer uç *yardımcı* uç adını alır. Ana uca yazılan veri yardımcı uç tarafından sıradan bir uçbirime kullanıcı tarafından yazılmış gibi alınır ve yardımcı uca yazılan veri ana uca sıradan bir terminale yazılmış gibi gönderilir.

Uçbirimsiler genellikle **xterm** ve **emacs** gibi uygulamalar tarafından uçbirim benzetimi amacıyla kullanılırlar.

8.1. Uçbirimsilerin Ayrılması

Bu bölümde bir uçbirim ayırmak için kullanılan işlevlerden sözedilecektir. Bu işlevler `stdlib.h` başlık dosyasında bildirilmiştir.

`int` **getpt**`(void)` işlev

getpt işlevi kullanılabilir ilk ana uçbirimsi için yeni bir dosya tanıtıcı ile döner. İşlevin normal dönüş değeri negatif olmayan bir tamsayı olarak dosya tanıtıcıdır. Bir hata oluşması durumunda –1 ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

ENOENT
: Serbest bir ana uçbirimsi yok

Bu işlev bir GNU oluşumudur.

`int` **grantpt**`(int` *dosyatanıtıcı*`)` işlev

grantpt işlevi *dosyatanıtıcı* dosya tanıtıcısı ile ilişkili ana uçbirimsi aygıtının diğer ucu olan yardımcı uçbirimsinin sahipliğini ve erişim izinlerini değiştirir. Sahiplik, işlevi çağıran sürecin gerçek kullanıcı kimliği olur (bkz. *Bir Sürecin Aidiyeti* (sayfa: 743)). Erişim yetkileri ise sahibi tarafından yazılabilir/okunabilir ve sadece grubu tarafından okunabilir olarak ayarlanır.

Bazı sistemlerde bu işlev özel bir **setuid** root yazılım çağrılarak gerçeklenir (bkz. *Bir Sürecin Aidiyeti Nasıl Değiştirilir?* (sayfa: 743)). Dolayısıyla, **SIGCHLD** sinyali için bir sinyal eylemci oluşturulması bir **grantpt** çağrısı ile çelişebilir (bkz. *İş Denetim Sinyalleri* (sayfa: 608)).

İşlevin normal dönüş değeri sıfırdır. Bir hata oluşması durumunda –1 ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* geçerli bir dosya tanıtıcı değil

EINVAL
: *dosyatanıtıcı* bir ana uçbirimsi ile ilişkili değil

EACCES
: *dosyatanıtıcı* ile ilişkili olan ana uçbirimsinin karşı ucu olan yardımcı uçbirimsiye erişilemiyor.

`int unlockpt(int dosyatanıtıcı)` işlev

unlockpt işlevi *dosyatanıtıcı* dosya tanıtıcısı ile ilişkili ana uçbirimsi aygıtının diğer ucu olan yardımcı uçbirimsinin kilidini kaldırır. Bazı sistemlerde, yardımcı uçbirimsi sadece kilidi kaldırıldığında açılabilir, bu nedenle taşınabilir uygulamalar yardımcı uçbirimsiyi açarken daima bir **unlockpt** çağrısı yapmalıdır.

İşlevin normal dönüş değeri sıfırdır. Bir hata oluşması durumunda –1 ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* geçerli bir dosya tanıtıcı değil

EINVAL
: *dosyatanıtıcı* bir ana uçbirimsi ile ilişkili değil

`char *ptsname(int dosyatanıtıcı)` işlev

dosyatanıtıcı dosya tanıtıcısı bir ana uçbirimsi aygıtı ile ilişkili ise, **ptsname** işlevi ilişkili yardımcı uçbirimsinin dosya ismini, boş karakter sonlandırmalı durağan ayrılmış bir dizgeye gösterici ile döndürür. Bu dizge daha sonraki **ptsname** çağrıları ile değişebilir.

```
int ptsname_r(int    dosyatanıtıcı,                                  işlev
              char   *tampon,
              size_t uzunluk)
```

ptsname_r işlevi **ptsname** işlevine benzemekle birlikte, sonucu kullanıcı tarafından belirtilen *uzunluk* uzunluktaki *tampon* içinde döndürmesi ile farklıdır.

Bu işlev bir GNU oluşumudur.

Taşınabilirlik Bilgisi

System V'den türetilmiş sistemlerde, **ptsname** ve **ptsname_r** işlevleri akım temelli olabilir ve bu nedenle onları açtıktan sonra, kullanmadan önce bir uçbirimsi olarak davranmaları için ek işlemler gerekebilir.

Bu işlevlerin genel kullanım biçimi örnekte gösterilmiştir:

```
int
open_pty_pair (int *amaster, int *aslave)
{
  int master, slave;
  char *name;

  master = getpt ();
  if (master < 0)
    return 0;

  if (grantpt (master) < 0 || unlockpt (master) < 0)
    goto close_master;
  name = ptsname (master);
  if (name == NULL)
    goto close_master;

  slave = open (name, O_RDWR);
  if (slave == -1)
    goto close_master;
```

```
  if (isastream (slave))
    {
      if (ioctl (slave, I_PUSH, "ptem") < 0
          || ioctl (slave, I_PUSH, "ldterm") < 0)
        goto close_slave;
    }

  *amaster = master;
  *aslave = slave;
  return 1;

close_slave:
  close (slave);

close_master:
  close (master);
  return 0;
}
```

8.2. Bir Uçbirimsi Çiftinin Açılması

Bu işlevler BSD'den alınmıştır, ayrı bir kütüphane olarak **libutil** kütüphanesinde bulunur ve pty.h başlık dosyasında bildirilmişlerdir.

```
int openpty(int             *ana,                                    işlev
            int             *yardımcı,
            char            *isim,
            struct termios  *termp,
            struct winsize  *winp)
```

Bu işlev bir uçbirimsi çiftini ayırdıktan sonra açar ve ana uçbirimsinin dosya tanıtıcısını *ana* göstericisinde, yardımcı uçbitimsinin dosya tanıtıcısını *yardımcı* göstericisinde döndürür. *isim* argümanı bir boş gösterici değilse, yardımcı uçbirimsi aygıtının dosya ismi *isim* içinde saklanır. *termp* bir boş gösterici değilse, yardımcı uçbirimsinin uçbirim öznitelikleri *termp* ile gösterilen yapıdan ayarlanır (bkz. *Uçbirim Kipleri* (sayfa: 444)). Benzer şekilde, *winp* bir boş gösterici değilse, uçbirimsinin ekran boyutları *winp* ile gösterilen yapıdan ayarlanır.

İşlevin normal dönüş değeri sıfırdır. Bir hata oluşması durumunda –1 ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

ENOENT
: Serbest uçbirimsi çifti yok

Uyarı

openpty işlevinin *isim* argümanının **NULL** olması *çok tehlikelidir*, çünkü işlev *isim* dizgesinin taşmasına karşı korunmamıştır. Yardımcı uçbirimsi aygıtının ismini bulmak için işlevdeki argüman yerine **ttyname** işlevini **openpty** tarafından döndürülen *yardımcı* dosya tanıtıcısı ile kullanmalısınız.

```
int forkpty(int             *ana,                                    işlev
            char            *isim,
            struct termios  *termp,
            struct winsize  *winp)
```

Bu işlev **openpty** işlevine benzer, ek olarak *yeni bir süreç çatallar* (sayfa: 687) ve yeni açılan yardımcı uçbirimsi aygıtını *alt sürecin denetim uçbirimi yapar* (sayfa: 717).

İşlem başarılı olursa hem çağıran hem de alt süreç işlevin döndüğünü görür fakat işlev farklı değerlerle döner: alt süreçte 0 değeriyle dönerken, çağıran sürece alt sürecin süreç kimliğini döndürür.

Uçbirimsi çifti ayrılamamışsa ya da alt süreç oluşturulamamışsa işlev başarısız olmuş demektir, bu durumda işlev çağrıldığı sürece –1 değerini döndürür.

Uyarı

forkpty işlevinde de **openpty** gibi *isim* argümanı sorunu vardır.

XVIII. Syslog

İçindekiler

Bu oylumda sistem yönetimini ilgilendiren iletilerin günlüklenmesi ve çıktılanması ile ilgili oluşumlardan bahsedilecektir. Bu oylumda bahsedilen oluşumlar yazılımların kullanıcılarına ileti çıktılaması ve özel günlükler tutması ile ilgili hiçbir şey yapmazlar (Bunları yapan oluşumlar *Akımlar Üzerinde Giriş/Çıkış* (sayfa: 236) bölümünde açıklanmıştır).

Çoğu sistemde "Syslog" diye bilinen bir oluşum vardır; yazılımların sistem yöneticisini ilgilendiren iletileri teslim edeceği ve bu iletileri aktarmak için konsola çıktılama, belli bir şahsa postalama ya da ileride başvurulmak üzere bir günlük dosyasına kaydetme gibi çeşitli yolları yapılandırabileceği bir oluşumdur.

Bir yazılım bu oylumdaki oluşumları böyle iletileri teslim etmek için kullanır.

1. Syslog'a Bir Bakış

Sistem yöneticileri her sistemin içindeki bir yığın alt sistemden ve bazı başka sistemlerden gelen farklı çeşitte iletiyle uğraşmak zorunda kalır. Örneğin, bir FTP sunucu aldığı her bağlantıyı raporlayabilir. Çekirdek bir disk sürücüdeki donanımsal hataları raporlayabilir. Bir DNS sunucu düzenli aralıklarla istatistikleri raporlayabilir.

Bu iletilerden bazıları bir sistem yöneticisinin dikkatinin hemen çekilmesini gerektirebilir. Ve o sadece herhangi bir sistem yöneticisi olmayabilir – belli ileti çeşitleri ile uğraşan belli sistem yöneticileri olabilir. Diğer iletilerin sadece ileride gerektiğinde bakılmak üzere sadece kaydedilmesi gerekebilir. Kalan diğerlerinin aylık raporlar üreten kendiliğinden çalışan süreçler tarafından çıkarılan bilgiler olması gerekebilir.

Bu iletilerle uğraşmak için çoğu Unix sisteminde "Syslog" diye bilinen bir oluşum vardır ve genellikle "Syslogd" diye bilinen bir ardalan süreci olarak çalışır. Syslogd iletiler için, **`/dev/log`** isminde bir Unix alan soketini dinler. İletilerdeki sınıflama bilgisine ve yapılandırma dosyasına (genellikle **`/etc/syslog.conf`**) bağlı olarak Syslogd onları çeşitli yollara yönlendirir. Çok bilinen bazı yöntemler:

- Sistem konsoluna yazma
- Belli bir kullanıcıya ileti postalama
- Bir günlük dosyasına yazma
- Başka bir ardalan sürecine aktarma
- İptal etme

Syslogd ayrıca başka sistemlerdeki iletileri de işleyebilir. İletiler için hem **`syslog`** UDP portunu hem de yerel soketi dinler.

Syslog çekirdeğin kendisinden gelen iletileri de işleyebilir. Fakat çekirdek **`/dev/log`**'a yazmaz; onun yerine başka bir artalan süreci (bazan "Klogd" denir) çekirdekten iletileri çıkarır ve onları herhangi bir başka sürecin yaptığı gibi Syslog'a aktarır (ve iletilerin çekirdekten geldiğini belirtir).

Syslog çekirdeğin Syslogd ve Klogd çalıştırılmadan önce çıktıladığı iletileri de işleyebilir. Örneğin Linux çekirdeği başlatma iletilerini bir çekirdek ileti zincirinde saklar ve daha sonra Klogd başlatıldığında onlar normal olarak

hala orada duruyor olur. Klogd başlatıldığında Syslogd'nin de başlatılmış olduğu varsayımıyla ileti zincirindeki herşeyi ona aktarır.

Düzenlemek amacıyla iletilerin sınıflanması için, Syslogd her sürecin bir iletiyi teslim ederken iki parçalı sınıflama bilgisini iletiyle birlikte sağlamasını gerektirir:

oluşum
: İletiyi kimin teslim ettiğini belirtir. Tanımlı çok az sayıda oluşum vardır. Çekirdek, posta altsistemi ve FTP sunucusu tanınan oluşumlara birer örnektir. Tam bir liste için *syslog, vsyslog* (sayfa: 471) bölümüne bakınız. Bunların tamamen keyfi bir sınıflama olduğunu unutmayın. "Posta altsistemi" denince sistem yöneticisinin ona verdiğinden daha fazla bir anlam içermez.

öncelik
: Bu teslim edilen iletinin ne kadar önemli olduğunu gösterir. Tanımlı önceliklere örnekler: hata ayıklama, bilgilendirme, uyarı, ölümcül. Tam bir liste için *syslog, vsyslog* (sayfa: 471) bölümüne bakınız. Önceliklerin tanımlı bir sıralamasının dışında bu önceliklerin anlamları sistem yöneticisi tarafından belirlenir.

Bir "oluşum/öncelik" çifti hem oluşumu hem de önceliği belirten bir sayıdır.

Uyarı

Bu terminoloji evrensel değildir. Bazıları önceliğe "seviye", öncelikle oluşumun oluşturduğu birleşime "öncelik" der. Linux çekirdeğinde ise bir iletinin seviyesi hem Syslog önceliği hem de Syslog oluşum/öncelik çiftine karşılık gelir (oluşum kodu çekirdek için sıfır olduğundan her ikisi de aslında aynı değere sahip olur).

GNU C kütüphanesi iletileri Syslog'a teslim etmek için işlevler içerir. Bu işlemi **`/dev/log`** soketine yazarak yaparlar. Bkz. *Syslog İletilerinin Teslim Edilmesi* (sayfa: 469).

GNU C kütüphanesi işlevleri iletileri sadece aynı sistemdeki Syslog oluşumuna teslim ederler. Syslog iletilerini başka bir sistemdeki Syslog oluşumuna teslim etmek için sistemdeki **`syslog`** UDP portuna bir UDP datagramı yazan soket G/Ç işlevleri kullanılır. Bkz. *Soketler* (sayfa: 398).

2. Syslog İletilerinin Teslim Edilmesi

GNU C kütüphanesi iletileri Syslog'a teslim etmek için işlevler içerir:

Bu işlevler iletileri sadece aynı sistemdeki Syslog oluşumuna teslim ederler. Syslog iletilerini başka bir sistemdeki Syslog oluşumuna teslim etmek için sistemdeki **`syslog`** UDP portuna bir UDP datagramı yazan soket G/Ç işlevleri kullanılır. Bkz. *Soketler* (sayfa: 398).

2.1. openlog

Bu böümdeki semboller `syslog.h` dosyasında bildirilmiştir.

```
void openlog(const char *kimlik,                                    işlev
             int          seçenek,
             int          oluşum)
```

`openlog` işlevi iletileri teslim etmek için Syslog'a bir bağlantı açar ya da bir bağlantıyı yeniden açar.

kimlik ileride yapılacak **`syslog`** çağrılarında iletinin başına eklenecek keyfi bir kimliklendirme dizgesidir. Bu iletinin kaynağını belirtmek için düşünülmüştür ve teamülen iletileri teslim edecek yazılımın ismidir.

Eğer *kimlik* bir boş gösterici ise ya da **`openlog`** çağrısı yapılmazsa, ön tanımlı kimliklendirme dizgesi `argv[0]`'dan alınan yazılım ismi olacaktır.

kimlik dizge göstericisinin Syslog yordamları tarafından dahili olarak saklanacağını lütfen hatırlayın. *kimlik* ile gösterilen belleği serbest bırakmamalısınız. Etki alanından çıkmak değişkenin yaşamının sonu demek olduğundan dizgeyi bir özdevinimli değişkende saklamak ayrıca tehlikelidir de. *kimlik* dizgesini değiştirmek isterseniz **openlog** işlevini tekrar çağırmalısınız; *kimlik* ile gösterilen dizgeye yazma işlemi evresel değildir.

Syslog yordamlarının *kimlik* başvurusunu bırakmasını sağlayabilir ve **closelog** çağrısı ile öntanımlı dizgeye (`argv[0]`'dan alınan yazılım ismi) geri dönebilirsiniz: *closelog* (sayfa: 473).

Kodu, yüklenebilen bir paylaşımlı kütüphane (örn, bir PAM modülü) için yazıyorsanız ve **openlog** kullanıyorsanız, kütüphaneyi yüklenmemiş duruma getirmeden önce bir noktada **closelog** çağrısı yapmalısınız. Bir örnek:

```
#include <syslog.h>

void
shared_library_function (void)
{
  openlog ("mylibrary", option, priority);

  syslog (LOG_INFO, "shared library has been invoked");

  closelog ();
}
```

closelog çağrılmaksızın kütüphane yüklenmemiş duruma getirilip **"mylibrary"** dizgesini içeren bellek anlamsız hale getirilirse, kütüphaneyi kullanan yazılım tarafından ileride yapılacak bir **syslog** çağrısı bir çökmeye sebep olabilir. Bu BSD syslog arayüzünün bir sınırlamasıdır.

openlog işlevi *seçenek* değerine bağlı olarak **/dev/log** soketini açabilir de açmayabilir de. Eğer açması gerekiyorsa, onu açmayı dener ve bir datagram soketi olarak ona bağlanır. Soket, "Close on Exec" özniteliğine sahiptir, bu bakımdan eğer süreç bir **exec** çağrısı yaparsa çekirdek onu kapatacaktır.

openlog kullanmak zorunda değilsiniz. Eğer **openlog** çağrısı yapmaksızın bir **syslog** çağrısı yaprsanız, *kimlik* ve *options* için öntanımlı değerler kullanılarak bağlantı anında açılır.

options bir bit dizgesidir. Bitler aşağıdaki tek bitlik maskelerle tanımlanmıştır:

`LOG_PERROR`
: Bit varsa, **openlog** bağlantıyı öyle ayarlar ki, bu bağlantı üzerinden yapılan herhangi bir **syslog** çağrısı iletiyi Syslog'a teslim ederken ek olarak iletiyi sürecin standart hata akımına yazar. Aksi takdirde, **syslog** iletiyi standart hataya yazmaz.

`LOG_CONS`
: Bit varsa, **openlog** bağlantıyı öyle ayarlar ki, bu bağlantı üzerinden yapılan herhangi bir **syslog** çağrısında bir ileti Syslog'a teslim edilemezse ileti sistem konsoluna yazılır. Aksi takdirde, **syslog** çağrısı sistem konsoluna yazmaz (ama şüphesiz kendi konsoluna yazabilir).

`LOG_PID`
: Bit varsa, **openlog** bağlantıyı öyle ayarlar ki, bu bağlantı üzerinden yapılan herhangi bir **syslog** çağrısında, çağrıldığı sürecin süreç kimliğini (PID) iletiye yerleştirir. Aksi takdirde, **openlog** PID'i yerleştirmez.

`LOG_NDELAY`
: Bit varsa, **openlog**, **/dev/log** soketini açar ve bağlanır. Aksi tadirde, ileride yapılacak bir **syslog** çağrısı soketi açmalı ve bağlanmalıdır.

Taşınabilirlik Bilgisi

Erken dönemdeki sistemlerde, bu bit tamamen ters çalışırdı.

`LOG_ODELAY`
: Bu bit hiçbir şey yapmaz. Sadece geriye uyumluluk adına vardır.

options içindeki diğer bitler birse sonuç belirsizdir.

oluşum bu bağlantı için öntanımlı oluşum kodudur. Bu bağlantı üzerinden öntanımlı oluşum belirten bir **syslog** çağrısı, iletinin bu oluşumla ilişkilendirilmesine sebep olur. Olası değerler için **syslog** işlevinin açıklamasına bakın. Belirtilen bir sıfır değeri öntanımlı olan **LOG_USER**'ı öntanımlı oluşum yapar.

openlog çağrısı yaptığınızda, eğer bir Syslog bağlantısı zaten açıksa, **openlog** bağlantıyı "yeniden" açar. Yeniden açma öntanımlı oluşum kodu için sıfır belirtilmesi dışında açma gibidir, öntanımlı kod basitçe değişmeden kalır ve **LOG_NDELAY** belirtilmişse, soket zaten açık ve bağlı olduğundan **openlog** onu olduğu gibi bırakır.

2.2. syslog, vsyslog

Bu bölümdeki semboller `syslog.h` dosyasında bildirilmiştir.

```
void syslog(int    oluşum_öncelik,                                    işlev
            char *biçim,
            ...)
```

syslog bir iletiyi Syslog'a teslim eder. Bunu Unix alan soketi olan **/dev/log**'a yazarak yapar.

syslog iletiyi *oluşum_öncelik* ile belirtilen oluşum ve öncelikle teslim eder. Bir oluşum ve öncelikten bir oluşum/öncelik değeri **LOG_MAKEPRI** makrosu ile şöyle üretilir:

```
LOG_MAKEPRI(LOG_USER, LOG_WARNING)
```

Oluşum kodu için olası değerler (makrolar):

`LOG_USER`
: Bir kullanıcı süreci

`LOG_MAIL`
: Posta

`LOG_DAEMON`
: Bir sistem artalan süreci

`LOG_AUTH`
: Güvenlik (kimlik denetimi)

`LOG_SYSLOG`
: Syslog

`LOG_LPR`
: Merkezi yazıcı

`LOG_NEWS`
: Ağ haberleri (yani, Usenet)

`LOG_UUCP`

UUCP

`LOG_CRON`
Cron ve At

`LOG_AUTHPRIV`
Özel güvenlik (kimlik denetimi)

`LOG_FTP`
Ftp sunucu

`LOG_LOCAL0`
`LOG_LOCAL1`
`LOG_LOCAL2`
`LOG_LOCAL3`
`LOG_LOCAL4`
`LOG_LOCAL5`
`LOG_LOCAL6`
`LOG_LOCAL7`
Yerel olarak tanımlı

Oluşum kodu bunların dışında bir değerse sonuç tanımsızdır.

Bilgi

`syslog` başka bir oluşum kodunu daha tanır: bu çekirdektir. Ama onun oluşum kodunu bu işlevlerle belirtemezsiniz. Eğer denerseniz **`syslog`** öntanımlı oluşum isteği yapılmış gibi davranır. Fakat bunu hiçbir şekilde denememelisiniz, çünkü çekirdek GNU C kütüphanesini kullanan bir yazılım değildir.

oluşum_öncelik olarak bir öncelik kodu da belrtebilirsiniz. Bu durumda, **`syslog`** çağrısı Syslog bağlantısı açıldığında öntanımlı oluşum kurulmuş kabul eder. Bkz. *Syslog Örneği* (sayfa: 474).

Öncelik kodu için olası değerler (makrolar):

`LOG_EMERG`
İleti, sistem işe yaramaz diyor.

`LOG_ALERT`
İleti hemen birşeyler yapılmasını istiyor.

`LOG_CRIT`
İleti ölümcül bir durumu belirtiyor.

`LOG_ERR`
İleti, bir hatanın açıklaması

`LOG_WARNING`
İleti, bir uyarı.

`LOG_NOTICE`
İleti, bir normal ama önamli bir olayı bildiriyor.

`LOG_INFO`
İleti sadece bilgilendirme amaçlı.

`LOG_DEBUG`

İleti sadece hata ayıklama ile ilgili.

Öncelik kodu bunların dışında bir değerse sonuç tanımsızdır.

Eğer süreç açık bir Syslog bağlantısına sahip değilse (yani **openlog** çağrısı yapılmamışsa), **syslog** işlevi **openlog**'un yaptığı gibi hemen bağlantıyı açar. Bağlantıyı açarken, aksi sadece bir **openlog** çağrısı ile belirtilebilecek bilgi için şu öntanımlıları kullanır: Öntanımlı kimliklendirme dizgesi yazılımın ismidir. Öntanımlı oluşum **LOG_USER**'dır. *seçenek* ile belirtilen bitlerin hepsi sıfırdır. **syslog** bağlantıyı açık bırakır.

Eğer **dev/log** soketi açık ve bağlı değilse, **syslog**, **openlog** işlevinin **LOG_NDELAY** seçeneğiyle yaptığı gibi onu açar ve bağlanır.

syslog iletiyi göndermeye çalışırkan başarısız olmazsa, **/dev/log**'u açık ve bağlı bırakır, aksi takdirde **syslog** onu kapatır (ileride bir örtük açılışın Syslog bağlantısını kullanışlı bir durumda açacağını umarak).

Örnek:

```
#include <syslog.h>
  syslog (LOG_MAKEPRI(LOG_LOCAL1, LOG_ERROR),
          "Unable to make network connection to %s.  Error=%m", host);
```

```
void vsyslog(int     oluşum_öncelik,                                    işlev
             char    *biçim,
             va_list arglist)
```

BSD tarzı değişken sayıda argümanlı olarak **syslog**'un benzeridir.

2.3. closelog

Bu bölümdeki semboller `syslog.h` dosyasında bildirilmiştir.

```
void closelog(void)                                                     işlev
```

closelog var olan Syslog bağlantısını kapatır. Bu işlem açıksa **/dev/log** soketini de kapatır. İşlev ayrıca Syslog iletilerininin kimliklendirme dizgesini, eğer **openlog** bağlantıyı boş olmayan bir *kimlik* argümanı ile açmışsa, geriye öntanımlı değerine ayarlar. Öntanımlı kimliklendirme dizgesi `argv[0]`'dan alınan yazılım ismidir.

openlog işlevini özelleştirilmiş syslog çıktısı üretmek için kullanan bir paylaşımlı kütüphane kodu yazıyorsanız, işiniz bittiğinde, GNU C kütüphanesinin *kimlik* göstericisi ile ilişkisini kesmek için **closelog** kullanmalısınız. Daha fazla bilgi için *openlog* (sayfa: 469) bölümünü okuyun.

closelog tamponları boşaltmaz. Bir Syslog bağlantısını **openlog** ile yeniden açmadan önce **closelog** çağrısı yapmak zorunda değilsiniz. Syslog bağlantıları **exec** veya **exit** çağrıları ile özdevinimli olarak kapanır.

2.4. setlogmask

Bu bölümdeki semboller `syslog.h` dosyasında bildirilmiştir.

```
int setlogmask(int mask)                                                işlev
```

setlogmask işlevi ileride hangi **syslog** çağrılarının yoksayılacağını belirleyen bir maske ("log-mask") tanımlar. Eğer bir yazılım **setlogmask** çağrısı yapmazsa, hiçbir **syslog** çağrısı yoksayılmaz. **setlogmask** işlevini hangi öncelikteki iletilerin ileride yoksayılacağını belirtmek için kullanabilirsiniz.

Her **setlogmask** çağrısı daha önceki bir **setlogmask** çağrısının etkisini ortadan kaldırır.

Logmask, Syslog bağlantılarının açılması ya da kapanmasından bağımsız olarak mevcut olabilir.

Logmaskın atanması, Syslog'un yapılandırmasına benzer bir etki oluşturmasına rağmen tamamen aynı değildir. Syslog yapılandırması, alınacak belli iletilerin tamamen yoksayılasına sebep olabilir, ama logmaskta belirterek yapılandırma ile yoksayılan iletilerin teslim edilmesini sağlayamazsınız.

mask her biti olası ileti önceliklerine karşılık düşen bir bit maskesidir. Eğer bu bitlerden biri birse, **syslog** o bitle ilgili öncelikteki iletileri iptal eder. Kullanılan ileti öncelik makroları *syslog, vsyslog* (sayfa: 471) bölümünde açıklanmıştır. **LOG_MASK** kullanım örnekleri:

```
LOG_MASK(LOG_EMERG) | LOG_MASK(LOG_ERROR)
```

veya

```
~(LOG_MASK(LOG_INFO))
```

Ayrıca, bu makroya benzer bir **LOG_UPTO** makrosu vardır:

```
LOG_UPTO(LOG_ERROR)
```

Makro isminin tahmin edilmesindeki zorluk nedeniyle bu makro dahili olarak düşük ileti öncelikleri için daha fazla kullanılır.

2.5. Syslog Örneği

Burada **openlog**, **syslog** ve **closelog** işlevlerinin kullanımı örneklenmiştir:

Bu örnekte hata ayıklama ve bilgilendirme iletileri logmaskta belirtilerek daha Syslog'a ulaşmadan iptal edilmektedir. Böylece örnekteki ikinci **syslog** hiçbir şey yapmamaktadır.

```
#include <syslog.h>

setlogmask (LOG_UPTO (LOG_NOTICE));

openlog ("exampleprog", LOG_CONS | LOG_PID | LOG_NDELAY, LOG_LOCAL1);

syslog (LOG_NOTICE, "Program started by User %d", getuid ());
syslog (LOG_INFO, "A tree falls in a forest");

closelog ();
```

XIX. Matematik

İçindekiler

Bu oylumda trigonometrik işlevler gibi bir takım matematiksel hesaplamaları yapan işlevlere yer verilmiştir. Bu işlevlerinin çoğunun prototipleri `math.h` başlık dosyasında, karmaşık değerli olanları `complex.h` başlık dosyasında bildirilmiştir.

Tüm matematiksel işlevler herbiri **`double`**, **`float`** ve **`long double`** türünde gerçek sayı argüman alan üçlülerdir. **`double`** türünde olanların çoğu ISO C89'da tanımlanmıştır. **`float`** ve **`long double`** türünde olan sürümleri ise ISO C99'da tanımlanmıştır.

Bu üçlüdeki hangi işlevin kullanılması gerektiği duruma bağlıdır. Çoğu hesaplamalarda **`float`** işlevler en hızlısıdır. Ancak **`long double`** işlevler en yüksek hassasiyete sahiptir. **`double`** işlevler ise ikisinin arasında kalır. En iyisi ihtiyacınızı görecek en dar türü seçmektir. Ayrıca, tüm makinalarda ayrı bir **`long double`** tür yoktur; **`double`** ile aynı olabilir.

1. Önceden Tanımlı Matemetiksel Sabitler

`math.h` başlık dosyasında çeşitli kullanışlı matematiksel sabitler tanımlanmıştır. Tüm değerler **`M_`** ile başlayan önişlemci makroları olarak tanımlanmıştır. Değerler şunlardır:

`M_E`
: Tabii logaritmanın tabanı.

`M_LOG2E`
: **`M_E`**'nin ikilik tabanda logaritması.

`M_LOG10E`
: **`M_E`**'nin onluk tabanda logaritması.

`M_LN2`
: **`2`**'nin tabii logaritması.

`M_LN10`
: **`10`**'un tabii logaritması.

`M_PI`
: Pi, çemberin çevresinin çapına oranı.

`M_PI_2`

Pi bölü iki.

`M_PI_4`
Pi bölü dört.

`M_1_PI`
Pi'nin tersi (1/pi).

`M_2_PI`
İki bölü pi.

`M_2_SQRTPI`
İki bölü karekök pi.

`M_SQRT2`
Karekök iki.

`M_SQRT1_2`
Karekök ikinin tersi (ya da karekök 1/2).

Bu sabitler Ubix98 standardından gelir ve ayrıca 4.4 BSD'de de vardır; bu bakımdan, sadece **`_BSD_SOURCE`** ya da **`_XOPEN_SOURCE=500`** tanımlıysa ya da bir daha genel özellik makrosu seçiliyse tanımlıdırlar. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

Tüm değerler **`double`** türündedir. Bir GNU C kütüphanesi oluşumu olarak bu sabitler ayrıca **`long double`** türünde de tanımlanmıştır. **`long double`** makroların isimlerinin sonuna küçük L (**`l`**) harfi eklenmiştir: **`M_El`**, **`M_Pil`**, ... Bunlar sadece **`_GNU_SOURCE`** tanımlıysa kullanılabilir.

Bilgi

Bazı yazılımlar **`M_PI`** makrosu ile aynı değerde **`PI`** isimli bir sabit kullanırlar. Bu sabit standart değildir; bazı eski ATT başlık dosyalarında görünür, bir de Stroustrup'un C++ kitabında bahsedilir. Kullanıcının isim uzayını ihlâl ettiğinden GNU C kütüphanesinde tanımlanmamıştır. Böyle yazılımları düzeltmek kolaydır: ya her **`PI`**'yi **`M_PI`** ile değiştirirsiniz ya da derleyici komut satırına **`-DPI=M_PI`** seçeneğini eklersiniz.

2. Trigonometrik İşlevler

Bunlar bildiğiniz **`sin`**, **`cos`** ve **`tan`** işlevleridir. Bu işlevlerin argümanları radyan cinsindendir; pi radyan 180 dereceye eşittir.

Matematik kütüphanesi (**`-lm`**) normalde pi sayısını **`double`** türünde bir yaklaşıklık olarak **`M_PI`** ile tanımlar. Kesin ISO ve/veya POSIX uyumluluğu istenirse, bu sabit tanımsızdır, ama onu kolayca kendiniz tanımlayabilirsiniz:

```
#define M_PI 3.14159265358979323846264338327
```

Ayrıca pi sayısını **`acos (-1.0)`** ifadesi ile de hesaplayabilirsiniz.

```
double sin(double x)                                    işlev
float sinf(float x)                                     işlev
long double sinl(long double x)                         işlev
```

Bu işlevler x radyan cinsinden verildiğinde x'in sinüsünü hesaplar. Dönüş değeri **`-1`** ile **`1`** arasındadır.

```
double cos(double x)                                    işlev
float cosf(float x)                                     işlev
long double cosl(long double x)                         işlev
```

Bu işlevler x radyan cinsinden verildiğinde x'in kosinüsünü hesaplar. Dönüş değeri **-1** ile **1** arasındadır.

```
double tan(double x)                                    işlev
float tanf(float x)                                     işlev
long double tanl(long double x)                         işlev
```

Bu işlevler x radyan cinsinden verildiğinde x'in tanjantını hesaplar.

Matemetiksel olarak tanjant işlevi pi/2'nin tek katlarında sonsuza gider. Eğer x argümanı bu değerlerden biri ise **tan** işlevi taşma sinyalleyecektir.

sin ve **cos** işlevlerinin kullanıldığı çoğu uygulamada aynı anda aynı açının hem sinüsü hem de kosinüsü gerekir. Böyle durumlar için kütüphanede bir işlev bulunur.

```
void sincos(double  x,                                  işlev
            double *sinx,
            double *cosx)
void sincosf(float  x,                                  işlev
             float *sinx,
             float *cosx)
void sincosl(long double  x,                            işlev
             long double *sinx,
             long double *cosx)
```

Bu işlevler x radyan cinsinden verildiğinde x'in sinüsünü $*sinx$ içinde, kosinüsünü $*cosx$ içinde döndürür. Her iki değer de **-1** ile**1** arasındadır.

Bu işlev bir GNU oluşumudur. Taşınabilir yazılımlar bu işlevin yokluğunu gerektirir.

ISO C99 trigonometrik işlevlerin karmaşık sayılarla çalışan çeşitlerini de tanımlamıştır. GNU C kütüphanesi bu işlevleri içerir, fakat sadece derleyici standart tarafından tanımlanan yeni karmaşık sayı türlerini destekliyorsa bunlar kullanılabilir. (Bu kılavuz yazılırken GCC karmaşık sayıları desteklemekteydi, ancak gerçeklemede hatalar vardı.)

```
complex double csin(complex double z)                   işlev
complex float csinf(complex float z)                    işlev
complex long double csinl(complex long double z)        işlev
```

Bu işlevler z karmaşık açısının karmaşık sinüsünü hesaplar. Karmaşık sinüsün matematiksel tanımı:

```
sin (z) = 1/(2*i) * (exp (z*i) - exp (-z*i))
```

```
complex double ccos(complex double z)                   işlev
complex float ccosf(complex float z)                    işlev
complex long double ccosl(complex long double z)        işlev
```

Bu işlevler z karmaşık açısının karmaşık kosinüsünü hesaplar. Karmaşık kosinüsün matematiksel tanımı:

```
cos (z) = 1/2 * (exp (z*i) + exp (-z*i))
```

```
complex double ctan(complex double z)                   işlev
complex float ctanf(complex float z)                    işlev
complex long double ctanl(complex long double z)        işlev
```

Bu işlevler z karmaşık açısının karmaşık tanjantını hesaplar. Karmaşık tanjantın matematiksel tanımı:

```
tan (z) = -i * (exp (z*i) - exp (-z*i)) / (exp (z*i) + exp (-z*i))
```

Karmaşık tanjantın `pi/2 + 2n`'de kutupları vardır; burada `n` bir tamsayıdır. **`ctan`** işlevi z bu kutba çok yakınsa taşma sinyalleyebilir.

3. Ters Trigonometrik İşlevler

Bunlar sinüs, kosinüs ve tanjant işlevlerinin ters işlevleri olan (sırasıyla) ark sinüs, ark kosinüs ve ark tanjant işlevleridir.

```
double asin(double x)                                          işlev
float asinf(float x)                                           işlev
long double asinl(long double x)                               işlev
```

Bu işlevler sinüsü x olan açı ile döner. Dönüş değerleri radyan cinsindendir. Matematiksel olarak böyle sonsuz sayıda değer vardır; işlev asıl kol üzerinde yani **`-pi/2`** ile **`pi/2`** (dahil) arasında bir değerle döner.

Ark sinüs işlevi matematiksel olarak **`-1`** ile **`1`** arasındaki sahada tanımlıdır. Eğer x bu sahanın dışındaysa, **`asin`** bir saha hatası sinyaller.

```
double acos(double x)                                          işlev
float acosf(float x)                                           işlev
long double acosl(long double x)                               işlev
```

Bu işlevler kosinüsü x olan açı ile döner. Dönüş değerleri radyan cinsindendir. Matematiksel olarak böyle sonsuz sayıda değer vardır; işlev asıl kol üzerinde yani **`0`** ile **`pi`** (dahil) arasında bir değerle döner.

Ark kosinüs işlevi matematiksel olarak **`-1`** ile **`1`** arasındaki sahada tanımlıdır. Eğer x bu sahanın dışındaysa, **`acos`** bir saha hatası sinyaller.

```
double atan(double x)                                          işlev
float atanf(float x)                                           işlev
long double atanl(long double x)                               işlev
```

Bu işlevler tanjantı x olan açı ile döner. Dönüş değerleri radyan cinsindendir. Matematiksel olarak böyle sonsuz sayıda değer vardır; işlev asıl kol üzerinde yani **`-pi/2`** ile **`pi/2`** (dahil) arasında bir değerle döner.

```
double atan2(double y,                                         işlev
             double x)
float atan2f(float y,                                          işlev
             float x)
long double atan2l(long double y,                              işlev
                   long double x)
```

Bu işlevler tanjantı y/x olan açı ile döner, fakat her iki argümanın işaretinden sonucun hangi dörtte bir çemberde olduğu saptanır. Dönüş değeri radyan cinsinden **`-pi`** ile **`pi`** (dahil) arasındadır.

Eğer x ve y düzlem üzerindeki bir noktanın koordinatları ise **`atan2`** işlevi, merkez ile bu nokta arasında çizilen doğrunun x ekseni ile yaptığı açıyı işareti ile döndürür. Bu durumda, **`atan2`** işlevi kartezyen koordinatları kutupsal koordinatlara dönüştürmek için kullanılabilir. (Açısal koordinatları hesaplamak için **`hypot`** kullanın; bkz. *Üstel ve Logaritmik İşlevler* (sayfa: 479).)

Eğer x ve y değerlerinin ikisi de sıfırsa, işlev sıfırla döner.

ISO C99 ters trigonometrik işlevlerin karmaşık sürümlerini de tanımlamıştır.

```
complex double casin(complex double z)                    işlev
complex float casinf(complex float z)                     işlev
complex long double casinl(complex long double z)         işlev
```

Bu işlevler karmaşık sinüsü z olan açı ile döner. Dönüş değerleri radyan cinsindendir.

Gerçek sayı değerli işlevlerin tersine, **casin** işlevi tüm z değerleri için tanımlıdır.

```
complex double cacos(complex double z)                    işlev
complex float cacosf(complex float z)                     işlev
complex long double cacosl(complex long double z)         işlev
```

Bu işlevler karmaşık kosinüsü z olan açı ile döner. Dönüş değerleri radyan cinsindendir.

Gerçek sayı değerli işlevlerin tersine, **cacos** işlevi tüm z değerleri için tanımlıdır.

```
complex double catan(complex double z)                    işlev
complex float catanf(complex float z)                     işlev
complex long double catanl(complex long double z)         işlev
```

Bu işlevler karmaşık tanjantı z olan açı ile döner. Dönüş değerleri radyan cinsindendir.

4. Üstel ve Logaritmik İşlevler

```
double exp(double x)                                      işlev
float expf(float x)                                       işlev
long double expl(long double x)                           işlev
```

Bu işlevler **e** (tabii logaritmanın tabanı) üssü x'i hesaplar.

Sonuç gösterilemeyecek kadar büyükse işlev taşma hatası sinyaller.

```
double exp2(double x)                                     işlev
float exp2f(float x)                                      işlev
long double exp2l(long double x)                          işlev
```

Bu işlevler **2** üssü x'i hesaplar. Matematiksel olarak, **exp2 (x)** ile **exp (x * log (2))** aynıdır.

```
double exp10(double x)                                    işlev
float exp10f(float x)                                     işlev
long double exp10l(long double x)                         işlev
double pow10(double x)                                    işlev
float pow10f(float x)                                     işlev
long double pow10l(long double x)                         işlev
```

Bu işlevler **10** üssü x'i hesaplar. Matematiksel olarak, **exp10 (x)** ile **exp (x * log (10))** aynıdır.

Bu işlevler GNU oluşumudur. **exp** ve **exp2** işlevleriyle isim benzerliğinden dolayı **exp10** tercih edilir.

```
double log(double x)                                      işlev
float logf(float x)                                       işlev
long double logl(long double x)                           işlev
```

Bu işlevler x'in tabii logaritmasını hesaplar. **exp (log (x))** ifadesi matematikte kesinlikle, C'de ise yaklaşık olarak x'e eşittir.

Eğer x negatifse, **log** işlevi bir saha hatası sinyaller. Eğer x sıfırsa, negatif sonsuz döner; x sıfıra çok yakınsa taşma sinyallenebilir.

```
double log10(double x)                                  işlev
float log10f(float x)                                   işlev
long double log10l(long double x)                       işlev
```

Bu işlevler x'in 10 tabanına göre logaritması ile döner. **log10 (x)** ifadesi **log (x) / log (10)** ifadesine eşittir.

```
double log2(double x)                                   işlev
float log2f(float x)                                    işlev
long double log2l(long double x)                        işlev
```

Bu işlevler x'in 2 tabanına göre logaritması ile döner. **log2 (x)** ifadesi **log (x) / log (2)** ifadesine eşittir.

```
double logb(double x)                                   işlev
float logbf(float x)                                    işlev
long double logbl(long double x)                        işlev
```

Bu işlevler x sonucunu veren üstel sayının üssünü bir gerçek sayı olarak döndürür. Eğer **FLT_RADIX** iki ise, **logb** işlevi büyük olasılıkla daha hızlı olmak dışında **floor (log2 (x))** çağrısına eşdeğerdir.

Eğer x normalleştirilememiş bir değerse, işlev normalleştirilmiş değerle döner. Eğer x sonsuzsa (pozitif ya da negatif) ya da sıfırsa, işlev sonsuzla döner ve hata sinyallemez.

```
int ilogb(double x)                                     işlev
int ilogbf(float x)                                     işlev
int ilogbl(long double x)                               işlev
```

Bu işlevler işaretli tamsayı değerler döndürmeleri dışında **logb** işlevlerine eşdeğerdir.

Tamsayılarla sonsuz ve NaN gösterilemediğinden **ilogb** bunların yerine sonucu yansıtmayan bir tamsayı ile döner. `math.h` dosyasında tanımlı sabitlerle bu durumu sınayabilirsiniz.

```
int FP_ILOGB0                                           makro
```

ilogb işlevi argümanı **0** ise bu değerle döner. Sayısal değeri ya **INT_MIN** ya da **-INT_MAX**'tır.

Bu makro ISO C99'da tanımlanmıştır.

```
int FP_ILOGBNAN                                         makro
```

ilogb işlevi argümanı **NaN** ise bu değerle döner. Sayısal değeri ya **INT_MIN** ya da **-INT_MAX**'tır.

Bu makro ISO C99'da tanımlanmıştır.

Bu değerler sisteme özeldir, hatta aynı bile olabilirler. **ilogb** işlevinin sonucunu sınamanın uygun yolu şöyledir:

```
i = ilogb (f);
if (i == FP_ILOGB0 || i == FP_ILOGBNAN)
  {
    if (isnan (f))
      {
```

```
        /* NaN'ı elde edelim.  */
      }
    else if (f  == 0.0)
      {
        /* 0.0'ı elde edelim.  */
      }
    else
      {
        /* Diğer değerler, +Inf gibi.  */
      }
  }
```

```
double pow(double taban,                                    işlev
           double üs)
float powf(float taban,                                     işlev
           float üs)
long double powl(long double taban,                         işlev
                 long double üs)
```

Bunlar genel amaçlı üstel işlevlerdir ve *taban* üssü *üs* ile dönerler.

Matematiksel olarak, *taban* negatifse ve *üs* bir tamsayı değer değilse **pow** bir karmaşık sayı ile dönerdi. **pow** böyle yapmaz, bir saha hatası sinyaller. **pow** ayrıca hedef türe bağlı olarak alttan ya da üstten taşabilir.

```
double sqrt(double x)                                       işlev
float sqrtf(float x)                                        işlev
long double sqrtl(long double x)                            işlev
```

Bu işlevler x'in negatif olmayan karekökü ile dönerler.

Eğer x negatifse, **sqrt** işlevi bir saha hatası sinyaller. Matematiksel olarak, bir karmaşık sayı ile dönerdi.

```
double cbrt(double x)                                       işlev
float cbrtf(float x)                                        işlev
long double cbrtl(long double x)                            işlev
```

Bu işlevler x'in kübkökü ile dönerler. Başarısız olamazlar; Gösterilebilen her gerçek sayının gösterilebilen bir kübkökü daima vardır.

```
double hypot(double x,                                      işlev
             double y)
float hypotf(float x,                                       işlev
             float y)
long double hypotl(long double x,                           işlev
                   long double y)
```

Bu işlevler **sqrt (x*x + y*y)** işleminin sonucu ile döner. Bu değer, dik kenarları x ve y olan bir dik üçgenin hipotenüsünün uzunluğudur; aynı zamanda (0, 0) ve (x, y) noktaları arasındaki mesafedir. Doğrudan formülü hesaplatmak yerine bu işlevi kullanırsanız hassasiyet kaybı olmaz. Ayrıca, *Mutlak Değer* (sayfa: 519) bölümündeki **cabs** işlevine de bakınız.

```
double expm1(double x)                                      işlev
float expm1f(float x)                                       işlev
long double expm1l(long double x)                           işlev
```

Bu işlevler **exp (x) - 1** ifadesine eşdeğer bir değerle döner. x sıfıra çok yakın olduğunda bile en doğru sonucu verecek şekilde hesaplama yapar (**exp (x) - 1** ifadesinde çıkarma işleminin iki terimi yaklaşık eşit olduğunda yeterince doğru sonuç elde edilemeyebilirdi).

```
double log1p(double x)                                          işlev
float log1pf(float x)                                           işlev
long double log1pl(long double x)                               işlev
```

Bu işlevler **log (1 + x)**. ifadesine eşdeğer bir değerle döner. x sıfıra çok yakın olduğunda bile en doğru sonucu verecek şekilde hesaplama yapar.

ISO C99 üstel ve logaritmik işlevlerin karmaşık sayılarla işlem yapanlarını da tanımlamıştır.

```
complex double cexp(complex double z)                           işlev
complex float cexpf(complex float z)                            işlev
complex long double cexpl(complex long double z)                işlev
```

Bu işlevler **e** üssü z ile döner. Bu işlemin matematiksel ifadesi:

```
exp (z) = exp (creal (z)) * (cos (cimag (z)) + I * sin (cimag (z)))
```

```
complex double clog(complex double z)                           işlev
complex float clogf(complex float z)                            işlev
complex long double clogl(complex long double z)                işlev
```

Bu işlevler z'nin tabii loaritmasını hesaplar. Bu işlemin matematiksel ifadesi:

```
log (z) = log (cabs (z)) + I * carg (z)
```

clog işlevinin 0'da bir kutbu vardır ve z sıfıra çok yakın olduğunda taşma sinyalleyecektir. z'nin diğer değerleri için en doğru sonucu verir.

```
complex double clog10(complex double z)                         işlev
complex float clog10f(complex float z)                          işlev
complex long double clog10l(complex long double z)              işlev
```

Bu işlevler z karmaşık sayısının 10 tabanına göre logaritmasını hesaplar. Bu işlemin matematiksel ifadesi:

```
log (z) = log10 (cabs (z)) + I * carg (z)
```

Bu işlevler GNU oluşumudur.

```
complex double csqrt(complex double z)                          işlev
complex float csqrtf(complex float z)                           işlev
complex long double csqrtl(complex long double z)               işlev
```

Bu işlevler z karmaşık sayısının karekökünü hesaplar. Gerçek sayılı eşdeğerlerinin tersine bu işlevler tüm z değerleri için tanımlıdır.

```
complex double cpow(complex double taban,                       işlev
                    complex double üs)
complex float cpowf(complex float taban,                        işlev
                    complex float üs)
complex long double cpowl(complex long double taban,            işlev
                          complex long double üs)
```

Bu işlevler *taban* üssü *üs* ile döner. Yapılan işlem **cexp (us * clog (taban))** çağrısına eşdeğerdir.

5. Hiperbolik İşlevler

Bu kısımdaki işlevler üstel işlevlerle ilişkilidir; bkz. *Üstel ve Logaritmik İşlevler* (sayfa: 479).

```
double sinh(double x)                                    işlev
float sinhf(float x)                                     işlev
long double sinhl(long double x)                         işlev
```

Bu işlevler x'in hiperbolik sinüsünü hesaplar. İşlem, **(exp (x) - exp (-x)) / 2** ifadesine eşdeğerdir. x çok büyükse işlevler taşma sinyalleyebilir.

```
double cosh(double x)                                    işlev
float coshf(float x)                                     işlev
long double coshl(long double x)                         işlev
```

Bu işlevler x'in hiperbolik kosinüsünü hesaplar. İşlem, **(exp (x) + exp (-x)) / 2** ifadesine eşdeğerdir. x çok büyükse işlevler taşma sinyalleyebilir.

```
double tanh(double x)                                    işlev
float tanhf(float x)                                     işlev
long double tanhl(long double x)                         işlev
```

Bu işlevler x'in hiperbolik tanjantını hesaplar. İşlem, **sinh (x) / cosh (x)** ifadesine eşdeğerdir. x çok büyükse işlevler taşma sinyalleyebilir.

Bu hiperbolik işlevlerin karmaşık sayılarla çalışan benzerleri vardır.

```
complex double csinh(complex double z)                   işlev
complex float csinhf(complex float z)                    işlev
complex long double csinhl(complex long double z)        işlev
```

Bu işlevler z'nin karmaşık hiperbolik sinüsünü hesaplar. İşlem, **(exp (z) - exp (-z)) / 2** ifadesine eşdeğerdir.

```
complex double ccosh(complex double z)                   işlev
complex float ccoshf(complex float z)                    işlev
complex long double ccoshl(complex long double z)        işlev
```

Bu işlevler z'nin karmaşık hiperbolik kosinüsünü hesaplar. İşlem, **(exp (z) + exp (-z)) / 2** ifadesine eşdeğerdir.

```
complex double ctanh(complex double z)                   işlev
complex float ctanhf(complex float z)                    işlev
complex long double ctanhl(complex long double z)        işlev
```

Bu işlevler z'nin karmaşık hiperbolik tanjantını hesaplar. İşlem, **csinh (z) / ccosh (z)** ifadesine eşdeğerdir.

```
double cosh(double x)                                    işlev
float asinhf(float x)                                    işlev
long double asinhl(long double x)                        işlev
```

Bu işlevler hiperbolik sinüsü x olan açıyı hesaplar.

```
double acosh(double x)                                   işlev
float acoshf(float x)                                    işlev
long double acoshl(long double x)                        işlev
```

Bu işlevler hiperbolik kosinüsü x olan açıyı hesaplar. Eğer $x < 1$ ise, **acosh** bir saha hatası sinyaller.

```
double atanh(double x)                                          işlev
float atanhf(float x)                                           işlev
long double atanhl(long double x)                               işlev
```

Bu işlevler hiperbolik tanjantı x olan açıyı hesaplar. x'in mutlak değeri 1'den büyükse, **atanh** bir daha hatası sinyaller; 1'e eşitse, **atanh** sonsuz ile döner.

```
complex double casinh(complex double z)                         işlev
complex float casinhf(complex float z)                          işlev
complex long double casinhl(complex long double z)              işlev
```

Bu işlevler karmaşık hiperbolik sinüsü z olan açıyı hesaplar.

```
complex double cacosh(complex double z)                         işlev
complex float cacoshf(complex float z)                          işlev
complex long double cacoshl(complex long double z)              işlev
```

Bu işlevler karmaşık hiperbolik kosinüsü z olan açıyı hesaplar. Gerçek sayılı eşdeğerlerinin tersine bu işlevler için z'nin değeri ile ilgili bir sınırlama yoktur.

```
complex double catanh(complex double z)                         işlev
complex float catanhf(complex float z)                          işlev
complex long double catanhl(complex long double z)              işlev
```

Bu işlevler karmaşık hiperbolik tanjantı z olan açıyı hesaplar. Gerçek sayılı eşdeğerlerinin tersine bu işlevler için z'nin değeri ile ilgili bir sınırlama yoktur.

6. Özel İşlevler

Bu işlevler kimi zaman faydalı olan biraz daha yabancıl matematiksel işlevlerdir. Şimdilik sadece gerçek sayılarla çalışan sürümleri vardır.

```
double erf(double x)                                            işlev
float erff(float x)                                             işlev
long double erfl(long double x)                                 işlev
```

Bu işlevler x'in hata işlevi ile döner. Hata işlevi şöyle tanımlanır:

```
erf (x) = 2/sqrt(pi) * 0'dan x'e integral exp(-t^2) dt
```

```
double erfc(double x)                                           işlev
float erfcf(float x)                                            işlev
long double erfcl(long double x)                                işlev
```

Bu işlevler **1.0 - erf(**x**)** ile döner, fakat hesaplama x çok büyük olduğunda yuvarlayamama hatalarından kaçınan bir yöntemle yapılır.

```
double lgamma(double x)                                         işlev
float lgammaf(float x)                                          işlev
long double lgammal(long double x)                              işlev
```

Bu işlevler x'in gamma işlevinin mutlak değerinin tabii logartiması ile döner. Gamma işlevi şöyle tanımlanır:

```
gamma (x) = 0'dan sonsuza integral  t^(x-1) e^-t dt
```

Gamma işlevinin işareti `math.h` dosyasında tanımlanmış olan *signgam* genel değişkeninde saklanır. Saklanan değer, eğer ara sonuç sıfıra eşit ya da büyükse **1**, değilse **-1**'dir.

Gerçek gamma işlevini hesaplamak için ya **tgamma** işlevini kullanın ya da değerleri şöyle hesaplayın:

```
lgam = lgamma(x);
gam  = signgam * exp(lgam);
```

Gamma işlevinin pozitif olmayan tamsayılarda belirsizlikleri vardır. **lgamma** işlevi böyle bir belirsizlikte sıfırla bölme olağandışılığı sinyaller.

```
double lgamma_r(double x,                                          işlev
                int    *signp)
float lgammaf_r(float x,                                           işlev
                int   *signp)
long double lgammal_r(long double x,                               işlev
                      int         *signp)
```

lgamma_r işlevi işareti *signgam* genel değişkenine yerleştirmek yerine kullanıcı tanımlı **signp*'ye yerleştirmesi dışında **lgamma** işlevinin benzeridir. Yani **lgamma** işlevinin evreselidir.

```
double gamma(double x)                                             işlev
float gammaf(float x)                                              işlev
long double gammal(long double x)                                  işlev
```

Bu işlevler uyumluluk adına vardır, **lgamma**'ların eşdeğerleridir. **lgamma** ISO C99'da tanımlanmış olarak **gamma** işlevinden daha standart olduğundan bu işlevler yerine **lgamma**'ları kullanmalısınız.

```
double tgamma(double x)                                            işlev
float tgammaf(float x)                                             işlev
long double tgammal(long double x)                                 işlev
```

Bu işlevler *x*'e gamma işlevini uygular. Gamma işlevi şöyle tanımlanır:

```
gamma (x) = 0'dan sonsuza integral t^(x-1) e^-t dt
```

Bu işlev ISO C99'da tanımlanmıştır.

```
double j0(double x)                                                işlev
float j0f(float x)                                                 işlev
long double j0l(long double x)                                     işlev
```

Bu işlevler *x* üssü sıfırın birinci türden Bessel işlevini hesaplar. *x* çok büyük olduğunda işlev taşma sinyalleyebilir.

```
double j1(double x)                                                işlev
float j1f(float x)                                                 işlev
long double j1l(long double x)                                     işlev
```

Bu işlevler *x* üssü birin birinci türden Bessel işlevini hesaplar. *x* çok büyük olduğunda işlev taşma sinyalleyebilir.

```
double jn(int    n,                                                işlev
          double x)
float jnf(int   n,                                                 işlev
          float x)
long double jnl(int          n,                                    işlev
                long double x)
```

Bu işlevler x üssü n'in birinci türden Bessel işlevini hesaplar. x çok büyük olduğunda işlev taşma sinyalleyebilir.

```
double y0(double x)                                                    işlev
float y0f(float x)                                                     işlev
long double y0l(long double x)                                         işlev
```

Bu işlevler x üssü sıfırın ikinci türden Bessel işlevini hesaplar. x çok büyük olduğunda işlev taşma sinyalleyebilir. Negatif olduğunda saha hatası sinyaller; sıfırsa taşma sinyaller ve eksi sonsuz ile döner.

```
double y1(double x)                                                    işlev
float y1f(float x)                                                     işlev
long double y1l(long double x)                                         işlev
```

Bu işlevler x üssü birin ikinci türden Bessel işlevini hesaplar. x çok büyük olduğunda işlev taşma sinyalleyebilir. Negatif olduğunda saha hatası sinyaller; sıfırsa taşma sinyaller ve eksi sonsuz ile döner.

```
double yn(int    n,                                                    işlev
          double x)
float ynf(int   n,                                                     işlev
          float x)
long double ynl(int         n,                                         işlev
                long double x)
```

Bu işlevler x üssü n'in ikinci türden Bessel işlevini hesaplar. x çok büyük olduğunda işlev taşma sinyalleyebilir. Negatif olduğunda saha hatası sinyaller; sıfırsa taşma sinyaller ve eksi sonsuz ile döner.

7. Matematiksel İşlevlerde Hatalar

Bu bölümde matematik kütüphanesindeki işlevlerin bilinen hataları listelenmiştir. Hatalar son görüldüğü yerdeki birim (ULP – Units of the Last Place) cinsinden ölçülür. Bu, göreli hatalar için bir ölçüdür. `d.d...d*2^e` ile ifade edilen bir `z` sayısı için ULP şöyle ifade edilir:

```
|d.d...d - (z / 2^e)| / 2^(p - 1)
```

Burada `p` kayan noktalı gösterimde ondalık kısımdaki bit sayısıdır. İdeal olarak tüm işlevler için hatalar daima `0.5ulps`'dan küçüktür. Bit yuvarlaması kullanarak bu ayrıca mümkündür ve normalde temel işlemler için gerçeklenmiştir. Karmaşık matematiksel işlevler için biraz daha çalışma gerekir ve bu henüz yapılmamıştır.

Dolayısıyla, matematik kütüphanesindeki çoğu işlev hata içerir. Tablo her işlevin sınamalar kapsamındaki mevcut sınamalarla tespit edilmiş en büyük hatasını listeler. Tablo oluşturulurken mümkün olduğunca çok sayıda hata listelenmeye çalışılmış, ama geniş arama uzayı sebebiyle pek mümkün olamamıştır.

Tabloda ULP değerleri farklı mimariler için listelenmiştir. Farklı mimarilerdeki donanım kayan noktalı sayı işlemlerini farklı desteklediğinden ve işlem çeşitleri de farklılık gösterdiğinden farklı mimarilerde sonuçlar da farklı olur.

İşlev adı	Alpha	Soysal	ix86	IA64	PowerPC
acosf	–	–	–	–	–
acos	–	–	–	–	–
acosl	–	–	622	–	1
acoshf	–	–	–	–	–
acosh	–	–	–	–	–
acoshl	–	–	–	–	1
asinf	–	–	–	–	–

asin	–	–	–	–	–
asinl	–	–	1	–	2
asinhf	–	–	–	–	–
asinh	–	–	–	–	–
asinhl	–	–	–	–	1
atanf	–	–	–	–	–
atan	–	–	–	–	–
atanl	–	–	–	–	–
atanhf	1	–	–	–	1
atanh	–	–	–	–	–
atanhl	–	–	1	–	–
atan2f	6	–	–	–	1
atan2	–	–	–	–	–
atan2l	–	–	–	–	1
cabsf	–	–	–	–	–
cabs	–	–	–	–	–
cabsl	–	–	–	–	1
cacosf	–	–	0 + i 1	0 + i 1	–
cacos	–	–	–	–	–
cacosl	–	–	0 + i 2	0 + i 2	1 + i 1
cacoshf	–	9 + i 4	7 + i 0	7 + i 3	
cacosh	1 + i 1	–	1 + i 1	1 + i 1	1 + i 1
cacoshl	–	–	6 + i 1	7 + i 1	1 + i 0
cargf	–	–	–	–	–
carg	–	–	–	–	–
cargl	–	–	–	–	–
casinf	1 + i 0	–	1 + i 1	1 + i 1	1 + i 0
casin	1 + i 0	–	1 + i 0	1 + i 0	1 + i 0
casinl	–	–	2 + i 2	2 + i 2	1 + i 1
casinhf	1 + i 6	–	1 + i 6	1 + i 6	1 + i 6
casinh	5 + i 3	–	5 + i 3	5 + i 3	5 + i 3
casinhl	–	–	5 + i 5	5 + i 5	4 + i 1
catanf	4 + i 1	–	0 + i 1	0 + i 1	4 + i 1
catan	0 + i 1	–	0 + i 1	0 + i 1	0 + i 1
catanl	–	–	–	–	1 + i 1
catanhf	0 + i 6	–	1 + i 0	–	0 + i 6
catanh	4 + i 0	–	2 + i 0	4 + i 0	4 + i 0
catanhl	–	–	1 + i 0	1 + i 0	–
cbrtf	–	–	–	–	–
cbrt	1	–	–	–	1
cbrtl	–	–	1	–	1
ccosf	1 + i 1	–	0 + i 1	0 + i 1	1 + i 1
ccos	1 + i 0	–	1 + i 0	1 + i 0	1 + i 0
ccosl	–	–	1 + i 1	1 + i 1	1 + i 1
ccoshf	1 + i 1	–	1 + i 1	1 + i 1	1 + i 1
ccosh	1 + i 0	–	1 + i 1	1 + i 1	1 + i 0
ccoshl	–	–	0 + i 1	0 + i 1	1 + i 2

ceilf	–	–	–	–	–
ceil	–	–	–	–	–
ceill	–	–	–	–	–
cexpf	1 + i 1	–	–	1 + i 1	1 + i 1
cexp	–	–	–	–	–
cexpl	–	–	1 + i 1	0 + i 1	2 + i 1
cimagf	–	–	–	–	–
cimag	–	–	–	–	–
cimagl	–	–	–	–	–
clogf	1 + i 3	–	1 + i 0	1 + i 0	1 + i 3
clog	–	–	–	–	–
clogl	–	–	1 + i 0	1 + i 0	2 + i 1
clog10f	1 + i 5	–	1 + i 1	1 + i 1	1 + i 5
clog10	0 + i 1	–	1 + i 1	1 + i 1	0 + i 1
clog10l	–	–	1 + i 1	1 + i 1	3 + i 1
conjf	–	–	–	–	–
conj	–	–	–	–	–
conjl	–	–	–	–	–
copysignf	–	–	–	–	–
copysign	–	–	–	–	–
copysignl	–	–	–	–	–
cosf	1	–	1	1	1
cos	2	–	2	2	2
cosl	–	–	1	1	1
coshf	–	–	–	–	–
cosh	–	–	–	–	–
coshl	–	–	–	–	1
cpowf	4 + i 2	–	4 + i 3	5 + i 3	5 + i 2
cpow	2 + i 2	–	1 + i 2	2 + i 2	2 + i 2
cpowl	–	–	763 + i 2	6 + i 4	2 + i 2
cprojf	–	–	–	–	–
cproj	–	–	–	–	–
cprojl	–	–	–	–	0 + i 1
crealf	–	–	–	–	–
creal	–	–	–	–	–
creall	–	–	–	–	–
csinf	–	–	1 + i 1	1 + i 1	–
csin	–	–	–	–	–
csinl	–	–	1 + i 0	1 + i 0	1 + i 0
csinhf	1 + i 1	–	1 + i 1	1 + i 1	1 + i 1
csinh	0 + i 1	–	1 + i 1	1 + i 1	0 + i 1
csinhl	–	–	1 + i 2	1 + i 2	1 + i 1
csqrtf	1 + i 0	–	–	1 + i 0	1 + i 0
csqrt	–	–	–	–	–
csqrtl	–	–	–	–	1 + i 1
ctanf	–	–	0 + i 1	0 + i 1	–
ctan	0 + i 1	–	1 + i 1	1 + i 1	1 + i 1

ctanl	–	–	439 + i 3	2 + i 1	1 + i 1
ctanhf	2 + i 1	–	1 + i 1	0 + i 1	2 + i 1
ctanh	1 + i 0	–	1 + i 1	1 + i 1	1 + i 0
ctanhl	–	–	5 + i 25	1 + i 24	1 + i 1
erff	–	–	–	–	–
erf	1	–	1	1	1
erfl	–	–	–	–	1
erfcf	–	–	1	1	1
erfc	1	–	1	1	1
erfcl	–	–	1	1	1
expf	–	–	–	–	–
exp	–	–	–	–	–
expl	–	–	–	–	1
exp10f	2	–	–	2	2
exp10	6	–	–	6	6
exp10l	–	–	8	3	8
exp2f	–	–	–	–	–
exp2	–	–	–	–	–
exp2l	–	–	–	–	2
expm1f	1	–	–	–	1
expm1	1	–	–	–	–
expm1l	–	–	–	1	–
fabsf	–	–	–	–	–
fabs	–	–	–	–	–
fabsl	–	–	–	–	–
fdimf	–	–	–	–	–
fdim	–	–	–	–	–
fdiml	–	–	–	–	–
floorf	–	–	–	–	–
floor	–	–	–	–	–
floorl	–	–	–	–	–
fmaf	–	–	–	–	–
fma	–	–	–	–	–
fmal	–	–	–	–	–
fmaxf	–	–	–	–	–
fmax	–	–	–	–	–
fmaxl	–	–	–	–	–
fminf	–	–	–	–	–
fmin	–	–	–	–	–
fminl	–	–	–	–	–
fmodf	–	–	–	–	–
fmod	–	–	–	2	–
fmodl	–	–	–	–	–
frexpf	–	–	–	–	–
frexp	–	–	–	–	–
frexpl	–	–	–	–	–
gammaf	–	–	–	–	–

gamma	–	–	1	–	–
gammal	–	–	1	1	1
hypotf	1	–	1	1	1
hypot	–	–	–	–	–
hypotl	–	–	–	–	1
ilogbf	–	–	–	–	–
ilogb	–	–	–	–	–
ilogbl	–	–	–	–	–
j0f	2	–	2	2	2
j0	2	–	3	3	3
j0l	–	–	1	2	1
j1f	2	–	1	2	2
j1	1	–	1	1	1
j1l	–	–	1	1	1
jnf	4	–	2	4	4
jn	4	–	5	3	3
jnl	–	–	2	2	4
lgammaf	2	–	2	2	2
lgamma	1	–	1	1	1
lgammal	–	–	1	1	3
lrintf	–	–	–	–	–
lrint	–	–	–	–	–
lrintl	–	–	–	–	–
llrintf	–	–	–	–	–
llrint	–	–	–	–	–
llrintl	–	–	–	–	–
logf	–	–	1	1	–
log	–	–	–	–	–
logl	–	–	–	–	1
log10f	2	–	1	1	2
log10	1	–	–	–	1
log10l	–	–	1	1	1
log1pf	1	–	–	–	1
log1p	–	–	–	–	–
log1pl	–	–	–	–	1
log2f	–	–	–	–	–
log2	–	–	–	–	–
log2l	–	–	–	–	1
logbf	–	–	–	–	–
logb	–	–	–	–	–
logbl	–	–	–	–	–
lroundf	–	–	–	–	–
lround	–	–	–	–	–
lroundl	–	–	–	–	–
llroundf	–	–	–	–	–
llround	–	–	–	–	–
llroundl	–	–	–	–	–

modff	–	–	–	–	–
modf	–	–	–	–	–
modfl	–	–	–	–	–
nearbyintf	–	–	–	–	–
nearbyint	–	–	–	–	–
nearbyintl	–	–	–	–	–
nextafterf	–	–	–	–	–
nextafter	–	–	–	–	–
nextafterl	–	–	–	–	–
nexttowardf	–	–	–	–	–
nexttoward	–	–	–	–	–
nexttowardl	–	–	–	–	–
powf	–	–	–	–	–
pow	–	–	–	–	–
powl	–	–	–	–	1
remainderf	–	–	–	–	–
remainder	–	–	–	–	–
remainderl	–	–	–	–	–
remquof	–	–	–	–	–
remquo	–	–	–	–	–
remquol	–	–	–	–	–
rintf	–	–	–	–	–
rint	–	–	–	–	–
rintl	–	–	–	–	–
roundf	–	–	–	–	–
round	–	–	–	–	–
roundl	–	–	–	–	–
scalbf	–	–	–	–	–
scalb	–	–	–	–	–
scalbl	–	–	–	–	–
scalbnf	–	–	–	–	–
scalbn	–	–	–	–	–
scalbnl	–	–	–	–	–
scalblnf	–	–	–	–	–
scalbln	–	–	–	–	–
scalblnl	–	–	–	–	–
sinf	–	–	–	–	–
sin	–	–	–	–	–
sinl	–	–	–	–	1
sincosf	1	–	–	1	1
sincos	1	–	–	1	1
sincosl	–	–	1	1	1
sinhf	–	–	–	–	–
sinh	–	–	–	–	–
sinhl	–	–	–	–	1
sqrtf	–	–	–	–	–
sqrt	–	–	–	–	–

sqrtl	–	–	–	–	–
tanf	–	–	–	–	–
tan	1	–	1	1	1
tanl	–	–	–	–	1
tanhf	–	–	–	–	–
tanh	–	–	–	–	–
tanhl	–	–	–	–	1
tgammaf	1	–	1	1	1
tgamma	1	–	2	1	1
tgammal	–	–	1	1	1
truncf	–	–	–	–	–
trunc	–	–	–	–	–
truncl	–	–	–	–	–
y0f	1	–	1	1	1
y0	2	–	2	2	2
y0l	–	–	1	1	1
y1f	2	–	2	2	2
y1	3	–	2	3	3
y1l	–	–	1	1	2
ynf	2	–	3	2	2
yn	3	–	2	3	3
ynl	–	–	4	2	2

Function	S/390	SH4	Sparc 32–bit	Sparc 64–bit	x86_64/fpu
acosf	–	–	–	–	–
acos	–	–	–	–	1
acosl	1	–	–	–	–
acoshf	–	–	–	–	–
acosh	–	–	–	–	–
acoshl	1	–	–	–	–
asinf	–	2	–	–	–
asin	–	1	–	–	–
asinl	–	–	–	–	1
asinhf	–	–	–	–	–
asinh	–	–	–	–	–
asinhl	–	–	–	–	–
atanf	–	–	–	–	–
atan	–	–	–	–	–
atanl	–	–	–	–	–
atanhf	1	–	1	1	1
atanh	–	1	–	–	–
atanhl	–	–	–	–	1
atan2f	1	4	6	6	1
atan2	–	–	–	–	–
atan2l	1	–	1	1	–
cabsf	–	1	–	–	–
cabs	–	1	–	–	–

cabsl	–	–	–	–	–
cacosf	–	1 + i 1	–	–	0 + i 1
cacos	–	1 + i 0	–	–	–
cacosl	0 + i 1	–	0 + i 1	0 + i 1	0 + i 2
cacoshf	7 + i 3	7 + i 3	7 + i 3	7 + i 3	7 + i 3
cacosh	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
cacoshl	0 + i 1	–	5 + i 1	5 + i 1	6 + i 1
cargf	–	–	–	–	–
carg	–	–	–	–	–
cargl	–	–	–	–	–
casinf	1 + i 0	2 + i 1	1 + i 0	1 + i 0	1 + i 1
casin	1 + i 0	3 + i 0	1 + i 0	1 + i 0	1 + i 0
casinl	0 + i 1	–	0 + i 1	0 + i 1	2 + i 2
casinhf	1 + i 6	1 + i 6	1 + i 6	1 + i 6	1 + i 6
casinh	5 + i 3	5 + i 3	5 + i 3	5 + i 3	5 + i 3
casinhl	4 + i 2	–	4 + i 2	4 + i 2	5 + i 5
catanf	4 + i 1	4 + i 1	4 + i 1	4 + i 1	4 + i 1
catan	0 + i 1	0 + i 1	0 + i 1	0 + i 1	0 + i 1
catanl	0 + i 1	–	0 + i 1	0 + i 1	–
catanhf	0 + i 6	1 + i 6	0 + i 6	0 + i 6	0 + i 6
catanh	4 + i 0	4 + i 1	4 + i 0	4 + i 0	4 + i 0
catanhl	1 + i 1	–	1 + i 1	1 + i 1	1 + i 0
cbrtf	–	–	–	–	–
cbrt	1	1	1	1	1
cbrtl	1	–	1	1	1
ccosf	1 + i 1	0 + i 1	1 + i 1	1 + i 1	1 + i 1
ccos	1 + i 0	1 + i 1	1 + i 0	1 + i 0	1 + i 0
ccosl	1 + i 1	–	1 + i 1	1 + i 1	1 + i 1
ccoshf	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
ccosh	1 + i 0	1 + i 1	1 + i 0	1 + i 0	1 + i 1
ccoshl	1 + i 1	–	1 + i 1	1 + i 1	0 + i 1
ceilf	–	–	–	–	–
ceil	–	–	–	–	–
ceill	–	–	–	–	–
cexpf	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
cexp	–	1 + i 0	–	–	–
cexpl	1 + i 1	–	1 + i 1	1 + i 1	0 + i 1
cimagf	–	–	–	–	–
cimag	–	–	–	–	–
cimagl	–	–	–	–	–
clogf	1 + i 3	0 + i 3	1 + i 3	1 + i 3	1 + i 3
clog	–	0 + i 1	–	–	–
clogl	1 + i 0	–	1 + i 0	1 + i 0	1 + i 0
clog10f	1 + i 5	1 + i 5	1 + i 5	1 + i 5	1 + i 5
clog10	0 + i 1	1 + i 1	0 + i 1	0 + i 1	1 + i 1
clog10l	1 + i 1	–	1 + i 1	1 + i 1	1 + i 1
conjf	–	–	–	–	–

conj	–	–	–	–	–
conjl	–	–	–	–	–
copysignf	–	–	–	–	–
copysign	–	–	–	–	–
copysignl	–	–	–	–	–
cosf	1	1	1	1	1
cos	2	2	2	2	2
cosl	1	–	1	1	1
coshf	–	–	–	–	–
cosh	–	–	–	–	–
coshl	–	–	–	–	–
cpowf	4 + i 2	4 + i 2	4 + i 2	4 + i 2	5 + i 2
cpow	2 + i 2	1 + i 1.1031	2 + i 2	2 + i 2	2 + i 2
cpowl	10 + i 1	–	10 + i 1	10 + i 1	5 + i 2
cprojf	–	–	–	–	–
cproj	–	–	–	–	–
cprojl	–	–	–	–	–
crealf	–	–	–	–	–
creal	–	–	–	–	–
creall	–	–	–	–	–
csinf	–	0 + i 1	–	–	0 + i 1
csin	–	–	–	–	0 + i 1
csinl	1 + i 1	–	1 + i 1	1 + i 1	1 + i 0
csinhf	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
csinh	0 + i 1	0 + i 1	0 + i 1	0 + i 1	1 + i 1
csinhl	1 + i 0	–	1 + i 0	1 + i 0	1 + i 2
csqrtf	1 + i 0	1 + i 1	1 + i 0	1 + i 0	1 + i 0
csqrt	–	1 + i 0	–	–	–
csqrtl	1 + i 1	–	1 + i 1	1 + i 1	–
ctanf	–	1 + i 1	–	–	0 + i 1
ctan	1 + i 1	1 + i 1	1 + i 1	1 + i 1	1 + i 1
ctanl	1 + i 2	–	1 + i 2	1 + i 2	439 + i 3
ctanhf	2 + i 1	2 + i 1	2 + i 1	2 + i 1	2 + i 1
ctanh	1 + i 0	2 + i 2	1 + i 0	1 + i 0	1 + i 1
ctanhl	1 + i 1	–	1 + i 1	1 + i 1	5 + i 25
erff	–	–	–	–	–
erf	1	–	1	1	1
erfl	–	–	–	–	–
erfcf	1	12	–	–	–
erfc	1	24	1	1	1
erfcl	1	–	1	1	1
expf	–	–	–	–	–
exp	–	–	–	–	–
expl	–	–	–	–	–
exp10f	2	2	2	2	2
exp10	6	6	6	6	6
exp10l	1	–	1	1	1

exp2f	–	–	–	–	–
exp2	–	–	–	–	–
exp2l	2	–	2	2	–
expm1f	1	1	1	1	1
expm1	1	–	1	1	1
expm1l	1	–	1	1	–
fabsf	–	–	–	–	–
fabs	–	–	–	–	–
fabsl	–	–	–	–	–
fdimf	–	–	–	–	–
fdim	–	–	–	–	–
fdiml	–	–	–	–	–
floorf	–	–	–	–	–
floor	–	–	–	–	–
floorl	–	–	–	–	–
fmaf	–	–	–	–	–
fma	–	–	–	–	–
fmal	–	–	–	–	–
fmaxf	–	–	–	–	–
fmax	–	–	–	–	–
fmaxl	–	–	–	–	–
fminf	–	–	–	–	–
fmin	–	–	–	–	–
fminl	–	–	–	–	–
fmodf	–	1	–	–	–
fmod	–	2	–	–	–
fmodl	1	–	–	–	–
frexpf	–	–	–	–	–
frexp	–	–	–	–	–
frexpl	–	–	–	–	–
gammaf	–	–	–	–	–
gamma	–	–	–	–	–
gammal	1	–	1	1	1
hypotf	1	1	1	1	1
hypot	–	1	–	–	–
hypotl	–	–	–	–	–
ilogbf	–	–	–	–	–
ilogb	–	–	–	–	–
ilogbl	–	–	–	–	–
j0f	2	2	1	2	2
j0	3	2	2	2	2
j0l	2	–	2	2	1
j1f	2	2	2	2	2
j1	1	1	1	1	1
j1l	4	–	4	4	1
jnf	4	4	4	4	4
jn	4	6	4	4	4

jnl	4	–	4	4	2
lgammaf	2	2	2	2	2
lgamma	1	1	1	1	1
lgammal	1	–	1	1	1
lrintf	–	–	–	–	–
lrint	–	–	–	–	–
lrintl	–	–	–	–	–
llrintf	–	–	–	–	–
llrint	–	–	–	–	–
llrintl	–	–	–	–	–
logf	–	1	–	–	–
log	–	1	–	–	–
logl	–	–	–	–	–
log10f	2	1	2	2	2
log10	1	1	1	1	1
log10l	1	–	1	1	1
log1pf	1	1	1	1	1
log1p	–	1	–	–	–
log1pl	1	–	1	1	–
log2f	–	1	–	–	–
log2	–	1	–	–	–
log2l	1	–	1	1	–
logbf	–	–	–	–	–
logb	–	–	–	–	–
logbl	–	–	–	–	–
lroundf	–	–	–	–	–
lround	–	–	–	–	–
lroundl	–	–	–	–	–
llroundf	–	–	–	–	–
llround	–	–	–	–	–
llroundl	–	–	–	–	–
modff	–	–	–	–	–
modf	–	–	–	–	–
modfl	–	–	–	–	–
nearbyintf	–	–	–	–	–
nearbyint	–	–	–	–	–
nearbyintl	–	–	–	–	–
nextafterf	–	–	–	–	–
nextafter	–	–	–	–	–
nextafterl	–	–	–	–	–
nexttowardf	–	–	–	–	–
nexttoward	–	–	–	–	–
nexttowardl	–	–	–	–	–
powf	–	–	–	–	–
pow	–	–	–	–	–
powl	1	–	–	–	–
remainderf	–	–	–	–	–

remainder	–	–	–	–	–
remainderl	–	–	–	–	–
remquof	–	–	–	–	–
remquo	–	–	–	–	–
remquol	–	–	–	–	–
rintf	–	–	–	–	–
rint	–	–	–	–	–
rintl	–	–	–	–	–
roundf	–	–	–	–	–
round	–	–	–	–	–
roundl	–	–	–	–	–
scalbf	–	–	–	–	–
scalb	–	–	–	–	–
scalbl	–	–	–	–	–
scalbnf	–	–	–	–	–
scalbn	–	–	–	–	–
scalbnl	–	–	–	–	–
scalblnf	–	–	–	–	–
scalbln	–	–	–	–	–
scalblnl	–	–	–	–	–
sinf	–	–	–	–	–
sin	–	–	–	–	–
sinl	1	–	–	–	–
sincosf	1	1	1	1	1
sincos	1	1	1	1	1
sincosl	1	–	1	1	1
sinhf	–	1	–	–	–
sinh	–	1	–	–	–
sinhl	–	–	–	–	–
sqrtf	–	–	–	–	–
sqrt	–	–	–	–	–
sqrtl	1	–	1	1	–
tanf	–	–	–	–	–
tan	1	0.5	1	1	1
tanl	1	–	–	–	–
tanhf	–	1	–	–	–
tanh	–	1	–	–	–
tanhl	1	–	1	1	–
tgammaf	1	1	1	1	1
tgamma	1	1	1	1	1
tgammal	1	–	1	1	1
truncf	–	–	–	–	–
trunc	–	–	–	–	–
truncl	–	–	–	–	–
y0f	1	1	1	1	1
y0	2	2	2	2	2
y0l	3	–	3	3	3

y1f	2	2	2	2	2
y1	3	3	3	3	3
y1l	1	–	1	1	1
ynf	2	2	2	2	2
yn	3	3	3	3	3
ynl	5	–	5	5	4

8. Rasgeleymiş gibi Görünen Sayılar

Bu kısımda rasgeleymiş gibi görünen sayıları üreten GNU oluşumlarından bahsedilecektir. Üreilen sayılar gerçekten rasgele değildir; genellikle sürekli tekrarlanan ama tekrar dönemleri çok uzun olduğundan tekrarlandığı farkedilemeyen dizilimler şeklindedirler. Rasgele Sayı Üreteci sonraki rasgele sayıyı hesaplamak için ve yeni tohum değerini hesaplamak için bir ***tohum değeri*** kullanır.

Üretilen sayılar yazılımın çalışması esnasında önceden tahmin edilemez gibi göründüğü halde, aynı yazılım başka süreçlerde *tamamen aynı* sayı dizilimlerini kullanır. Bu dahili tohumun hep aynı olması sebebiyle böyledir. Yazılımda hata ayıklamaya kalkışırsanız bunun böyle olduğunu görürsünüz ama yazılımın önceden tahmin edilemeyen tarzda davranmasını isterseniz bu size pek yardımcı olmaz. Yazılımınızın her süreçte farklı rasgele sayı dizilimleri ile çalışmasını isterseniz, her seferinde farklı bir tohum belirtmeniz gerekir. Amaç sıradansa, tohum olarak daima eşsiz bir değer olan o anki mutlak zaman seçilir.

Belli bir makina türünde sayıları tekrarlanan dizilimler şeklinde elde etmek isterseniz, rassgele sayı üretecine aynı tohum değerini belirtmeniz gerekir. Standart olmak anlamında belirli bir tohum değeri yoktur; farklı C kütüphaneleri ya da farklı işlemci türlerinin herbirinin kendine özgü tohum değeri vardır ve bunlar size daima farklı rasgele sayılar verirler.

GNU C kütüphanesi standart ISO C rasgele sayı işlevleri yanında BSD ve SVID'den türetilmiş olanları da destekler. BSD ve ISO C işlevleri biraz sınırlı bir işlevsellikle benzerdirler. Sadece az sayıda rasgele bit gerekliyse, ISO C arayüzünü, **rand** ve **srand** işlevlerini kullanmanızı öneririz. SVID işlevleri daha esnek bir arayüz sağlar; daha iyi rasgele sayı üreteci algoritmaları içerdiğinden, her çağrıda daha fazla rasgele bit (48bite kadar) üretirler ve rasgele gerçek sayılar üretebilirler.

8.1. ISO C Rasgele Sayı İşlevleri

Bu bölümde ISO C standardının bir parçası olan rasgele sayı işlevleri ele alınacaktır.

Bu oluşumları kullanabilmek için yazılımınıza **stdlib.h** başlık dosyasını dahil etmelisiniz.

`int` **RAND_MAX** makro

Bu makronun değeri, **rand** işlevinin döndürebileceği en büyük değeri gösteren bir tamsayı sabittir. GNU C kütüphanesinde bu makronun değeri 32 bitle gösterilebilen en büyük işaretli tamsayı olan **2147483647**'dir. Diğer kütüphanelerde **32767** gibi düşük bir değer olabilir.

`int` **rand**`(void)` işlev

rand işlevi dizilimdeki sonraki rasgeleymiş gibi görünen sayıyı döndürür. Değerler **0** ile **RAND_MAX** arasındadır.

`void` **srand**`(unsigned int` *tohum*`)` işlev

Rasgeleymiş gibi görünen yeni bir sayı dizilimi için tohum olarak *tohum*'u etkin kılar. **srand** ile yeni bir tohum atamadan önce **rand** çağrısı yaparsanız işlev öntanımlı tohum değeri olan **1** değerini kullanır.

Yazılımınızın her çalıştırılışında farklı bir rasgele sayı dizilimi kullanmasını istiyorsanız **srand (time (0))** çağrısını kullanın.

POSIX.1, standart C işlevlerini çok evreli yazılımlarda yeniden üretilebilir rasgele sayıları destekleyen işlevlerle genişletmişse de bu genişletme biraz kötü tasarlandığından kullanışsızdır.

```
int rand_r(unsigned int tohum)                                    işlev
```

Bu işlev **rand** işlevinin yaptığı gibi 0 ile **RAND_MAX** arasında bir rasgele sayı ile döner. Ancak, onun tüm durumu *tohum* argümanında saklanır. Bunun anlamı rasgele sayı üretecinin durumunun sadece **unsigned int** türünün sahip olabildiği bit sayısı kadar olduğudur. Bu iyi bir rasgele sayı üreteci için çok çok azdır.

Eğer yazılımınız bir evresel rasgele sayı üreteci gerektiriyorsa, SVID rasgele sayı üretecinde evresel GNU oluşumlarını kullanmanızı öneririz. POSIX.1 arayüzünü ise sadece GNU oluşumlarını kullanamadığınız zaman kullanmalısınız.

8.2. BSD Rasgele Sayı İşlevleri

Bu bölümde BSD'den türetilmiş rasgele sayı üretim işlevlerine yer verilmiştir. GNU kütüphenesiyle bu işlevleri kullanmanın bir getirisi yoktur; onları sadece BSD uyumluluğu adına destekliyoruz.

Bu işlevlerin prototipleri **stdlib.h** dosyasında bildirilmiştir.

```
long int random(void)                                             işlev
```

Bu işlev dizilimdeki sonraki rasgele gibi görünen sayıyı döndürür. Dönen değer **0** ile **RAND_MAX** arasındadır.

Bilgi

Bu işlev geçici olarak, **long int** daha geniş olsa bile dönüş değerinin daima 32 bit olduğunu belirten **int32_t** türünde bir değer döndürecek şekilde tanımlanmıştır. Standart farklı olmasını ister. Kullanıcıların 32 bitlik sınırlamanın farkında olması için böyle yapılmıştır.

```
void srandom(unsigned int tohum)                                  işlev
```

srandom işlevi rasgele sayı üretecinin durumunu *tohum* tamsayısına tabanlandırır. *tohum* olarak **1** değerini verirseniz işlevin rasgele sayıların öntanımlı dizilimini üretmesine sebep olursunuz.

Yazılımınızın her çalıştırılışında farklı bir rasgele sayı dizilimi kullanmasını istiyorsanız **srandom (time (0))** çağrısını kullanın.

```
void *initstate(unsigned int tohum,                               işlev
                void         *durum,
                size_t        boyut)
```

initstate işlevi rasgele sayı üretecini ilklendirmek için kullanılır. *durum* argümanı durum bilgisini tutan *boyut* baytlık bir dizidir. *tohum*'a bağlı olarak ilklendirilir. *boyut* değeri 8 ile 256 arasında ve ikinin üstel katlarında olmalıdır. Daha büyük *durum* dizisi daha iyidir.

İşlev önceki durum bilgisi dizisinin değerini döndürür. Bu değeri daha sonra durumu eski haline getirmek için **setstate** işlevinde argüman olarak kullanabilirsiniz.

```
void *setstate(void *durum)                                       işlev
```

setstate işlevi rasgele sayı durumunu *durum* durum bilgisi ile eski haline getirir. Argüman evvelce yapılmış bir *initstate* veya *setstate* çağrısından dönmüş bir değer olmalıdır.

İşlev önceki durum bilgisi dizisinin değerini döndürür. Bu değeri daha sonra durumu eski haline getirmek için **setstate** işlevinde argüman olarak kullanabilirsiniz.

İşlev başarısız olursa **NULL** ile döner.

Bu bölümde buraya kadar bahsedilen dört işlevin hepsi durum bilgisini tüm evrelerle paylaşır. Durum bilgisi kullanıcı tarafından erişilebilir değildir, sadece bu işlevlerle erişilebilir. Bu, kendi rasgele sayı üreteçlerinin olmasını gerektiren evreler bakımından çalışılması zor bir durum oluşturur.

GNU C kütüphanesi bu dört işleve ek olarak evreye özel rasgele sayı üreteçlerini mümkün kılmak için durumu ayrı bir parametre olarak alan dört işlev daha içerir. Aslında, bundan önce bahsedilen dört işlev de dahili olarak aşağıdaki arayüz ile gerçeklenmiştir.

stdlib.h başlık dosyası şu türün bir tanımını içerir:

```
struct random_data                                                    veri türü
```

struct random_data türündeki nesneler rasgele sayı üretecinin durumunu ifade etmek için gereken bilgiyi içerir. Türün tam bir tanımı mevcut olduğu halde türün şeffaf olmadığı varsayılmalıdır.

Durum bilgisini değiştiren işlevler bundan önce açıklanan işlevlerin yaptıklarını evresel kullanıma uygun olarak yaparlar.

```
int random_r(struct random_data *restrict tampon,                     işlev
             int32_t *restrict            sonuç)
```

random_r işlevi genel durum bilgisi yerine ilk parametresi ile gösterilen nesnedeki durumu kullanması dışında **random** işlevi gibi davranır.

```
int srandom_r(unsigned int         tohum,                             işlev
              struct random_data *tampon)
```

srandom_r işlevi genel durum bilgisi yerine ikinci parametresi ile gösterilen nesnedeki durumu kullanması dışında **srandom** işlevi gibi davranır.

```
int initstate_r(unsigned int                  tohum,                  işlev
                char *restrict                durum–tamponu,
                size_t                        durum–uzunluğu,
                struct random_data *restrict tampon)
```

initstate_r işlevi genel durum bilgisi yerine dördüncü parametresi ile gösterilen nesnedeki durumu kullanması dışında **initstate** işlevi gibi davranır.

```
int setstate_r(char *restrict                durum–tamponu,           işlev
               struct random_data *restrict tampon)
```

setstate_r işlevi genel durum bilgisi yerine ikinci parametresi ile gösterilen nesnedeki durumu kullanması dışında **setstate** işlevi gibi davranır.

8.3. SVID Rasgele Sayı İşlevleri

SVID sistemlerdeki C kütüphanesi rasgele sayı üretim işlevlerinin daha farklı çeşitlerini içerir. Kullanıcı farklı biçimlerde rasgele bitler döndüren işlevler arasından seçim yapabilir.

Genel olarak iki çeşit işlev vardır. İlki çeşitli işlevler ve sürecin tüm evreleri tarafından paylaşılan bir rasgele sayı üreteci durumunu kullanır. İkincisi ise durumu kullanıcının elde etmesini gerektirir.

Tüm işlevler aynı sabitlerle aynı benzeş[7] formülü kullanırlar. Bu formül:

```
Y = (a * X + c) mod m
```

Burada, X üretecin başlangıçtaki durumu, Y ise son durumudur. **a** ve **c** üretecin çalışma yöntemini belirleyen sabitlerdir. Öntanımlı olarak bunlar:

```
a = 0x5DEECE66D = 25214903917
c = 0xb = 11
```

Ancak, bunlar kullanıcı tarafından değiştirilebilir. **m** ise, durum bilgisi 48 bitlik bir diziden oluştuğundan 2^48'dir.

Bu işlevlerin prototipleri **stdlib.h** başlık dosyasındadır.

`double` **drand48**`(void)` işlev

Bu işlev **double** türünde **0.0** ile **1.0** (hariç) arasında bir değerle döner. Rasgele bitler C kütüphanesindeki rasgele sayı üretecinin genel durum bilgisi tarafından belirlenir.

IEEE 754'de **double** türü 52 bitlik ondalık kısımdan oluştuğundan bunun 4 biti rasgele sayı üreteci tarafından ilklendirilemez. Bunlar (şüphesiz) en kıymetsiz bitlerden seçilir ve **0**'a ilklendirilir.

`double` **erand48**`(unsigned short int` *xsubi*`[3])` işlev

Bu işlev **drand48** işlevi gibi **double** türünde **0.0** ile **1.0** (hariç) arasında bir değerle döner. Argüman rasgele sayı üretecinin durumunu ifade eden bir dizidir.

Bu işlev rasgele sayıları garantilemek için diziyi güncellediğinden ardarda çağrılabilir. Dizi, yeniden üretilebilir sonuç sağlamak için ilk kullanımdan önce ilklendirilmelidir.

`long int` **lrand48**`(void)` işlev

lrand48 işlevi **0** ile **2^31** (hariç) arasında bir tamsayı değerle döner. Dönen değerin türü **long int** olsa bile 32 bitten daha geniş değildir, dolayısıyla daha yüksek bir değer dönmez. Rasgele bitler C kütüphanesindeki rasgele sayı üretecinin genel durum bilgisi tarafından belirlenir.

`long int` **nrand48**`(unsigned short int` *xsubi*`[3])` işlev

Bu işlev **lrand48** işlevi gibi **0** ile **2^31** (hariç) arasında bir tamsayı değerle döner, ama rasgele bitleri üretmekte kullanılan rasgele sayı üreteci durum bilgisi işleve parametre olarak verilen dizi tarafından belirlenir.

Bu işlev rasgele sayıları garantilemek için diziyi güncellediğinden ardarda çağrılabilir. Dizi, yeniden üretilebilir sonuç sağlamak için ilk kullanımdan önce ilklendirilmelidir.

`long int` **mrand48**`(void)` işlev

mrand48 işlevi **-2^31** ile **2^31** (hariç) arasında sayı döndürmesi dışında **lrand48** işlevi gibidir.

`long int` **jrand48**`(unsigned short int` *xsubi*`[3])` işlev

jrand48 işlevi **-2^31** ile **2^31** (hariç) arasında sayı döndürmesi dışında **lrand48** işlevi gibidir. **xsubi** parametresi için gereksinimler aynıdır.

Rasgele sayı üretecinin ilk durumu çeşitli yollarla ilklendirilebilir. Yöntemler sağlanan bilginin bütünlüğüne bağlı olarak değişir.

`void` **srand48**`(long int` *tohum*`)` işlev

srand48 işlevi rasgele sayı üretecinin dahili durumunun en kıymetli 32 bitine, *tohum* parametresinin en kıymetsiz 32 bitini yerleştirir. Alt 16 bit **0x330E** değeri ile ilklendirilir. **long int** türü 32 bitten daha fazla bit içerse bile sadece alt 32 biti kullanılır.

Bu sınırlamayla, bu işlevin durumu ilklendirmesi çok kullanışlı değildir, ama **srand48 (time (0))** gibi bir kullanım onu kolay kullanılır yapar.

Bu işlevin bir yan etkisi, dahili durumda benzetimli formüldeki **a** ve **c** değerleri yukarıda verilen öntanımlı değerlere ayarlanır. This is of importance once the user has called the **lcong48** function (see below).

```
unsigned short int *seed48(unsigned short int tohum16[3])                    işlev
```

seed48 işlevi dahili rasgele sayı üretecinin durumunun 48 bitinin tamamını *tohum16* parametresinin içeriğinden ilklendirir. Burada *tohum16* dizisinin ilk elemanının alt 16 biti dahili durumun en kıymetsiz 16 bitini ilklendirir, *tohum16*[1]'in alt 16 biti durumun ortadaki 16 bitini ilklendirir, *tohum16*[2]'nin alt 16 biti de durumun en kıymetli 16 bitini ilklendirir.

srand48 işlevinin tersine bu işlev kullanıcının durumun tüm 48 bitini ilklendirmesini sağlar.

İşlevin dönüş değeri dahili durumun değişmeden önceki değerini içeren bir dizidir. Bu dizi rasgele sayı üretecini belirli bir durumdan tekrar başlatmak için kullanılabilir. Aksi takdirde değer yoksayılabilir.

srand48'deki gibi benzetimli formüldeki **a** ve **c** değerleri öntanımlı değerlere ayarlanır.

Rasgele sayı üretecini ilklendirmek için benzetimli formüldeki parametreleri değiştirebilmenizi sağlayarak daha fazla bilgi belirtebilmenizi mümkün kılan bir işlev daha vardır.

```
void lcong48(unsigned short int param[7])                                     işlev
```

lcong48 işlevi rasgele sayı üretecinin durumunu tamamen değiştirebilmenizi mümkün kılar. **srand48** ve **seed48** işlevlerinin aksine, bu işlev ile ayrıca benzetimli formüldeki sabitleri de değiştirebilirsiniz.

param dizisindeki 7 elemandan 0, 1, 2 indisli elemanlardaki girdilerin en kıymetsiz 16 biti ilk durumu belirler; 3, 4, 5 indisli elemanlardaki girdiler 48 bitlik **a** sabitini, 6. elemanın girdisi ise 16 bitlik **c** değerini belirler.

Yukardaki işlevlerin hepsi ortak olarak benzetimli formüldeki genel parametreleri kullanır. Çok evreli yazılımlarda bazan farklı evrelerde farklı parametrelere sahip olmak kullanışlı olabilir. Bu sebeple yukardaki işlevlerin tümünün rasgele sayı üreteci durumunu genel durum yerine kullanıcı tanımlı bir tamponda tutarak çalışan sürümleri de vardır.

Tüm evrelerin durumu içeren bir diziye gösterici alan işlevleri kullanması durumunda böyle bir sorunun olmayacağına dikkat edin. Aşağıda hesaplanan rasgele sayılar aynı döngüyü izler ama eğer dizideki durum farklıya tüm evrelerin kendilerine özel bir rasgele sayı üreteci olacaktır.

Kulanıcı tanımlı tampon **struct drand48_data** türünde olmalıdır. Bu türün şeffaf olmadığı kabul edilir, dolayısıyla doğrudan değiştirilmemelidir.

```
int drand48_r(struct drand48_data *tampon,                                    işlev
              double              *sonuç)
```

Bu işlev, genel rasgele sayı üreteci parametrelerini değiştirmemesi, ama bunun yerine *tampon* göstericisi üzerinden sağlanan tampondaki parametreleri değiştirmesi ile **drand48** işlevinden farklıdır. Rasgele sayı *sonuç* ile gösterilen değişkende döndürülür.

İşlevin dönüş değeri çağrının başarısını gösterir. Eğer dönüş değeri **0**'dan küçükse bir hata oluşmuştur. Bu durumda *errno* değişkenine hata durumu atanır.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

```
int erand48_r(unsigned short int   xsubi[3],                                  işlev
              struct drand48_data *tampon,
              double              *sonuç)
```

erand48_r işlevi **erand48** gibi çalışır, ama rasgele sayı üretecini ifade eden bir *tampon* argümanı alır. Rasgele sayı üretecinin durumu **xsubi** dizisinden alınır. Rasgele sayı *sonuç* ile gösterilen değişkende döndürülür.

Çağrı başarılı olmuşsa dönüş değeri negatif bir değer değildir.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

```
int lrand48_r(struct drand48_data *tampon,                                    işlev
              double              *sonuç)
```

Bu işlev **lrand48** işlevine benzer, ama ek olarak **drand48** işlevindeki gibi rasgele sayı üretecinin durumunu ifade eden tampona bir gösterici alır.

İşlevin dönüş değeri negatif değilse sonuç *sonuç* ile gösterilen değişkende döndürülür. Aksi takdirde bir hata oluşmuş demektir.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

```
int nrand48_r(unsigned short int   xsubi[3],                                  işlev
              struct drand48_data *tampon,
              long int            *sonuç)
```

nrand48_r işlevi **nrand48** işlevi gibi **0** ile **2^31** arasında bir rasgele sayı üretir. Fakat, benzetimli formül için genel parametreleri kullanmak yerine *tampon* ile gösterilen tampondaki bilgiyi kullanır. Durum *xsubi* dizisindeki değerlerle ifade edilir.

İşlevin dönüş değeri negatif değilse sonuç *sonuç* ile gösterilen değişkende döndürülür.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

```
int mrand48_r(struct drand48_data *tampon,                                    işlev
              double              *sonuç)
```

Bu işlev **mrand48** işlevine benzer. Fakat bu işlev ailesinin diğer evresel işlevleri gibi bu işlev de *tampon* ile gösterilen tampondaki değerle ifade edilen rasgele sayı üretecini kullanır.

İşlevin dönüş değeri negatif değilse sonuç *sonuç* ile gösterilen değişkende döndürülür.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

```
int jrand48_r(unsigned short int   xsubi[3],                                  işlev
              struct drand48_data *tampon,
              long int            *sonuç)
```

jrand48_r işlevi **jrand48** işlevine benzer. Fakat bu işlev ailesinin diğer evresel işlevleri gibi bu işlev de benzetimli formülün parametrelerini *tampon* ile gösterilen tampondan kullanır.

İşlevin dönüş değeri negatif değilse sonuç *sonuç* ile gösterilen değişkende döndürülür.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

Yukarıdaki evresel işlevleri kullanmadan önce **struct drand48_data** türündeki tampon ilklendirilmelidir. Bunu yapmanın en kolay yolu tüm tamponu boş karakterle doldurmaktır:

```
memset (buffer, '\0', sizeof (struct drand48_data));
```

Bundan sonra bu ailenin evresel işlevlerinin kullanılması rasgele sayı üretecini durum ve benzetimli formülün parametreleri için öntanımlı değerlere özdevinimli olarak ilklendirecektir.

Diğer bir olasılık, işlevleri tamponu açıkça ilklendirerek kullanmaktır. İşlevin parametrelerine bakarak tamponun nasıl ilklendirileceği açıkça belli olduğuna göre sonuç daima umduğunuz olmayacağından bu işlevleri kullanmanızı hararetle tavsiye ediyoruz.

```
int srand48_r(long int tohum,                                    işlev
              struct drand48_data *tampon)
```

Rasgele sayı üreteci **srand48** işlevindeki gibi *tampon* içindeki bilgi ile ifade edilir. Durum *tohum* parametresinden ilklendirilir ve benzetimli formülün parametreleri de öntanımlı değerleriyle ilkelendirilir.

İşlevin dönüş değeri negatif değilse işlev başarılıdır.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

```
int seed48_r(unsigned short int tohum16[3],                      işlev
             struct drand48_data *tampon)
```

Bu işlev **srand48_r** işlevine benzer ama **seed48** işlevi gibi durumun 48 bitinin tamamını *tohum16* parametresinden ilklendirir.

İşlevin dönüş değeri negatif değilse işlev başarılıdır. **seed48** işlevinin yaptığı gibi rasgele sayı üretecinin önceki durumuna bir gösterici döndürmez. Eğer kullanıcı daha sonra kullanmak üzere durum bilgisini saklamak isterse *tampon* ile gösterilen tamponun tamamını kopyalayabilir.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

```
int lcong48_r(unsigned short int param[7],                       işlev
              struct drand48_data *tampon)
```

Bu işlev *tampon* ile ifade edilen rasgele sayı üretecininin tüm özelliklerini *param* içindeki veri ile ilklendirir. Burada işlevin *param* ve *tampon* içeriklerini kopyalamaktan daha fazlasını yaptığı bir gerçektir. Daha fazla çalışma gerektirmesi ve bundan dolayı bu işlevi kullanmak rasgele sayı üretecini doğrudan ilklendirmekten daha iyidir.

İşlevin dönüş değeri negatif değilse işlev başarılıdır.

Bu işlev bir GNU oluşumu olduğundan taşınabilir yazılımlarda kullanılmamalıdır.

9. Hızlı Kod mu, Küçük Kod mu Tercih Edilir?

Eğer bir uygulama çok sayıda gerçek sayı işlevi kullanıyorsa işlev çağrılarının maliyetinin ihmal edilemediği duruma sık rastlanır. Günümüz işlemcileri çoğunlukla kendi işlemlerini çok hızlı yapar, fakat işlev çağrıları komut boruhattını ikiye ayırır.

Bu sebeple GNU C kütüphanesi sık kullanılan matematik işlevlerinin bir çoğu için eniyilemeler sağlar. GNU CC kullanıldığında ve kullanıcı eniyileyiciyi etkinleştirmişse, çeşitli yeni satıriçi işlevler ve makrolar tanımlanır. Bu yeni işlevler ve makrolar kütüphanedeki işlevlerle aynı isimdedirler ve dolayısıyla onların yerine kullanılırlar. Satıriçi işlevler durumunda onların kullanılmasının makul olup olmayacağına derleyici karar verir ve bu karar genellikle doğru olur.

Bu, kütüphane işlevlerinin çağrılmasının gerekli olmadığı ve üretilen kodun hızı belirgin biçimde arttrabileceği anlamına gelir. Sakıncası kod boyutunun artacak olmasıdır ve bu artış her zaman ihmal edilebilir olmayacaktır.

Satıriçi işlevlerin iki çeşidi vardır: kütüphane işlevleriyle aynı sonucu verenler ve **errno** değişkenine değer atamayabilen ve kütüphane işlevleri ile karşılaştırıldığında hassasiyeti ve/veya argüman aralığını azaltabilen diğerleri. İkinci çeşit satıriçi işlevler sadece GNU CC'ye **-ffast-math** seçeneği verilirse mümkündür.

Satıriçi işlevlerin ve makroların istenmediği durumlarda **__NO_MATH_INLINES** sembolü sistem başlık dosyaları yazılıma dahil edilmeden önce tanımlanmış olmalıdır. Bu sadece kütüphane işlevlerinin kullanılacağını garanti eder. Şüphesiz, bu seçeneğin tercih edilip edilmediği projedeki her dosya için belirlenebilir.

Tüm donanımlar IEEE 754 standardını gerçeklemediğinden ve gerçeklenmiş olsa bile onun özelliklerinden bazılarının kullanımı önemli bir başarım kaybına yol açabilir. Örneğin, sinyal eylemcilerinin kurulması bazı işlemcilerin kayan nokta birimini çifte hesaplama zamanından fazlasına sebep olan borulanmamış çalışmaya zorlar.

XX. Aritmetik İşlevleri

İçindekiler

Bu kısımda, bir gerçek sayıyı tamsayı ve ondalık bileşenlerine ayırma, bir karmaşık sayının sanal bileşenini alma, vb. temel aritmetik işlemleri yapmak için kullanılan işlevlere yer verilmiştir. Bu işlevler `math.h` ve `complex.h` başlık dosyalarında bildirilmiştir.

1. Tamsayılar

C dilinde çeşitli tamsayı veri türleri tanımlanmıştır: tamsayı, küçük tamsayı, büyük tamsayı ve karakter; bunların her birinin işaretli ve işaretsiz türleri vardır. GNU C derleyici dili bunlara çok büyük tamsayıları da katarak kapsamı genişletir.

C tamsayı türleri, yapısal bakımdan farklı veri genişliklerine (sözcük genişlikleri) sahip makinalar arasında kodun taşınabilirliğini mümkün kılmak amacıyla tasarlanmıştır. Bu sorun, yazılımların hangi makinada çalışacağından bağımsız olarak, çoğunlukla belli tamsayı aralıklarına ihtiyaç duyacak şekilde, bazan da belli bir saklama alanı gerektirecek şekilde geliştirilmesi gerekliliğinden kaynaklanır.

Bu soruna yanıt olarak, GNU C kütüphanesi ihtiyaçlarınıza uygun olarak bildirebileceğiniz C tür tanımları içerir. Çünkü GNU C kütüphanesinin başlık dosyaları belli bir makinaya özel duruma getirilebilir, böylece yazılımınızın kaynak kodunda bunları bulundurmak zorunda kalmazsınız.

Bu tür tanımları, yani **`typedef`**'ler `stdint.h` başlık dosyasında bildirilmiştir.

Tam olarak N bitlik genişliğe sahip bir tamsayıya ihtiyacınız varsa, bit sayısı bakımından belli genişliklere ve işaretliliğe sahip aşağıdaki türlerden size uygun olan birini kullanın:

- int8_t
- int16_t
- int32_t
- int64_t
- uint8_t
- uint16_t
- uint32_t
- uint64_t

Eğer C derleyiciniz ya da hedef makina bu tamsayı genişliklerinden bazılarına izin vermiyorsa bunların eşdeğeri olan veri türleri bulunmayacaktır.

Eğer, belirli bir saklama alanına ihtiyacınız varsa ama en küçük veri yapısının *en azından* (at least) N bitlik olmasını istiyorsanız bunlardan birini kullanın:

- int_least8_t
- int_least16_t
- int_least32_t
- int_least64_t
- uint_least8_t
- uint_least16_t
- uint_least32_t
- uint_least64_t

Eğer belirli bir saklama alanından ziyade en azından N bitlik genişlik yanında en hızlı (fast) erişime sahip (ya da daha küçük olanı ile aynı erişim hızına sahip) veri yapılarına ihtiyacınız varsa bunlardan birini kullanın:

- int_fast8_t
- int_fast16_t
- int_fast32_t
- int_fast64_t
- uint_fast8_t
- uint_fast16_t
- uint_fast32_t
- uint_fast64_t

Eğer, platformun destekleyebildiği en büyük tamsayı genişliğini istiyorsanız, aşağıdakilerden birini kullanın. Bunları kullandığınız takdirde yazılımınızdaki değişken boyutlarında ve tamsayı aralıklarında bunları hesaba katmalısınız.

- intmax_t
- uintmax_t

GNU C kütüphanesi ayrıca, her tamsayı veri türü için olası asgari ve azami değerleri içeren makrolar içerir. Bu makro isimlerine bazı örnekler: **INT32_MAX**, **UINT8_MAX**, **INT_FAST32_MIN**, **INT_LEAST64_MIN**, **UINTMAX_MAX**, **INTMAX_MAX**, **INTMAX_MIN**. İşaretsiz tamsayı türlerin olası en küçük değerleri daima sıfır olduğundan bunlar için makrolar tanımlanmamıştır.

C'nin yerleşik tamsayı türleri ile kullanmak için C derleyiciniz ile gelen benzer makrolar vardır. Bunlardan *Veri Türü Ölçüleri* (sayfa: 821) bölümünde bahsedilmiştir.

Bu veri türlerinin saklandığı alanı bayt cinsinden öğrenmek için C'nin **sizeof** işlevini kullanabileceğinizi unutmayın.

2. Tamsayı Bölme

Bu kısımda tamsayı bölme işlemleri yapan işlevlere yer verilmiştir. Bu işlevler GNU C derleyicisi kullanıldığında gereksizdir, çünkü GNU C **/** işleci daima sıfıra doğru yuvarlar (pozitif değerleri alta, negatif değerleri üste). Fakat diğer C gerçeklemeleri **/** işleci ile yapılan işlemleri negatif argümanlar kullanıldığında farklı yönlerde yuvarlarlar. Bu durumda, yuvarlama yönü belli (daima sıfıra doğru) olan **div** ve **ldiv** işlevleri kullanışlı olur. Bölme işleminden kalan bölünen ile aynı işaretlidir.

Bu işlevler, *r*.**quot** * *bölen* + *r*.**rem** eşittir *bölünen* (bölüm çarpı bölen artı kalan eşittir bölünen) şeklinde ifadesini bulan bir *r* değeri ile döner.

Bu oluşumları yazılımınızda kullanmak için `stdlib.h` başlık dosyasını yazılımınıza dahil etmelisiniz.

div_t veri türü

Bu yapı **div** işlevinden dönen sonucu tutan veri türüdür. Şu üyelere sahiptir:

`int` **quot**
: Bölüm.

`int` **rem**
: Kalan.

```
div_t div(int bölünen,
          int bölen)
```
işlev

div işlevi *bölünen*'ın *bölen*'e bölünmesinden kalanı ve bölümü **div_t** türündeki yapı içinde döndürür.

Sonuç ifade edilemiyorsa (sıfırla bölme gibi) işlevin davranışı belirsizdir.

Çok kullanışlı olmasa da bir örnek:

```
div_t sonuc;
sonuc = div (20, -6);
```

Burada **sonuc.quot** değeri **-3** ve **sonuc.rem** değeri **2**'dir.

ldiv_t veri türü

Bu yapı **ldiv** işlevinden dönen sonucu tutan veri türüdür. Şu üyelere sahiptir:

`long int` **quot**
: Bölüm.

`long int` **rem**
: Kalan.

(Bu yapı elemanlarının **int** yerine **long int** türünde olması dışında **div_t** ile aynıdır.)

```
ldiv_t ldiv(long int bölünen,                                          işlev
            long int bölen)
```

ldiv işlevi argümanlarının **long int** türünde olması ve sonucu **ldiv_t** türünde bir yapı içinde döndürmesi dışında **div** işlevinin benzeridir.

lldiv_t veri türü

Bu yapı **lldiv** işlevinden dönen sonucu tutan veri türüdür. Şu üyelere sahiptir:

`long long int` **quot**
: Bölüm.

`long long int` **rem**
: Kalan.

(Bu yapı elemanlarının **int** yerine **long long int** türünde olması dışında **div_t** ile aynıdır.)

```
lldiv_t lldiv(long long int bölünen,                                   işlev
              long long int bölen)
```

lldiv işlevi argümanlarının **long long int** türünde olması ve sonucu **lldiv_t** türünde bir yapı içinde döndürmesi dışında **div** işlevinin benzeridir.

lldiv işlevi ISO C99 ile eklendi.

imaxdiv_t veri türü

Bu yapı **imaxdiv** işlevinden dönen sonucu tutan veri türüdür. Şu üyelere sahiptir:

`intmax_t` **quot**
: Bölüm.

`intmax_t` **rem**
: Kalan.

(Bu yapı elemanlarının **int** yerine **intmax_t** türünde olması dışında **div_t** ile aynıdır.)

intmax_t türü hakkında daha fazla bilgi için *Tamsayılar* (sayfa: 506) bölümüne bakınız.

```
imaxdiv_t imaxdiv(intmax_t bölünen,                                    işlev
                  intmax_t bölen)
```

imaxdiv işlevi argümanlarının **intmax_t** türünde olması ve sonucu **imaxdiv_t** türünde bir yapı içinde döndürmesi dışında **div** işlevinin benzeridir.

intmax_t türü hakkında daha fazla bilgi için *Tamsayılar* (sayfa: 506) bölümüne bakınız.

imaxdiv işlevi ISO C99 ile eklendi.

3. Gerçek Sayılar

Çoğu bilgisayar donanımı iki çeşit sayıyı destekler: tamsayılar (…–3, –2, –1, 0, 1, 2, 3…) ve gerçek sayılar. Bir gerçek sayı üç parçadan oluşur: ***ondalık kısım*** (İng: mantissa), ***üstel kısım*** (İng: exponent) ve ***işaret biti*** (İng: sign bit). Bir gerçek sayı `(s ? -1 : 1) * 2^e * M` ifadesiyle belirtilen bir kayan noktalı sayı ile ifade

edilir (ifadedeki `s` işaret biti, `e` üstel kısım, `M` ise ondalık kısımdır). Daha ayrıntılı bilgi için *Gerçek Sayı Gösterimi ile İlgili Kavramlar* (sayfa: 823) bölümüne bakınız. (Üstel kısım için farklı ***taban***lar mümkündür ama günümüzdeki hemen bütün donanımlar taban olarak 2 kullanır.)

Kayan noktalı sayıların gerçek sayıların sonlu bir alt kümesi olduğundan bahsedilebilir. Bu alt küme çoğu amaç için yeterince genişken, bir gerçek sayının sadece ondalık kısmının ikilik tabanda sınırlı bir bitsel açılımından daha dar yer kaplayan rasyonel sayılar olarak ifade edilirler. Hatta 1/5 gibi basit bir bölmenin sonucu bile kayan noktalı olarak yaklaşık bir değer olarabilir.

Matematiksel işlemler ve işlevler çoğunlukla gösterilemeyen değerler üretir. Bu değerler genellikle uygulamaya dönük olarak yeterli yaklaşıklıkta olur, fakat bazan bu bile mümkün olmaz. Tarihsel olarak, hesaplama sonuçlarının yeterince hassas olmadığını söylemenin bir yolu yoktur. Günümüz bilgisayarlarında sayısal hesaplamalar IEE 754 standardına uygun yapılır. Standart, hesaplama sonuçları güvenilir olmadığında yazılıma bunu belirten bir çerçeve tanımlar. Bu çerçeve sonucun neden gösterilemediğini belirten bir ***olağandışılıklar*** kümesinden oluşur. Bunlar ***sonsuzluk*** ve ***bir sayı değil*** (NaN — Not a Number) gibi özel değerlerdir.

4. Gerçek Sayı Sınıflama İşlevleri

ISO C99 bir değişkendeki bir gerçek sayının çeşidini saptamanıza yardımcı olacak makrolar tanımlamıştır.

`int` **`fpclassify`** (*float–type x*) makro

Tüm gerçek sayı türleri ile çalışan ve **`int`** türünde bir değer ile dönen bir soysal makrodur. Olası dönüş değerleri:

`FP_NAN`
: x sayısı normal bir sayı değil. (Bkz. *Sonsuzluk ve NaN* (sayfa: 513))

`FP_INFINITE`
: x sayısı ya artı ya da eksi sonsuz. (Bkz. *Sonsuzluk ve NaN* (sayfa: 513))

`FP_ZERO`
: x sayısı sıfır. IEEE 754 benzeri kayan noktalı biçimlerde sıfırlar işaretli olabilir, dolayısıyla bu değer eksi sıfır için de döner.

`FP_SUBNORMAL`
: Mutlak değeri çok küçük olduğundan normal biçimde gösterilemeyen sayılar ***normalleştirilmemiş*** biçim denilen biçimde gösterilebilir (bkz. *Gerçek Sayı Gösterimi ile İlgili Kavramlar* (sayfa: 823)). Bu biçim pek hassas olmamakla birlikte sıfıra çok yakın değerleri gösterebilir. **`fpclassify`** işlevi bu diğer biçimle gösterilmekten başka çare olmayan x değerleri için bu makronun değerini döndürür.

`FP_NORMAL`
: x sayısına özel hiçbir durum olmadığını gösterir, yani sayı sıradan, normal bir sayıdır.

`fpclassify` işlevi bir sayının birden fazla özelliği olması durumunda çok kullanışlıdır. Bir defada bir özelliği sınayan daha özel makrolar da vardır. Bunlar için özel donanım desteği olduğundan bu makrolar **`fpclassify`** işlevinden daha hızlıdırlar. Bu bakımdan, mümkün olduğunca bu özel makroları kullanmalısınız.

`int` **`isfinite`** (*bir–float–tür x*) makro

Bu makro x değeri bir sonlu değerse sıfırdan farklı bir değerle döner (sonlu değer: eksi ya da artı sonsuz ve NaN olmayan). İşlem,

```
(fpclassify (x) != FP_NAN && fpclassify (x) != FP_INFINITE)
```

ifadesine eşdeğerdir. **isfinite** herhangi bir gerçek sayı türünü kabul eden bir makro olarak gerçeklenmiştir.

`int` **isnormal**(*bir–float–tür x*) makro

Bu makro *x* değeri bir sonlu ve normalleştirilmiş değerse sıfırdan farklı bir değerle döner. Eşdeğer ifade:

```
(fpclassify (x) == FP_NORMAL)
```

`int` **isnan**(*float–type x*) makro

Bu makro *x* değeri normal bir sayı değilse (NaN ise) sıfırdan farklı bir değerle döner. Eşdeğer ifade:

```
(fpclassify (x) == FP_NAN)
```

Gerçek sayı sınıflama işlevlerinin BSD sürümleri de vardır ve GNU C kütüphanesi bu işlevleri de destekler. Yine de yeni geliştireceğiniz yazılımlarda ISO C99 makrolarını kullanmanızı öneririz. Bunlar standarttır ve daha geniş çapta kullanılır olacaktır. Ayrıca, makro olduklarından argüman türleri bakımından endişelenmeniz gerekmez.

`int` **isinf**(double *x*) işlev
`int` **isinff**(float *x*) işlev
`int` **isinfl**(long double *x*) işlev

Bu işlevler *x*'in değeri negatif sonsuz ise **-1** ile, positif sonsuz ise **1** ile aksi takdirde **0** ile döner.

`int` **isnan**(double *x*) işlev
`int` **isnanf**(float *x*) işlev
`int` **isnanl**(long double *x*) işlev

Bu işlevler *x* bir sayı değilse (yani NaN ise), sıfırdan farklı bir değerle, aksi takdirde sıfır ile döner.

Bilgi

ISO C99 tarafından tanımlanan **isnan** makrosu BSD işlevinin yerine geçer. Bu durum normalde bir soruna yol açmaz çünkü bu ikisi birbirine eşdeğerdir. Buna rağmen, illaki BSD işlevine ihtiyacınız varsa şunu yazın:

```
(isnan) (x)
```

`int` **finite**(double *x*) işlev
`int` **finitef**(float *x*) işlev
`int` **finitel**(long double *x*) işlev

Bu işlevler *x* bir sayı değilse (yani NaN ise) ya da sonlu bir sayı ise, sıfırdan farklı bir değerle, aksi takdirde sıfır ile döner.

Taşınabilirlik Bilgisi

Bu bölümde listelenen işlevler BSD oluşumudur.

5. Gerçek Sayı Hesaplamalarında Hatalar

5.1. Kayan Noktalı Sayı Olağandışılıkları

IEEE 754 standardı hesaplama sırasında oluşan beş tane ***olağandışılık*** tanımlar. Her biri üstten taşma gibi belli başlı bir hataya karşılıktır.

Olağandışılıklar oluştuğunda (standardın dilinde, olağandışılıklar *ortaya çıktığında*) iki şeyden biri olur. Öntanımlı olarak olağandışılık basitçe kayan noktalı *durum sözcüğü*ne kaydedilir ve yazılım hiçbir şey olmamış gibi çalışmaya devam eder. İşlem olağandışılığa bağlı bir öntanımlı değer üretir (aşağıdaki tabloya bakınız). Yazılımınız durum sözcüğüne bakarak hangi olağandışılığın oluştuğunu saptar.

Bundan başka, olağandışılıklar için *kapanları* etkinleştirebilirsiniz. Bu durumda, bir olağandışılık ortaya çıktığında, yazılımınız **SIGFPE** sinyali alacaktır. Bu sinyalin öntanımlı eylemi yazılımın sonlanmasıdır. Sinyallerin etkilerinin nasıl değiştirildiği *Sinyal İşleme* (sayfa: 601) bölümünde anlatılmıştır.

System V matematik kütüphanesinde, kütüphane işlevleri içinde bu olağandışılıklardan bir oluştuğunda kullanıcı tanımlı **matherr** işlevi çağrılır. Ancak Unix98 standardı bu arayüzün kullanılmasını önermez. Bu işlevi geçmişe uyumluluk adına destekliyoruz, ama yeni yazılımlarda kullanmamanızı öneriyoruz.

IEEE 754 standardında tanımlanan olağandışılıklar:

Geçersiz İşlem

Belirtilen terimler uygulanacak işlem için geçersiz ise bu olağandışılık ortaya çıkar. Örnekler (IEEE 754, bölüm 7'ye bakın):

1. Toplama ve çıkarma: `oo - oo`. (ama `oo + oo = oo`).
2. Çarpma: `0 * oo`.
3. Bölme: `0/0` or `oo/oo`.
4. Kalan: `y` sıfır ya da `x` sonsuz olduğunda `x REM y`.
5. Karekök alma işleminde terim sıfırdan küçükse. Daha genel olarak kendi işlem sahası dışında işlem yapmaya zorlanan matematiksel işlevler bu olağandışılığı üretir.
6. Gerçek sayıların tamsayıya ya da ondalık dizgeye dönüştürülmesinde sayı hedef biçimde gösterilemiyorsa (üstten taşma, sonsuzluk ya da NaN sebebiyle).
7. Tanınamayan bir girdi dizgesinin dönüşümü.
8. İlişkisel işleçlerle (< veya >) yapılan karşılaştırmalarda, terimlerden birinin NaN olması. Bu işleçleri kullanmak yerine düzensiz karşılaştırma işlevlerini kullanarak bu olağandışılıktan kaçınabilirsiniz; bkz. *Gerçek Sayı Karşılaştırma İşlevleri* (sayfa: 525).

Olağandışılık yakalanmazsa işlemin sonucu NaN'dır.

Sıfırla Bölme

Bu olağandışılık sıfırdan farklı bir sonlu sayı sıfırla bölündüğünde ortaya çıkar. Bir yakalama gerçekleşmezse sonuç terimlerin işaretine bağlı olarak ya `+oo` ya da `-oo`'dur.

Üstten Taşma

Bu olağandışılık sonuç hedef hassasiyet biçiminde bir sonlu değer olarak gösterilemiyorsa ortaya çıkar. Bir yakalama gerçekleşmezse sonuç ara sonucun işaretine ve geçerli yuvarlama kipine bağlıdır (IEEE 754, bölüm 7.32e bakın):

1. En yakın sayıya yuvarlama tüm üstten taşmaları ara sonucun işareti ile sonsuza taşır.
2. Sıfıra yuvarlama tüm üstten taşmaları ara sonucun işareti ile gösterilebilir en büyük sonlu sayıya taşır.
3. Eksi sonsuza yuvarlama pozitif üstten taşmaları gösterilebilir en büyük sonlu sayıya, negatif üstten taşmaları ise eksi sonsuza taşır.
4. Sonsuza yuvarlama negatif üstten taşmaları gösterilebilir en negatif sonlu sayıya, pozitif üstten taşmaları ise sonsuza taşır.

Her üstten taşma olağandışılığı ayrıca kesin olmama olağandışılığının ortaya çıkmasına sebep olur.

Alttan Taşma
: Alttan taşma olağandışılığı bir ara sonuç tam olarak hesaplanmış olarak çok küçük olduğunda ya da hedefin hassasiyetine yuvarlama işleminin sonucu normalleştirilmiş olarak çok küçük olduğunda ortaya çıkar.

  Alttan taşma olağandışılığı için bir kapan kurulmadığında, alttan taşma sadece küçücüklük ve doğruluk kaybı birlikte saptandığında sinyallenir (alttan taşma bayrağı üzerinden). Bir sinyal eylemci kurulu değilse, işlem bir belirsiz küçük değer ile ya da hedefin hassasiyeti tam doğru küçük sonucu tutamayacaksa sıfır ile devam eder.

`Kesin Olmama`
: Bu olağandışılık eğer bir yuvarlanan sonuç kesin değilse (örneğin karekök iki hesaplanırken) ya da bir sonuç bir üstten taşma kapanı olmaksızın üstten taşarsa sinyallenir.

5.2. Sonsuzluk ve NaN

IEEE 754 kayan noktalı sayıları ile pozitif ve negatif sonsuz sayılar ve sayı olmama durumu (NaN) gösterilebilir. Bu üç değer hesaplamalardaki sonuç tanımsızsa ya da tam doğru olarak gösterilemiyorsa ortaya çıkar. Ayrıca bunlardan birini bir gerçek sayı değişkenine enine boyuna düşünerek faydası olacaksa atayabilirsiniz. Sonsuzluk ve NaN üreten bazı hesaplama örnekleri:

```
1/0 = oo
log (0) = -oo
sqrt (-1) = NaN
```

Bir hesaplama bu değerlerden birini ürettiğinde ayrıca bir olağandışılık oluşur; bkz. *Kayan Noktalı Sayı Olağandışılıkları* (sayfa: 511).

Temel işlemler ve matematik işlevlerinin hepsi sonsuzluk ve NaN değerlerini kabul ederler ve uygun bir çıktı üretirler. Sonsuzluğun hesaplamalarda kullanılışına örnekler: `2 + oo = oo`, `4/oo = 0`, `atan (oo) = pi/2`. Ancak, NaN bulaştığı bir hesaplamayı bozar. Hesaplama aynı sonucu üretmedikçe sonucun NaN mı olduğu yoksa gerçek sonucun yerini NaN'ın mı aldığının bir önemi yoktur.

Karşılaştırma işlemlerinde pozitif sonsuz, kendisi ve NaN hariç diğer tüm değerlerden büyüktür; negatif sonsuz ise kendisi ve NaN hariç diğer tüm değerlerden küçüktür. NaN sıradışıdır: kendisi dahil ne bir şeye eşittir, ne birşeyden küçük, ne de birşeyden büyüktür. **`x == x`** karşılaştırması değer NaN olduğunda yanlıştır. Bu durumu bir değerin NaN olup olmadığını sınamak için kullanabilirsiniz, ancak önerilen yöntem **`isnan`** işlevini kullanmaktır (bkz. *Gerçek Sayı Sınıflama İşlevleri* (sayfa: 510)). Ek olarak, NaN'lara uygulandığında **`<`**, **`>`**, **`<=`** ve **`>=`** işleçleri bir olağandışılık ortaya çıkarırlar.

Bir değişkene sonsuzluk veya NaN atamayı mümkün kılan makrolar `math.h` dosyasında tanımlanmıştır.

`float` **`INFINITY`** makro

Pozitif sonsuzu gösteren bir ifade. **`1.0 / 0.0`** gibi matematiksel bir işlemin sonucuna eşittir. Negatif sonsuz **`-INFINITY`**'dir.

Bir gerçek sayının sonsuz olup olmadığını bu makro ile karşılaştırarak sınayabilirsiniz. Ancaki bu önerilmez; bunun yerine **`isfinite`** makrosunu kullanmalısınız; bkz. *Gerçek Sayı Sınıflama İşlevleri* (sayfa: 510).

Bu makro ISO C99 standardında tanıtıldı.

`float` **`NAN`** makro

Sayı olmayan bir değeri gösteren bir ifade. Bu makro bir GNU oluşumudur ve sadece IEEE kayan noktalı sayıları destekleyen tüm makinalarda desteklenir.

Bir makinada NaN desteği olup olmadığını sınamak için **#ifdef NAN** kullanabilirsiniz. (Şüphesiz, **_GNU_SOURCE** tanımlayarak ve `math.h` başlık dosyasını yazılımınıza dahil ederek GNU oluşumlarının görünür olmasını sağlamalısınız.)

IEEE 754 ayrıca başka bir faydasız değeri mümkün kılar: negatif sıfır. Bu değer, bir pozitif sayıyı negatif sonsuza böldüğünüzde ya da bir negatif sonuç gösterim sınırından daha küçükse üretilir. Negatif sıfır, işaret bitini **signbit** veya **copysign** ile doğrudan sınamadıkça tüm hesaplamalarda sıfır gibi davranır.

5.3. Kayan Nokta Birimi Durum Sözcüğünün İncelenmesi

ISO C99, kayan nokta durum sözcüğünün sorgulanması ve değiştirilmesi için işlevler tanımlamıştır. Yakalanmayan olağandışılıklar nedeniyle hesaplamanın ortasında endişelenmek yerine münasip bir zamanda bunları bu işlevleri kullanarak sınayabilirsiniz.

Bu sabitler çeşitli IEEE 754 olağandışılıklarını gösterir. Tüm kayan nokta birimleri tüm olağandışılık çeşitlerini raporlamaz. Her sabit yalnız ve yalnız kayan nokta birimi ilgili olağandışılık desteği verilerek derlenmişse tanımlıdır. Kayan nokta birimi desteğini **#ifdef** ile sınayabilirsiniz. Bu sabitler `fenv.h` başlık dosyasında tanımlıdır..

`FE_INEXACT`
: Kesin olmama olağandışılığı.

`FE_DIVBYZERO`
: Sıfırla bölme olağandışılığı.

`FE_UNDERFLOW`
: Üstten taşma olağandışılığı.

`FE_OVERFLOW`
: Alttan taşma olağandışılığı.

`FE_INVALID`
: Geçersizlik olağandışılığı.

FE_ALL_EXCEPT makrosu kayan nokta gerçeklemesi tarafından desteklenen tüm olağandışılık makrolarının bit seviyesinde VEYAlanmışıdır.

Bu işlevler olağandışılık bayraklarını temizlenmesini, olağandışılıkların sınanmasını ve bayraklı olağandışılık kümesinin kaydedilmesini ve eski haline getirilmesini mümkün kılar.

`int` **feclearexcept**`(int` *olağandışılıklar*`)` işlev

Bu işlev *olağandışılıklar* ile belirtilenlerden desteklenen tüm olağandışılık bayraklarını temizler.

İşlev başarılı olduğunda sıfırla, aksi takdirde sıfırdan farklı bir değerle döner.

`int` **feraiseexcept**`(int` *olağandışılıklar*`)` işlev

Bu işlev *olağandışılıklar* ile belirtilenlerden desteklenen olağandışılıkları oluşturur. *olağandışılıklar* ile birden fazla olağandışılık biti belirtilmişse, üstten (**FE_OVERFLOW**) ya da alttan (**FE_UNDERFLOW**) taşmaların kesin olmama (**FE_INEXACT**) olağandışılığından önce ortaya çıkması dışında hangi olağandışılıkların oluşturulacağı tanımsızdır. Üstten veya alttan taşma için kesin olmama olağandışılığının ayrıca ortaya çıkıp çıkmayacağı ayrıca gerçeklemeye de bağlıdır.

İşlev başarılı olduğunda sıfırla, aksi takdirde sıfırdan farklı bir değerle döner.

`int` **fetestexcept**`(int` *olağandışılıklar*`)` işlev

olağandışılıklar parametresi ile belirtilen olağandışılık bayraklarının o an etkin olup olmadıklarını sınar. Bunlardan biri varsa, hangi olağandışılığın varolduğunu belirten sıfırdan farklı bir değerle döner. Aksi takdirde dönüş değeri sıfırdır.

Bu işlevlerin yaptığı işlemleri anlayabilmek için, durum sözcüğünün *durum* isimli bir tamsayı değişkeni olduğunu farzedelim. **feclearexcept** işlevi **durum &= ~olağandışılıklar** ifadesine, **fetestexcept** işlevi ise **durum & olağandışılıklar** ifadesine eşdeğerdir. Şüphesiz, asıl gerçekleme farklı olabilir.

Olağandışılık bayrakları yazılım içinden sadece doğrudan **feclearexcept** çağrısı yapılarak temizlenebilir. Bir hesaplama sırasında oluşan olağandışılıklara bakmak isterseniz, önce bayrakların hepsini temizlemelisiniz. **fetestexcept** işlevinin kullanımına basit bir örnek:

```
{
  double f;
  int raised;
  feclearexcept (FE_ALL_EXCEPT);
  f = compute ();
  raised = fetestexcept (FE_OVERFLOW | FE_INVALID);
  if (raised & FE_OVERFLOW) { /* … */ }
  if (raised & FE_INVALID) { /* … */ }
  /* … */
}
```

Durum sözcüğünün bitlerini doğrudan etkinleştiremezsiniz. Ancak, durum sözcüklerinin tamamını önce kaydeder ve daha sonra eski haline getirebilirsiniz. Bu işlem aşağıdaki işlevlerle yapılabilir:

```
int fegetexceptflag(fexcept_t *bayrak,                                  işlev
                    int        olağandışılıklar)
```

Bu işlev *olağandışılıklar* ile belirtilenlerden o an geçerli gerçekleme tanımlı değerleri *bayrak* ile gösterilen değişkende saklar.

İşlev başarılı olduğunda sıfırla, aksi takdirde sıfırdan farklı bir değerle döner.

```
int fesetexceptflag(const fexcept_t *bayrak,                            işlev
                    int              olağandışılıklar)
```

Bu işlev *olağandışılıklar* ile belirtilen olağandışılık bayraklarını *bayrak* ile gösterilen değişkenin değerinden eski durumuna getirir.

İşlev başarılı olduğunda sıfırla, aksi takdirde sıfırdan farklı bir değerle döner.

Bilgi

fexcept_t türünde saklanan değerin **fetestexcept** işlevinden dönen bit maskesine bir benzerliği sözkonusu değildir. Hatta tür bir tamsayı bile olmayabilir. **fexcept_t** türünde bir değişkeni doğrudan değiştirmeye çalışmayın.

5.4. Hataların Matematiksel İşlevlerce Raporlanması

Matematiksel işlevlerin çoğu sadece gerçek ya da karmaşık sayıların bir alt kümesi ile ilgilidir. Matematiksel olarak tanımlı olsalar bile, sonuçları dönüş türleri ile gösterilebilen aralıktan daha büyük ya da daha küçük olabilir. Bunlar sırasıyla ***saha hataları***, ***üstten taşmalar*** ve ***alttan taşmalar*** olarak bilinir. Bu hatalardan biri oluştuğunda matematiksel işlevler çeşitli şeyler yaparlar. Bu kılavuzda bu yanıtların hepsine birden bir saha hatasının, üstten ya da alttan taşmanın *sinyallenmesi* olarak bakacağız.

Bir matematiksel işlev bir saha hatasına maruz kaldığında, geçersizlik olağandışılığı oluşturur ve NaN ile döner. Ayrıca, *errno* değişkenine **EDOM** hata durumunu atar; bu, IEEE 754 olağandışılıklarından nasibini almamış eski sistemlerle uyumluluk adına böyledir. Benzer şekilde üstten taşma oluştuğunda matematiksel işlevler üstten taşma olağandışılığı oluşturur ve duruma göre `oo` ya da `-oo` döndürür. Ayrıca, *errno* değişkenine **ERANGE** hata durumunu atar. Alttan taşma oluştuğunda matematiksel işlevler alttan taşma olağandışılığı oluşturur ve sıfırla (işaretli olabilir) dönerler. *errno* değişkenine **ERANGE** hata durumu atanmış olabilir, ama bu garanti değildir.

Bazı matematiksel işlevler matematiksel olarak bir karmaşık değerle sonuçlanacak şekilde tanımlanmıştır. Çok bilinen bir örnek, negatif bir sayının karekökünün alınmasıdır. **csqrt** gibi karmaşık sayılarla ilgili matematiksel işlevler böyle bir durumda uygun bir değerle dönecektir. **sqrt** gibi gerçek sayılarla ilgili işlevler ise böyle bir durumda bir saha hatasını sinyalleyecektir.

Bazı eski donanımlar sonsuzlukları desteklemez. Böyle bir donanımda üstten taşmalar gösterilebilen en büyük sayı ile ya da belli bir çok büyük sayı ile döner. `math.h` dosyasında tanımlanan makroları üstten taşmaları hem eski hem de yeni donanımlarda sınamak için kullanabilirsiniz.

`double` **HUGE_VAL** — makro

`float` **HUGE_VALF** — makro

`long double` **HUGE_VALL** — makro

Belli bir çok büyük sayıyı gösteren bir ifade. IEEE 754 kayan nokta biçimi kullanılan yeni donanımlarda makronun değeri sonsuzdur. Diğer makinalarda, genellikle gösterilebilen en büyük pozitif sayıdır.

Matematiksel işlevler sonucun gösterilebilenden çok büyük olması durumunda **HUGE_VAL** veya **-HUGE_VAL** makrolarının uygun türdeki sürümlerini döndürür.

6. Yuvarlama Kipleri

Gerçek sayılarla yapılan hesaplamalarda dahili hassasiyet daha yüksektir ve hedef türe sığacak şekilde yuvarlama yapılır. Bu, sonuçların girilen veri kadar hassas olmasını sağlar. IEEE 754 dört olası yuvarlama kipi tanımlar:

En yakına yuvarlama
Bu öntanımlı kiptir. Diğer kiplerden birine özellikle ihtiyaç duyulmadıkça kullanılması gereken kip budur. Bu kipte sonuçlar gösterilebilen en yakın değere yuvarlanır. Sonuç iki gösterilebilen değerin ortasındaysa çift sayı olanı seçilir. Burada "çift" en düşük seviyeli biti sıfır olan anlamındadır. Bu yuvarlama kipi istatistiksel sapmadan korur ve sayısal kararlılığı garantiler: uzunca bir hesaplamada yuvarlayamama hataları **FLT_EPSILON**'un yarısından daha küçük kalacaktır.

Artı sonsuza yuvarlama
Tüm sonuçlar, sonuçtan daha büyük olan gösterilebilir en küçük değere yuvarlanır.

Eksi sonsuza yuvarlama
Tüm sonuçlar, sonuçtan daha küçük olan gösterilebilir en büyük değere yuvarlanır.

Sıfıra yuvarlama
Tüm sonuçlar, sıfıra en yakın gösterilebilir en büyük değere yuvarlanır. Başka bir deyişle, negatif sonuçlar üste, pozitif sonuçlar alta yuvarlanır.

`fenv.h` dosyasında tanımlanan sabitleri yuvarlama kiplerini belirtmek için kullanabilirsiniz. Her biri yalnız ve yalnız kayan nokta birimi ilgili yuvarlama kipini destekliyorsa tanımlıdır.

`FE_TONEAREST`
En yakına yuvarlama.

`FE_UPWARD`
: Artı sonsuza yuvarlama.

`FE_DOWNWARD`
: Eksi sonsuza yuvarlama.

`FE_TOWARDZERO`
: Sıfıra yuvarlama.

Alttan taşma gereksiz bir durumdur. Normalde, IEEE 754 kayan noktalı sayıları daima normalleştirilirler (bkz. *Gerçek Sayı Gösterimi ile İlgili Kavramlar* (sayfa: 823)). `2^r`'den küçük sayılar (burada r en küçük üstür ve *float* türü için **`FLT_MIN_RADIX-1`**'dir) normallşetirilmiş sayılar olarak gösterilemezler. Böyle sayıların sıfıra ya da `2^r`'ye yuvarlanması bazı algoritmaların sıfırda başarısız olmasına sebep olur. Bu bakımdan bunlar normalleştirilmemiş şekilde kalır. Ondalık kısmın bazı bitleri ondalık noktayı göstermek için kaynayıp gittiğinden bu hassasiyet kaybına sebep olur.

Eğer bir sonuç bir normalleştirilmemiş sayı olarak göstremek için bile çok küçükse sıfıra yuvarlanır. Yine de, sonucun işareti korunur; eğer hesaplama negatifse sonuç eksi sıfırdır. Negatif sıfır değeri ayrıca `4/-oo` gibi sonsuz değeri kullanılan işlemlerinde sonucu olur. Negatif sıfır, işaret bitini **`signbit`** veya **`copysign`** ile doğrudan sınanmadıkça tüm hesaplamalarda sıfır gibi davranır.

Herhangi bir anda yukarıdaki dört yuvarlama kipinden biri seçilidir. Hangi kipin seçili olduğunu şu işlevle bulabilirsiniz:

```
int fegetround(void)                                                    işlev
```

O an seçili olan yuvarlama kipini yuvarlama kipini tanımlayan makrolardan birinin değeri olarak döndürür.

Yuvarlama kipini değiştirmek için şu işlevi kullanın:

```
int fesetroundint yuvarlama)                                            işlev
```

O an seçili yuarlama kipini *yuvarlama* kipi olarak değiştirir. Eğer *yuvarlama* değeri desteklenen yuvarlama kiplerinden birine karşılık değilse hiçbir değişiklik yapılmaz. İşlev kipi değiştirebilmişse sıfırla, kip desteklenmiyorsa sıfırdan farklı bir değerle döner.

Mümkünse, yuvarlama kipini değiştirmekten kaçınmalısınız. Masraflı bir işlem haline gelebilir; ayrıca bazı donanımlar yazılımınızın çalıştığından farklı derlenmesini gerektirebilir. Sonuçlanan kod daha yavaş çalışabilir. Daha ayrıntılı bilgi için derleyicinizin belgelerine bakınız.

7. Kayan Nokta Denetim İşlevleri

IEEE 754 kayan nokta gerçeklemeleri denetim sözcüğündeki bitleri ayarlayarak oluşacak her olağandışılık için kapanlar olup olmayacağına yazılımcının karar vermesini mümkün kılar. C'de kapanlar yazılımın **`SIGFPE`** sinyali almasıyla sonuçlanır; bkz. *Sinyal İşleme* (sayfa: 601).

Bilgi

IEEE 754 sinyal eylemcilerin olağandışı durumların ayrıntılarının verildiği ve sonuç değerin ayarlandığı yerler olduğunu söyler. C sinyalleri bu bilginin ileri ya da geri aktarılması için herhangi bir mekanizma sağlamaz. C'de olağandışılıkların yakalanması bundan dolayı çok elverişli değildir.

Bazı hesaplamaları yaparken kayan nokta biriminin durumunun kaydedilmesi bazan gerekli olur. Kütüphane, kapanları üreten olağandışılıkları ve yuvarlama kiplerini kaydetmeyi ve gerektiğinde eski durumuna getirmeyi sağlayan işlevler sağlar. Bu bilgiler *kayan nokta ortamı* olarak bilinir.

Kayan nokta ortamını kaydeden ve eski durumuna getiren işlevlerin hepsi bilgiyi saklamak için **fenv_t** türünde bir değişken kullanırlar. Bu veri türü `fenv.h` dosyasında tanımlanmıştır. Türün genişliği ve içeriği gerçeklemeye bağlıdır. Bu türdeki bir değişkenin değerini doğrudan değiştirmeye çalışmamalısınız.

Kayan nokta biriminin durumunu kaydetmek için bu işlevlerden birini kullanın:

`int` **fegetenv**`(fenv_t *`*ortam*`)` işlev

Kayan nokta ortamını *ortam* ile gösterilen değişkende saklar.

İşlem başarılıysa işlev sıfırla aksi takdirde sıfırdan farklı bir değerle döner.

`int` **feholdexcept**`(fenv_t *`*ortam*`)` işlev

O anki kayan nokta ortamını *ortam* ile gösterilen nesnede saklar ve tüm olağandışılık bayraklarını temizledikten sonra kayan nokta birimini hiçbir olağandışılığı yakalamamaya ayarlar. Hiçbir olağandışılığın yakalanmamasını kayan nokta birimlerinin hepsi desteklemez; eğer **feholdexcept** bu kipi ayarlayamıyorsa, sıfırdan farklı bir değer döner. Başarırsa sıfırla döner.

Kayan nokta ortamını eski durumuna getiren işlevlerin alabildiği argüman çeşitleri:

- Önceki bir **fegetenv** vaya **feholdexcept** çağrısı ile ilklendirilen **fenv_t** türündeki nesnelere göstericiler.
- Yazılım başlangıcında kullanılabilir olarak, kayan noktalı ortamı ifade eden bir özel makro: **FE_DFL_ENV**.
- **fenv_t *** türünde isimleri **FE_** ile başlayan gerçekleme tanımlı makrolar.

  Mümkünse, GNU C kütüphanesi, bir kapanın oluşmasına sebep olan her olağandışılığın ortaya çıkışında ortamı ifade eden **FE_NOMASK_ENV** makrosunu tanımlar. Bu makronun varlığını **#ifdef** kullanarak sınayabilirsiniz. Bu makro sadece **_GNU_SOURCE** tanımlıysa tanımlıdır.

  Bazı platformlar diğer önceden tanımlanmış ortamları tanımlayabilir.

Kayan noktalı ortamı etkin kılmak için şu işlevleri kullanabilirsiniz:

`int` **fesetenv**`(const fenv_t *`*ortam*`)` işlev

ortam ile açıklanan kayan noktalı ortamı etkin kılar.

İşlev başarılı olduğunda sıfırla, aksi takdirde sıfırdan farklı bir değerle döner.

`int` **feupdateenv**`(const fenv_t *`*ortam*`)` işlev

fesetenv gibi bu işlev de *ortam* ile açıklanan kayan noktalı ortamı etkin kılar. Ancak, bu işlevin çağrısından önce durum sözcüğü bayraklı herhangi bir olağandışılık varsa çağrıdan sonra da bayraklı olarak kalır. Başka bir deyişle, bu işlevin çağrısından sonra önceki durum sözcüğü ile *ortam* içinde kayıtlı olan bit seviyesinde VEYAlanır.

İşlev başarılı olduğunda sıfırla, aksi takdirde sıfırdan farklı bir değerle döner.

Olağandışılıklar ortaya çıktığında bir kapanın oluşmasına sebep oluyorsa, bu olağandışılıkları tek tek aşağıdaki iki işlevle denetim altında tutabilirsiniz.

Taşınabilirlik Bilgisi

Bu işlevlerin hepsi GNU oluşumudur.

`int` **feenableexcept**`(int` *olağandışılıklar*`)` işlev

olağandışılıklar parametresi ile belirtilen olağandışılıkların her biri için kapanları etkinleştirir. Olağandışılıklar tek tek *Kayan Nokta Birimi Durum Sözcüğünün İncelenmesi* (sayfa: 514) bölümünde açıklanmıştır. Sadece belirtilen olağandışılıklar etkinleştirilir, diğer olağandışılıkların durumu değiştirilmez.

İşlev başarılı olduğunda evvelce etkinleştirilmiş olağandışılıklarla döner, aksi takdirde **-1** ile döner.

`int` **fedisableexcept**`(int` *olağandışılıklar*`)` işlev

olağandışılıklar parametresi ile belirtilen olağandışılıkların her biri için kapanları iptal eder. Olağandışılıklar tek tek *Kayan Nokta Birimi Durum Sözcüğünün İncelenmesi* (sayfa: 514) bölümünde açıklanmıştır. Sadece belirtilen olağandışılıklar iptal edilir, diğer olağandışılıkların durumu değiştirilmez.

İşlev başarılı olduğunda evvelce etkinleştirilmiş olağandışılıklarla döner, aksi takdirde **-1** ile döner.

`int` **fegetexcept**`(int` *olağandışılıklar*`)` işlev

Bu işlev o an etkin olan olağandışılıklar ile döner. **-1** dönmüşse bir hata oluşmuş demektir.

8. Aritmetik İşlevleri

C kütüphanesi gerçek sayılar üzerinde temel işlemleri yapan işlevler içerir. Bunlar arasında mutlak değer, en küçük ve en büyük değer, normalleştirme, bit değiştirme ve yuvarlama işlemleri sayılabilir.

8.1. Mutlak Değer

Bu işlevler ***mutlak değerler***i (ya da *genlik*leri) elde etmek için kullanılır. x gerçek sayısının mutlak değeri, x pozitifse x, negatifse $-x$'tir. z, gerçek kısmı x, sanal kısmı y olan bir karmaşık sayı ise, z'nin mutlak değeri x'in karesi ile y'nin karesinin toplamının karekökü'dür.

abs, **labs** ve **llabs** işlevleri `stdlib.h` başlık dosyasında bildirilmiştir. **imaxabs** işlevi `inttypes.h` dosyasında, **fabs**, **fabsf** ve **fabsl** işlevleri `math.h` dosyasında, **cabs**, **cabsf** ve **cabsl** işlevleri ise `complex.h` dosyasında bildirilmiştir.

`int` **abs**`(int` *sayı*`)` işlev
`long int` **labs**`(long int` *sayı*`)` işlev
`long long int` **llabs**`(long long int` *sayı*`)` işlev
`intmax_t` **imaxabs**`(intmax_t` *sayı*`)` işlev

Bu işlevler *sayı* tamsayısının mutlak değeri ile döner.

INT_MIN (olası en küçük **int**) değerinin mutlak değerinin gösterilemediği bazı bilgisayarlarda tamsayı gösterimleri ikinin tümleyenini kullanırlar; böyle bir durumda **abs (INT_MIN)** tanımsızdır.

llabs ve **imaxdiv** işlevleri ISO C99 ile tanımlanmış daha yeni işlevlerdir.

intmax_t türü hakkında daha fazla bilgi için *Tamsayılar* (sayfa: 506) bölümüne bakınız.

`double` **fabs**`(double` *sayı*`)` işlev
`float` **fabsf**`(float` *sayı*`)` işlev
`long double` **fabsl**`(long double` *sayı*`)` işlev

Bu işlevler *sayı* gerçek sayısının mutlak değeri ile döner.

`double` **cabs**`(complex double` *z*`)` işlev
`float` **cabsf**`(complex float` *z*`)` işlev
`long double` **cabsl**`(complex long double` *z*`)` işlev

Bu işlevler z karmaşık sayısının mutlak değeri ile döner (bkz. *Karmaşık Sayılar* (sayfa: 527)). Bir karmaşık sayının mutlak değeri şöyle hesaplanır:

```
sqrt (creal (z) * creal (z) + cimag (z) * cimag (z))
```

Hassasiyet kayıplarından kaçınmak için doğrudan formülü kullanmak yerine bu işlevin kullanılması gerekir. Bu işlem ayrıca donanım desteği avantajına da sahiptir. *Üstel ve Logaritmik İşlevler* (sayfa: 479) bölümündeki **hypot**'a bakınız.

8.2. Normalleştirme İşlevleri

Bu bölümde, dahili olarak bir ikilik üs değeri kullanarak gösterilen kayan noktalı sayıların düşük seviyeli işlemlerini verimli bir yöntem olarak sağlayan işlevlere öncelikle yer verilmiştir; bkz. *Gerçek Sayı Gösterimi ile İlgili Kavramlar* (sayfa: 823). Bu işlevlerin, gösterim ikilik üs kullanmıyorsa bile eşdeğer davranış göstermeleri gerekir, fakat şüphesiz bunlar bu durumların tersi durumlarda kısmen bile verimli olmaz.

Bu bölümdeki işlevlerin hepsi `math.h` dosyasında bildirilmiştir.

```
double frexp(double değer,                                          işlev
             int    *üs)
float frexpf(float değer,                                           işlev
             int   *üs)
long double frexpl(long double değer,                               işlev
                   int         *üs)
```

Bu işlevler *değer* ile belirtilen sayıyı normalleştirilmiş ondalık kısım ile üstel kısıma ayırır.

Eğer *değer* argümanı sıfır değilse, 2 üssü **üs* ile işlevin dönüş değerinin çarpımı, *değer*'i verir. İşlevin dönüş değeri daima 1/2 (dahil) ile 1 (hariç) arasındadır. Üstel kısım **üs* içinde saklanır.

Örneğin, **frexp (12.8, &exponent)** çağrısı **0.8** ile döner ve *exponent* içinde **4** değerini saklar.

değer sıfır ise işlev sıfırla döner ve **üs* içinde sıfır saklanır.

```
double ldexp(double değer,                                          işlev
             int    *üs)
float ldexpf(float değer,                                           işlev
             int   *üs)
long double ldexpl(long double değer,                               işlev
                   int         *üs)
```

Bu işlev, *değer* gerçek sayısının 2 üssü *üs* ile çarpımını döndürür. (Bu işlev, **frexp** işlevinden dönen parçalarla gerçek sayıları yeniden elde etmek amacıyla kullanılabilir.)

Örneğin, **ldexp (0.8, 4)** çağrısı **12.8** ile döner.

Aşağıdaki işlevler BSD'den gelir ve **ldexp** ve **frexp** işlevlerinin eşdeğeri oluşumlardır. Ayrıca, ISO C işlevi olan, aynı zamanda da orjinal olarak bir BSD işlevi olan **logb** işlevine de bakınız.

```
double scalb(double değer,                                          işlev
             int    *üs)
float scalbf(float değer,                                           işlev
             int   *üs)
long double scalbl(long double değer,                               işlev
                   int         *üs)
```

scalb işlevi **ldexp** işlevinin BSD ismidir.

```
long long int scalbn(double   x,                                    işlev
                     long int n)
long long int scalbnf(float    x,                                   işlev
                      long int n)
long long int scalbnl(long double x,                                işlev
                      long int    n)
```

scalbn işlevi, n üssünün bir gerçek sayı değil **int** türünde bir değer olması dışında **scalb** işlevinin benzeridir.

```
long long int scalbln(double x,                                     işlev
                      int    n)
long long int scalblnf(float x,                                     işlev
                       int   n)
long long int scalblnl(long double x,                               işlev
                       int         n)
```

scalbln işlevi, n üssünün bir gerçek sayı değil **long int** türünde bir değer olması dışında **scalb** işlevinin benzeridir.

```
long long intsignificand(double x)                                  işlev
long long int significandf(float x)                                 işlev
long long intsignificandl(long double x)                            işlev
```

significand işlevi x'in ondalık kısmını [1, 2] aralığında oranlayarak döndürür. **scalb (x, (double) -ilogb (x))** çağrısına denktir.

Bu işlev esas olarak IEEE 754 uyumlu standartlaştırılmış belli sınamalarda kullanmak için vardır.

8.3. Yuvarlama İşlevleri

Burada listelenen işlevler gerçek sayıları yuvarlamak ya da ondalık kısmını kırpmak gibi işlemleri gerçekleştirirler. Bu işlevlerden bazıları gerçek sayıları tamsayılara dönüştürmek için kullanılır. Bu işlevler `math.h`[8] başlık dosyasında bildirilmiştir.

Gerçek sayıları tamsayılara dönüştürmek için ayrıca **int** türüne tür dönüşümü yapabilirsiniz. Bu gerçek sayının ondalık kısmını iptal eder, daha doğrusu sıfıra yuvarlar. Ancak, sonuç gerçekte **int** türünde gösterilebiliyorsa bu çalışır—çok büyük sayılarda bu mümkün olmaz. Burada listelenen işlevler bu sorunun çevresinden dolaşmak için sonucu **double** türünde döndürüler.

```
double ceil(double x)                                               işlev
float ceilf(float x)                                                işlev
long double ceill(long double x)                                    işlev
```

Bu işlevler x sayısını en yakın büyük tamsayıya yuvarlar ve sonucu gerçek sayı türünde döndürür. **ceil (1.5)** çağrısı **2.0** değerini döndürür.

```
double floor(double x)                                              işlev
float floorf(float x)                                               işlev
long double floorl(long double x)                                   işlev
```

Bu işlevler x sayısını en yakın küçük tamsayıya yuvarlar ve sonucu gerçek sayı türünde döndürür. **floor (1.5)** çağrısı **1.0** ile ve **floor (-1.5)** çağrısı **-2.0** ile döner.

```
double trunc(double x)                                              işlev
float truncf(float x)                                               işlev
long double truncl(long double x)                                   işlev
```

Bu işlevler x sayısını sıfıra doğru en yakın tamsayıya yuvarlar ve sonucu gerçek sayı türünde döndürür. `trunc (1.5)` çağrısı `1.0` ile ve `trunc (-1.5)` çağrısı `-1.0` ile döner.

```
double rint(double x)                                    işlev
float rintf(float x)                                     işlev
long double rintl(long double x)                         işlev
```

Bu işlevler x sayısını o anki yuvarlama kipine uygun olarak bir tamsayıya yuvarlar. Çeşitli yuvarlama kipleri hakkında bilgi almak için *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824) bölümüne bakınız. Öntanımlı yuvarlama kipi en yakın tamsayıya yuvarlamadır; bazı makinalar diğer kipleri destekleyebilir ama açıkça başka bir kip belirtilmedikçe daima en yakına yuvarlama kipi kullanılır.

x sayısının baş tarafı bir tamsayı değilse, bu işlevler kesin olmama olağandışılığı oluşturur.

```
double nearbyint(double x)                               işlev
float nearbyintf(float x)                                işlev
long double nearbyintl(long double x)                    işlev
```

Bu işlevler `rint` işlevleri ile aynı değeri döndürür ama x'in baş tarafı bir tamsayı değilse, kesin olmama olağandışılığını oluşturmaz.

```
double round(double x)                                   işlev
float roundf(float x)                                    işlev
long double roundl(long double x)                        işlev
```

Bu işlevler `rint` işlevlerine benzer, farklı olarak x'in işaretiyle, ondalık kısmın 1/2'den (dahil) büyük değerlerini en yakın büyük mutlak tamsayıya, 1/2'den küçük değerlerini en yakın küçük mutlak tamsayıya yuvarlar.

```
long int lrint(double x)                                 işlev
long int lrintf(float x)                                 işlev
long int lrintl(long double x)                           işlev
```

Bu işlevler `rint` işlevlerine benzer, fakat bir gerçek sayı yerine bir `long int` türünde değerle döner.

```
long long int llrint(double x)                           işlev
long long int llrintf(float x)                           işlev
long long int llrintl(long double x)                     işlev
```

Bu işlevler `rint` işlevlerine benzer, fakat bir gerçek sayı yerine bir `long long int` türünde değerle döner.

```
long int lround(double x)                                işlev
long int lroundf(float x)                                işlev
long int lroundl(long double x)                          işlev
```

Bu işlevler `round` işlevlerine benzer, fakat bir gerçek sayı yerine bir `long int` türünde değerle döner.

```
long long int llround(double x)                          işlev
long long int llroundf(float x)                          işlev
long long int llroundl(long double x)                    işlev
```

Bu işlevler `round` işlevlerine benzer, fakat bir gerçek sayı yerine bir `long long int` türünde değerle döner.

```
double modf(double  değer,                                          işlev
            double *tamkısım)
float modff(float  değer,                                           işlev
            float *tamkısım)
long double modfl(long double  değer,                               işlev
                  long double *tamkısım)
```

Bu işlevler *değer* gerçek sayısını tam ve ondalık kısımlarına (**-1** ile **1** arasında) ayırır. Toplamları *değer*'e eşittir. Her iki kısmın işareti de *değer*'in işareti ile aynıdır. Tam kısım daima sıfıra yuvarlanır.

modf sayının tamsayı kısmını **tamkısım* içinde saklayıp ondalık kısım ile döner. Örneğin, **modf (2.5, &intpart)** çağrısı **0.5** ile dönerken *intpart* içine **2.0** değerini yerleştirir.

8.4. Kalan İşlevleri

Bu bölümdeki işlevler iki gerçek sayının birbirine bölünmesinden kalanı hesaplar. Her biri biraz farklıdır; sorunlarınıza çözüm olacak biri vardır.

```
double fmod(double bölünen,                                         işlev
            double bölen)
float fmodf(float bölünen,                                          işlev
            float bölen)
long double fmodl(long double bölünen,                              işlev
                  long double bölen)
```

Bu işlevler *bölünen*'in *bölen*'e bölünmesinden kalanı verir. Yani, dönüş değeri `bölünen - bölüm * bölen` işleminin sonucudur. Örneğin, **fmod (6.5, 2.3)** çağrısı **1.9** ile döner. Bu işlev *bölüm*'ü sıfıra en yakın tamsayıya yuvarlayarak işlem yapar.

İşlevin dönüş değeri *bölünen* ile aynı işaretlidir ve *bölen*'den küçüktür.

Eğer *bölen* sıfırsa, **fmod** bir saha hatası sinyaller.

```
double drem(double bölünen,                                         işlev
            double bölen)
float dremf(float bölünen,                                          işlev
            float bölen)
long double dreml(long double bölünen,                              işlev
                  long double bölen)
```

Bu işlevler **fmod** gibidir, farklı olarak *bölüm*'ü sıfıra en yakın tamsayıya değil, en yakın tamsayıya yuvarlayarak işlem yapar. Örneğin, **drem (6.5, 2.3)** çağrısı **-0.4** ile döner; burada yapılan işlem, **6.5** eksi **6.9**'dur.

Sonucun mutlak değeri *bölen*'in mutlak değerine eşit ya da küçüktür. **fmod (***bölünen***, *bölen***) - drem (***bölünen***, *bölen***)** işleminin sonucu daima *bölen* veya eksi *bölen* ya da sıfırdır.

Eğer *bölen* sıfırsa, **drem** bir saha hatası sinyaller.

```
double remainder(double bölünen,                                    işlev
                 double bölen)
float remainderf(float bölünen,                                     işlev
                 float bölen)
long double remainderl(long double bölünen,                         işlev
                       long double bölen)
```

drem işlevlerinin diğer isimleridir.

8.5. Kayan Noktalı Sayılarda İşaret Bitinin Ayarlanması

Kayan noktalı sayılarda elle yapılamayacak kadar karmaşık ve zahmetli bazı işlemler vardır. ISO C99 bu işlemleri yapmak için işlevler tanımlamıştır. Bu işlevler çoğunlukla tek bitlik değişiklikler yaparlar.

```
double copysign(double x,                                          işlev
                double y)
float copysignf(float x,                                           işlev
                float y)
long double copysignl(long double x,                               işlev
                      long double y)
```

Bu işlevler x değerini y'nin işareti ile döndürür. x ya da y NaN ya da sıfır olsa bile bu işlevler çalışır, ayrıca her ikisinin de işareti olabilir (tüm gerçeklemeler bunu desteklemese de). Bu, farklı olduğu söylenebilecek bir kaç işlemden biridir.

copysign hiçbir zaman bir olağandışılığa sebep olmaz.

Bu işlev IEC 559'da tanımlanmıştır (ve IEEE 754/IEEE 854'de, eklerde, önerilen işlevlerden biri olarak).

```
int signbit(bir–float–türü x)                                      işlev
```

signbit tüm gerçek sayı türleri ile çalışabilen soysal bir makrodur. x değerinin işaret biti birse işlev sıfırdan farklı bir değerle döner.

Bu, **x < 0.0** olacak anlamına gelmez, çünkü IEEE 754 kayan noktalılarında sıfırın işaretli olması mümkündür. **-0.0 < 0.0** karşılaştırmasının sonucu yanlıştır, fakat **signbit (-0.0)** çağrısı daima sıfırdan farklı bir değerle döner.

```
double nextafter(double x,                                         işlev
                 double y)
float nextafterf(float x,                                          işlev
                 float y)
long double nextafterl(long double x,                              işlev
                       long double y)
```

nextafter işlevi x'in y değerine doğru gösterilebilir en yakın komşusu ile döner. Sonuç ile x arasındaki adım boyu sonucun türüne bağlıdır. Eğer $x = y$ ise işlev basitçe y ile döner. Eğer ikisinden biri **NaN** ise **NaN** döner. Aksi takdirde ondalık kısım içindeki en kıymetli bitin değerine karşı düşen bir değer yöne bağlı olarak eklenir ya da çıkarılır. Eğer sonuç normalleştirilmiş sayılar aralığının dışına çıkarsa işlev üstten ya da alttan taşma olağandışılığını sinyaller.

Bu işlev IEC 559'da tanımlanmıştır (ve IEEE 754/IEEE 854'de, eklerde, önerilen işlevlerden biri olarak).

```
double nexttoward(double      x,                                   işlev
                  long double y)
float nexttowardf(float       x,                                   işlev
                  long double y)
long double nexttowardl(long double x,                             işlev
                        long double y)
```

Bu işlevler **nextafter** ailesi işlevlere benzer, farklı olarak ikinci argüman **long double** türünde bir değerdir.

```
double nan(const char *tagp)                                    işlev
float nanf(const char *tagp)                                    işlev
long double nanl(const char *tagp)                              işlev
```

nan işlevi hedef platform tarafından desteklenen NaN gösterimi ile döner. **nan ("***karakterler***")** ile **strtod ("NAN(***karakterler***)")** eşdeğerdir.

tagp argümanı olarak belirtilebilecekler belirlenmemiştir denebilir. IEEE 754 sistemlerde bir çok NaN gösterimi vardır ve *tagp* ile bunlardan biri seçilir. Diğer sistemlerde işlev hiçbir şey yapmaz.

8.6. Gerçek Sayı Karşılaştırma İşlevleri

Standart C karşılaştırma işlevleri terimlerden biri NaN olduğunda bir olağandışılık tetikler. örneğin,

```
int v = a < 1.0;
```

ifadesinde *a* bir sayı değilse bir olağandışılık oluşur. (Olağandışılık, karşılaştıma **==** ve **!=** işleçleriyle yapıldığında oluşmaz; sadece sırasıyla yanlış ve doğru döner.) Çoğunlukla bu olağandışılığın oluşması istenmez. Bu bakımdan, ISO C99 NaN saptandığında olağandışılık oluşturmayan karşılaştıma işlevleri tanımlamıştır. Tüm işlevler argümanları herhangi bir gerçek sayı veri türünde olabilen makrolar olarak gerçeklenmiştir. Makrolar argümanlarının sadece bir kere değerlendirileceğini garanti eder (yani aradeğer yuvarlaması yapılmaz).

```
int isgreater(bir–float–tür x,                                  makro
              bir–float–tür y)
```

Bu makro *x* argümanının *y* argümanından büyük olup olmadığına bakar. **(***x***) > (***y***)** ifadesine eşdeğerdir, fakat argümanlarından birinin bir sayı olmaması durumunda bir olağandışılık oluşturmaz.

```
int isgreaterequal(bir–float–tür x,                             makro
                   bir–float–tür y)
```

Bu makro *x* argümanının *y* argümanından büyük ya da eşit olup olmadığına bakar. **(***x***) >= (***y***)** ifadesine eşdeğerdir, fakat argümanlarından birinin bir sayı olmaması durumunda bir olağandışılık oluşturmaz.

```
int isless(bir–float–tür x,                                     makro
           bir–float–tür y)
```

Bu makro *x* argümanının *y* argümanından küçük olup olmadığına bakar. **(***x***) < (***y***)** ifadesine eşdeğerdir, fakat argümanlarından birinin bir sayı olmaması durumunda bir olağandışılık oluşturmaz.

```
int islessequal(bir–float–tür x,                                makro
                bir–float–tür y)
```

Bu makro *x* argümanının *y* argümanından küçük ya da eşit olup olmadığına bakar. **(***x***) <= (***y***)** ifadesine eşdeğerdir, fakat argümanlarından birinin bir sayı olmaması durumunda bir olağandışılık oluşturmaz.

```
int islessgreater(bir–float–tür x,                              makro
                  bir–float–tür y)
```

Bu makro *x* argümanının *y* argümanından küçük ya da büyük olup olmadığına bakar. **(***x***) < (***y***) || (***x***) > (***y***)** ifadesine eşdeğerdir (ama makroda *x* ve *y* sadece bir kere işleme sokulur), fakat argümanlarından birinin bir sayı olmaması durumunda bir olağandışılık oluşturmaz.

Bu makro *x* **!=** *y* ifadesine eşdeğerdir, çünkü terimlerden biri bir sayı değilse ifadenin sonucu doğru olacaktır.

```
int isunordered(bir–float–tür x,                                    makro
                bir–float–tür y)
```

Bu makro argümanlarının NaN olup olmadığına bakar. x ya da x bir sayı değilse doğru, aksi takdirde yanlış döner.

Tüm makinalar bu işlemlere donanım desteği sağlamaz. Bu bakımdan, NaN ile işiniz yoksa bu işlevleri kullanmamalısınız.

Bilgi

isequal veya **isunequal** makroları yoktur. Çünkü **==** ve **!=** işleçleri terimlerden birinin bir sayı olmaması durumunda bir olağandışılık oluşturmadıklarından bunlar için birer makro gereksizdir.

8.7. Çeşitli Gerçek Sayı Aritmetik İşlevleri

Bu bölümdeki işlevler C işleçleriyle yapıldığında kullanışsız olan bazı işlemleri yaparlar. Bazı işlemcilerde bu işlevlerin yaptığı işlemleri eşdeğer C kodundan daha hızlı yapan özel makina komutları vardır ve bu işlevler onları kullanabilmektedir.

```
double fmin(double x,                                               işlev
            double y)
float fminf(float x,                                                işlev
            float y)
long double fminl(long double x,                                    işlev
                  long double y)
```

fmin işlevi x ve y değerlerinden daha küçük olanla döner.

```
((x) < (y) ? (x) : (y))
```

ifadesine benzer bir işlem yapar ama x ve y sadece bir kere işleme sokulur.

Eğer argümanlardan biri bir sayı değilse, diğer argüman döner. Her iki argüman da NaN ise NaN döner.

```
double fmax(double x,                                               işlev
            double y)
float fmaxf(float x,                                                işlev
            float y)
long double fmaxl(long double x,                                    işlev
                  long double y)
```

fmax işlevi x ve y değerlerinden daha büyük olanla döner.

Eğer argümanlardan biri bir sayı değilse, diğer argüman döner. Her iki argüman da NaN ise NaN döner.

```
double fdim(double x,                                               işlev
            double y)
float fdimf(float x,                                                işlev
            float y)
long double fdiml(long double x,                                    işlev
                  long double y)
```

fdim işlevi x ve y arasındaki pozitif farkla döner. x – y işleminde $x > y$ ise işlemin sonucu olan pozitif fark döner, değilse 0 döner.

Eğer argümanlardan biri ya da her ikisi de NaN ise NaN döner.

```
double fma(double x,                                                    işlev
           double y,
           double z)
float fmaf(float x,                                                     işlev
           float y,
           float z)
long double fmal(long double x,                                         işlev
                 long double y,
                 long double z)
```

fma işlevi gerçek sayılarla çarpıp toplama işlemi yapar. İşlev, (x * y) + z işlemini yapar ama ara sonuç hedef türe yuvarlanmaz.

Bazı işlemciler bu işlemi gerçekleştiren özel komutlara sahip olduğundan bu işlev tasarlanmıştır. C derleyicisi bu işlemi bir defada yapamaz, çünkü **x*y + z** ifadesinde ara sonuç yuvarlanır. İşlemin bir defada ve hassasiyet kaybı olmadan yapılmasını istiyorsanız **fma** işlevini seçin.

Çarpıp toplama işleminin donanımda gerçeklenmediği işlemcilerde, **fma** işlevi ara sonuç yuvarlamasından kaçınmak zorunda olduğundan çok yavaş çalışacaktır. `math.h` dosyasında **x*y + z** ifadesinden daha yavaş olmayan **fma** işlevinin sonuç türüne bağlı olarak **FP_FAST_FMA**, **FP_FAST_FMAF** ve **FP_FAST_FMAL** sembolleri tanımlanmıştır. GNU C kütüphanesinde bu, işlemin daima donanımda gerçeklendiği anlamına gelir.

9. Karmaşık Sayılar

ISO C99 standardı ile C'de karmaşık sayılar için destek geldi. Bu yeni bir tür, **complex** ile sağlandı. Sadece `complex.h` dosyası yazılıma dahil edildiğinde geçerli olan bir anahtar sözcüktür. Karmaşık sayı türleri üç tanedir ve üç gerçek sayı türüne karşılıktır: **float complex**, **double complex**, and **long double complex**.

Karmaşık sayıları oluşturabilmek için sayının sanal kısmını belirtecek bir yöntem gerekir. Sanal gerçek sayı sabitler için standart bir sembolleştirme yöntemi yoktur. Bunun yerine, `complex.h` dosyasında karmaşık sayıları oluşturmakta kullanılabilen iki makro tanımlanmıştır.

```
const float complex _Complex_I                                          makro
```

Bu makro `0 + 1i` karmaşık sayısının bir gösterimidir. Bir gerçek sayının **_Complex_I** ile çarpılması tamamen sanal kısımdan oluşan bir karmaşık sayı verir. Bunu karmaşık sabitleri oluşturmakta kullanabilirsiniz:

```
3.0 + 4.0i = 3.0 + 4.0 * _Complex_I
```

_Complex_I * _Complex_I çarpımının **-1** değerini vereceğine ama bu değerin **complex** türünde olacağına dikkat edin.

_Complex_I söylenmesi de yazılması da biraz zor bir sözcüktür. `complex.h` dosyasında aynı sabit için bir de daha kısa bir isim tanımlanmıştır.

```
const float complex I                                                   makro
```

Bu makro **_Complex_I** ile tamamen aynı değerdedir. Çoğu zaman tercih edilir. Ancak, yazılımın herhangi bir yerinde **I**'yi bir belirteç olarak kullanırsanız sorun çıkar. **I**'yi kendi amaçlarınız için kullanmak isterseniz basitçe şunu yazabilirsiniz:

```
#include <complex.h>
#undef I
```

Böyle bir durumda, **_Complex_I** için başka bir kısa isim (örneğin, **J**) tanımlamanız önerilir.

10. Karmaşık Sayıların İzdüşümleri, Eşlenikleri ve Analizi

ISO C99 ayrıca, karmaşık sayıların temel işlemlerini (kısımlarına ayırmak, eşleniğini almak gibi) yapmak için işlevler de tanımlamıştır. Bu işlevler `complex.h` dosyasında bildirilmiştir. Tüm işlevlerin 3 karmaşık sayı türünün herbiri için bir eşdeğeri vardır.

```
double creal(complex double z)                                  işlev
float crealf(complex float z)                                   işlev
long double creall(complex long double z)                       işlev
```

Bu işlevler *z* karmaşık sayısının gerçel kısmı ile döner.

```
double cimag(complex double z)                                  işlev
float cimagf(complex float z)                                   işlev
long double cimagl(complex long double z)                       işlev
```

Bu işlevler *z* karmaşık sayısının sanal kısmı ile döner.

```
complex double conj(complex double z)                           işlev
complex float conjf(complex float z)                            işlev
complex long double conjl(complex long double z)                işlev
```

Bu işlevler *z* karmaşık sayısının eşleniği ile döner. Eşlenik karmaşık sayılar, gerçel kısımları aynı, sanal kısımları ise ters işaretli olarak aynı olan birer karmaşık sayıdır. Başka bir deyişle, eşlenik karmaşık sayılar karmaşık düzlemde gerçel eksene göre simetriktir. Yani, **conj(a + bi) = a - bi**'dir.

```
double carg(complex double z)                                   işlev
float cargf(complex float z)                                    işlev
long double cargl(complex long double z)                        işlev
```

Bu işlev *z* karmaşık sayısının kutupsal koordinatlardaki argümanı ile döner. Argüman, karmaşık düzlemde karmaşık sayının oluşturduğu vektörün pozitif gerçel eksenle oluşturduğu açıdır. Bu açı radyan cinsinden 0 ile 2pi arasındadır.

```
complex double cproj(complex double z)                          işlev
complex float cprojf(complex float z)                           işlev
complex long double cprojl(complex long double z)               işlev
```

Bu işlevler *z* karmaşık sayısının Riemann küresindeki izdüşümü ile döner. Sonsuz sanal kısımlı değerlerin gerçel eksende (gerçel kısım NaN olsa bile) pozitif sonsuza izdüşümü alınır. Eğer gerçel kısım sonsuzsa sonuç şuna eşdeğerdir:

```
INFINITY + I * copysign (0.0, cimag (z))
```

11. Dizgelerdeki Sayıların Çözümlenmesi

Bu bölümdeki işlevler tamsayıların ya da gerçek sayıların dizgelerden okunması için kullanılır. Bu işlem için **sscanf** veya ilgili işlevler kullanmak daha iyi olabilir; bkz. *Biçimli Girdi* (sayfa: 277). Fakat dizge içindeki dizgecikleri elle bulup onları tek tek sayılara dönüştürmek yazılımı daha güçü yapabilirr.

11.1. Tamsayıların Çözümlenmesi

str işlevleri stdlib.h dosyasında, **wcs** ile başlayanlar ise wchar.h dosyasında bildirilmiştir. Bu bölümdeki işlevlerin prototiplerinde **restrict** kullanımı şaşırtıcı olabilir. Kullanışsız görünür ama ISO C standardı onu kullandığı için (burada tanımlı işlevler için) biz de kullandık.

```
long int strtol(const char *restrict dizge,                    işlev
                char **restrict      kalan–dizge,
                int                  taban)
```

strtol ("string–to–long" kısaltması) işlevi *dizge* dizgesinin baştarafını bir işaretli tamsayıya dönüştürür ve sonucu **long int** türünde bir değer olarak döndürür.

İşlev *dizge*'yi şu şekilde analiz etmeye çalışır:

- Boşluk karakterlerinden oluşan bir dizge (muhtemelen boş). Boşluk karakterleri **isspace** işlevi ile saptanır ve bunlar iptal edilir (bkz. *Karakterlerin Sınıflandırılması* (sayfa: 82)).
- İsteğe bağlı artı ya da eksi işareti (**+** veya **−**).
- Tabanı *taban* ile belirtilen aralarında boşluk olmayan rakamlar.

  taban sıfırsa, rakam dizgesi **0** (sekizlik taban belirtir) ile ya da **0x** veya **0X** (onaltılık taban belirtir)ile başlamıyorsa rakamların onluk tabanda verildiği varsayılır; yani C'deki tamsayı sabit sözdizimi kullanılır.

  Aksi takdirde *taban*, **2** ile **36** arasında olmalıdır. Eğer *taban* olarak **16** verilmişse, rakam dizgesi isteğe bağlı olarak **0x** veya **0X** ile başlayabilir. Eğer taban olarak kuraldışı bir değer belirtilmişse **0l** değeri dömer ve **errno** değişkenine **EINVAL** atanır.
- Dizgede kalan karakterler. Eğer *kalan–dizge* bir boş gösterici değilse işlev bu artıkları **kalan–dizge* içine yerleştirir.

Eğer dizge boşsa, sadece boşluk karakterleri içeriyorsa ya da başlangıçtaki altdizge *taban* ile belirtilen bir tamsayı için umulan sözdizimine sahip değilse hiçbir dönüşüm yapılmaz. Bu durumda, işlev sıfırla döner ve *dizge* değeri **kalan–dizge* içine yerleştirir.

Yerel, standart **"C"** yerelinden farklıysa, bu işlev gerçeklemeye bağlı olarak ek sözdizimini tanıyabilir.

Eğer dizge geçerli sözdizimine sahip olduğu halde taşmadan dolayı değer gösterilemiyorsa, değerin işaretine bağlı olarak **LONG_MAX** ya da **LONG_MIN** döner (bkz. *Bir Tamsayı Türün Aralığı* (sayfa: 821)). Ayrıca taşmayı belirtmek üzere **errno** değişkenine **ERANGE** atanır.

strtol işlevinin döndürdüğü değere bakarak hata sınaması yapmayın, çünkü dizge **0l**, **LONG_MAX** veya **LONG_MIN** gibi sözdizimsel olarak geçerli bir sayı olabilir. Bunun yerine *kalan–dizge*'nin gösterdiği dizgenin sayıdan sonra umduğunuz dizgeyi içerip içermediğine bakın. Örneğin dizge sayıdan sonra başka bir karakter içermiyorsa dönüş değerinde **'\0'** varlığına bakarsınız. Ayrıca çağrı öncesi *errno* değişkenine sıfır atayıp çağrıdan sonra değişkenin değerine bakarak taşma durumu olup olmadığını saptayabilirsiniz.

Bu bölümün sonunda bir örnek bulacaksınız.

```
long int wcstol(const wchar_t *restrict dizge,                 işlev
                wchar_t **restrict      kalan–dizge,
                int                     taban)
```

wcstol işlevi geniş karakter kabul etmesi dışında hemen herşeyiyle **strtol** işlevine benzer.

wcstol işlevi ISO C90'ın 1. düzeltmesinde vardı.

```
unsigned long int strtoul(const char *retrict dizge,                    işlev
                          char **restrict     kalan–dizge,
                          int                 taban)
```

strtoul ("string–to–unsigned–long" kısaltması) işlevi sonucu bir **unsigned long int** değer olarak döndürmesi dışında **strtol** işlevi gibidir. Sözdizimi **strtol** işlevindeki gibidir. Taşma durumunda dönüş değeri **ULONG_MAX**'tır (bkz. *Bir Tamsayı Türün Aralığı* (sayfa: 821)).

dizge içinde bir negatif sayı varsa, **strtoul** işlevi *strtol* işlevi gibi davranır ama sonucu bir işaretsiz tamsayıya dönüştürür. Örneğin, işlev **"-1"** dizgesi için **ULONG_MAX** ile, girdi **LONG_MIN**'den daha negatifse (**ULONG_MAX** + 1) / 2 ile döner.

Eğer *taban* kapsamdışı ise işlev *errno* değişkenine **EINVAL** değerini, taşma durumunda ise **ERANGE** değerini atar.

```
unsigned long int wcstoul(const wchar_t *restrict dizge,                işlev
                          wchar_t **restrict      kalan–dizge,
                          int                     taban)
```

wcstoul işlevi geniş karakteleri kabul etmesi dışında hemen herşeyiyle **strtoul** işlevinin benzeridir.

wcstoul işlevi ISO C90'ın 1. düzeltmesinde vardı.

```
long long int strtoll(const char *restrict dizge,                       işlev
                      char **restrict      kalan–dizge,
                      int                  taban)
```

strtoll işlevi daha büyük değer kabul ederek sonucu **long long int** türünde bir değer olarak döndürmesi dışında **strtol** gibidir.

Eğer dizge geçerli sözdizimine sahip olduğu halde taşmadan dolayı değer gösterilemiyorsa, değerin işaretine bağlı olarak **LONG_LONG_MAX** ya da **LONG_LONG_MIN** döner (bkz. *Bir Tamsayı Türün Aralığı* (sayfa: 821)). Ayrıca taşmayı belirtmek üzere **errno** değişkenine **ERANGE** atanır.

strtoll işlevi ISO C99'da tanıtıldı.

```
long long int wcstoll(const wchar_t *restrict dizge,                    işlev
                      wchar_t **restrict      kalan–dizge,
                      int                     taban)
```

wcstoll işlevi geniş karakter kabul etmesi dışında hemen herşeyiyle **strtoll** işlevine benzer.

wcstoll işlevi ISO C90'ın 1. düzeltmesinde vardı.

```
long long int strtoq(const char *restrict dizge,                        işlev
                     char **restrict      kalan–dizge,
                     int                  taban)
```

strtoq ("string–to–quad–word" kısaltması) işlevi**strtoll** işlevinin BSD ismidir.

```
long long int wcstoq(const wchar_t *restrict dizge,                     işlev
                     wchar_t **restrict      kalan–dizge,
                     int                     taban)
```

wcstoq işlevi geniş karakter kabul etmesi dışında hemen herşeyiyle **strtoq** işlevine benzer.

wcstoq işlevi bir GNU oluşumdur.

```
unsigned long long int strtoull(const char *restrict dizge,                    işlev
                                char **restrict      kalan–dizge,
                                int                  taban)
```

strtoul işlevinin **strtol** işlevine ilgisi gibi **strtoull** işlevi de **strtoll** işleviyle benzer ilgiye sahiptır.

strtoull işlevi ISO C99'da tanıtıldı.

```
unsigned long long int wcstoull(const wchar_t *restrict dizge,                 işlev
                                wchar_t **restrict      kalan–dizge,
                                int                     taban)
```

wcstoull işlevi geniş karakter kabul etmesi dışında hemen herşeyiyle **strtoull** işlevine benzer.

wcstoull işlevi ISO C90'ın 1. düzeltmesinde vardı.

```
unsigned long long int strtouq(const char *restrict dizge,                     işlev
                               char **restrict      kalan–dizge,
                               int                  taban)
```

strtouq işlevi **strtoull** işlevinin BSD ismidir.

```
unsigned long long int wcstouq(const wchar_t *restrict dizge,                  işlev
                               wchar_t **restrict      kalan–dizge,
                               int                     taban)
```

wcstouq işlevi geniş karakter kabul etmesi dışında hemen herşeyiyle **strtouq** işlevine benzer.

wcstouq işlevi bir GNU oluşumdur.

```
intmax_t strtoimax(const char *restrict dizge,                                 işlev
                   char **restrict      kalan–dizge,
                   int                  taban)
```

strtoimax işlevi daha büyük değerler kabul ederek sonucu **intmax_t** türünde döndürmesi dışında **strtol** gibidir.

Eğer dizge geçerli sözdizimine sahip olduğu halde taşmadan dolayı değer gösterilemiyorsa, değerin işaretine bağlı olarak **INTMAX_MAX** ya da **INTMAX_MIN** döner (bkz. *Bir Tamsayı Türün Aralığı* (sayfa: 821)). Ayrıca taşmayı belirtmek üzere **errno** değişkenine **ERANGE** atanır.

intmax_t türü hakkında daha fazla bilgi için *Tamsayılar* (sayfa: 506) bölümüne bakınız. **strtoimax** işlevi ISO C99'da tanıtıldı.

```
intmax_t wcstoimax(const wchar_t *restrict dizge,                              işlev
                   wchar_t **restrict      kalan–dizge,
                   int                     taban)
```

wcstoimax işlevi geniş karakter kabul etmesi dışında hemen herşeyiyle **strtoimax** işlevine benzer.

wcstoimax işlevi ISO C99'da tanıtıldı.

```
uintmax_t strtoumax(const char *restrict dizge,                                işlev
                    char **restrict      kalan–dizge,
                    int                  taban)
```

strtoul işlevinin **strtol** işlevine ilgisi gibi **strtoul** işlevi **strtoumax** işleviyle de benzer ilgiye sahiptır.

intmax_t türü hakkında daha fazla bilgi için *Tamsayılar* (sayfa: 506) bölümüne bakınız. **strtoumax** işlevi ISO C99'da tanıtıldı.

```
uintmax_t wcstoumax(const wchar_t *restrict dizge,                    işlev
                    wchar_t **restrict      kalan–dizge,
                    int                     taban)
```

wcstoumax işlevi geniş karakter kabul etmesi dışında hemen herşeyiyle **strtoumax** işlevine benzer.

wcstoumax işlevi ISO C99'da tanıtıldı.

```
long int atol(const char *dizge)                                      işlev
```

Bu işlev *taban* argümanı **10** olarak belirtilen **strtol** işlevine benzer, ancak taşma hatalarının saptanmasını gerektirmez. **atol** işlevi mevcut kodla uyumluluk adına vardır; **strtol** kullanım bakımından daha güçlüdür.

```
int atoi(const char *dizge)                                           işlev
```

Bu işlev **int** türünde bir değer döndürmesi dışında **atol** işlevi gbidir. Ayrıca, **atoi** işlevinin atıl olduğu varsayılır; yerine **strtol** kullanın.

```
long long int atoll(const char *dizge)                                işlev
```

Sonucu **long long int** türünde döndürmesi dışında **atol** işlevine benzer.

atoll işlevi ISO C99'da tanıtıldı. Tamamen atıldır; yerine **strtoll** kullanın.

Buraya kadar bahsedilen işlevler artık yerel veride tanımlanan diğer gösterimleri tanımamaktadır. Bazı yereller çok büyük sayıların daha rahat okunabilmesi için binler ayracı kullanmaktadır. Böyle sayıları okutmak için **scanf** işlevini **'** imi ile kullanın.

Bu örnekte bir işlev bir dizgeyi çözümleyip elde ettiği tamsayıların toplamı ile dönmektedir:

```
int
sum_ints_from_string (char *string)
{
  int sum = 0;

  while (1) {
    char *tail;
    int next;

    /* Baştaki boşlukları atlayalım.  */
    while (isspace (*string)) string++;
    if (*string == 0)
      break;

    /* Artık boşluk karakeri kalmadı,  */
    /* rakamlara bakabiliriz.  */
    errno = 0;
    /* Çözümle.  */
    next = strtol (string, &tail, 0);
    /* Taşmamışsa, ekle.  */
    if (errno)
```

```
      printf ("Taştı\n");
    else
      sum += next;
    /* Kalan dizgeyi tekrar işleme sokalım.  */
    string = tail;
  }

  return sum;
}
```

11.2. Gerçek Sayıların Çözümlenmesi

str işlevleri `stdlib.h` dosyasında, **wcs** ile başlayan işlevler `wchar.h` dosyasında bildirilmiştir. Bu bölümdeki işlevlerin prototiplerinde **restrict** kullanımı şaşırtıcı olabilir. Kullanışsız görünür ama ISO C standardı onu kullandığı için (burada tanımlı işlevler için) biz de kullandık.

```
double strtod(const char *restrict dizge,                    işlev
              char **restrict      kalan–dizge)
```

strtod ("string–to–double" kısaltması) işlevi *dizge* dizgesinin baştarafını bir gerçek sayıya dönüştürür ve sonucu **double** türünde bir değer olarak döndürür.

İşlev *dizge*'yi şu şekilde analiz etmeye çalışır:

- Boşluk karakterlerinden oluşan bir dizge (muhtemelen boş). Boşluk karakterleri **isspace** işlevi ile saptanır ve bunlar iptal edilir (bkz. *Karakterlerin Sınıflandırılması* (sayfa: 82)).
- İsteğe bağlı artı ya da eksi işareti (**+** veya **−**).
- Onluk ya da onaltılık biçimde bir gerçek sayı. Onluk biçim şöyle çözümlenmeye çalışılır:
  - İsteğe bağlı olarak gösterimi yerele bağlı bir ondalık nokta (normalde **.**) içeren ve aralarında boşluk bulunmayan rakamlar. (Bkz. *Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169)).
  - İsteğe bağlı üstel kısım. **e** veya **E** karakteri ile isteğe bağlı bir işaret ve rakamlardan oluşur.

  Onaltılık biçim şöyle çözümlenmeye çalışılır:
  - 0x veya 0X ile başlayan, isteğe bağlı olarak gösterimi yerele bağlı bir ondalık nokta (normalde **.**) içeren ve aralarında boşluk bulunmayan rakamlar. (Bkz. *Soysal Sayısal Biçimleme Parametreleri* (sayfa: 169)).
  - İsteğe bağlı ikilik üstel kısım. **p** veya **P** karakteri ile isteğe bağlı bir işaret ve rakamlardan oluşur.
- Dizgede kalan karakterler. Eğer *kalan–dizge* bir boş gösterici değilse işlev bu artıkları **kalan–dizge* içine yerleştirir.

Eğer dizge boşsa, sadece boşluk karakterleri içeriyorsa ya da başlangıçtaki altdizge bir gerçek sayı için umulan sözdizimine sahip değilse hiçbir dönüşüm yapılmaz. Bu durumda, işlev sıfırla döner ve *dizge* değeri **kalan–dizge* içine yerleştirir.

Yerel, standart **"C"** ya da **"POSIX"** yerellerinden farklıysa, bu işlev gerçeklemeye bağlı olarak ek sözdizimini tanıyabilir.

Eğer dizge bir gerçek sayı için geçerli sözdizimine sahip olduğu halde değer **double** türün kapsamı dışındaysa, **strtod** işlevi *Hataların Matematiksel İşlevlerce Raporlanması* (sayfa: 515) bölümünde açıklandığı gibi üstten ya da alttan taşma sinyalleyecektir.

strtod işlevi dört özel girdi dizgesi tanır. Bu dizgelerden **"inf"** ve **"infinity"** ya ∞'a ya da gerçek sayı biçimi sonsuzlukları desteklemiyorsa gösterilebilir en büyük sayıya dönüştürülür. Önlerine işareti belirtmek için bir **"+"** veya **"-"** konulabilir. Bu dizgelerde harf büyüklüğünün önemi yoktur.

"nan" ve **"nan(***karakterler***)"** dizgeleri ise NaN'a dönüştürülür. Yine harf büyüklüğünün önemi yoktur. Eğer *karakterler* belirtilmişse, NaN'ın kısmi bir gösterimine (herşey olabilir) karşılık olarak kullanılır.

Sıfır geçerli bir sonuç olduğu kadar hata oluştuğunu da belirtebilir. Hatayı sınamak için *errno* ve *kalan–dizge*'yi **strtol** işlevinin açıklamasında anlatıldığı gibi kullanmalısınız.

```
float strtof(const char *dizge,                                          işlev
             char      **kalan–dizge)
long doublestrtold(const char *dizge,                                    işlev
                   char      **kalan–dizge)
```

Bu işlevler **strtod** işlevinin benzeri olmakla birlikte, sırayla **float** ve **long double** değerle dönerler. Hataları **strtod** gibi raporlarlar. Hassasiyeti daha düşük olduğundan **strtof** işlevi **strtod** işlevinden daha hızlı olabilir; tersine, hassasiyeti daha yüksek olduğundan **strtold** daha yavaş olabilir (**long double** türünün ayrı bir tür olduğu sistemlerde).

Bu işlevler ISO C99'da yeni olmasına rağmen evvelce GNU oluşumuydular.

```
double wcstod(const wchar_t *restrict dizge,                             işlev
              wchar_t **restrict      kalan–dizge)
float wcstof(const wchar_t *dizge,                                       işlev
             wchar_t      **kalan–dizge)
long double wcstold(const wchar_t *dizge,                                işlev
                    wchar_t      **kalan–dizge)
```

wcstod, **wcstof** ve **wcstol** işlevleri sırayla **strtod**, **strtof** ve **strtold** işlevleriyle geniş arakterleri kabul etmeleri dışında hemen herşeyleriyle benzerdirler.

wcstod işlevi ISO C90'ın 1. düzeltmesinde vardı. **wcstof** ve **wcstold** işlevleri ise ISO C99'da tanıtıldı.

```
double atof(const char *dizge)                                           işlev
```

Bu işlev alttan ve üstten taşma hatalarının saptanmasını gerektirmemesi dışında **strtod** işlevine benzer. **atof** işlevi mevcut kodla uyumluluk adına vardır; **strtod** kullanım bakımından daha güçlüdür.

GNU C kütüphanesi ayrıca bu işlevlerin dönüşümde yerel kullanmak için bir ek argüman alan **_l** sürümlerini de içerir. Bkz. *Tamsayıların Çözümlenmesi* (sayfa: 528).

12. Eski Moda System V Sayıdan Dizgeye Dönüşüm İşlevleri

Eski System V C kütüphanesi sayıları dizgelere çeviren, kullanımı zor ve kullanışsız üç işlev içerir. GNU C kütüphanesi bu işlevleri bazı doğal oluşumlarla birlikte içerir.

Bu işlevler sadece glibc'de ve AT&T Unix soyundan gelen sistemlerde vardır. Bu bakımdan, bunlara özellikle ihtiyaç duymuyorsanız bunların yerine standart olan **sprintf**'i kullanmak daha iyidir.

Bu işlevlerin tamamı `stdlib.h` dosyasında bildirilmiştir.

```
char *ecvt(double değer,                                                 işlev
           int    hane–sayısı,
           int   *ondalık–nokta,
           int   *negatif)
```

ecvt işlevi *değer* gerçek sayısını en fazla *hane–sayısı* onluk rakama dönüştürür. Dönen dizge ondalık noktayı ve işareti içermez. Dizgenin ilk rakamı sıfırdan farklıdır (*değer* sıfır olmadıkça) ve son rakan en yakına yuvarlanır. **ondalık–nokta*'ya ondalık noktadan sonraki ilk hanenin indisi yerleşir. *değer* negatifse **negatif*'e sıfırdan farklı bir değer, değilse sıfır yerleşir.

Eğer *hane–sayısı* rakam **double** türünün hassasiyetini aşarsa sisteme özel değere düşürülür.

Dönen dizge durağan olarak ayrıldığından işlevin sonraki çağrıları bunun üzerine yazar.

Eğer *değer* sıfırsa, **ondalık–nokta*'nın **0** mı yoksa **1** mi olacağı gerçeklemeye bağlıdır.

Örnek: **ecvt (12.3, 5, &d, &n)** çağrısı **"12300"** ile döner ve *d*'ye **2**, *n*'ye **0** atanır.

```
char *fcvt(double değer,                                                    işlev
           int    hane–sayısı,
           int   *ondalık–nokta,
           int   *negatif)
```

fcvt işlevi *hane–sayısı*'nın ondalık noktadan sonraki hane sayısını göstermesi dışında **ecvt** gibidir. Eğer *hane–sayısı* sıfırdan küçükse, *değer* ondalık noktanın solundaki *hane–sayısı*+1'inci haneye yuvarlanır. Örneğin, *hane–sayısı* **-1** ise, *değer* en yakın 10'a yuvarlanır. Eğer *hane–sayısı* negatifse ve *değer*'deki ondalık noktanın solundaki hane sayısından daha büyükse, *değer* bir en kıymetli rakama yuvarlanacaktır.

Eğer *hane–sayısı* rakam **double** türünün hassasiyetini aşarsa sisteme özel değere düşürülür.

Dönen dizge durağan olarak ayrıldığından işlevin sonraki çağrıları bunun üzerine yazar.

```
char *fcvt(double değer,                                                    işlev
           int    hane–sayısı,
           char  *tampon)
```

gcvt işlevi **sprintf(tampon, "%*g", hanesayisi, deger** çağrısına eşdeğerdir. Sadece uyumluluk adına vardır. *tampon* ile döner.

Eğer *hane–sayısı* rakam **double** türünün hassasiyetini aşarsa sisteme özel değere düşürülür.

Bu üç işleve ek olarak, GNU C kütüphanesi bu işlevlerin **long double** argüman alan sürümlerini de içerir.

```
char *qecvt(long double değer,                                              işlev
            int         hane–sayısı,
            int        *ondalık–nokta,
            int        *negatif)
```

Bu işlev ilk parametresinde **long double** argüman alması ve *hane–sayısı*'nı **long double** hassasiyeti ile sınırlaması dışında **ecvt** işlevinin benzeridir.

```
char *qfcvt(long double değer,                                              işlev
            int         hane–sayısı,
            int        *ondalık–nokta,
            int        *negatif)
```

Bu işlev ilk parametresinde **long double** argüman alması ve *hane–sayısı*'nı **long double** hassasiyeti ile sınırlaması dışında **fcvt** işlevinin benzeridir.

```
char *qgcvt(long double değer,                                          işlev
            int          hane–sayısı,
            char        *tampon)
```

Bu işlev ilk parametresinde **long double** argüman alması ve *hane–sayısı*'nı **long double** hassasiyeti ile sınırlaması dışında **gcvt** işlevinin benzeridir.

ecvt ve **fcvt** işlevleri ve onların **long double** eşdeğerlerinde, dönen dizge durağan olarak ayrıldığından işlevin sonraki çağrıları bunun üzerine yazar. GNU C kütüphanesi ek işlevlerin dönen dizgeyi kullanıcı tanımlı tampona yazarak döndüğü sürümlerini de içerir. Bunların isimleri teamüen **_r** soneki alır.

gcvt işlevi zaten kullanıcı tanımlı tampon kullandığından onun için bir **gcvt_r** işlevi yoktur.

```
int ecvt_r(double değer,                                                işlev
           int    hane–sayısı,
           int   *ondalık–nokta,
           int   *negatif,
           char  *tampon,
           size_t uzunluk)
```

ecvt_r işlevi, sonucu *uzunluk* uzunluktaki kullanıcı tanımlı *tampon*'a yerleştirmesi dışında **ecvt** işlevi ile aynıdır. Bir hata durumunda dönüş değeri **-1**, aksi takdirde sıfırdır.

Bu işlev bir GNU oluşumudur.

```
int fcvt_r(double değer,                                                işlev
           int    hane–sayısı,
           int   *ondalık–nokta,
           int   *negatif,
           char  *tampon,
           size_t uzunluk)
```

fcvt_r işlevi, sonucu *uzunluk* uzunluktaki kullanıcı tanımlı *tampon*'a yerleştirmesi dışında **fcvt** işlevi ile aynıdır. Bir hata durumunda dönüş değeri **-1**, aksi takdirde sıfırdır.

Bu işlev bir GNU oluşumudur.

```
int qecvt_r(long double değer,                                          işlev
            int          hane–sayısı,
            int         *ondalık–nokta,
            int         *negatif,
            char        *tampon,
            size_t       uzunluk)
```

qecvt_r işlevi, sonucu *uzunluk* uzunluktaki kullanıcı tanımlı *tampon*'a yerleştirmesi dışında **qecvt** işlevi ile aynıdır. Bir hata durumunda dönüş değeri **-1**, aksi takdirde sıfırdır.

Bu işlev bir GNU oluşumudur.

```
int qfcvt_r(long double değer,                                          işlev
            int          hane–sayısı,
            int         *ondalık–nokta,
            int         *negatif,
            char        *tampon,
            size_t       uzunluk)
```

qfcvt_r işlevi, sonucu *uzunluk* uzunluktaki kullanıcı tanımlı *tampon*'a yerleştirmesi dışında **qfcvt** işlevi ile aynıdır.Bir hata durumunda dönüş değeri **-1**, aksi takdirde sıfırdır.

Bu işlev bir GNU oluşumudur.

XXI. Tarih ve Zaman

İçindekiler

Bu oylumda tarihler, saatler yani zamanla ilgili, zamanı saptamak, değiştirmek ve zaman gösterimleri arasında dönüşüm gibi işlemleri gerçekleştiren işlevlerden bahsedilecektir.

1. Zaman Kavramları

"Time" sözcüğü ingilizcede çok fazla anlama geldiğinden bir teknik kılavuzda zamanı tartışmak zor olabilir (Ç.N.:Katlanacağız `:-)`).

"Takvim zamanı" zamanın sürekliliği içinde bir noktayı ifade eder, örneğin: 4 Kasım 1990 18:02.5 UTC. Buna genellikle ***mutlak zaman*** denir. Buna "tarih" demiyoruz, çünkü tarih mutlak zamanın doğasında var. Bir ***zaman aralığı*** zamanın sürekliliği içinde iki mutlak zaman arasındaki kesintisiz parçadır. Örnek: 4 Temmuz 1980 tarihinde saat 9:00 ile 10:00 arası. ***Süre*** bir aralığın uzunluğudur. Örnek: 35 dakika.

Toplam süre, adı üstünde, sürelerin toplamıdır. Örnek: Parça parça okunan bir kitabın toplam okuma süresi 9 saattir, gibi.

Dönem sürekli belirli aralıklarla yinelenen iki olay arasındaki süredir.

İşlemci zamanı mutlak zaman gibidir, bir farkla zamanın başlangıcı, bir sürecin işlemciyi kullanmaya başladığı andır. İşlemci zamanı bu nedenle sürece bağlıdır.

İşlemci süresi ise işlemcinin kullanım süresidir. Temel bir sistem özkaynağıdır. Çok işlemcili sistemlerde bir işlemcinin ne kadar süreyle işlem yapacağını belirten süredir.

2. Süre

Süreyi göstermenin tek yolu basit bir aritmetik veri türünü kullanmaktır. Aşağıdaki işlev iki mutlak zaman arasındaki süreyi hesaplar. Bu işlev `time.h` başlık dosyasında tanımlanmıştır.

```
double difftime(time_t zaman1,                                          işlev
                time_t zaman0)
```

difftime işlevi *zaman1* ve time *zaman0* mutlak zamanları arasındaki süreyi **double** türünde saniye cinsinden bir değer olarak hesaplar. Artık süre desteği etkinleştirilmedikçe farkın artık süresi yoksayılır.

GNU sisteminde, **time_t** değerleri arasında basitçe bir çıkarma işlemi yapabilirsiniz. Fakat diğer sistemlerde **time_t** başka bir kodlamaya karşılık olabileceğinden doğrudan çıkarma işlemi ile doğru sonuç alınamayabilir.

GNU C kütüphanesinde özellikle süreyi ifade edebilen iki veri türü vardır. Bunlar çeşitli kütüphane işlevlerince kullanılmıştır ve siz de amaçlarınıza uygun olarak bunları kullanabilirsiniz. İkisi de aslında aynı olmakla birlikte biri saniye cinsinden bir çözünürlük içinken diğeri nanosaniye cinsinden bir çözünürlük içindir.

```
struct timeval                                                          veri türü
```

`struct` **timeval** yapısı süreyi göstermekte kullanılır. `sys/time.h` başlık dosyasında bildirilmiş olan yapı şu üyelere sahiptir:

`long int` **tv_sec**
: Sürenin tam saniyelerden oluşan kısmını içerir (ondalık ayracın solundaki kısım).

`long int` **tv_usec**
: Sürenin saniyeden küçük kısmını (ondalık ayracın sağındaki kısım) mikrosaniye cinsinden içerir. Daima bir milyondan küçüktür.

```
struct timespec                                                         veri türü
```

`struct` **timespec** yapısı süreyi göstermekte kullanılır. `time.h` başlık dosyasında bildirilmiş olan yapı şu üyelere sahiptir:

`long int` **tv_sec**
: Sürenin tam saniyelerden oluşan kısmını içerir (ondalık ayracın solundaki kısım).

`long int` **tv_nsec**
: Sürenin saniyeden küçük kısmını (ondalık ayracın sağındaki kısım) nanosaniye cinsinden içerir. Daima bir milyardan küçüktür.

Çoğunlukla **struct timeval** veya **struct timespec** türündeki iki değeri birbirinden çıkarmak gerekir. Burada en iyi yol budur. **tv_sec** üyesinin veri türünün işaretsiz veri türlerinden biri olduğu kendine özgü işletim sistemlerinde bile bu çalışır.

```
/* 'struct timeval' değerleri olan X ve Y arasında çıkarma yap
    ve sonucu RESULT içinde sakla.
    Fark negatifse 1 ile değilse 0 ile dön.  */

int
timeval_subtract (result, x, y)
      struct timeval *result, *x, *y;
{
  /* Elde 1 diyelim. */
  if (x->tv_usec < y->tv_usec) {
    int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
    y->tv_usec -= 1000000 * nsec;
    y->tv_sec += nsec;
  }
  if (x->tv_usec - y->tv_usec > 1000000) {
```

```
    int nsec = (x->tv_usec - y->tv_usec) / 1000000;
    y->tv_usec += 1000000 * nsec;
    y->tv_sec -= nsec;
  }

  /* Şimdi hesaplayalım. tv_usec kesinlikle pozitif. */
  result->tv_sec = x->tv_sec - y->tv_sec;
  result->tv_usec = x->tv_usec - y->tv_usec;

  /* Sonuç negatifse 1 dönsün. */
  return x->tv_sec < y->tv_sec;
}
```

struct timeval kullanan işlevler **gettimeofday** ve **settimeofday** işlevleridir.

Doğrudan özellikle sürelerle çalışan hiçbir GNU C kütüphanesi işlevi olmamakla birlikte, mutlak zaman, işlemci süresi, alarm ve uyku ile ilgili işlevler sürelerle ilgili birşeyler yaparlar.

3. İşlemci Zamanı ve İşlemci Süresi

Bir yazılımı eniyilemeye ya da verimliliğini ölçmeye çalışıyorsanız, ne kadar işlemci süresi kullanıldığını bilmek iyidir. Bunun için, mutlak zaman ve süreler kullanışsızdır, çünkü bir süreç işlemciyi kullanırken arada bazı G/Ç işlemlerini veya başka süreçleri bekleyebilir. Buna karşın, bu bölümdeki işlevleri kullanarak işlemci kullanımı ile ilgili bilgi alabilirsiniz.

İşlemci zamanı (bkz. *Zaman Kavramları* (sayfa: 538)), ***saat tikleri***'nin sayısı olarak **clock_t** türünde bir veridir. İşlemci bazı keyfî ancak belli başlı olaylarda kullanıldığından bir sürecin işlemciyi etkin olarak kullandığı sürelerin toplamı hesaplanabilir. GNU sistemlerinde sürecin oluşturulması böyle bir olaydır. Keyfîlik yanında genelde, sürecin belli bir parçası için olay daima aynı olaydır, bu durumda işlemci zamanını hesaplamanın başlangıcında ve bitişinde saptayarak belli bir hesaplama için ne kadar işlemci süresi kullanıldığı ölçülebilir.

GNU sisteminde, **clock_t** ile **long int** eşdeğerdir ve **CLOCKS_PER_SEC** bir tamsayı değerdir. Fakat diğer sistemlerde, **clock_t** ve **CLOCKS_PER_SEC** birer tamsayı ya da birer gerçek sayı türü olabilir. Önceki bölümdeki örnekte olduğu gibi, işlemci zamanı değerlerinin **double**'a yükseltgenmesi, aritmetik ve gösterim işlemlerinin düzgün olarak yapılmasını sağlarken sistemlerin olası farklı türlerdeki gösterimlerinden etkilenmemesini de sağlar.

Saat tikleri sayısının, sayı türlerinin genişliklerine bağlı olarak belli bir sınırı olduğunu unutmayın. **CLOCKS_PER_SEC**'in bir milyon olarak atandığı 32 bitlik bir sistemde bu işlev yaklaşık 72 dakikada bir aynı değeri döndürecektir.

Bir sürecin kullandığı işlemci süresini incelemek ve denetlemek için de işlevler vardır, bunlar hakkında *Özkaynak Kullanımı ve Sınırlaması* (sayfa: 572) bölümünde bilgi verilmiştir.

3.1. İşlemci Zamanının Sorgulanması

Bir sürecin işlemci zamanını almak için **clock** işlevini kullanabilirsiniz. Bu oluşum `time.h` başlık dosyasında bildirilmiştir.

Genel olarak, **clock** işlevi ilgilenilen bir zaman aralığının başında ve sonunda çağrılır ve bu değerler birbirinden çıkarılıp sonuç **CLOCKS_PER_SEC** ile bölünerek işlemci süresi aşağıdaki gibi elde edilir:

```
#include <time.h>

clock_t start, end;
double cpu_time_used;
```

```
start = clock();
... /* Burada bir takım işlemler yapılıyor. */
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Belirli süreyi elde etmek için tek bir işlemci zamanı değerini kullanmayın.[9] Ya yukarıdaki gibi bir çıkartma işlemi yapın ya da doğrudan işlemci süresini sorgulayın. Bkz. *İşlemci Süresinin Sorgulanması* (sayfa: 541).

Farklı makinalar ve işletim sistemlerinde işlemci zamanının hesabı az ya da çok çılgıncadır. Bunlarda, dahili işlemci saati saniyenin yüzde biri ile milyonda biri arasında bir çözünürlüğe sahiptir.

`int` **`CLOCKS_PER_SEC`** makro

Bu makronun değeri **`clock`** işlevi tarafından ölçülen saniyedeki saat tiki sayısını verir. POSIX standardı bu değerin gerçek çözünürlükten bağımsız olarak bir milyon olmasını gerektirir.

`int` **`CLK_TCK`** makro

Bu, **`CLOCKS_PER_SEC`** değerinin atıl olmuş eşdeğeridir.

`clock_t` veri türü

Bu, **`clock`** işlevinin dönüş değerinin türüdür. **`clock_t`** türünden değerler saat tiklerinin sayısıdır.

`clock_t` **`clock`**`(void)` işlev

Bu işlev çağrıldığı sürecin o anki işlemci zamanı ile döner. İşlemci zamanı ölçülmüyorsa ya da gösterilemiyorsa, **`clock`** işlevi **`clock_t`** cinsinden **`(-1)`** ile döner.

3.2. İşlemci Süresinin Sorgulanması

`times` işlevi bir sürecin harcadığı işlemci süresi ile ilgili bilgileri, *işlemci zamanına* (sayfa: 538) ek olarak **`struct tms`** cinsinden bir nesne olarak döndürür. Bu oluşum `sys/times.h` başlık dosyasında bildirilmiştir.

`struct` **`tms`** veri türü

`tms` yapısı bir sürecin harcadığı işlemci süresi ile ilgili bilgileri döndürmek için kullanılır. En azından şu üyelere sahip olmalıdır:

`clock_t` **`tms_utime`**
: Çağrıldığı sürecin çalıştırdığı komutların harcadığı toplam işlemci süresidir.

`clock_t` **`tms_stime`**
: Çağrıldığı sürece "sistem" tarafından harcanan işlemci süresidir.

`clock_t` **`tms_cutime`**
: Çağrıldığı sürecin sonlandırılmış alt süreçlerinin harcadığı **`tms_utime`** ve **`tms_cutime`** sürelerinin toplamıdır. Sonlandırılmış alt süreçlerin durumu sürece **`wait`** veya **`waitpid`** ile raporlandığından (bkz. *Süreç Tamamlama* (sayfa: 690)), bu değer **`wait`** veya **`waitpid`** ile raporlanmamış süreleri içermez.

`clock_t` **`tms_cstime`**
: Çağrıldığı sürecin sonlandırılmış alt süreçleri için "sistemin" harcadığı işlemci süresini göstermesi dışında **`tms_cutime`** gibidir.

Tüm süreler saat tikleri cinsinden verilmiştir. İşlemci zamanının tersine, bunlar bir olaya göreli olmayan birer süre gösterirler. Bkz. *Bir Sürecin Oluşturulması* (sayfa: 687).

`clock_t` **`times`**`(struct tms *`*tampon*`)` işlev

`times` işlevi kendisini çağıran sürecin işlemci süresi ile ilgili bilgileri *tampon* içinde saklar.

İşlevin dönüş değeri çağrıldığı sürecin işlemci zamanıdır. Başarısızlık durumunda işlev **`(clock_t) (-1)`** ile döner.

Taşınabilirlik Bilgisi

İşlemci Zamanının Sorgulanması (sayfa: 540) bölümünde anlatılan **`clock`** işlevi ISO C standardında belirtilmiştir. GNU sisteminde işlemci zamanı **`times`** tarafından döndürülen yapının **`tms_utime`** ve **`tms_stime`** alanlarındaki değerlerin toplamı olarak tanımlanmıştır.

4. Mutlak Zaman

Bu kısımda *mutlak zaman* (sayfa: 538)'ın hesabı ile ilgili oluşumlara yer verilmiştir.

GNU C kütüphanesinde mutlak zaman üç türlü gösterilir:

- ***Basit zaman*** (**`time_t`** veri türü) gerçeklemeye özgü bir başlangıç zamanına göre geçen sürenin saniye cinsinden bir gösterimidir.
- Bir de ***yüksek çözünürlüklü zaman*** gösterimi vardır. Basit zamandaki gibi, bir başlangıca göre geçen süre olarak gösterilmesine rağmen ölçülen değer saniye cinsinden değil, saniyenin kesirlerini de içeren **`struct timeval`** türünde bir veridir. Basit zamana göre daha yüksek bir çözünürlüğe ihtiyacınız olursa bu zaman gösterimini kullanın.
- ***Yerel zaman*** veya ***bozuk zaman***[10] (**`struct tm`** veri türü) belirli bir zaman dilimi için zaman birimlerinin yıl, ay, gün v.s. olarak belirtildiği bir zaman gösterimidir. Bu zaman gösterimi sadece insanlar arası iletişimde kullanılır.

4.1. Basit Zaman

Bu bölümde mutlak zamanı basit zaman olarak göstermekte kullanılan **`time_t`** veri türü ile basit zaman nesneleri üzerinde işlem yapan işlevlerden bahsedilecektir. Bu oluşumlar `time.h` başlık dosyasında bildirilmiştir.

`time_t` veri türü

Bu basit zaman gösteriminde kullanılan veri türüdür. Bazan, bir süreyi de belirttiği olur. Bir mutlak zaman değeri olarak yorumlandığında, Koordinatlı Evrensel Zamana (UTC ya da GMT denen şey) göre, 1 Ocak 1970, saat 00:00:00'dan beri geçen saniye sayısını gösterir. (Mutlak zaman başlangıcına ingilizcede "epoch" deniyor.[11] POSIX standardının bu sayının artık saniyeleri içermemesini gerektirmesine rağmen bazı sistemlerde **`TZ`** ilgili değere ayarlanırsa artık saniyeler de dahil edilir. (Bkz. *Zaman Diliminin TZ ile Belirtilmesi* (sayfa: 565)).

Basit zamanın yerel zaman dilimleri kavramı ile ilgisi yoktur. Mutlak zamanın, bilgisayarınızın dünyanın neresinde olduğuna bakılmaksızın aynı değeri göstereceğini unutmayın.

GNU C kütüphanesinde, **`time_t`** ile **`long int`** türleri eşdeğerdir. Başka sistemlerde ise, **`time_t`** bir tamsayı tür olabildiği gibi gerçek sayı türlerinden biri ile de eşdeğer olabilir.

`difftime` işlevi iki basit mutlak zaman arasında geçen süreyi verir ve bu sonuç her zaman bir çıkarma işleminin sonucu değildir. Bkz. *Süre* (sayfa: 538).

`time_t` **`time`**`(time_t *`*sonuç*`)` işlev

time işlevi, o anki mutlak zamanı **time_t** türünde bir değer olarak döndürür. *sonuç* bir boş gösterici değilse bu değer ayrıca **sonuç* içinde saklanır. O anki mutlak zaman alınamıyorsa işlev **(time_t) (-1)** ile döner.

`int` **stime**`(time_t *`*yenizaman*`)` işlev

stime sistem saatini ayarlar. *yenizaman*, yukarıda **time_t** tanımında açıklandığı gibi yorumlanarak sistem zamanı bu değere ayarlanır.

settimeofday işlevi ise sistem zamanını bir saniyeden daha hassas bir çözünürlükle ayarlamak için tasarlanmış daha yeni bir işlevdir. **settimeofday** işlevi genellikle **stime** işlevine göre daha iyi bir seçimdir. Bkz. *Yüksek Çözünürlüklü Zaman* (sayfa: 543).

Sistem saatini sadece süper kullanıcı (`root`) ayarlayabilir.

Eğer işlev başarılı olursa sıfır değeri ile döner. Aksi takdirde, **-1** ile döner ve **errno** değişkenine şu değer atanır:

`EPERM`
: Sistem saatini sadece süper kullanıcı ayarlayabilir.

4.2. Yüksek Çözünürlüklü Zaman

time_t veri türü basit zamanları bir saniyelik hassasiyet ile göstermekte kullanılır. Bazı uygulamalarda ise daha yüksek hassasiyet gerekir.

Bu nedenle, GNU C kütüphanesi mutlak zamanı bir saniyeden daha yüksek hassasiyetle hesaplayabilen işlevler de içerir. Bu bölümde bahsi geçen veri türleri ve işlevler `sys/time.h` başlık dosyasında bildirilmiştir.

`struct` **timezone** veri türü

struct timezone yapısı yerel zaman dilimi ile ilgili mümkün olan en az bilgiyi saklamak için kullanılır. Şu üyelere sahiptir:

`int` **tz_minuteswest**
: UTC'nin batısındaki dakika sayısıdır.

`int` **tz_dsttime**
: Sıfırdan farklıysa, yılın belli bir döneminde yaz saati uygulanır.

struct timezone türü artık atıl olmuştur ve asla kullanılmamalıdır. Buradaki bu veri türü ile ilgili oluşumlar yerine, *Zaman Dilimi Değişkenleri ve İşlevleri* (sayfa: 566) bölümünde anlatılan oluşumlar kullanılmalıdır.

```
int gettimeofday(struct timeval  *zaman,                    işlev
                 struct timezone *zamandilimi)
```

gettimeofday işlevi, mutlak zaman başlangıcından beri geçen süreyi *zaman* ile gösterilen **struct timeval** yapısı ile döndürür. Zaman dilimi bilgisi ise *zamandilimi* ile gösterilen yapı içinde döner. *zamandilimi* bir boş gösterici olarak verilmişse, zaman dilimi bilgisi yoksayılır.

Eğer işlev başarılı olursa sıfır değeri ile döner. Aksi takdirde, **-1** ile döner ve **errno** değişkenine şu değer atanır:

`ENOSYS`

İşletim sistemi zaman dilimi bilgisinin alınmasını desteklemiyor. GNU işletim sistemi zaman dilimi gösteriminde 4.3 BSD'nin atıl bir özelliği olan **struct timezone** kullanımını desteklemez. Bunun yerine *Zaman Dilimi Değişkenleri ve İşlevleri* (sayfa: 566) bölümünde açıklanan oluşumları kullanın.

```
int settimeofday(const struct timeval  *zaman,                    işlev
                 const struct timezone *zamandilimi)
```

settimeofday işlevi sistem saatini argümanlarındaki değerlerle ayarlar. **gettimeofday** işlevindeki gibi mutlak zaman, mutlak zaman başlangıcından (epoch) beri geçen süre olarak ifade edilirken zaman dilimi bilgisi de *zamandilimi* bir boş gösterici ise yoksayılır.

settimeofday işlevini kullanırken süper kullanıcı olmak zorundasınız.

Bazı çekirdekler sistem saatini donanım saati gibi bir kaynaktan açılış sırasında ayarlar. Linux gibi diğerleri ise, sistem saatini "geçersiz" bir duruma getirir (saat okunmaya çalışılır ve başarısız olur). Sistem başlatma betikleri ile yapılan bir **stime** çağrısı ile sistem saati geçersiz durumdan çıkarılır.

settimeofday işlevi, sistem saatinin beklenmedik şekilde ileri ya da geri kalmasına ve sistemde çeşitli sorunlar çıkmasına sebep olur. Sistem saatini geçici olarak hızlandırarak ya da yavaşlatarak bir zamandan diğerine yumuşak bir geçiş yapmak için aşağıdaki **adjtime** işlevini kullanın.

Linux çekirdeği ile, **adjtimex** işlevi aynı şeyi yapar, ayrıca sistem saatinin hızında kalıcı değişiklikler yaparak sık sık düzeltme yapma ihtiyacını ortadan kaldırır.

Eğer işlev başarılı olursa sıfır değeri ile döner. Aksi takdirde, **-1** ile döner ve **errno** değişkenine şu değer atanır:

EPERM
: Yetkileri yetersiz olduğundan bu süreç saati ayarlayamaz.

ENOSYS
: *zamandilimi* bir boş gösterici değil ama işletim sistemi zaman dilimi bilgisi ayarını desteklemiyor.

```
int adjtime(const struct timeval *delta,                          işlev
            struct timeval       *eskidelta)
```

Bu işlev sistem saatini aşamalı olarak hızlandırır ya da yavaşlatır. Böyle ayarlanan sistem saati mutlak zamanı düzenli bir artışla gösterir. Saatin ayarını basitçe değiştirirseniz bu böyle olmaz.

delta argümanı bir göreli ayar belirtir. Negatifse belirtilen süre tamamlanıncaya kadar saat yavaşlatılır, pozitifse hızlandırılır.

eskidelta argümanı bir boş gösterici değilse, işlev henüz tamamlanmamış olan ayarın önceki ayar bilgisi ile döner.

Bu işlev genellikle sistem saatini yerel ağın saati ile eşzamanlamak için kullanılır. İşlevi kullanabilmek için yetkili kullanıcı olmalısınız.

Linux çekirdeği ile saat hızını kalıcı olarak değiştirmek için **adjtimex** işlevini kullanabilirsiniz.

Eğer işlev başarılı olursa sıfır değeri ile döner. Aksi takdirde, **-1** ile döner ve **errno** değişkenine şu değer atanır:

EPERM
: Saati ayarlamak için yetkileriniz yetersiz.

Taşınabilirlik Bilgisi

gettimeofday, **settimeofday** ve **adjtime** işlevleri BSD'den alınmıştır.

Aşağıdaki işlevle ilgili semboller `sys/timex.h` başlık dosyasında bildirilmiştir.

`int` **adjtimex**`(struct timex *`*timex*`)` işlev

adjtimex işlevi **ntp_adjtime** işlevi ile aynıdır. Bkz. *Yüksek Doğrulukta Saat* (sayfa: 547).

Bu işlev sadece Linux çekirdeği ile kullanılabilir.

4.3. Yerel Zaman

Mutlak zaman, GNU C kütüphanesi işlevleri tarafından sabit bir mutlak zamanı başlangıç kabul ederek bu zamandan itibaren geçen süre olarak ifade edilir. Bu hesaplama açısından uygun olmakla birlikte, insanları düşündüğü takvim zamanıyla ilişkilendirilmesinin bir yolu yoktur. Tersine olarak ***yerel zaman***, mutlak zamanın yıl, ay, gün, v.s. şeklinde ayrık bir ikilik gösterimidir. Yerel zaman değerleri hesaplama için elverişli olmadığı halde zaman bilgisini insanların okuyabileceği biçimde ifade etmek için elverişlidir.

Bir yerel zaman değeri daima zaman dilimi seçimine göredir ve ayrıca zaman dilimi de belirtilir.

Bu bölümdeki semboller `time.h` başlık dosyasında bildirilmiştir.

`struct` **tm** veri türü

Bu veri türü bir yerel zamanı ifade etmek için kullanılır. Yapı, farklı sıralamayla da olsa en azından aşağıdaki üyeleri içermelidir:

`int` **tm_sec**
Bu üye bir dakikadan küçük tam saniyelerin sayısıdır (normalde **0**'dan **59**'a kadar olmakla birlikte gerçek üst sınır, eğer artık saniye desteği varsa artık saniyeleri de ifade edebilmek için **60** saniyedir).

`int` **tm_min**
Bu üye bir saatten küçük tam dakikaların sayısıdır (**0**'dan **59**'a kadar).

`int` **tm_hour**
Bu üye geceyarısından itibaren geçen tam saatlerin sayısıdır (**0**'dan **23**'e kadar).

`int` **tm_mday**
Bu üye ayın gününü ifade eden bir sayıdır (**1**'den **31**'e kadar). Dikkat edin!, yapıdaki sıradan bir sayı olarak, bu üye yapının kalanı ile bağdaşmaz.

`int` **tm_mon**
Yılın başlangıcından itibaren tam ayların sayısıdır (**0**'dan **11**'e kadar). Dikkat edin!, normalde insanlar ilk ayın (Ocak ayı) numarası olarak 1'i kullanırlar.

`int` **tm_year**
Bu, 1900'den beri geçen tam yılların sayısıdır.

`int` **tm_wday**
Bu, hafta başından (Pazar) beri geçen tam günlerin sayısıdır (**0**'dan **6**'ya kadar).

`int` **tm_yday**
Bu, yılın başından beri geçen tam gün sayısıdır (**0**'dan**365**'e kadar).

`int` **tm_isdst**

Bu yaz saati uygulamasının geçerli olup olmadığını gösterir. Değer pozitifse uygulanacak, sıfırsa uygulanmayacak, negatifse hiç yaz saati uygulaması yapılmıyor demektir.

`long int` **`tm_gmtoff`**
Yerel zamanı hesaplamakta kullanılan zaman dilimini belirtir. Yaz saati uygulamasını da içererek, UTC'ye göre saniye sayısını içerir. Bu üyenin değerini UTC'nin doğusundaki saniye sayısı olarak da düşünebilirsiniz. Örneğin, Türkiye için bu değer normalde **`2*60*60`**, yaz saati uygulaması yapılan dönemde ise **`3*60*60`** saniyedir. Yapının **`tm_gmtoff`** üyesi BSD'den alınmıştır, bir GNU kütüphanesi oluşumudur; kesin ISO C uyumluluğu istenen ortamda görülmez.

`const char *` **`tm_zone`**
Yerel zamanı hesaplamakta kullanılan zaman diliminin ismidir. Yapının bu üyesi de BSD'den alınmıştır, bir GNU kütüphanesi oluşumudur; kesin ISO C uyumluluğu istenen ortamda görülmez.

`struct tm *`**`localtime`**`(const time_t *`*`zaman`*`)` işlev

`localtime` işlevi, *zaman* ile belirtilen basit zamanı, kullanıcının belirtilmiş zaman dilimine göre ifade edilen yerel zaman gösterimine dönüştürür.

Dönüş değeri bir durağan yerel zaman yapısına bir göstericidir ve yapı içeriği sonraki **`ctime`**, **`gmtime`** veya **`localtime`** çağrılarıyla değişebilecektir. (Kütüphanede bu nesnenin içeriğini değiştirebilen başka işlev yoktur.)

zaman, bir yerel zaman olarak ifade edilemiyorsa dönüş değeri bir boş göstericidir; bu, genellikle yıl değeri **`int`** türüne sığmazsa ortaya çıkar.

`localtime` çağrısı bir etkiye daha sahiptir: **`tzname`** değişkenine o anki zaman dilimi bilgisi ile ilintili bir değer atar. Bkz. *Zaman Dilimi Değişkenleri ve İşlevleri* (sayfa: 566).

`localtime` işlevi çok evreli yazılımlarda büyük bir soruna yolaçar. Sonuç bir durağan tamponda döndüğünden bu tüm evreler tarafından kullanılamaz. POSIX.1c'de bu işlevin çok evreli kullanımına yönelik bir benzerinden bahsedilir.

`struct tm *`**`localtime_r`**`(const time_t *`*`zaman`*`,`
`struct tm *`*`sonuç`*`)` işlev

The **`localtime_r`** işlevi, **`localtime`** işlevi gibi çalışır. Bir basit zaman içeren bir değişkene gösterici alır ve onu yerel zaman gösterimine dönüştürür.

Ancak, sonuç durağan bir tampona yerleştirilmez. *sonuç* parametresi ile gösterilen **`struct tm`** türündeki bir nesneye yerleştirilir.

Dönüşüm başarılı olursa işlev, sonucun yazıldığı nesneye bir gösterici ile, yani *sonuç* ile döner.

`struct tm *`**`gmtime`**`(const time_t *`*`zaman`*`)` işlev

Bu işlev, yerel zamanı bir yerel zaman diliminden ziyade UTC'ye uyarlaması dışında **`localtime`** gibidir.

`localtime` işlevinde olduğu gibi sonucun bir durağan değişkene atanması sebebiyle bu işlev de sorunludur. POSIX.1c **`gmtime`** işlevi yerine sorunsuz kullanılabilen bir benzerinden bahseder.

`struct tm *`**`gmtime_r`**`(const time_t *`*`zaman`*`,`
`struct tm *`*`sonuç`*`)` işlev

Bu işlev **`localtime_r`** işlevine benzemekle birlikte, **`gmtime`** gibi zamanı UTC'ye göre verilmiş gibi dönüştürür.

Dönüşüm başarılı olursa işlev, sonucun yazıldığı nesneye bir gösterici ile, yani *sonuç* ile döner.

`time_t` **`mktime`**`(struct tm *`*yerelzaman*`)` işlev

`mktime` işlevi, bir yerel zaman yapısını basit zaman gösterimine dönüştürmekte kullanılır. Ayrıca yerel zaman yapısının içeriğini, haftanın gününü ve yılın gününü diğer tarih ve zaman elemanlarına göre doldurarak "normalleştirir".

`mktime` işlevi yapının **`tm_wday`** ve **`tm_yday`** üyelerini yoksayar. Mutlak zamanı saptamak için yapının diğer üyelerini kullanır. Bu nedenl yapının yoksayılan üyelerinde normaldışı değerlerin bulunması sorun oluşturmaz. **`mktime`** işlevi ayrıca *yerelzaman* yapısının elemanlarını da (**`tm_wday`** ve **`tm_yday`** dahil) ayarlar.

Belirtilen yerel zaman bir basit zaman olarak gösterilemiyorsa işlev, **`(time_t) (-1)`** değeri ile döner ve *yerelzaman* ile gösterilen yapının içeriğine dokunmaz.

`mktime` çağrısı ayrıca, **`tzname`** değişkenine o anki zaman dilimi bilgisi ile ilintili bir değer atar. Bkz. *Zaman Dilimi Değişkenleri ve İşlevleri* (sayfa: 566).

`time_t` **`timelocal`**`(struct tm *`*yerelzaman*`)` işlev

`timelocal` işlevi **`mktime`** ile aynıdır, sadece, **`localtime`** işlevinin yaptığının tersini yaptığından ismi daha kolay hatırlanacak şekilde seçilmiştir.

Taşınabilirlik Bilgisi

`mktime` işlevi özellikle evrensel anlamda geçerli olmakla birlikte **`timelocal`** daha az yaygındır.

`time_t` **`timegm`**`(struct tm *`*yerelzaman*`)` işlev

`timegm` işlevi girdi olarak yerel zaman dilimine bakılmaksızın UTC'ye göre bir değer alması dışında **`mktime`** işlevi gibidir.

`timegm` işlevi, **`gmtime`** işlevinin yaptığının tersini yapar.

Taşınabilirlik Bilgisi

`mktime` işlevi özellikle evrensel anlamda geçerli olmakla birlikte **`timegm`** daha az yaygındır. UTC zamanından basit zamana dönüşüm yapan çoğu taşınabilir işlev **`TZ`** ortam değişkenine UTC atar, **`mktime`** işlevi ise değişkene yerel zaman dilimini atar.

4.4. Yüksek Doğrulukta Saat

`ntp_gettime` ve **`ntp_adjtime`** işlevleri doğruluğunu arttırmak amacıyla sistem saatini izlemek ve değiştirmek için bir arayüz sağlar. Örneğin, saat hızını daha iyi bir ayar için hızlandırabilir ya da başka bir kaynaktan saati ayarlayabilirsiniz.

Bu işlevler özellikle çok sayıda sistemin saatini yüksek hassasiyetli bir saat kullanarak ayarlamak için ağ zaman protokolü (NTP – Network Time Protocol) ile gerçeklenmiş sunucular tarafından kullanılır.

Bu işlevler `sys/timex.h` başlık dosyasında bildirilmiştir.

`struct` **`ntptimeval`** veri türü

Bu yapı sistem saati ile ilgili bilgiler için kullanılır. Şu üyelere sahiptir:

`struct` **`timeval time`**
: Mutlak zaman başlangıcından (epoch) itibaren geçen süre olarak ifade edilen, o anki mutlak zamandır. **`struct timeval`** veri türü *Süre* (sayfa: 538) bölümünde açıklanmıştır.

`long int` **`maxerror`**
: Mikrosaniyeler cinsinden ölçülen en büyük hata miktarıdır. **`ntp_adjtime`** ile düzenli aralıklarla güncellenmedikçe, bu değer platforma özel en büyük değere ulaşacaktır.

`long int` **`esterror`**
: Mikrosaniyeler cinsinde ölçülen tahmini hatadır. Bu değer gerçek mutlak zamandaki sistem saatinin tahmini ayar noktasını belirtmek için **`ntp_adjtime`** tarafından ayarlanabilir.

`int` **`ntp_gettime`**`(struct ntptimeval *`*`tptr`*`)` işlev

`ntp_gettime` işlevi *tptr* tarafından gösterilen yapıyı o anki değerlerle doldurur. Bundan sonra yapının elemanları çekirdeğin kabul ettiği zamanlayıcı gerçeklemesinin değerlerini içerir. Böyle değilse, bir **`ntp_adjtime`** çağrısı gerekir.

Başarı durumunda dönüş değeri sıfır, aksi takdirde sıfırdan farklıdır. Aşağıdaki **`errno`** değeri bu işlev için atanmıştır:

`TIME_ERROR`
: Hassas saat modeli şu an için düzgün olarak ayarlanamadı; saat eşzamanlanmış kabul edilemediğinden bu değerleri dikkatli kullanmalısınız.

`struct` **`timex`** veri türü

Bu yapı sistem saatini izlemek ve denetlemek için kullanılır. Şu üyeleri içerir:

`unsigned int`**`modes`**
: Hangi kiplerin geçerli olduğunu belirtir. Etkin kip belirtmek için çeşitli sembolik sabitler "ikil veya"lanarak birleştirilebilir. Bu sabitler **`MOD_`** ile başlar.

`long int` **`offset`**
: Gerçek mutlak zamandaki sistem saatinin o anki ayar noktasını belirtir. Değer mikrosaniye cinsindendir. **`modes`** üyesinde **`MOD_OFFSET`** biti varsa ayar noktası (ve olası başka bağımlı değerler) atanmış olabilir. Ayar noktasının mutlak değeri **`MAXPHASE`**'dan büyük olmamalıdır.

`long int` **`frequency`**
: Gerçek mutlak zamanla sistem saati arasındaki farkı frekans olarak belirtir. Değer bir PPM oranıdır (milyonda birlik değerler, 0.0001%). Oranlama **`1 << SHIFT_USEC`**'dir. Değer, **`MOD_FREQUENCY`** biti ile atanabilir, fakat **`MAXFREQ`** değerinden büyük olamaz.

`long int` **`maxerror`**
: Mikrosaniyeler cinsinden ölçülen en büyük hata miktarıdır. Yeni bir değer **`MOD_MAXERROR`** biti kullanılarak atanabilir. **`ntp_adjtime`** ile düzenli aralıklarla güncellenmedikçe, bu değer platforma özel en büyük değere ulaşacaktır.

`long int` **`esterror`**
: Mikrosaniyeler cinsinden ölçülen tahmini hata miktarıdır. Yeni bir değer **`MOD_ESTERROR`** biti kullanılarak atanabilir.

`int` **`status`**

Bu üye, saat çarkının çeşitli durumlarını gösterir. Bunlar önemli bitlerle ifade edilen sembolik sabitlerdir ve **STA_** ile başlarlar. Bu bitlerin bazıları **MOD_STATUS** biti kullanılarak güncellenebilir.

`long int` **constant**
Çekirdek içinde gerçeklenmiş PLL'in (phase locked loop – faz kilitlemeli çevrim) değişmezliğini ya da band genişliğini ifade eder. Bu değer **MOD_TIMECONST** biti kullanılarak değiştirilebilir.

`long int` **precision**
Sistem saati okumasındaki en büyük hatayı ya da doğruluğu ifade eder. Değer mikrosaniye cinsindendir.

`long int` **tolerance**
Sistem saatindeki en büyük frekans hatasını milyonda birlik değerler (PPM) olarak ifade eder. Bu değer **maxerror**'ü her saniyede bir arttırmakta kullanılır.

`struct timeval` **time**
O anki mutlak zamanı gösterir.

`long int` **tick**
Mikrosaniye cinsinden saat tikleri arasındaki süredir. Bir saat tiki sistem saatinin temel aldığı sürekli tekrarlanan bir zamanlayıcı kesmesidir.

`long int` **ppsfreq**
Sistem saatinin zapturapt altına alınması için saniyede bir darbelik (PPS) bir sinyalin kullanılması durumunda anlamlı olan bir kaç isteğe bağlı üyeden ilkidir. Değer milyonda birlik (PPM) bir oran olarak, sistem saati ile PPS sinyali arasındaki frekans farkını belirtir.

`long int` **jitter**
PPS sinyalindeki saçılmanın ortasına göre mikrosaniye cinsinden bir ortalamasıdır.

`int` **shift**
PPS kalibrasyon aralığının **PPS_SHIFT**'den **PPS_SHIFTMAX**'a kadar ikilik üstel değeridir.

`long int` **stabil**
PPS sinyalindeki ortasına göre saçılmanın milyonda birlik (PPM) oranıdır.

`long int` **jitcnt**
Seğirmenin (jitter) izin verilen en büyük değer olan **MAXTIME**'ı aştığında darbe sayısını gösteren bir sayaçtır.

`long int` **calcnt**
Başarılı kalibrasyon aralıklarının sayısını gösteren bir sayaçtır.

`long int` **errcnt**
Kalibrasyon hatalarının sayısını gösteren bir sayaçtır (hatalar çok geniş ayar aralıklarından ya da seğirmelerden kaynaklanır).

`long int` **stbcnt**
Kararlılık eşiği aşıldığında yapılan kalibrasyonların sayısını gösteren bir sayaçtır.

`int` **ntp_adjtime**`(struct timex *`*tptr*`)` işlev

ntp_adjtime işlevi *tptr* ile gösterilen yapıyı o anki değerlerle doldurur.

Ek olarak, **ntp_adjtime** işlevi *tptr içinde aktardığınız değerlerle eşleşen bazı ayarları günceller. Güncellenecek ayarları belirtmek için *tptr'nin **modes** elemanı kullanılır. Bu yolla, **offset**, **freq**, **maxerror**, **esterror**, **status**, **constant** ve **tick** değerlerini güncelleyebilirsiniz.

modes= sıfır ise hiçbir şey yapılmaz.

Yalnız süper kullanıcı bu ayarları güncelleyebilir.

Başarı durumunda dönüş değeri sıfır, aksi takdirde sıfırdan farklıdır. Aşağıdaki **errno** değerleri bu işlev için atanmıştır:

`TIME_ERROR`
: Hassas saat modeli şu an için düzgün olarak ayarlanamadı; saat eşzamanlanmış kabul edilemediğinden bu değerleri dikkatli kullanmalısınız. Başka bir sebep de izin verilmediği halde yeni değerlerin belirtilmesi olabilirdi.

`EPERM`
: Süreç bir ayarın güncellenmesini belirtiyor ama süper kullanıcı değil.

Daha ayrıntılı bilgi için RFC1305 (Network Time Protocol, Version 3 [Ağ Zaman Protokolü, 3. sürüm]) ve onunla ilgili belgelere bakınız.

Taşınabilirlik Bilgisi

GNU C kütüphanesinin eski sürümlerinde bu işlev yoktu ama eşanlamlısı olan **adjtimex** işlevi vardı.

4.5. Zaman Değerlerinin Biçimlenmesi

Bu bölümdeki mutlak zaman değerlerini dizgeler dönüştürerek biçimlendirmekte kullanılan işlevlerden bahsedilecektir. Bu işlevler `time.h` başlık dosyasında bildirilmiştir.

```
char *asctime(const struct tm *yerelzaman)                                  işlev
```

asctime işlevi *yerelzaman* ile gösterilen yerel zaman değerini

```
"Sat Aug 28 13:49:43 2004\n"
```

standart biçimine dönüştürür. Bu işlevde kullanılan gün isimlerinin kısaltmaları: **Sun**, **Mon**, **Tue**, **Wed**, **Thu**, **Fri** ve **Sat**.

Ay isimlerinin kısaltmaları: **Jan**, **Feb**, **Mar**, **Apr**, **May**, **Jun**, **Jul**, **Aug**, **Sep**, **Oct**, **Nov** ve **Dec**.

Dönüş değeri bir durağan ayrılmış dizgeye göstericidir ve bu değer sonraki **asctime** veya **ctime** işlevleri ile değiştirilebilir. (Kütüphanede bu değeri değiştiren başka işlev yoktur.)

```
char *asctime_r(const struct tm *yerelzaman,                                işlev
                char            *tampon)
```

Bu işlev **asctime** işlevine benzemekle birlikte, sonucu bir durağan ayrılmış tampona değil, *tampon* ile gösterilen tampona yazar. Bu tampon sonlandırıcı boş karakter dahil en az 26 baytlık bir yere sahip olmalıdır.

Bir hata oluşmazsa işlev sonucun yazıldığı tampona bir gösterici ile yani *tampon* ile döner. Aksi takdirde **NULL** döner.

```
char *ctime(const time_t *zaman)                                    işlev
```

ctime işlevi argüman olarak bir yerel zaman değeri yerine basit zaman değeri alması dışında **asctime** işlevi gibidir.

```
asctime (localtime (zaman))
```

ile eşdeğerdir. **localtime** işlevi **tzname** değerini atadığından **ctime** işlevi de aynısını yapar. Bkz. *Zaman Dilimi Değişkenleri ve İşlevleri* (sayfa: 566).

```
char *ctime_r(const time_t *zaman,                                  işlev
              char         *tampon)
```

Bu işlev **ctime** işlevine benzemekle birlikte, sonucu bir durağan ayrılmış tampona değil, *tampon* ile gösterilen tampona yazar.

```
({ struct tm tm; asctime_r (localtime_r (time, &tm), buf); })
```

ile eşdeğerdir (GCC oluşumları kullanılarak yazılmıştır). Bir hata oluşmazsa işlev sonucun yazıldığı tampona bir gösterici ile yani *tampon* ile döner. Aksi takdirde **NULL** döner.

```
size_t strftime(char             *dizge,                            işlev
                size_t            boyut,
                const char       *şablon,
                const struct tm  *yerelzaman)
```

Bu işlev **sprintf** işlevine benzer (bkz. *Biçimli Girdi* (sayfa: 277)), fakat *yerelzaman* içindeki tarih ve saatin basılması için *şablon* biçim dizgesi *o an belirtilmiş olan yerele özgü* (sayfa: 164) zaman dönüşümlerinin yapılabilmesi için özelleştirilmiştir.

şablon dizgesindeki belirteç olmayan karakterler *dizge* çıktı dizgesine oldukları gibi kopyalanırlar; Bu dizge çokbaytlı karakterleri de içerebilir. Dönüşüm belirteçleri bir `%` karakteri ile başlar ve bunu isteğe bağlı bir seçenek izler. Bu seçeneklerin hepsi GNU oluşumudur. İlk üçü sadece sayıların çıktılanması ile ilgilidir:

`_`
: Sayılar için ayrılan genişlikteki boş yerler boşlukla doldurulur.

`-`
: Sayılar için ayrılan genişlikteki boş yerler hiçbir şeyle doldurulmaz.

`0`
: Sayılar için ayrılan genişlikteki boş yerlerin boşlukla doldurulması belirtilmiş olsa bile sıfırlarla doldurulur.

`^`
: Çıktıda büyük harfler kullanılır, fakat bu sadece mümkün olabiliyorsa yapılır (Bkz. *Büyük–Küçük Harf Dönüşümleri* (sayfa: 84)).

Öntanımlı eylem sayılar için ayrılan genişliğin sıfırlarla doldurulmasıdır. Bir aralık dahilinde değişmeyen, dolayısıyla doğal bir genişliği olmayan sayılar için dolgu yapılmaz.

Bundan sonra, genişlik belirtilmeyi mümkün kılan isteğe bağlı bir genişlik belirtimi gelir. Bu ondalık sayı gösterimi belirtir. Çıktılanan sayının karakter sayısı belirtilen genişlikten azsa sonuç sağa yanaştırılarak yazılırken soldaki alan boşluklarla doldurulur.

Seçenek ve genişlik belirtiminden sonra isteğe bağlı bir değiştirici gelebilir. Değiştiriciler ilk olarak, POSIX.2–1992 ve ISO C99 standartları ile standartlaştırılmıştır:

E

Tarih ve saat gösterimi için yerele özgü diğer gösterim kullanılır. Bu değiştirici **%c**, **%C**, **%x**, **%X**, **%y** ve **%Y** biçim belirteçlerine uygulanır. Japon yerelinde örneğin, **%Ex** Japon İmparator'unun hüküm sürdüğü çağa göre biçimlenmiş tarih olabilir.

O

Sayılar için yerele özgü diğer sayısal semboller kullanılır. Bu değiştirici sadece sayısal biçim belirteçlerine uygulanır.

Biçimde değiştirici desteği varsa ama yerelde diğer bir gösterim yoksa değiştirici yoksayılır.

Dönüşüm belirteçleri biçim belirteçlerinden biri ile biter. Aşağıda **%** ile başlayan belirtimler ve çıktı dizgesinde nasıl yorumlandıkları liste halinde verilmiştir:

%a

Yerele özgü kısaltılmış gün ismi.

%A

Yerele özgü gün ismi.

%b

Yerele özgü kısaltılmış ay ismi.

%B

Yerele özgü ay ismi.

%B'nin **%d** ile birlikte kullanımı bazı yerellerde imla kurallarına uygun olmayan sonuçlar üretir.

%c

Yerele özgü tarih ve saat gösterimi.

%C

Yılın yüzyılllık tam parçası (yüzyıl değeri değil). Yılın 100'e bölünmesi ile elde edilen sonucun tamsayı kısmıdır.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

%d

01 ile **31** arasında bir değer olarak ayın gün numarası.

%D

%m/%d/%y biçiminde tarih.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

%e

1 ile **31** arasında bir değer olarak ayın gün numarası (tek rakamlı sayıların önüne 0 konmaz).

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

%F

%Y-%m-%d biçimimde tarih. Bu biçim ISO 8601 standardında belirtilmiştir ve çok tercih edilen bir biçimdir.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

%g

00 ile **99** arasında yılın yüzyıllık parçası olmaksızın yıl numarasıdır (örneğin, 1999 için 99, 2004 için 04). **%y** ile aynı biçim ve değerdedir, ancak ilk ve son ISO hafta numarası (bakınız **%V**) önceki ya da sonraki hangi yıla karşılıksa o yıla karşılık olan değer gösterilir.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

%G

%Y ile aynı biçim ve değerdedir, ancak ilk ve son ISO hafta numarası (bakınız **%V**) önceki ya da sonraki hangi yıla karşılıksa o yıla karşılık olan değer gösterilir.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır. Fakat bir GNU oluşumu olarak evvelce de vardı.

%h

%b ile aynıdır ve yerele özgü kısaltılmış ay ismidir.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

%H

00 ile **23** arasında bir sayısal değer olarak 24 saatlik saat gösterimi.

%I

01 ile **12** arasında bir sayısal değer olarak 12 saatlik saat gösterimi.

%j

001 ile **366** arasında yılın gün numarası.

%k

0 ile **23** arasında bir sayısal değer olarak 24 saatlik saat gösterimi (tek rakamlı sayıların önüne 0 konmaz).

Bu biçim bir GNU oluşumudur.

%l

1 ile **12** arasında bir sayısal değer olarak 12 saatlik saat gösterimi (tek rakamlı sayıların önüne 0 konmaz).

Bu biçim bir GNU oluşumudur.

%m

01 ile **12** arasında ayın numarası.

%M

00 ile **59** arasında dakika değeri.

%n

Tek bir **\n** (satırsonu) karakteri.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

%p

Verilen değere bağlı olarak yerele özgü **AM** veya **PM** dizgesidir. Öğleden önceki saatler için **AM** ile öğleden sonraki saatler için **PM** karşılığı olan yerele özgü dizge basılır. Bu dizgeleri desteklemeyen yerellerde **"%p"** bir boş dizgeye karşılıktır.

%P

Verilen değere bağlı olarak yerele özgü `am` veya `pm` dizgesidir. Öğleden önceki saatler için `am` ile öğleden sonraki saatler için `pm` karşılığı olan yerele özgü dizge basılır. Bu dizgeleri desteklemeyen yerellerde `"%p"` bir boş dizgeye karşılıktır.

Bu biçim bir GNU oluşumudur.

`%r`

Yerele özgü AM/PM biçimli saat gösterimi (örn, 04:38:53 ÖS)

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır. POSIX yerelinde, bu biçim `%I:%M:%S %p` biçim dizgesine eşdeğerdir.

`%R`

`%H:%M` biçiminde saat ve dakika.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır. Fakat bir GNU oluşumu olarak evvelce de vardı.

`%s`

Mutlak zaman başlangıcından (epoch) yani 1970–01–01 00:00:00 UTC'den beri geçen saniye sayısı. Artık saniye desteği yoksa artık saniyeler hesaba katılmaz.

Bu biçim bir GNU oluşumudur.

`%S`

`00` ile `60` arasında saniye.

`%t`

Tek bir `\t` (sekme) karateri.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

`%T`

`%H:%M:%S` biçiminde saat.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

`%u`

`1` ile `7` arasında haftadaki günün numarası. Pazartesi, haftanın birinci günüdür.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

`%U`

`00` ile `53` arasında hafta numarası. Yılın ilk Pazar günü ile başlayan hafta yılın ilk haftası sayılır ve `00` ile gösterilir.

`%V`

`01` ile `53` arasında ISO 8601:1988 hafta numarası. ISO haftaları Pazartesi günü başlar, Pazar günü biter. Yılın `01`. haftası, günlerinin çoğunluğu yeni yıl içinde kalan ilk haftadır; örneğin, yılın ilk günü Perşembe ve son günü 4 Ocak olan hafta `01`. haftadır (7 günden 4'ü yeni yıl içinde). Yani, yılın `01`. haftası bir önceki yıldan da günler içerebileceği gibi bunun tersine `01`. haftadan önceki hafta yeni yıldan bir kaç gün içerse bile yılın son haftası (`52`. veya `53`. hafta) olabilir.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır.

`%w`

`0` ile `6` arasında haftadaki günün numarası. Pazar, haftanın sıfırıncı günüdür.

`%W`

00 ile **53** arasında hafta numarası. Yılın ilk Pazartesi günü ile başlayan hafta yılın ilk haftası sayılır ve **00** ile gösterilir.

`%x`

Yerele özgü kısa tarih gösterimi.

`%X`

Yerele özgü saat gösterimi.

`%y`

00 ile **99** arasında yılın yüzyıllık parçası olmaksızın yıl numarasıdır (örneğin, 1999 için 99, 2004 için 04).

`%Y`

Gregoryen takvimine göre yıl. **1**. yıldan önceki yıllar **0**, **-1**, **-2** diye gider.

`%z`

RFC 822/ISO 8601:1988 tarzı zaman dilimi (örn, **+0300** veya **-0600**); zaman dilimi saptanamazsa hiçbir şey basılmaz.

Bu biçim ilk olarak POSIX.2–1992 ve ISO C99 tarafından standartlaştırılmıştır. Fakat bir GNU oluşumu olarak evvelce de vardı.

POSIX yerelinde, bir RFC 822 tam tarih ve saat **"%a, %d %b %Y %H:%M:%S %z"** (veya **"%a, %d %b %Y %T %z"** eşdeğeri) biçimindedir.

`%Z`

Zaman dilimi kısaltması (örn, EEST); saptanamazsa boş dizge.

`%%`

Tek bir **%** karakteri.

boyut parametresi *dizge* dizisi içinde sonlandırıcı boş karakter dahil saklanabilecek karakterlerin sayısı kadar ya da daha büyük olmalıdır. Biçimlenen zaman dizgesi *boyut* karakterden fazla ise **strftime** sıfır ile döner ve *dizge* dizisi tanımsız bırakılır. Aksi takdirde işlev *dizge* dizisine yerleştirilen karakter sayısı ile döner, bu sayıya sonlandırıcı boş karakter dahil değildir.

Uyarı

Dönüş değeri için ISO C'de açıklanan bu uzlaşım bazı durumlarda sorunlara yol açabilir. Bazı yerellerde bazı biçim dizgeleri gerçekten de boş dizge çıktılar ve bu durum dönüş değerine bakarak saptanamaz. Örneğin çoğu yerel AM/PM biçimini kullanmaz (çoğu ülkede 24 saatlik biçim kullanılır) ve bu yerellerde boş dizge çıktılanır, bu durumda işlevin dönüş değeri sıfır olur. Buna benzer bir durumu saptamak için şöyle bir kod kullanılmalıdır:

```
buf[0] = '\1';
len = strftime (buf, bufsize, format, tp);
if (len == 0 && buf[0] != '\0')
  {
    /* strftime çağrısında birşeyler yanlış gitmiş.  */
    ...
  }
```

dizge bir boş gösterici ise, **strftime** hiçbir yazma işlemi yapmaz, ama *dizge*'ye yazılmış gibi karakterlerin sayısı ile döner.

POSIX.1 gereğince her **strftime** çağrısı bir **tzset** çağrısına sebep olur. Bu durumda, herhangi bir çıktı üretilmeden önce **TZ** ortam değişkeninin içeriği incelenebilir.

Bir **strftime** örneği *Zaman İşlevleri Örneği* (sayfa: 567) bölümünde verilmiştir.

```
size_t wcsftime(wchar_t          *dizge,                                    işlev
                size_t            boyut,
                const wchar_t    *şablon,
                const struct tm  *yerelzaman)
```

wcsftime işlevi geniş karakter dizgeleriyle çalışması dışında **strftime** işleviyle aynıdır. Sonucun saklandığı *dizge* bir geniş karakter dizisi olmalıdır. *boyut* parametresi ile bayt sayısı değil, geniş karakter sayısı belirtilir.

Ayrıca *şablon* biçim dizgesi de bir geniş karakterli dizgedir. Biçim dizgesindeki tüm karakterlerin temel karakter kümesindeki karakterler olması gerektiğinden taşınabilirlik açısından C kaynak kodunda biçim dizgesi **L"…"** sözdizimi ile yazılmalıdır. *yerelzaman* parametresi **strftime** çağrısındaki ile aynı anlamdadır.

wcsftime işlevi **strftime** işlevinin desteklediği tüm seçenekleri, değiştiricileri ve biçim belirteçlerini destekler.

wcsftime işlevinin dönüş değeri *dizge* dizisine saklanan geniş karakterlerin sayısıdır. *dizge* çıktılanacak karakterler bakımından yetersizse işlev sıfır ile döner, ayrıca **strftime** işlevinin açıklamasında bahsedilen sorunlar burada da geçerlidir.

4.6. Tarih ve Saatin Yerel Zamana Dönüştürülmesi

ISO C standardı **strftime** işlevinin çıktısının, girdisi olan ikilik biçime dönüştürülmesi için herhangi bir işlev belirtmemiştir. Bu, ilk yıllarda farklı arayüzlerin az ya da çok başarılı çeşitli gerçeklemeler geliştirmeleri ile sonuçlanmıştır. Bunların ardından Unix standardı iki ek işlev tanımladı: **strptime** ve **getdate**. Her ikisi de tuhaf arayüzlere sahip olmasına karşın geniş kullanım alanı bulmuştur.

4.6.1. Düşük Seviyede Çözümleme

İlk işlev oldukça düşük seviyededir. Buna rağmen, iyi bilindiğinden yazılım geliştirirken sıklıkla kullnılır. Arayüzü ve gerçeklenmesi ağırlıkla, **strptime** çağrısı kurallarıyla tanımlanmış ve gerçeklenmiş olan **getdate** işlevinden etkilenmiştir.

```
char *strptime(const char *dizge,                                           işlev
               const char *biçim,
               struct tm  *zaman)
```

strptime işlevi *dizge* dizgesini *biçim* dizgesine göre çözümleyerek sonucu *zaman* yapısına yazar.

Girdi dizgesi **strftime** işleviyle ya da başka bir yöntemle üretilmiş olabilir. Girdi dizgesinin içeriğinin insanlar açısından anlamlı olması gerekmez, *biçim* dizgesi ile eşleşebilen herhangi bir dizge olabilir (örn, **"02:1999:9"**).

Kullanıcı ne yaptığını bilmelidir, yoksa girdi saçma sapan çözümlenebilir. Örneğin **"1999112"** dizgesi **"%Y%m%d"** biçimi kullanılarak 1999–1–12, 1999–11–2 ve hatta 19991–1–2 olarak çözümlenebilir. Kararlı sonuçlar almak için girdi dizgesine uygun ayraçların eklenmesi gerekir.

Biçim dizgesi, **strftime** işlevinin biçim dizgesinde kullanılan elemanlarla oluşturulur. Tek farkla; **_**, **-**, **0** ve **^** seçeneklerine izin verilmez. Girdide harf büyüklükleri gibi farklara bakılmadığından **strftime**

işlevinin bazı belirli biçimleri bile **strptime**'da aynı işi yapar. İki işlev arasanıda bakışımın sağlanması için tüm biçimler desteklenmiştir.

E ve **O** değiştiricilerine **strftime** işlevinin kullanılmasına izin verdiği her yerde izin verilmiştir.

Biçimler şunlardır:

`%a`
`%A`
: Yerele özgü kısaltılmış ya da tam gün ismi.

`%b`
`%B`
`%h`
: Yerele özgü kısaltılmış ya da tam ay ismi.

`%c`
: Yerele özgü tarih ve saat gösterimi.

`%Ec`
: Yerele özgü diğer tarih ve saat biçimi olması dışında **%c** ile aynıdır.

`%C`
: Yılın yüzyıllık parçası.

  Bu biçim, biçim dizgesi ayrıca bir **%y** biçimi içeriyorsa anlamlı olarak çözümlenir.

`%EC`
: Yerele özgü dönem gösterimi.

  %C'den farklı olarak, bazı kültürlerde yıllar, Gregoryen yıllarına göreli bir yılı başlangıç yılı alarak gösterilebilmektedir.

`%d`
`%e`
: 1 ile 31 arasında bir değer olarak ayın gün numarası (tek rakamlı sayıların önüne 0 konulsa da olur konulmasa da).

`%Od`
`%Oe`
: Yerele özgü diğer sayısal sembollerin kullanılması dışında **%d** ile aynıdır.

  Tek rakamlı sayıların önüne 0 konulsa da olur konulmasa da.

`%D`
: **%m/%d/%y** biçimine eşdeğerdir.

`%F`
: ISO 8601 tarih biçimi olan **%Y-%m-%d** ile eşdeğerdir.

  strftime'da ISO C99 oluşumu olan bir GNU oluşumudur.

`%g`
: **00** ile **99** arasında yılın yüzyıllık parçası olmaksızın ISO hafta numarasına bağlı yıl numarasıdır.

Bilgi

Şimdilik tam olarak gerçeklenmemiştir. Biçim olarak tanınır ama *tm* yapısında ilgili alana birşey yazılmaz.

strftime'da da GNU oluşumu olan bir GNU oluşumudur.

`%G`

ISO hafta numarasına göre yıl.

Bilgi

Şimdilik tam olarak gerçeklenmemiştir. Biçim olarak tanınır ama *tm* yapısında ilgili alana birşey yazılmaz.

strftime'da da GNU oluşumu olan bir GNU oluşumudur.

`%H`
`%k`

00 ile **23** arasında bir sayısal değer olarak 24 saatlik saat gösterimi.

%k, **strftime**'da da GNU oluşumu olan bir GNU oluşumudur.

`%OH`

Yerele özgü diğer sayısal sembollerin kullanılması dışında **%H** ile aynıdır.

`%I`
`%l`

01 ile **12** arasında bir sayısal değer olarak 12 saatlik saat gösterimi.

%l, **strftime**'da da GNU oluşumu olan bir GNU oluşumudur.

`%OI`

Yerele özgü diğer sayısal sembollerin kullanılması dışında **%I** ile aynıdır.

`%j`

1 ile **366** arasında yılın gün numarası.

Başa 0 konulsa da olur konulmasa da.

`%m`

1'den **12**'ye kadar ay numarası.

Başa 0 konulsa da olur konulmasa da.

`%Om`

Yerele özgü diğer sayısal sembollerin kullanılması dışında **%m** ile aynıdır.

`%M`

0'dan **59**'a kadar dakika.

Başa 0 konulsa da olur konulmasa da.

`%OM`

Yerele özgü diğer sayısal sembollerin kullanılması dışında **%M** ile aynıdır.

`%n`
`%t`

Boşluklarla eşleşirler.

`%p`
`%P`

Yerele özgü **AM** veya **PM** dizgesi.

Bu biçim, **%I** veya **%l** ayrıca kullanılmamışsa kullanışsızdır. Bu değerlerin atanmamış olduğu yerellerde başka sorunlara da yol açar ve bu nedenle dönüşüm başarısız olur.

%P strftime'da da GNU oluşumu olan bir GNU oluşumudur..

`%r`

The complete time using the AM/PM format of the current locale.

A complication is that the locale might not define this format at all and therefore the conversion fails.

`%R`

The hour and minute in decimal numbers using the format **%H:%M**.

%R, **strftime**'da da GNU oluşumu olan bir GNU oluşumudur.

`%s`

Mutlak zaman başlangıcından (epoch) yani 1970–01–01 00:00:00 UTC'den beri geçen saniye sayısı. Artık saniye desteği yoksa artık saniyeler hesaba katılmaz.

%s, **strftime**'da da GNU oluşumu olan bir GNU oluşumudur..

`%S`

0'dan **60**'a kadar saniye.

Başa 0 konulsa da olur konulmasa da.

Bilgi

Unix belirtimi iki artık saniyenin gösterilmesinin de mümkün olması varsayımıyla bu değerin üst sınırının **61** olduğunu söyler. Bu değeri **61** olarak hiç görmeyeceksiniz, çünkü 1 artık saniyeden fazlasına sahip bir dakika yoktur ama efsane sürüyor.

`%OS`

Yerele özgü diğer sayısal sembollerin kullanılması dışında **%S** ile aynıdır.

`%T`

%H:%M:%S biçimine eşdeğer olarak yorumlanır.

`%u`

1'den **7**'ye kadar gün isminin numarası. Pazartesi, haftanın birinci günüdür.

Başa 0 konulsa da olur konulmasa da.

Bilgi

Şimdilik tam olarak gerçeklenmemiştir. Biçim olarak tanınır ama *tm* yapısında ilgili alana birşey yazılmaz.

`%U`

0'dan **53**'e kadar hafta numarası.

Başa 0 konulsa da olur konulmasa da.

`%OU`

Yerele özgü diğer sayısal sembollerin kullanılması dışında **`%U`** ile aynıdır.

`%V`

`1`'den **`53`**'e kadar ISO 8601:1988 hafta numarası.

Başa 0 konulsa da olur konulmasa da.

Bilgi

Şimdilik tam olarak gerçeklenmemiştir. Biçim olarak tanınır ama *tm* yapısında ilgili alana birşey yazılmaz.

`%w`

`0`'dan **`6`**'ya kadar gün isminin numarası. Pazar, haftanın ilk günüdür.

Başa 0 konulsa da olur konulmasa da.

Bilgi

Şimdilik tam olarak gerçeklenmemiştir. Biçim olarak tanınır ama *tm* yapısında ilgili alana birşey yazılmaz.

`%Ow`

Yerele özgü diğer sayısal sembollerin kullanılması dışında **`%w`** ile aynıdır.

`%W`

`0`'dan **`53`**'e kadar hafta numarası.

Başa 0 konulsa da olur konulmasa da.

Bilgi

Şimdilik tam olarak gerçeklenmemiştir. Biçim olarak tanınır ama *tm* yapısında ilgili alana birşey yazılmaz.

`%x`

Yerele özgü kısa tarih gösterimi.

`%Ex`

Yerele özgü diğer veri gösterimlerinin kullanılması dışında **`%x`** gibidir.

`%X`

Yerele özgü saat gösterimi.

`%EX`

Yerele özgü diğer saat gösterimlerinin kullanılması dışında **`%x`** gibidir.

`%y`

`0` ile **`99`** arasında yılın yüzyıllık parçası olmaksızın yıl numarası.

Başa 0 konulsa da olur konulmasa da.

`%C` olmaksızın bu biçim nasıl kullanılacak diye bir soru sorabilirsiniz. **`strptime`** işlevi 69'dan 99'a kadar değerleri 1969'dan 1999'a kadar, 0'dan 68'e kadar değerleri de 2000'den 2068'e kadar yıllara karşılık olarak ele alacaktır. Fakat bazı veri girdilerinde bu ampirik yaklaşım başarısız olabilir.

Bu nedenle **%y** biçiminden uzak durup yerine **%Y** kullanmak en iyisidir.

%Ey
Yerelin diğer gösteriminde **%EC**'den itibaren geçen yıl sayısı.

%Oy
Yerelin diğer gösteriminde **%C** başlangıcından itibaren geçen yıl sayısı.

%Y
Gregoryen takvimine göre tam yıl.

%EY
Diğer tam yıl gösterimi.

%z
RFC 822/ISO 8601:1988 tarzı zaman dilimi.

%Z
Zaman dilimi kısaltması.

Bilgi

Şimdilik tam olarak gerçeklenmemiştir. Biçim olarak tanınır ama *tm* yapısında ilgili alana birşey yazılmaz.

%%
Tek bir **%** karakteri.

Biçim dizgesindeki diğer karakterler girdi dizgesindekilerle eşleşmelidir. Ancak girdi dizgesindeki boşluklar biçim dizgesinde bulunmayabileceği gibi daha fazla sayıda da bulunabilir.

Taşınabilirlik Bilgisi

XPG standardı, iki dönüşüm belirtimi arasında en azından bir boşluk (**isspace** ile belirtilenler) ya da bir alfanumerik olmayan karakter konulmasını tavsiye eder. GNU C kütüphanesi böyle bir sınırlama yapmaz ama diğer kütüphaneler **"%d%m%Y%H%M%S"** gibi bir biçim dizgesini yanlış çözümleyebilir.

strptime işlevi dizgeleri sağdan sola doğru işleme tabi tutar. Her olası girdi elemanı (boşluklar, kendisi olan karakterler, biçimler) için sırayla bakılır. Eğer girdiyle biçim eşleştirilemezse işlev işlemi durdurur. Girdi ve biçim dizgelerinin kalanı işleme tabi turulmaz.

İşlev, işlenmeyen ilk karaktere bir gösterici ile döner. Eğer girdi dizgesi, biçim dizgesinde gerekli karakterlerden fazla karakter içeriyorsa, dönüş değeri son işlenen girdi karakterinden hemen sonraki karakteri gösterir. Eğer girdi dizgesi tamamen tüketilmişse dönüş değeri dizgenin sonundaki boş karakteri gösterir. Bir hata oluşmuşsa, yani işlev biçim dizgesini tam eşleştirememişse, işlev **NULL** ile döner.

İşlevin XPG standardındaki belirtimi, bilginin bir kaç önemli parçasını dışarda bıraktığından oldukça muğlaktır. En önemlisi, doğrudan doğruya farklı biçimlerle ilklendirilmemiş *tm* elemanlarının ne olacağı belirlenmemiştir. Farklı Unix sistemlerindeki gerçeklemeler bu bakımdan değişiklikler gösterir.

GNU libc gerçeklemesi doğrudan ilklendirilmemiş alanlara dokunmaz. **tm_wday** ve **tm_yday** alanları bunun dışındadır. Yıl, ay veya tarih elemanları değişmişse bu alanların değerleri yeniden hesaplanır. Bunun iki faydası vardır:

- **strptime** işlevini yeni bir girdi dizgesi ile çağırmadan önce, işleve aktaracağınız *tm* yapısını hazılamalısınız. Bu normalde yapının tüm elemanlarının sıfır olacağı anlamına gelir. Bunun yerine tüm alanlara **INT_MAX** değerini de atayabilirsiniz. Böylece işlev çağrısının hangi elemanlara değer atadığını saptayabilirsiniz. Bunu sıfırlarla yapamazdınız, çünkü atanan bazı değerler zaten sıfır olacaktır.

  tm yapısının ilgilendiğiniz alanlarına işlev çağrısı sırasında bir değer atanıp atanmadığını saptamak isterseniz yapıyı dikkatli ilklendirmelisiniz.

- **struct tm** değerini peşpeşe yapacağınız **strptime** çağrıları ile oluşturabilirsiniz. Bir dizgede tarih başka bir dizgede de saat gösterimi bulunan iki dizgeyi ayrı ayrı çözümleten bir uyulama için bu yararlıdır. Bir dizgeyi çözümledikten sonra yapıyı temizlemden yapacağınız ikinci çözümleme ile tam yerel zamanı elde edebilirsiniz.

Aşağıdaki örnekte ya US tarzında ya da ISO 8601 biçiminde olabilen bir tarih bilgisi içeren bir dizgeyi çözümleyen bir işlev gösterilmiştir:

```
const char *
parse_date (const char *input, struct tm *tm)
{
  const char *cp;

  /* Önce yapıyı temizleyelim.  */
  memset (tm, '\0', sizeof (*tm));

  /* İlk olarak ISO biçimine bakalım.  */
  cp = strptime (input, "%F", tm);
  if (cp == NULL)
    {
      /* Eşleşme yok, o halde US biçimini deneyelim.  */
      cp = strptime (input, "%D", tm);
    }

  return cp;
}
```

4.6.2. Genel Zaman Gösterimi Çözümlemesi

Unix standardı tarih dizgelerini çözümlemek için başka bir işlev tanımlar. Arayüz tuhaftır ama uygulamanıza uygun düşecek olursa işlev en iyisidir. Bir durağan ayrılmış değişkene gösterici ile döndüğünden ve bir genel değişkenle genel bir durum (bir ortam değişkeni) kullandığından çok evreli uygulama ve kütüphanelerde sorun çıkarır.

getdate_err değişken

Son başarısız **getdate** çağrısının hata kodunu içeren **int** türünde bir değişkendir. Tanımlı değerleri şunlardır:

1
: **DATEMSK** ortam değişkeni tanımsız ya da boş.

2
: **DATEMSK** ortam değişkeni ile belirtilen şablon dosyası açılamadı.

3
: Şablon dosyası ile ilgili bilgiler alınamadı.

4

Şablon dosyası normal bir dosya değil.

5

Şablon dosyası okunurken bir G/Ç hatası oluştu.

6

İşlevin çalışması için bellek yetersiz.

7

Şablon dosyası eşleşen bir şablon içermiyor.

8

Girdi olarak verilen tarih geçersiz. Bu tarihler Şubat'ın 31'ini içeren ya da **time_t** türünde bir değişkenle ifade edilemeyen tarihler olabilir.

```
struct tm *getdate(const char *dizge)                                    işlev
```

getdate arayüzü bir dizgeyi çözümleyip bir değer döndürecek bir işlev için olası en basit arayüzdür. *dizge* bir girdi dizgesidir ve sonuç bir dutağan olarak ayrılmış değişkende döndürülür.

Dizgenin nasıl işleme sokulduğu ile ilgili ayrıntılar kullanıcıdan gizlenmiştir. Aslında, yazılımın da denetiminin dışında olabilir. Karşılaştırma yapılacak biçim dizgeleri **DATEMSK** ortam değişkeni ile belirtilen bir dosyadadır. Bu dosya **strptime** işlevine aktarılabilecek biçim dizgelerinden oluşmuş satırlar içerir.

getdate işlevi bu biçim dizgelerini sırayla okuyarak girdi dizgesi ile eşleştirmeye çalışır. Girdi dizgesi ile tamamen eşleşen ilk satır kullanılır.

Biçim dizgesi ile ilklendirilmemiş elemanlar tekrar bir **getdate** çağrısı yapılana kadar öylece kalır.

getdate tarafından tanınan biçimler **strptime** iişlevininkilerle aynıdır. Bunlar için önceki bölümdeki açıklamalara bakabilirsiniz. **strptime** davranışına ek olarak bir kaç ek davranış vardır:

- Eğer **%Z** biçimi yerel zamanda verilmişse, çalışma anı ortamının o anki zaman diliminin zamanına değil, zaman diliminin eşleştiği o anki zamana tabanlanır.

  

  Bilgi

  Bu şimdilik gerçeklenmemiştir. Sorun, zaman dilimi isimlerinin eşsiz olmamasıdır. ABD ile diğer ülkeler için aynı olarak kabul edilen bir sabit zaman dilimi (örn, **EST** ABD Doğu Sahili Zamanı anlamında) verildiğinde ABD dışındaki ülkeler açısından işlev başarısız olacaktır. Şimdiye kadar buna iyi bir çözüm bulamadık.

- Eğer sadece haftanın günü belirtilmişse, seçilen gün o anki tarihe bağlı olacatır. Eğer o anki haftanın günü **tm_wday** değerine eşit ya da ondan büyükse o anki haftanın günü, aksi takdirde sonraki haftanın günü seçilir.
- Benzer bir ampirik yöntem de sadece ayın verildiği yılın belirtilmediği durumla ilgilidir. Eğer ay, o anki aya eşit ya da büyükse, o anki yıl, aksi takdirde bir sonraki yıl kullanılır. Açıkça belirtilmemişse ayın ilk günü verilmiş varsayılır.
- Biçimde açıça belirtilmemişse o anki saat, dakika ve saniye değeri kullanılır.
- Bir tarih belirtilmemişse ve zaman, o anki zamandan daha küçükse ertesi günün tarihi aksi takdirde o günkü tarih alınır.

Şablon dosyasındaki biçimin sadece biçim elemanlarını içermediği unutulmamalıdır. Aşağıda olası biçim dizgelerinin bir listei verilmiştir (Unix standardından alınmıştır):

```
%m
%A %B %d, %Y %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

Gördüğünüz gibi, şablon listesi **`run job at %I %p,%B %dnd`** gibi çok özel dizgeler içerebilir. Bu listedeki şablonları kullanarak ve o anki zamanın Mon Sep 22 12:19:47 EDT 1986 olduğunu kabul ederek verilen girdiler için şu sonuçları aldık:

Girdi	Eşleşen	Sonuç
Mon	%a	Mon Sep 22 12:19:47 EDT 1986
Sun	%a	Sun Sep 28 12:19:47 EDT 1986
Fri	%a	Fri Sep 26 12:19:47 EDT 1986
September	%B	Mon Sep 1 12:19:47 EDT 1986
January	%B	Thu Jan 1 12:19:47 EST 1987
December	%B	Mon Dec 1 12:19:47 EST 1986
Sep Mon	%b %a	Mon Sep 1 12:19:47 EDT 1986
Jan Fri	%b %a	Fri Jan 2 12:19:47 EST 1987
Dec Mon	%b %a	Mon Dec 1 12:19:47 EST 1986
Jan Wed 1989	%b %a %Y	Wed Jan 4 12:19:47 EST 1989
Fri 9	%a %H	Fri Sep 26 09:00:00 EDT 1986
Feb 10:30	%b %H:%S	Sun Feb 1 10:00:30 EST 1987
10:30	%H:%M	Tue Sep 23 10:30:00 EDT 1986
13:30	%H:%M	Mon Sep 22 13:30:00 EDT 1986

İşlevin dönüş değeri **`struct tm`** türündeki bir durağan ayrılmış değişkene göstericidir, bir hata oluşmuşsa boş gösterici döner. Sonuç sadece sonraki **`getdate`** çağrısına kadar geçerlidir. Bu, işlevi çok evreli uygulamalar açısından kullanışsız yapar.

`errno` değişkeni *değiştirilmez*. Hta değerleri **`getdate_err`** genel değişkeninde saklanır. Olası hata değerleri ve açıklamaları için *yukarıya bakınız* (sayfa: 562).

Uyarı

`getdate` işlevi *hiçbir zaman* SUID'li yazılımlarda kullanılmamalıdır. Sebep belli: işlev **`DATEMSK`** ortam değişkeninde belirtilen bir dosyayı kullanıyor; dosyada bazı bozuk girdiler (ikilik veri gibi) varsa yazılım çökecektir.

```
int getdate_r(const char *dizge,                                    işlev
              struct tm  *zaman)
```

`getdate_r` işlevi **`getdate`** işlevinin evresel sürümüdür. **`getdate_err`** genel değişkenini kullanmaz, hata kodunu işlevin kendisi döndürür. Hata oluşmamışsa işlev sıfır döndürür. İşlevin döndürdüğü hata kodları **`getdate_err`** değişkeninin açıklamasındaki *hata kodları* (sayfa: 562) ile aynıdır.

Bundan başka, **getdate_r** işlevi yerel zamanı bir durağan değişkende değil, **struct tm** türünde bir değişken olan ikinci argümanda saklar.

Bu işlev Unix standardında tanımlanmamıştır. Buna rağmen birçok Unix sisteminde kullanılmaktadır.

getdate işlevi için belirtilen SUID'li yazılımlarla ilgili uyarı bu işlev için de geçerlidir.

4.7. Zaman Diliminin TZ ile Belirtilmesi

POSIX sistemlerde, bir kullanıcı zaman dilimini **TZ** ortam değişkeni ile belirtebilir. Ortam değişkenlerine nasıl değer atandığı ile ilgili belgi almak isterseniz *Ortam Değişkenleri* (sayfa: 676) bölümüne bakınız. Zaman dilimine erişim için kullanılan işlevler `time.h` başlık dosyasında bildirilmiştir.

Normalde **TZ** değişkenine bir değer atamak zorunda kalmazsınız. Eğer sistem olması gerektiği gibi yapılandırılmışsa öntanımlı zaman dilimi doğru olacaktır. Eğer bilgisayarınız farklı zaman dilimindeki bir ağa bağlıysa ve zaman diliminizin kendi zaman diliminiz değil de ağın zaman dilimi olarak raporlanmasını istiyorsanız **TZ** değişkenine ağın zaman dilimini atayabilirsiniz.

POSIX.1 sistemlerde **TZ** değişkeninin değeri üç biçimden biri olabilir. GNU C kütüphanesi ile en ortak biçim olan sonuncusunda zaman dilimi geniş bir veri tabanından yapılan bir seçimle belirtilir. İlk iki biçim ise zaman dilimi bilgisini doğrudan açıklamak için kullanılır, daha az kesin ve çok hantaldırlar. Ama POSIX.1 standardı sadece bu ilk iki biçimin ayrıntılarını belirlemiştir. Bu nedenle zaman dilimi bilgileri veritabanı desteği bulunmayan POSIX.1 sistemlerde bunlarla karşılaşabileceğinizden onlar hakkında bilgi sahibi olmanız iyi olacaktır.

İlk biçim yaz saati uygulaması yapılmayan zaman dilimlerinde kullanılır:

```
stdfark
```

std dizgesi zaman diliminin ismidir. En az üç karakter uzunlukta olmalı, başlangıcında iki nokta üstüste olmamalı, içinde rakam, virgül, artı ya da – işareti bulunmamalıdır. string specifies the name of the time zone. It must be three or more characters long and must not contain a leading colon, embedded digits, commas, nor plus and minus signs. There is no space character separating the time zone name from the *far* ile zaman dilimi arasında herhangi bir boşluk karakteri olmamalıdır. Bu sınırlamalar belirtimin çözümlenebilmesi için gereklidir.

fark, yerel zamanı elde etmek için UTC değerine eklenecek saat farkıdır. Sözdizimi şöyledir:

```
[+|-]ss[:dd[:SS]]
```

Yerel zaman dilimi ilk meridyenin batısındaysa pozitif, aksi yöndeyse negatiftir. Saatler **0** ile **23** arasında, dakika ve saniyeler ise **0** ile **59** arasına belirtilmelidir.

Örneğin, Doğu Standart Zamanını yaz saati uygulaması olmaksızın belirtmek isterseniz:

```
EST+5
```

```
std fark yer [fark],başlangıç[/saat],bitiş[/saat]
```

İlk *std* ve *fark* yukarıda açıklandığı gibi standart zaman dilimini belirtir. Sonraki *yer* dizgesi ve *fark* ise yaz saati uygulama bölgesinin ismini ve saat farkını belirtir. *fark* belirtilmemişse bir saatlik fark öntanımlıdır.

Belirtimin kalanı yaz saatinin nasıl uygulanacağını belirtir. *başlangıç* yaz saatinin uygulanmaya başlayacağı zamanı, *bitiş* ise uygulamanın sonlandırılıp normal zamana dönülecek zamanı belirtir. Bu alanlarda şu biçimler kullanılabilir:

J*n*

Jülyen takvimine göre gün belirtir. *n*, **1** ile **365** arasında bir değer alabilir. 29 Şubat, artık yıllarda bile sayılmaz.

n

Jülyen takvimine göre gün belirtir. *n*, **0** ile **365** arasında bir değer alabilir. 29 Şubat, artık yıllarda hesaba katılır.

M*a.h.g*
: *a*ıncı ayın *h*ıncı haftasının *g*inci gününü belirtir. *g* haftanın ilk günü Pazar olmak üzere **0** ile **6** arasında, *h* ise **1** ile **5** arasında olmalıdır; **1**. hafta *g*inci günü içeren ilk hafta, **5**. hafta ise *g*inci günü içeren son haftadır. *a* ise **1** ile **12** arasında olmalıdır.

saat alanı, yaz saati uygulamasının başlayacağı ve biteceği saati belirtmek için kullanılır. Verilmezse, öntanımlı olarak **02:00:00** kabul edilir.

Örnek olarak ABD'de Doğu zaman diliminde yaz saati uygulamasının nasıl belirtileceğine bakalım. UTC'ye göre saat farkı 5 saat; ilk meridyenin batısında olduğundan işareti pozitif; yaz saati uygulaması Nisan ayının ilk Pazar günü öğleden önce 2:00'da başlayıp Ekim ayının son Pazar günü öğleden önce 2:00'da sona ersin dersek, bunu şöyle belirtiriz:

```
EST+5EDT,M4.1.0/2,M10.5.0/2
```

Yaz saati uygulamasının zamanlaması bir takım karar organlarınca zaman içinde değiştirilebilir. Ancak bu biçim zamanlamanın yıldan yıla nasıl değişeceğinin belirtilebileceği hiçbir oluşuma sahip değildir. Tam doğru zaman dilimi belirtimi için en iyisi zaman dilimi bilgileri veritabanı kullanmaktır.

Üçüncü biçim ise şöyledir:

: *:karakterler*

Her işletim sistemi bu biçimi farklı yorumlar; GNU C kütüphanesinde *karakterler*, zaman dilimi bilgilerini içeren bir dosyanın ismidir.

TZ ortam değişkenine bir değer atanmamışsa, işlem öntanımlı olarak bir zaman dilimi seçer. GNU C kütüphanesinde öntanımlı zaman dilimi **TZ=:/etc/localtime** (veya GNU C kütüphanesinin derlenişine bağlı olarak, **TZ=:/usr/local/etc/localtime**; bkz. *GNU C Kütüphanesinin Kurulması* (sayfa: 950)) gibi bir belirtimdir. Diğer C kütüphaneleri öntanımlı zaman dilimin seçerken kendi kurallarını kullanırlar ve bunlar hakkında burada pek az şey söyleyebiliriz.

karakterler bir bölü çizgisi ile başlıyorsa, bir mutlak dosya ismi gösterir; aksi takdirde kütüphane **/share/lib/zoneinfo/***karakterler* dosyasına bakar. The **zoneinfo** dizini dünyanın farklı yerlerinde yerel zaman dilimleri hakkında bilgiler içeren veri dosyaları içerir. Bunlar, coğrafik bölge dizinleri içinde büyük şehirlerin isimlerini taşıyan dosyalardır; örneğin,**America/New_York**, **Europe/London**, **Asia/Hong_Kong** gibi. Bu veri dosyalarını sistem yöneticisi kurar ve yerel zaman dilimini **/etc/localtime** dosyasıyla belirtir. GNU C kütüphanesi, dünyanın hemen her yerindeki gönüllüler tarafından sağlanmış zaman dilimi bilgilerinden oluşturulmuş büyük bir veritabanı ile gelir.

4.8. Zaman Dilimi Değişkenleri ve İşlevleri

```
char *tzname [2]                                                    değişken
```

tzname dizisi kullanıcı tarafından seçilen bir standart isimle diğeri yaz saati ile ilgili iki dizge içerir. **tzname[0]** standart zaman diliminin ismi (örn, **"EST"**), **tzname[1]** ise yaz saati uygulaması ismidir (örn, **"EDT"**). Bunlar sırayla **TZ** ortam değişkeninin *std* ve *fark* dizgelerine karşılıktır. Yaz saati uygulaması yapılmıyorsa, **tzname[1]** boş bir dizge olur.

tzname dizisi, **TZ** ortam değişkeninden başka **tzset**, **ctime**, **strftime**, **mktime** veya **localtime** işlev çağrıları ile değiştirilebilir. Eğer bir zaman dilimini belirtmek için çok sayıda kısaltma varsa (örn, U.S. Eastern War Time ve Eastern Daylight Time için **"EWT"** ve **"EDT"**), kullanımdaki en son kısaltma belirtilmelidir.

tzname dizisi POSIX.1 uyumluluğu gerektirir. Ancak GNU yazılımlarının yerel zaman yapısının **tm_zone** üyesi en son kullanımda olanı olmasa bile en doğru kısaltmayı içerdiğinden bu üyeyi kullanmaları daha iyi olur.

Dizgeler **char *** olarak bildirildiğinden kullanıcı bu dizgeleri değiştirmekten kaçınmalıdır. Bu dizgelerin değiştirilmesi hemen hemen daima sorunlara yol açar.

`void tzset(void)` işlev

tzset işlevi **tzname** değişkenini **TZ** ortam değişkeninin değerinden günceller. Normalde bu işlevi çağırmanız gerekmez. Çünkü zaman dilimine bağlı zaman dönüşüm işlevlerini kullandığınızda bu işlev kendiliğinden çağrılır.

Aşağıdaki değişkenler System V Unix uyumluluğu için tanımlanmıştır. **tzname** gibi bu değişkenlere de **tzset** veya diğer zaman dönüşüm işlevleri ile değer atanır.

`long int timezone` değişken

Bu, UTC'nin batısına doğru saniye cinsinden, UTC ile en son yerel standart zaman arasındaki saat farkıdır. Örneğin ABD Doğu zaman dilimi için bu değer **5*60*60** saniyedir. Yerel zaman yapısının **tm_gmtoff** üyesinin aksine, bu değer yaz saati uygulaması ile ilgili değildir ve işareti de terstir. GNU yazılımlarında, en son uygulananı olmasa da en doğru değeri içerdiğinden, yerel zaman yapısının **tm_gmtoff** üyesini kullanmak daha iyidir.

`int daylight` değişken

Bu değişkenin değeri sıfırdan farklıysa yaz saati uygulaması kuralları uygulanır. Değerin sıfırdan farklı olması, yaz saatinin o an uygulanmakta olduğunun değil, zamanı gelince uygulanacağının göstergesidir.

4.9. Zaman İşlevleri Örneği

Burada yerel zamanla ilgili bazı işlevlerin kullanımı örneklenmiştir.

```
#include <time.h>
#include <stdio.h>
#include <locale.h>

#define SIZE 256

int
main (void)
{
  char buffer[SIZE];
  time_t curtime;
  struct tm *loctime;

  /* Yerel zamanı gerçekten yerele özgü göstereceksek
     bu lazım, yoksa C yereline özgü gösterilir. */
  setlocale (LC_ALL, "");

  /* Şimdiki zamanı öğrenelim. */
  curtime = time (NULL);

  /* Yerel zaman gösterimine dönüştürelim. */
  loctime = localtime (&curtime);

  /* Tarihi ve saati standart C biçiminde basalım. */
```

```
  fputs (asctime (loctime), stdout);

  /* Şimdi de yerele özgü ve istediğimiz biçimde basalım. */
  strftime (buffer, SIZE, "Bugün %d %B, %A.\n", loctime);
  fputs (buffer, stdout);
  strftime (buffer, SIZE, "Saat %I:%M %p.\n", loctime);
  fputs (buffer, stdout);

  return 0;
}
```

Çıktısı şöyle olur:

```
Sat Aug 28 13:49:43 2004
Bugün 28 Ağustos, Cumartesi.
Saat 01:49 ÖS.
```

5. Bir Alarmın Ayarlanması

alarm ve **setitimer** işlevleri bir sürecin kendisine gelecekte bir kesme göndermesini sağlar. Bunu bir zamanlayıcıyı ayarlayarak yaparlar; zamanlayıcının süresi dolduğunda süreç bir sinyal alır.

Her süreç üç bağımsız zamanlayıcı kullanabilir:

- Bir gerçek zamanlı zamanlayıcı kalan süreyi sayar. Bu zamanlayıcı zamanaşımına uğradığında sürece bir **SIGALRM** sinyali gönderir.
- Bir sanal zamanlayıcı süreç tarafından kullanılan işlemci süresini sayar. Bu zamanlayıcı zamanaşımına uğradığında sürece bir **SIGVTALRM** sinyali gönderir.
- Bir profil zamanlayıcı hem süreç tarafından kullanılan işlemci süresini hem de sistemin süreç adına kullandığı işlemci süresini sayar. Bu zamanlayıcı zamanaşımına uğradığında sürece bir **SIGPROF** sinyali gönderir.

  Bu zamanlayıcı yorumlayıcılarda profil çıkarma için kullanışlıdır. Zamanlayıcı mekanizmasının birim sayma süresi doğal kodun profilinin çıkarılması için gereken hassasiyete sahip değildir.

Herhangi bir anda bu üç zamanlayıcıdan sadece birini kullanabilirsiniz. Henüz zamanaşımına uğramamış bir zamanlayıcıyı tekrar kullanmaya çalışırsanız, zamanlayıcıyı basitçe yeni değerle yeniden başlatmış olursunuz.

Bir **setitimer** veya **alarm** işlevini çağırmadan önce **signal** veya **sigaction** işlevini kullanarak ilgili alarm sinyali için bir sinyal eylemci oluşturmalısınız. Aksai takdirde, bir olağandışı olaylar zinciri yazılımınız daha bir sinyal eylemci kurmaya fırsat bulamadan zamanlayıcının zamanaşımına uğramasına sebep olabilir. Bu durumda, alarm sinyalleri için öntanımlı eylem sürecin sonlandırılması olduğundan yazılımınız sonlanırdı. Bkz. *Sinyal İşleme* (sayfa: 601).

Durdurulmadığı takdirde sonsuza kadar engelleme yapabilecek bir sistem çağrısını durdurmak amacıyla alarm işlevinin kullanılmasını mümkün kılmak için sinyal eylemcinin **sigaction** ile **SA_RESTART** seçeneği atanmadan oluşturulması önemlidir. **sigaction** kullanılmadığı durumda işler biraz daha karışıktır, çünkü **signal** işlevinin yeniden başlatma açısından işleyiş mantığı sabittir: BSD mantığına göre bu seçenek atanır. Dolayısıyla herhangi bir nedenle **sigaction** kullanılamazsa **signal** değil **sysv_signal** kullanmak gerekir.

Bir alarm ayarlanmak istendiğinde ilk akla gelen **setitimer** işlevidir. Bu oluşum `sys/time.h` başlık dosyasında bildirilmiştir. Gerçek zamanlı bir zamanlayıcıyı etkinleştirmek için biraz daha basit bir arayüz sunan **alarm** işlevi ise `unistd.h` dosyasında bildirilmiştir.

`struct` **itimerval** veri türü

Bu yapı bir zamanlayıcıyı ayarlamak için kullanılır. Şu üyeleri içerir:

`struct timeval` **`it_interval`**
: Zamanlayıcı kesmeleri arasındaki süredir. Sıfırsa alarm sadece bir kere gönderilir.

`struct timeval` **`it_value`**
: İlk zamanlayıcı kesmesine kadar geçen süredir. Sıfırsa alarm iptal edilir.

The **`struct timeval`** data type is described in *Süre* (sayfa: 538).

```
int setitimer(int                tür,                                    işlev
              struct itimerval *yeni,
              struct itimerval *eski)
```

`setitimer` işlevi *tür* türündeki zamanlayıcıyı *yeni* değere ayarlarlar. *tür* argümanı **`ITIMER_REAL`**, **`ITIMER_VIRTUAL`** ve **`ITIMER_PROF`** değerlerinden biri olabilir.

eski bir boş gösterici değilse, işlev henüz zamanaşımına uğramamış aynı türde bir zamanlayıcı varsa onunla ilgili yapıyı bu gösterici ile döndürür.

İşlev başarılı ise dönüş değeri **`0`**, değilse **`-1`**'dir. Aşağıdaki **`errno`** hata değeri bu işlev için tanımlanmıştır:

`EINVAL`
: Zamanlayıcının süresi çok uzun.

```
int getitimer(int                tür,                                    işlev
              struct itimerval *eski)
```

`getitimer` işlevi *tür* türündeki zamanlayıcı ile ilgili bilgileri *eski* ile gösterilen yapı içinde döndürür.

Dönüş değeri ve hata durumları **`setitimer`** ile aynıdır.

`ITIMER_REAL`
: Bu sabit **`setitimer`** ve **`getitimer`** işlevlerinin *tür* argümanında kullanıldığında gerçek zamanlı bir zamanlayıcı belirtir.

`ITIMER_VIRTUAL`
: Bu sabit **`setitimer`** ve **`getitimer`** işlevlerinin *tür* argümanında kullanıldığında sanal bir zamanlayıcı belirtir.

`ITIMER_PROF`
: Bu sabit **`setitimer`** ve **`getitimer`** işlevlerinin *tür* argümanında kullanıldığında gerçek bir profil zamanlayıcı belirtir.

```
unsigned int alarm(unsigned int saniye)                                  işlev
```

`alarm` işlevi gerçek zamanlı zamanlayıcıyı *saniye* saniyede zamanaşımına uğrayacak şekilde ayarlar. Mevcut bir alarmı iptal etmek isterseniz, işlevi *saniye* argümanında sıfır değerini aktararak çağırmanız yeterlidir.

Dönüş değeri, önceki alarmın kalan süresidir. Daha önce bir alarm yoksa işlev sıfır ile döner.

alarm işlevi **setitimer** kuralları ile şöyle tanımlanabilirdi:

```
unsigned int
alarm (unsigned int seconds)
{
  struct itimerval old, new;
  new.it_interval.tv_usec = 0;
  new.it_interval.tv_sec = 0;
  new.it_value.tv_usec = 0;
  new.it_value.tv_sec = (long int) seconds;
  if (setitimer (ITIMER_REAL, &new, &old) < 0)
    return 0;
  else
    return old.it_value.tv_sec;
}
```

alarm işlevinin kullanım örneğini *Dönen Sinyal Yakalayıcılar* (sayfa: 618) bölümünde bulabilirsiniz.

Sürecinizin belli bir süre beklemesini isterseniz **sleep** işlevini kullanmalısınız. Bkz. *Uyku* (sayfa: 570).

Zamanlayıcı zamanaşımına uğrar uğramaz sinyalin gelmesini beklememelisiniz. Çok işlemcili ortamlarda genellikle biraz gecikme olur.

Taşınabilirlik Bilgisi

setitimer ve **getitimer** işlevlerinin BSD Unix'den türetilmiş olmasına karşın **alarm** işlevi POSIX.1 standardında belirtilmiştir. **setitimer** işlevi daha güçlü olduğu halde **alarm** işlevi daha yaygın olarak kullanılır.

6. Uyku

sleep işlevi yazılımınızın kısa bir süre için beklemeye alınmasını sağlayan basit bir yöntem sunar. Yazılımınız sinyalleri kullanmıyorsa (sonlanma hariç) **sleep** ile belli bir süre bekleme sağlayacağınızı varsayabilirsiniz. Ancak, eğer bir sinyal gelirse **sleep** daha erken dönebilir. Bekleme süresinin sinyallerden bağımsız olmasını isterseniz **select** (bkz. *Girdi ve Çıktının Beklenmesi* (sayfa: 323)) işlevini beklenecek herhangi bir tanımlayıcı belirtmeden kullanmalısınız.

`unsigned int` **sleep**`(unsigned int` *saniye*`)` işlev

sleep işlevi bir sinyal alınmadıkça çağrıldığı süreci *saniye* saniye bekletir.

sleep işlevi istenen süreyi tamamlamışsa, sıfır ile döner; bir sinyal nedeniyle tamamlayamamışsa kalan süreyi belirten bir sayı ile döner.

sleep işlevi `unistd.h` başlık dosyasında bildirilmiştir.

sleep işlevinin dönüş değerini kullanarak ve sıfırdan farklıysa çağrıyı tekrarlayarak bekleme süresinin dolmasını sağlamaya karşı bir direnç oluşur. Bu çalışma sonucunda elde edilen sonucun doğruluğu sinyallerin geliş sıklığına göre değişir. Her sinyal alınışında araya ek birkaç saniye eklenecektir. Çok sık aralıklarla birkaç sinyalin birden gelmesi kötü bir şans olurdu, bu durumda bekleme süresinin uzunluğunun ya da kısalığının bir sınırı olmayacaktır.

Bunun yerine yazılımın beklemeyi sonlandıracağı zamanı bir mutlak zamana ayarlayın. Bu bir saniyeden fazla şaşmaz. Biraz daha fazla bir çalışma ile daha kesin bir bekleme süresini **select** işlevini kullanarak sağlayabilirsiniz. (Bu durumda da makina uygulamanıza adanmamışsa ağır sistem yükleri kaçınılmaz ek gecikmeler oluşturur ve bundan kaçınmanın bir yolu yoktur.)

Bazı sistemlerde, **sleep** işlevi yazılımın doğrudan **SIGALRM** kullanmasına bağlı olarak tuhaf şeyler yapabilir. **sleep** çağrıldığı sırada **SIGALRM** sinyalleri yoksayılsa ya da engellense bile, bir **SIGALRM** sinyali alındığında işlev zamanaşımı süresi bitmeden dönecektir. **SIGALRM** sinyalleri için bir sinyal eylemci oluşturursanız ve sürecin uykusu esnasında bir **SIGALRM** sinyali gelirse, bu, eylemciniz çağrılmadan önce **sleep** işlevinin dönmesine sebep olur. Eğer **sleep** işlevi, eylemcisi bir alarm isteğinde bulunan ya da **SIGALRM** sinyalinin eylemini değiştiren bir sinyalle kesildiğinde bu eylemci ile **sleep** işlevi birbiriyle etkileşecektir.

GNU sisteminde, **sleep** işlevi **SIGALRM**'la etkileşecek şekilde çalışmadığından **sleep** ve **SIGALRM** aynı yazılımda güvenle kullanılabilir.

```
int nanosleep(const struct timespec *istenen,                    işlev
              struct timespec       *kalan)
```

Bir saniyelik çözünürlük yetersizse **nanosleep** işlevi kullanılabilir. İsminden de anlaşılacağı üzere uyku süresi nanosaniyeler cinsinden belirtilebilir. Asıl uyku süresini, sistemin uygulayabileceği çözünürlüğün katları olan bir tamsayıya yuvarlanacağından istenenden biraz daha uzun olabilir.

struct timespec yapısı *Süre* (sayfa: 538) bölümünde açıklanmıştır.

***istenen** uyku süresini belirtmek için kullanılır.

İşlev kesintiye uğramadan istenen süreyi doldurursa sıfır ile döner, aksi takdirde kalan süre ile ilgili bilgileri **kalan* içinde saklayarak döner. İşlev –1 değeri ile döndüğünde *errno* değişkeninde aşağıdaki değerler atanabilir.

EINTR
: Bir sinyal alındığından çağrı durduruldu. İşlevin *kalan* parametresi bir boş gösterici değilse kalan süre ile ilgili bilgiler *kalan* ile gösterilen yapıya yerleştirilir.

EINVAL
: *istenen* parametresi bir kuraldığı değer içeriyor. Değer ya negatif ya da 1000 milyona eşit veya büyük.

Bu işlev çok evreli yazılımlarda bir iptal noktasıdır. **nanosleep** çağrısı sırasında evre bazı özkaynakları ayırıyorsa (bellek, dosya tanıtıcı, semafor,v.s) bu bir sorun olabilir. Evre özkaynak ayırdıktan sonra bir iptal alırsa bu özkaynaklar uygulama sonlandırılana kadar ayrılmış olarak kalacaktır. İptal eylemcileri kullanarak **nanosleep** çağrılarının oluşturacağı bu olumsuz durum engellenmelidir.

nanosleep işlevi `time.h` başlık dosyasında bildirilmiştir.

XXII. Özkaynak Kullanımı ve Sınırlaması

İçindekiler

Bu oylumda bir sürecin çalışması esnasında kullanacağı çeşitli sistem özkaynaklarının (İşlemci zamanı, bellek, v.s.) sınırlarını öğrenmek ve belirtmek için kullanabileceği işlevlere yer verilmiştir.

1. Özkaynak Kullanımı

Bir sürecin özkaynak kullanımını incelemek için kullanılan **getrusage** işlevi ve **struct rusage** veri yapısı `sys/resource.h` başlık dosyasında bildirilmiştir.

```
int getrusage(int                süreçler,                                  işlev
              struct rusage *kaynakkullanımı)
```

Bu işlev *süreçler* ile belirtilen süreçlerin toplam özkaynak kullanımını **kaynakkullanımı* içinde saklayarak döner.

Çoğu sistemde, *süreçler* için sadece iki geçerli değer vardır:

`RUSAGE_SELF`
: Sadece işlevi çalıştıran süreci belirtir.

`RUSAGE_CHILDREN`
: Sonlanmış olan tüm alt süreçler (Doğrudan ya da dolaylı).

GNU sisteminde, belli bir alt sürecin özkaynak kullanımını süreç kimliğini belirterek sorgulayabilirsiniz.

İşlevin normal dönüş değeri sıfırdır, **-1** dönüş değeri bir hata oluştuğunu gösterir. Bu işlev için tanımlanmış **errno** hata durumu:

`EINVAL`
: *süreçler* argümanı geçersiz

Belli bir alt sürecin özkaynak kullanımını sorgulamanın tek yolu, sonlandığında kullandığı özkaynak toplamlarını döndüren **wait4** işlevidir. Bkz. *BSD Süreç Bekleme İşlevleri* (sayfa: 693).

`struct` **rusage** veri türü

Bu veri türü çeşitli özkaynakların kullanım istatistiklerini saklar. En azından şu üyelere sahiptir:

`struct timeval` **ru_utime**
Kullanıcı komutları çalıştırılırken harcanan süre.

`struct timeval` **ru_stime**
İşletim sistemi tarafından *süreçler* yararına kullanılan toplam süre.

`long int` **ru_maxrss**
süreçler'in aynı anda kullandığı fiziksel belleğin kilobayt cinsinden azami miktarı.

`long int` **ru_ixrss**
Diğer süreçlerle paylaşılan metin tarafından kullanılan bellek miktarını belirten icra tiklerini kilobayt cinsinden ifade eden bir tamsayı değer.

`long int` **ru_idrss**
Veri için kullanılan paylaşımsız bellek miktarının aynı yöntemle ifade edilen tamsayı değeri.

`long int` **ru_isrss**
Yığıt alanı için kullanılan paylaşımsız bellek miktarının aynı yöntemle ifade edilen tamsayı değeri.

`long int` **ru_minflt**
Herhangi bir G/Ç gerektirmeksizin sunulan sayfalama hatalarının sayısı.

`long int` **ru_majflt**
G/Ç yaparak sunulan sayfalama hatalarının sayısı.

`long int` **ru_nswap**
süreçler'in tamamen ana belleğe takaslanma sayısı.

`long int` **ru_inblock**
Dosya sisteminin *süreçler* yararına diskten yaptığı okumaların sayısı.

`long int` **ru_oublock**
Dosya sisteminin *süreçler* yararına diske yaptığı yazmaların sayısı.

`long int` **ru_msgsnd**
Gönderilen süreçlerararı iletişim (IPC) iletilerinin sayısı.

`long int` **ru_msgrcv**
Alınan süreçlerararı iletişim (IPC) iletilerinin sayısı.

`long int` **ru_nsignals**
Alınan sinyallerin sayısı.

`long int` **ru_nvcsw**
süreçler'in bir bağlamsal seçiciye kasıtlı yaptığı çağrı sayısı (genellikle bazı hizmetler beklenirken).

`long int` **ru_nivcsw**
İstemdışı olarak bir bağlamsal seçicide yeralma sayısı (belli bir süre geçmiş ya da daha yüksek öncelikli başka bir süreç öne geçmiş olmasından dolayı).

vtimes işlevi **getrusage** işlevinin yaptığını yapan tarihi bir işlevdir. **getrusage** işlevinin kullanımı tercih edilmelidir.

vtimes ve onun **vtimes** veri yapısı `sys/vtimes.h` başlık dosyasında bildirilmiştir.

`int` **vtimes**`(struct vtimes` *şimdiki*`, struct vtimes` *altsüreç*`)` işlev

vtimes bir sürecin özkaynak kullanım toplamlarını raporlar.

Eğer *şimdiki* boş gösterici değilse, işlev sadece çağıran sürecin özkaynak kullanım toplamlarını bu argümanın gösterdiği yapıda saklar. *altsüreç* bir boş gösterici değilse, çağıran sürecin sonlanmış alt süreçlerinin özkaynak kullanım toplamlarını bu argümanın gösterdiği yapıda saklar.

`struct` **vtimes** veri türü

Bu veri türü çeşitli özkaynakların kullanımı ile ilgili bilgileri saklar. Bu yapının her üyesi yukarıda açıklanan **struct rusage** yapısının üyelerinin karşılığıdır.

`vm_utime`
: Kullanıcı işlemci zamanı. **struct rusage** yapısının **ru_utime** üyesinin karşılığıdır.

`vm_stime`
: Sistem işlemci zamanı. **struct rusage** yapısının **ru_stime** üyesinin karşılığıdır.

`vm_idsrss`
: Veri ve yığıt belleği. **struct rusage** yapısının **ru_idrss** ve **ru_isrss** üyesisinin raporladığı değerlerin toplamıdır.

`vm_ixrss`
: Paylaşımlı bellek. **struct rusage** yapısının **ru_ixrss** üyesinin karşılığıdır.

`vm_maxrss`
: Süreçlerin aynı anda kullandığı fiziksel belleğin kilobayt cinsinden azami miktarı. **struct rusage** yapısının **ru_maxrss** üyesinin karşılığıdır.

`vm_majflt`
: G/Ç'lı sayfalama hatalarının sayısı.**struct rusage** yapısının **ru_majflt** üyesinin karşılığıdır.

`vm_minflt`
: G/Ç'sız sayfalama hatalarının sayısı. **struct rusage** yapısının **ru_minflt** üyesinin karşılığıdır.

`vm_nswap`
: Takaslama sayısı. **struct rusage** yapısının **ru_nswap** üyesinin karşılığıdır.

`vm_inblk`
: Diskten okuma sayısı. **struct rusage** yapısının **ru_inblk** üyesinin karşılığıdır.

`vm_oublk`
: Diske yazma sayısı. **struct rusage** yapısının **ru_oublk** üyesinin karşılığıdır.

İşlev başarılı olursa sıfır değeri ile, aksi takdirde **-1** ile döner.

2. Özkaynak Kullanımın Sınırlanması

Bir sürecin özkaynak kullanımı için sınırlar belirtebilirsiniz. Süreç bir sınırı aşmaya çalışırsa bir sinyal alabilir ya da özkaynağa bağımlı bir sistem çağrısı başarısız olabilir. Her süreç kendi özkaynak sınır değerlerini kendini çalıştıran süreçten miras alır, ancak onları üst sürecinden bağımsız olarak değiştirebilir.

Bir özkaynağın her süreç için iki sınırı vardır:

mevcut sınır
: Mevcut sınır sistemin aşılmasına izin vermediği sınırdır. "Sanal sınır" olarak da bilinir, çünkü bu sınırı sürecin kendisi belirler.

üst sınır
: Bir sürecin mevcut sınır olarak belirleyebileceği değerin üst sınırıdır. "Kesin sınır" olarak da bilinir, çünkü sürecin bu sınırın etrafından dolanması mümkün değildir. Bir süreç mevcut sınırını bu değere kadar arttırabilirken, sadece sistem yöneticisi bu üst sınırı arttırabilir.

getrlimit, **setrlimit**, **getrlimit64** ve **setrlimit64** işlevleri ve bunlarla kullanılan semboller `sys/resource.h` başlık dosyasında bildirilmiştir.

```
int getrlimit(int özkaynak,                                                    işlev
              struct rlimit *sınır)
```

özkaynak özkaynağının mevcut ve üst sınırlarını okur ve bunları **sınır* içinde saklar.

İşlev başarılı olursa **0**, aksi takdirde **−1** ile döner. Olası tek **errno** hata durumu **EFAULT**'tur.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği 32 bitlik bir sistemde bu işlev aslında **getrlimit64** işlevine denktir. Yani LFS arayüzü eski arayüzün yerine geçer.

```
int getrlimit64(int özkaynak,                                                  işlev
                struct rlimit64 *sınır)
```

Bu işlev ikinci argümanın **struct rlimit64** türünde olması dışında **getrlimit** işlevinin eşdeğeridir.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği 32 bitlik bir sistemde bu işlevin ismi **getrlimit** olur, böylece LFS arayüzü eski arayüzün yerine geçer.

```
int setrlimit(int özkaynak,                                                    işlev
              const struct rlimit *sınır)
```

özkaynak özkaynağının mevcut ve üst sınırlarını **sınır* içindeki değerlere ayarlar.

İşlev başarılı olursa **0**, aksi takdirde **−1** ile döner. Bu işlev için tanımlanmış **errno** hata durumları:

`EPERM`
- Süreç mevcut sınırı üst sınırın üstünde belirtmeyi denedi
- Süreç üst sınırı arttırmayı denedi ama yetkisi yetersiz

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği 32 bitlik bir sistemde bu işlev aslında **setrlimit64** işlevine denktir. Yani LFS arayüzü eski arayüzün yerine geçer.

```
int setrlimit64(int özkaynak,                                                  işlev
                const struct rlimit64 *sınır)
```

Bu işlev ikinci argümanın **struct rlimit64** türünde olması dışında **setrlimit** işlevinin eşdeğeridir.

Kaynakların **_FILE_OFFSET_BITS == 64** ile derlendiği 32 bitlik bir sistemde bu işlevin ismi **setrlimit** olur, böylece LFS arayüzü eski arayüzün yerine geçer.

`struct` **rlimit** veri türü

Bu yapı **getrlimit** işlevinde kullanıldığında sınır değerlerin alınmasını sağlar, **setrlimit** işlevinde kullanıldığında ise belli bir süreç ve özkaynak için sınır değerleri belirtir. İki üyesi vardır:

`rlim_t` **rlim_cur**
Mevcut sınır.

`rlim_t` **rlim_max**
Üst sınır.

getrlimit işlevi açısından yapı bir çıktı alanıdır; o anki değerleri alır. **setrlimit** işlevinde ise yeni değerleri belirtir.

LFS işlevleri için benzer bir tür `sys/resource.h` başlık dosyasında tanımlanmıştır.

`struct` **rlimit64** veri türü

Bu yapı üyelerinin daha geniş aralıklara sahip olması dışında yukarıdaki **rlimit** yapısının karşılığıdır. İki üyesi vardır:

`rlim64_t` **rlim_cur**
rlimit.rlim_cur karşılığıdır, ama türü farklıdır.

`rlim64_t` **rlim_max**
rlimit.rlim_max karşılığıdır, ama türü farklıdır.

Bir sınır belirtilebilecek özkaynakların listesi aşağıda verilmiştir. Bellek ve dosyalarla ilgili sınırlar bayt cinsindendir.

`RLIMIT_CPU`
Sürecin kullanabileceği işlemci zamanının azami miktarı. Süreç bundan daha uzun süre çalışırsa, **SIGXCPU** sinyalini alacaktır. Değer saniye cinsindendir. Bkz. *İşlemsel Hata Sinyalleri* (sayfa: 609).

`RLIMIT_FSIZE`
Sürecin oluşturabileceği azami dosya boyutu. Daha büyük bir dosya yazma denemesi **SIGXFSZ** sinyaline sebep olur. Bkz. *İşlemsel Hata Sinyalleri* (sayfa: 609).

`RLIMIT_DATA`
Süreç için ayrılan azami veri belleği miktarı. Süreç bu miktardan daha büyük bellek ayırmaya çalışırsa bellek ayırma işlevi başarısız olur.

`RLIMIT_STACK`
Süreç için azami yığıt boyutu. Süreç yığıtını bundan fazlasına genişletmeye çaışırsa bir **SIGSEGV** sinyali alacaktır. Bkz. *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

`RLIMIT_CORE`
Bu sürecin oluşturabileceği core dosyasının azami boyutu. Eğer süreç sonlanır ve bu boyuttan daha büyük bir core dosyası dökümlemeyi denerse core dosyası oluşmaz. Bu bakımdan, bu sınırı sıfıra indirerek core dosyalarının asla oluşmamasını sağlayabilirsiniz.

`RLIMIT_RSS`

Bu sürecin alabileceği fiziksel belleğin azami miktarı. Bu parametre sistemin zamanlayıcısı ve bellek ayırıcısı için bir kılavuzdur. Sistem, sürece ihtiyaç duyduğunda bundan daha fazla bellek verebilir.

`RLIMIT_MEMLOCK`
Fiziksel bellekte kilitlenebilecek belleğin azami miktarı (böylece bu bellek takaslanmayacaktır).

`RLIMIT_NPROC`
Aynı kullanıcı kimlikle oluşturulabilecek süreçlerin azami sayısı. Bu sınırı kendi kullanıcı kimliğiniz için aşmaya çalışırsanız, **fork** işlevi **EAGAIN** ile başarısız olacaktır. Bkz. *Bir Sürecin Oluşturulması* (sayfa: 687).

`RLIMIT_NOFILE`
`RLIMIT_OFILE`
Sürecin açabileceği dosyaların azami sayısı. Bundan daha fazla dosya açmaya çalışırsa, işlem **EMFILE** hata kodu ile başarısız olur. Bu sınır tüm sistemlerde desteklenmez; GNU ve 4.4 BSD böyledir.

`RLIMIT_AS`
Bu sürecin alabileceği toplam belleğin azami miktarı. Eğer süreç bunu aşan miktarı örneğin, **brk**, **malloc**, **mmap** veya **sbrk** ile ayırmaya çalışırsa işlev başarısız olur.

`RLIM_NLIMITS`
Farklı özkaynak sınırlarının sayısı. Geçerli *özkaynak* terimlerinin sayısı **RLIM_NLIMITS** değerinden küçük olmalıdır.

`int` **RLIM_INFINITY** değişken

Bu sabit, **setrlimit** içinde sınır değer olarak belirtildiğinde "sonsuz" değerine karşılıktır.

Aşağıdaki tarihi işlevler şimdiye kadar bahsedilen işlevlerin yaptıklarını yaparlar. Bunların yerine yukarıda bahsedilen işlevlerin kullanılması daha iyi bir seçim olacaktır.

ulimit ve komut sembolleri `ulimit.h` başlık dosyasında bildirilmiştir.

`int` **ulimit**`(int` *komut*`, ...)` işlev

ulimit işlevi çağrıldığı süreç ile ilgili olarak *komut* ile belirtildiği gibi özkaynağın ya o anki mevcut sınırıyla döner ya da mevcut ve üst sınırını belirtir.

Bir sınır değeri döndürmek için komut argümanından başka argüman gerekmez. Bir sınırı belirtmek için ikinci bir argüman gerekir: **long int***sınır*.

komut argümanında kullanılabilecek değerler ve belirttikleri işlemler:

`GETFSIZE`
Bir dosya boyutunun mevcut sınırını 512 baytlık birimler cinsinden döndürür.

`SETFSIZE`
Bir dosya boyutunun mevcut ve üst sınırını *sınır* * 512 bayta ayarlar.

Bazı sistemlerde başka *komut* değerleri de desteklenmektedir ama onlar GNU kütüphanesinde desteklenmemektedir.

Bir üst sınırı sadece sistem yöneticisi arttırabilir.

Bir sınır başarıyla alınmışsa işlevin dönüş değeri bu sınırın değeridir ve asla negatif değildir. Bir sınır başarıyla değiştirilmişse işlevin dönüş değeri sıfır olur. İşlev başarısız olduğunda **-1** ile döner ve olası **errno** hata durumu şu olabilir:

EPERM
: Süreç bir üst sınırı arttırmayı denedi ama yetkisi yetersiz

vlimit ve onunla ilgili özkaynak sembolleri `sys/vlimit.h` başlık dosyasında bildirilmiştir.

```
int vlimit(int özkaynak,                                                    işlev
           int sınır)
```

vlimit bir sürecin bir özkaynağı için mevcut sınırı değiştirir.

özkaynak şunlardan biri olabilir:

LIM_CPU
: Azami işlemci zamanı. **setrlimit** için **RLIMIT_CPU** ile aynıdır.

LIM_FSIZE
: Azami dosya boyutu. **setrlimit** için **RLIMIT_FSIZE** ile aynıdır.

LIM_DATA
: Azami veri belleği. **setrlimit** için **RLIMIT_DATA** ile aynıdır.

LIM_STACK
: Azami yığıt boyutu. **setrlimit** için **RLIMIT_STACK** ile aynıdır.

LIM_CORE
: Azami core dosyası boyutu. **setrlimit** için **RLIMIT_COR** ile aynıdır.

LIM_MAXRSS
: Azami fiziksel bellek. **setrlimit** için **RLIMIT_RSS** ile aynıdır.

İşlevin normal dönüş değeri sıfırdır, **-1** dönüş değeri bir hata oluştuğunu gösterir. Bu işlev için tanımlanmış **errno** hata durumu:

EPERM
: Süreç mevcut sınırı üst sınırın üzerine çıkarmaya çalıştı

3. Sürecin İşlemci Önceliği ve Zamanlama

Çok sayıda süreç aynı anda işlemci zamanını kullanmak isterse, sistemin zamanlama kurallarına ve sürecin işlemci önceliğine bakılarak onu hangi sürecin alacağı saptanır. Bu kısımda GNU C kütüphanesinde onu denetleyen işlevlerle bu saptamanın nasıl yapıldığından bahsedilecektir.

Metin içinde bir özkaynak olarak işlemci ile ilgili olarak, işlemci zamanlamasından basitçe zamanlama ve sürecin işlemci önceliğinden ise sürecin önceliği olarak bahsedeceğiz. İşlemci zamanının sadece bir sürecin kullandığı bir özkaynak olduğu ya da süreçlerin işlemci zamanı için savaştıkları gibi bir anlam da çıkarmayın. Hatta bazı durumlarda kısmen bile önemli değildir. Bir sürece yüksek bir "öncelik" verilmesi bu sürecin diğer süreçlerden daha hızlı olmasına çok küçük bir etkisi olabilir. Bu bölümde öncelik deyince sadece işlemci zamanına uygulanan öncelikten bahsediyor olacağız.

İşlemci zamanlaması karmaşık bir konudur ve farklı sistemler bunu oldukça farklı yollarla yaparlar. Yeni fikirler sürekli geliştirilmekte ve çeşitli sistemlerin zamanlama algoritmalarının griftliği içinde yollar bulunmaya çalışılmaktadır. Bu bölümde genel kavramlar, GNU C kütüphanesini kullanan sistemlerin bazı özellikleri ve standartlar üzerinde durulacaktır.

Basitleştirmek için, sistemde sadece bir işlemci ve o işlemci içinde sadece bir işlem birimi varmış gibi davranacağız. Ancak bazı prensipler bir işlemci birden fazla işlem birimi içerdiğinde uygulanır ve eşit sayıda işlem birimi içeren çok sayıda işlemci olduğunda bu bilgiyi kolayca genelleştirebilirsiniz.

Bu bölümde bahsedilen işlevlerin hepsi POSIX.1 ve POSIX.1b standartlarında bulunur (**sched…** işlevleri, POSIX.1b'dedir). Bununla birlikte, POSIX bu işlevlerin okuduğu ya da belirlediği değerler için anlambilimsel bir tanım yapmamıştır. Bu kısımda, anlama dair kabuller POSIX standardının Linux çekirdeği gerçeklemesi üzerine oturtulmuştur. Göreceğiniz gibi, Linux gerçeklemesi POSIX sözdizimi yazarlarlarınınkilerden biraz terstir.

3.1. Mutlak Öncelik

Her sürecin bir mutlak önceliği vardır ve bir sayı ile ifade edilir. Daha yüksek sayı daha yüksek mutlak öncelik demektir.

Geçmişteki bazı sistemlerde ve günümüzde çoğu sistemde, tüm süreçlerin mutlak önceliği 0'dır ve bu bölümün konusu dışındadır. Bu durumla ilgili olarak, *Geleneksel Zamanlama* (sayfa: 584) bölümüne bakınız. Mutlak öncelikler, belli süreçlerin dış olaylara yanıt vermesinin hayati önemde olduğu gerçek zamanlı sistemler için tasarlanmıştır. Bu sistemlerde çalışmak *isteyen* süreçler işlemciyi tutarken çalışması *gereken* süreçleri bekletmemesi amaçlanmıştır.

İşlemciyi herhangi bir anda kullanmaya çalışacak iki süreçten daha yüksek önceliği olan onu alır. Bunlardan biri işlemciyi zaten kullanmaktaysa ve önceliği düşükse önceliği yüksek olan işlemciyi yine alacaktır (yani, zamanlama ayrıcalıklıdır). Şüphesiz, burada bahsettiğimiz süreçler zaten başlatılmış yani çalışabilir olan ya da o an komutlarını çalıştırmaya hazır olmak anlamında "çalışmaya hazır" süreçlerdir. Bir süreç G/Ç işlemi gibi bir işlem nedeniyle beklemedeyse, onun önceliği konumuzun dışındadır.

Bilgi

"çalışabilir olmak" ile "çalışmaya hazır olmak" aynı anlamda kullanılmıştır.

İki süreç de çalışabilir durumdaysa ya da çalışmaya hazırsa ve ikisinin de mutlak önceliği aynıysa, bu daha ilginçtir. Bu durumda işlemciyi hangisinin alacağını zamanlama kuralları belirler. Eğer süreçlerin mutlak öncelikleri 0 ise, *Geleneksel Zamanlama* (sayfa: 584) bölümünde anlatılan geleneksel zamanlama kurallarına göre bu saptanır. Aksi takdirde *Anlık Zamanlama* (sayfa: 580) bölümünde anlatılan kurallar uygulanır..

Normalde 0'ın üstünde bir mutlak önceliği, işlemciyi etkisiz bırakmayacağından emin olduğunuz bir sürece verirsiniz. Böyle süreçler kısa bir işlemci kullanımından sonra beklemeye geçmek ya da sonlanmak üzere tasarlanır.

Bir süreç, kendini başlatan süreç ile aynı mutlak öncelikle oluşturulur. Bu durumu değiştirebilen işlevler *Temel Zamanlama İşlevleri* (sayfa: 581) bölümünde açıklanmıştır.

Sadece ayrıcalıklı bir süreç kendi mutlak önceliğini **0**'dan farklı bir değere ayarlayabilir. Sadece ayrıcalıklı bir süreç ya da hedef sürecin sahibi mutlak önceliği değiştirebilir.

POSIX, gerçekzamanlı zamanlama kuralları ile kullanılan mutlak öncelik değerlerinin 32'den başlayan ve ardışık sıralanan değerler olmasını gerektirir. Linux'ta 1 ile 99 arasındadır. Taşınabilirlik açısından, **sched_get_priority_max** ve **sched_set_priority_min** işlevleri belli bir sistem üzerinde geçerli aralığın ne olduğunu söyler.

3.1.1. Mutlak Önceliğin Kullanımı

Gerçek zamanlı uygulamaları tasarlarken unutmamanız gereken tek şey diğer süreçlerden daha yüksek bir mutlak önceliğin sürecin çalışma sürekliliğini garanti etmeyeceğidir. İşlemcinin çalışmasını kazaya uğratan iki şey vardır: kesmeler ve sayfalama hataları.

Kesme eylemcileri süreçler arasında unutulur gider. Komutlarını işlemci çalıştırır ama onlar bir sürecin parçası değillerdir. Bir kesme en yüksek öncelikli süreci bile durdurur. Bu durumda önemsiz derecede küçük gecikmelere izin vermelisiniz ve sürecinizin komutları arasında çok uzun gecikmelere sebep olabilecek kesme eylemcilere sahip aygıtların olmadığından da emin olmalısınız.

Benzer şekilde, bir sayfalama hatası, basit bir komut dizisinin uzun bir zaman alıyormuş gibi görünmesine sebep olur. Aslında, sayfalama hatası sırasında bundan etkilenmeyen süreçler çalışmaya devam eder, çünkü G/Ç'ların tamamlanması gerekir, ancak yüksek öncelikli süreç onları atıp tekrar çalışmaya devam eder. Esas sorun G/Ç için, öncelikli sürecin kendisinin bekleme durumuna geçmesi olurdu. Bu evreyi etkisiz hale getirmek için **`mlock`** veya **`mlockall`** kullanılır.

Bir öncelik vermeyi seçerken ve ayrıca yüksek mutlak öncelikli bir yazılım da çalıştırıyorsanız, aklınızda tutmanız gereken şey, tek işlem birimli tek işlemcili bir sistemde bu önceliğin mutlaklığının bunlar arasında bölüneceğidir. Diğer süreçlere göre daha yüksek mutlak öncelikli bir sürecin, yazılımındaki bir hatadan dolayı sonsuz döngüye girdiğini varsayalım. Çalışması sırasında işlemciden asla vazgeçmeyecektir. Bir komut çalıştırmak için onu öldürmekten başka çareniz kalmaz. Hatalı yazılım denetimi tamamen, her yönden ele geçirir.

Bundan kaçınmanın iki yolu vardır: 1) bir yerlerde çalışmakta olan daha yüksek öncelikli bir kabuk bulundurursunuz (root'un açtığı bir kabuk örneğin). 2) yüksek öncelikli süreç grubu ile ilişkili bir denetim uçbirimi tutarsınız. Çalışmaya başlayan bir kesme eylemciden kaçacak ya da <**C–c**> tuşladığınızda sinyal alıp da durmayacak bir öncelik mevcut değildir.

Bazı sistemler mutlak önceliği, işlemci zamanının belli bir yüzdesini bir sürece ayırmak manasında kullanırlar. Bir süper yüksek öncelikli ve ayrıcalıklı bir sürecin işlemci kullanımını sürekli gözlemesini sağlayarak, böylece paylaşıma girmeyen bir sürecin mutlak önceliğini yükselterek ve onu aşan bir sürecin mutlak önceliğini düşürerek bunu yaparlar.

Bilgi

Mutlak öncelik bazan "durağan öncelik" diye de anılır; bu kılavuzda bu terimi kullanmıyoruz, çünkü mutlak önceliğin en önemli özelliğini, mutlaklığını kaybederiz.

3.2. Anlık Zamanlama

Aynı mutlak önceliğe sahip iki süreç aynı anda çalışmaya hazırsa, çekirdek bir karar vermek zorundadır, çünkü bir kerede sadece biri çalışabilir. Eğer süreçlerin mutlak öncelikleri 0 ise çekirdek bu kararı *Geleneksel Zamanlama* (sayfa: 584) bölümünde anlatıldığı gibi verir. Aksi takdirde vereceği kararı bu bölümde anlatacağız.

Farklı mutlak önceliklere sahip iki süreç çalışmaya hazırsa verilecek karar basittir, bu *Mutlak Öncelik* (sayfa: 579) bölümünde açıklanmıştı.

Her sürecin kendine ait zamanlama kuralları vardır. Sıfırdan farklı mutlak önceliği olan süreçler için bunlar iki tanedir:

1. ilk gelen alır
2. döner turnuva düzenlenir

En duyarlı durum, tüm süreçlerin aynı zamanlama kurallarına sahip olduğu ama farklı mutlak önceliklere sahip olduğu durumdur ki, bundan daha önce söz edilmişti.

Turnuvada, süreçler işlemciyi paylaşırlar, her biri küçük bir zaman diliminde çalışırlar ve döner turnuva bağlamında bunu her turda bir kere yaparlar. Şüphesiz, bu turnuvaya sadece aynı mutlak önceliğe sahip ve aynı anda çalışmaya hazır süreçler katılırlar.

İlk gelen alır durumunda ise, en uzun bekleyen süreç işlemciyi alır ve işlemciyi bırakmaya gönüllü olana kadar, bırakmaktan başka çare kalmayana dek (beklemeye geçmek gibi) ya da daha yüksek öncelikli bir süreç işlemciyi alana kadar işlemciyi tutar.

İlk gelen alır kuralı, en yüksek mutlak öncelik ve kesmelerle sayfalama hatalarının dikkati denetlenmesiyle, bir sürecin mutlak olarak ve olumlu manada işlemciyi tam hızında çalıştırdığı takdirde kullanılması vazgeçilmezdir; değilse anlamlı değildir.

Süreçlerin **sched_yield** çağrılarında, zamanlama kuralını belirtirken turnuva ile ilk gelen alır arasında iyi bir uzlaşımın sonucu olarak ilk gelen alır kuralını kullanmak konusunda akıllıca davranmalıdırlar.

Farklı zamanlama kuralları olan aynı mutlak önceliğe sahip süreçler açısından zamanlamanın nasıl çalıştığını anlamak için, süreçlerin çalışmaya hazır süreçler listesine nasıl dahil edildiği ve nasıl listeden çıkarıldıkları ile ilgili ayrıntıları iyi bilmek zorundasınız:

Her iki durumda da, çalışmaya hazır süreçlerin listesi gerçek bir kuyruk olarak düzenlenir. Süreç çalışmaya hazır olduğunda kuyruğun sonuna eklenir ve zamanlayıcı onu çalıştırmaya karar verdiğinde kuyruğun başına çekilir.Çalışmaya hazır olmak ile çalışıyor olmanın aynı anda olan şeyler olmadıklarına dikkat edin. Zamanlayıcı bir süreci çalıştırıyorsa o artık çalışmaya hazır bir süreç değildir ve dolayısıyla artık çalışmaya hazır süreçler listesinde değildir. Sürecin çalışması durduğunda tekrar çalışmaya hazır duruma gelir.

Turnuva kuralına göre çalışacak bir süreç ile ilk gelen alır kuralına göre çalışacak bir süreç arasındaki tek fark, ilk durumdaki sürecin belli bir süre sonra işlemciyi bırakmak zorunda kalacağıdır. Bu olduğunda süreç tekrar çalışmaya hazır duruma gelir ve kuyruğa tekrar eklenir. Burada bahsedilen süre oldukça kısadır. Hem de gerçekten kısadır. Örneğin, Linux çekirdeğinde turnuva zaman dilimi, geleneksel zamanlama ile ilgili zaman diliminden bin kere daha kısadır.

Bir süreç, kendini başlatan süreç ile aynı zamanlama kuralları ile oluşturulur. Bu durumu değiştirebilen işlevler *Temel Zamanlama İşlevleri* (sayfa: 581) bölümünde açıklanmıştır.

Sadece ayrıcalıklı bir süreç, mutlak önceliği **0**'dan farklı bir sürecin zamanlama kuralını ayarlayabilir.

3.3. Temel Zamanlama İşlevleri

Bu bölümdeki işlevler, mutlak önceliği ve bir sürecin zamanlama kuralını ayarlamakta kullanılan GNU C kütüphanesindeki işlevlerdir.

Taşınabilirlik Bilgisi

Bu işlevlere sahip sistemlerde **_POSIX_PRIORITY_SCHEDULING** makrosu `unistd.h` başlık dosyasında tanımlanmıştır.

Zamanlama kurallarının geleneksel zamanlama ile ilgili olduğu durumda zamanlamanın daha hassas ayarlanabileceği işlevleri *Geleneksel Zamanlama* (sayfa: 584) bölümünde bulabilirsiniz.

Bu işlevlerin yapısı ve isimlendirilmesi bakımından denenecek çok fazla birşey yoktur. Bunlar POSIX.1b tarafından tanımlanmış olduğundan bu kılavuzda bahsedilen kavramlarla uyuşmamaları normaldir. Çünkü GNU C kütüphanesinin kullanıldığı sistemlerdeki gerçekleme POSIX yapısı ile ilgili kavramsallaştırmanın tersidir. POSIX şeması birincil zamanlama parametresinin zamanlama kuralları olduğunu ve öncelik değerinin (varsa), zamanlama kurallarının bir parametresi olduğunu kabul eder. Gerçeklemede ise ister istemez, öncelik değeri kraldır ve zamanlama kuralları, eğer varsa, önceliği etkileyen ince bir ayardır.

Bu bölümdeki semboller `sched.h` başlık dosyasında bildirilmiştir.

`struct` **sched_param** veri türü

Bu yapı bir mutlak önceliği tanımlar.

`int` **`ssched_priority`**
mutlak öncelik değeri

```
int sched_setscheduler(pid_t                        pid,                    işlev
                       int                          kural,
                       const struct sched_param *öncelik)
```

Bu işlev bir sürecin hem mutlak önceliğini hem de zamanlama kuralını ayarlar.

Süreç kimliği *pid* olan sürecin ya da *pid* olarak sıfır verildiğinde çağrıldığı sürecin mutlak önceliğini *öncelik* ile, zamanlama kuralını *kural* ile belirtilen değere ayarlar. Eğer *kural* negatifse işlev mevcut zamanlama kuralını değiştirmez.

İşlevin *kural* argümanında belirtilebilecek değerler şunlardır:

`SCHED_OTHER`
Geleneksel zamanlama

`SCHED_FIFO`
İlk gelen alır

`SCHED_RR`
Döner turnuva

İşlev başarılıysa **`0`** ile döner. **`-1`** ile dönmüşse bir hata oluşmuştur. Bu durumda **`errno`** değişkeni şu hata durumlarından birini içerir:

`EPERM`

- Çağıran süreç **`CAP_SYS_NICE`** yetkisine sahip değil ve *kural* değeri **`SCHED_OTHER`** değil (ya da değeri negatif ve mevcut kural **`SCHED_OTHER`** değil).
- Çağıran süreç **`CAP_SYS_NICE`** yetkisine sahip değil ve sahibi hedef sürecin sahibi değil. Yani çağıran sürecin etkin kullanıcı kimliği *pid* süreç kimlikli sürecin ne etkin ne de gerçek kullanıcı kimliğidir.

`ESRCH`
pid sıfırdan farklı olduğu halde *pid* süreç kimlikli bir süreç yok.

`EINVAL`

- *kural* mevcut bir zamanlama kuralını ifade etmiyor.
- **öncelik* ile belirtilen mutlak öncelik *kural* zamanlama kuralı için (ya da *kural* negatifse mevcut zamanlama kuralı için) geçerli aralığın dışında ya da *öncelik* bir boş gösterici. Geçerli aralığın ne olduğunu **`sched_get_priority_max`** ve **`sched_get_priority_min`** işlevleri ile öğrenebilirsiniz.
- *pid* negatif.

```
int sched_getscheduler(pid_t pid)                                           işlev
```

Süreç kimliği *pid* olan sürecin ya da *pid* olarak sıfır verildiğinde çağrıldığı sürece atanmış zamanlama kuralını döndürür.

İşlevin normal dönüş değeri zamanlama kuralıdır. Olası değerler için **sched_setscheduler** işlevine bakınız.

İşlev başarısız olursa **-1** döner ve **errno** değişkenine şu hata durumlarından biri atanır:

ESRCH
: *pid* sıfırdan farklı ve böyle bir süreç yok.

EINVAL
: *pid* negatif.

Bu işlevin **sched_setscheduler** işlevinin tam karşılığı olmadığına dikkat edin, **sched_setscheduler** işlevi hem zamanlama kuralını hem de mutlak önceliği belirlemek için kullanılabilirken bu işlev sadece zamanlama kuralını döndürür. Mutlak önceliği öğrenmek için **sched_getparam** işlevini kullanabilirsiniz.

```
int sched_setparam(pid_t                          pid,                    işlev
                   const struct sched_param *öncelik)
```

Bu işlev sürecin mutlak önceliğini değiştirmekte kullanılır.

İşlevselliği *kural*= **-1** olduğunda **sched_setscheduler** işlevindeki ile aynıdır.

```
int sched_getparam(pid_t                          pid,                    işlev
                   const struct sched_param *öncelik)
```

Bu işlev sürecin mutlak önceliği ile döner.

pid, mutlak önceliği öğrenilmek istenen sürecin süreç kimliğidir.

öncelik ise sürecin mutlak önceliğini içeren yapıya bir göstericidir.

Başarı durumunda işlevin dönüş değeri **0**'dır. Aksi takdirde **-1** döner ve **errno** değişkenine şu değerler biri atanır:

ESRCH
: *pid* sıfırdan farklı ve böyle bir süreç yok.

EINVAL
: *pid* negatif.

```
int sched_get_priority_min(int *kural)                                      işlev
```

Bu işlev *kural* zamanlama kuralı için bir sürecin alabileceği en düşük mutlak öncelik değeri ile döner.

Linux'ta, **SCHED_OTHER** için **0** diğerleri için **1**'dir.

Başarı durumunda işlevin dönüş değeri **0**'dır. Aksi takdirde **-1** döner ve **errno** değişkenine şu değerler biri atanır:

EINVAL
: *kural* mevcut bir zamanlama kuralı değil.

```
int sched_get_priority_max(int *kural)                                      işlev
```

Bu işlev *kural* zamanlama kuralı için bir sürecin alabileceği en yüksek mutlak öncelik değeri ile döner.

Linux'ta, **SCHED_OTHER** için **0** diğerleri için **99**'dur.

Başarı durumunda işlevin dönüş değeri **0**'dır. Aksi takdirde **-1** döner ve **errno** değişkenine şu değerler biri atanır:

EINVAL
kural mevcut bir zamanlama kuralı değil.

```
int sched_rr_get_interval(pid_t pid,                                    işlev
                          struct timespec *süre)
```

Bu işlev turnuva zamanlama kuralı kullanıldığında *pid* kimlikli sürecin işlemciyi kullanabileceği süreyi döndürür.

Sonuç *süre* ile döner.

Linux çekirdeği için turnuva zaman dilimi daima 150 mikrosaniyedir ve *pid* değerinin gerçek bir süreç kimliği olması bile gerekmez.

Başarı durumunda işlevin dönüş değeri **0**'dır. Aksi takdirde, imkansız olsa bile **-1** dönebilir. Ancak **errno** değişkenine atanmak üzere belirlenmiş bu işleve özgü özel bir hata durumu yoktur.

```
int sched_yield(void)                                                   işlev
```

Bu işlev, sürecin işlemci üzerindeki haklarından vazgeçilmesini sağlar.

Teknik olarak, **sched_yield** işlevi sürecin hemen tekrar çalışmaya hazır duruma (işlev çağrıldığında çalışmakta olmanın tersine) gelmesine sebep olur. Yani, sürecin mutlak önceliği sıfırdan farklıysa, mutlak öncelikleri mutlak önceliği ile aynı olan süreçlerin bulunduğu süreç listesinin kuyruğuna çalışmaya hazır süreç olarak eklenip sırasının gelmesini bekler. Mutlak önceliği sıfırsa bu işlem daha karmaşıklaşır, ancak işlemcinin başka bir sürece bırakılması yine de gerçekleşir.

Eğer mutlak öncelik bakımından eşdeğerde başka bir süreç yoksa bu işlev hiçbir şey yapmaz.

İşlevi içeren süreç açısından süreç, başka bir sürecin ne yaptığı ya da ne kadar hızlı çalıştığının farkında değildir, işlev bu bakımdan işlevsizdir.

Başarı durumunda işlevin dönüş değeri **0**'dır. Aksi takdirde, imkansız olsa bile **-1** dönebilir. Ancak **errno** değişkenine atanmak üzere belirlenmiş bu işleve özgü özel bir hata durumu yoktur.

3.4. Geleneksel Zamanlama

Bu bölüm mutlak önceliği sıfır olan süreçlerin zamanlanması hakkındadır. Yüksek mutlak öncelikli süreçler istediklerini aldıktan sonra geriye kalan işlemci zamanı artıkları, burada tanımlanan zamanlama kuralına göre el değmemiş irice süreçler arasında paylaştırılır.

3.4.1. Geleneksel Zamanlamaya Giriş

Mutlak öncelikler (sayfa: 579) ortaya çıkmadan çok önceleri Unix sistemleri işlemci zamanlamasını bu şekilde yapıyordu. Posix Romalılar gibi çıkagelip gerçek zamanlı işlemin gereksinimlerini karşılamak üzere mutlak öncelikleri ortaya attı ve köydeki Sıfır Mutlak Öncelikli süreçlerin kendilerini kendi bildikleri zamanlama kurallarına göre yönetmelerine izin verdi.

Gerçekten, sıfırdan büyük mutlak öncelikler, esas olarak gerçek zamanlı işlem yapmak üzere tasarlanmış bilgisayarların sistemleri dışında günümüzdeki çoğu sistemde kullanılmamaktadır. Bu bakımdan, bu bölümde sadece çoğu yazılımcının bilmek istediği zamanlamadan söz edilecektir.

Bu zamanlamanın kapsamı hakkında daha temiz bir giriş olarak: Herhangi bir anda mutlak önceliği sıfır olan bir süreç ile mutlak önceliği sıfırdan büyük bir süreç aynı anda çalışmaya hazırsa, mutlak önceliği sıfır olan çalışmaz. Eğer mutlak önceliği sıfır olan bir süreç çalışmaktayken mutlak önceliği sıfırdan büyük bir süreç çalışmaya hazır olursa mutlak önceliği sıfır olanın çalışması hemen durdurulur.

Mutlak önceliği sıfır olma durumuna ek olarak, sürecin çalışması esnasında değişen bir öncelik olarak ***özdevimli öncelik*** olarak bilinen bir öncelik daha vardır. Mutlak önceliği sıfırdan büyük süreçler için özdevimli öncelik anlamlı değildir.

Özdevimli öncelik bazan işlemciyi kimin alacağını, bazan işlemciyi ne kadar süreyle kullanacağını, bazan da bir sürecin başka bir süreci işlemciden kovup kovamayacağını belirler.

Linux'ta, değer bunların bir karışımı olarak ortaya çıkar. Fakat çoğunlukla bu değer sürecin işlemciyi kullanma süresini belirler. Özdevimli önceliği daha yüksek olan süreç, işlemciyi bir kere aldı mı, daha uzun süre onu kullanır. Eğer, G/Ç beklemek gibi şeyler yapmak için kendi zaman dilimi içinde işini bitiremezse, tekrar hazır olduğunda kendi zaman dilimini tamamlamak üzere işlemciyi alması için öne alınır. Bunun dışında, yeni zaman dilimleri için süreçlerin seçimi temel olarak turnuva sistemine göre yapılır. Fakat zamanlayıcı düşük öncelikli süreçlere bir kemik atar: Bir sürecin özdevimli önceliği, zamanlama işleminde her aşağılanışında yükselir. Linux'ta, oyunu hep şişko velet kazanır.

Bir sürecin özdevimli önceliğinin iniş çıkışları başka bir değerle düzene sokulur: ***nezaket değeri*** (ing.si "nice value" olan değer). Nezaket değeri bir tamsayıdır ve –20 ile 20 arasında olup, bir sürecin özdevimli önceliğinin uç değerlerini ifade eder. Daha yüksek bir nezaket daha düşük bir sınır gösterir.

Tipik bir Linux sisteminde, örneğin, nezaket değeri 20 olan bir süreç işlemci zamanının sadece 10 milisaniyesini alabilir, –20 olan bir süreç ise daha yüksek bir öncelikle işlemci zamanının 400 milisaniyesini alabilecektir.

Nezaket değeri gerçekten bir kibarlık, saygılılık belirtir. Başlarda, Unix'in cennet bahçesinde, tüm süreçler bilgisayar sisteminin nimetlerini eşit olarak paylaşırdı. Ama tüm süreçler aynı işlemci zamanını paylaşma ihtiyacı göstermez, bu durumda nezaket değeri, diğer süreçler yararına kibar bir sürecin payına düşen işlemci süresinden feragat etmesini sağlar. Dolayısıyla, nezaket değeri daha yüksek olan süreç daha kibar süreç olur. (Bir yılan geldi ve bir sürece bir negatif nezaket değeri sundu böylece bugün kaba özkaynak ayırma sistemi olarak bildiğimiz sistem ortaya çıktı).

Özdevimli öncelikler işlemci zamanının ayrılmasını nesnel olarak pürüzsüzleştirerek ve seyrek isteklere hızlı yanıt vererek yukarı ve aşağı doğru meylederler. Fakat kendi nezaket sınırlarını asla aşmazlar, böylece işlemcinin ağır yük altında olduğu durumda nezaket değeri etkin olarak bir sürecin ne kadar hızlı çalışacağını belirler.

Unix süreç önceliğinin toplumcu mirasına uymak için, bir süreç kendini çalıştıran süreçle aynı nezaket değeri ile oluşur ve onu yükseltebilir. Bir süreç ayrıca sahibi aynı kullanıcı olan (ya da aynı etkin kimlikli) başka bir sürecin de nezaket değerini yükseltebilir. Fakat sadece ayrıcalıklı süreç kendi nezaket değerini düşürebilir. Bir ayrıcalıklı süreç ayrıca başka bir sürecin nezaket değerini de arttırıp azaltabilir.

Nezaket değerlerini öğrenmek ve belirlemek için kullanılan GNU C kütüphanesi işlevleri *Geleneksel Zamanlama İşlevleri* (sayfa: 585) bölümünde açıklanmıştır.

3.4.2. Geleneksel Zamanlama İşlevleri

Bu bölümde bir sürecin nezaket değerinin nasıl okunabileceğinden ve nasıl belirtilebileceğinden bahsedilecektir. Bu sembollerin tamamı `sys/resource.h` başlık dosyasında bildirilmiştir.

İşlev ve makro isimleri POSIX tarafından tanımlanmıştır. POSIX'te ve bu kılavuzda isimlendirme için "öncelik" (priority) terimi kullanılmış olmasına rağmen işlevler aslında nezaket değerleri ile işlem yaparlar.

Geçerli nezaket değerleri çekirdeğe bağlı olmasına rağmen, genellikle **-20** ile **20** arasındadır. Daha düşük bir nezaket değeri daha yüksek önceliğe karşılıktır. Öncelik değerlerinin aralığını belirleyen sabitler şunlardır:

PRIO_MIN
: Geçerli en düşük nezaket değeri.

PRIO_MAX
: Geçerli en yüksek nezaket değeri.

```
int getpriority(int sınıf,                                                    işlev
                int kimlik)
```

Bir süreç kümesinin nezaket değerini döndürür; *sınıf* ve *kimlik* hangisi olduğunu belirtir (aşağıya bakınız). Eğer belirtilen süreçlerin hepsi aynı nezaket değerine sahip değilse, işlev bunların içinden en düşük nezaket değerini döndürür.

İşlev başarılı olursa **0** ile döner. Hata oluşmuşsa **-1** döner ve **ERRNO** değişkenine hata durumu atanır. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

ESRCH
: *sınıf* ve *kimlik* birlikte mevcut herhangi bir süreçle uyuşmuyor.

EINVAL
: *sınıf* değeri geçersiz.

Dönen değer **-1** olduğundan bunun bir başarısızlık mı yoksa bir nezaket değeri mi olduğunu anlayabilmek için tek yol **getpriority** çağrısından önce **errno** değişkenine **0** değeri atamaktır. Çağrının ardından **errno != 0** kullanarak başarısızlık sınaması yapabilirsiniz.

```
int setpriority(int sınıf,                                                    işlev
                int kimlik,
                int nezaket)
```

Bir süreç kümesinin nezaket değerini *nezaket* değerine ayarlar;*sınıf* ve *kimlik* hangisi olduğunu belirtir (aşağıya bakınız).

İşlev başarılı olursa **0** ile döner. Hata oluşmuşsa **-1** döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

ESRCH
: *sınıf* ve *kimlik* birlikte mevcut herhangi bir süreçle uyuşmuyor.

EINVAL
: *sınıf* değeri geçersiz.

EPERM
: Çağrı, sahibi çağıran süreçten farklı bir kulllanıcıya ait bir sürecin nezaket değerini değiştirmeye çalışıyor (yani, hedef sürecin gerçek ya da etkin kullanıcı kimliği işlevi çağıran sürecin ektin kullanıcı kimliği ile aynı değil) ve işlevi çağıran süreç **CAP_SYS_NICE** yetkisine sahip değil.

EACCES
: Çağrı daha düşük bir nezaket değeri isteği yaptı ama sürecin **CAP_SYS_NICE** yetkisi yok.

sınıf ve *kimlik* argümanları birlikte ilgilendiğiniz süreç kümesini belirtirler. *sınıf* için olası değerler şunlardır:

PRIO_PROCESS

Belirli bir süreç. *kimlik* sürecin süreç kimliğidir.

`PRIO_PGRP`
Belirli bir süreç grubundaki bütün süreçler. *kimlik* süreç grup kimliğidir.

`PRIO_USER`
Belirli bir kullanıcıya ait olan tüm süreçler (yani, gerçek kullanıcı kimliği kullanıcı ile aynı olan süreçler). *kimlik* kullanıcının kullanıcı kimliğidir.

Eğer *kimlik* değeri sıfırsa, bu çağıran süreci belirtir, süreç grubu ya da sürecin sahibi *sınıf* ile belirtilir.

`int` **`nice`**`(int` *artış*`)` işlev

Çağıran sürecin nezaket değerini *artış* kadar arttırır. İşlev başarılı olursa yeni nezaket değeri ile döner. **`-1`** dönüş değeri ise bir hata oluştuğunu gösterir. Bu durumda **`errno`** değişkenine **`setpriority`** işlevindeki değerler atanır.

`nice` işlevine eşdeğer bir işlev tanımı:

```
int
nice (int artis)
{
  int sonuc, eski = getpriority (PRIO_PROCESS, 0);
  sonuc = setpriority (PRIO_PROCESS, 0, eski + artis);
  if (sonuc != -1)
      return eski + artis;
  else
      return -1;
}
```

3.5. İşlemciler Arasında İcra Sınırlaması

Çok işlemcili bir sistemde, işletim sistemi sistemin en verimli çalışmasını mümkün kılan bir yolla farklı süreçleri mevcut işlemcilere dağıtır. Zamanlama işlevselliğini genişleterek hangi süreçlerin ve evrelerin çalışabileceği geçtiğimiz bölümlerde açıklanmıştı. Fakat hangi işlemcinin sonuçta hangi süreç ya da evreyi çalıştıracağı konumuzun dışındadır.

Bir yazılımın sistemi bu bakımdan denetimi altına almak zorunda bırakan bazı sebepler vardır:

- Mutlak olarak kritik bir işlem yürütme durumunda olan bir süreç ya da evre hiçbir şart altında durdurulmamalı veya işlemci özkaynaklarını kullanarak başka süreçler tarafından engellenmemelidir. Bu durumda özel süreç için, hiçbir süreç ya da evre tarafından kullanılmasına izin verilmeyen bir işlemci tahsis edilmelidir.

- Belirli özkaynaklara (RAM, G/Ç portları) farklı işlemcilerden erişim maliyetleri farklıdır. Böyle bir duruma NUMA (Non–Uniform Memory Architecture — Tektip olmayan bellek mimarisi) makinalarda rastlanır. Tercihan, belleğe yerel olarak erişilmelidir fakat bu gereksinim genelde zamanlayıcıya görünür değildir. Bundan dolayı kullanılan belleğe yerel erişime sahip işlemcilerin bir sürece ya da evreye ayrılması başarımın belirgin biçimde artmasına yardımcı olur.

- Çalışma anında denetim altında özkaynak ayrılması ve toparlama çalışmalarında (örn, çöp toplama) başarım işlemcilerin yerel işlem yapmasına bağlıdır. Eğer özkaynaklar farklı işlemcilerin rasgele erişimine karşı korunmamışsa, bu, kilitleme maliyetlerini düşürmeye yardımcı olur.

Şimdiye kadar POSIX standardı bu sorunun çözümüne pek yardımcı olmadı. Linux çekirdeği, bir işlemci için ***akrabalık kümeleri*** belirtilmesini mümkün kılan bir arayüz ailesi sağlar. Zamanlayıcı süreç ya da evreyi işlemciler üzerinde belirtilen akrabalık maskesine göre zamanlar. GNU C kütüphanesindeki arayüzler Linux çekirdeğindeki arayüz biraz genişletilerek tanımlanmıştır.

cpu_set_t veri türü

Bu veri türü her biri bir işlemciyi ifade eden bir bit kümesidir. İşlemcilerin bit kümesinin hangi bitleriyle eşleneceği sisteme bağlıdır. Veri türünün genişliği sabittir; sistemdeki işlemcileri ifade etmek için veri türünün genişliğini yetersiz kaldığı durumda başka bir arayüz kullanılmalıdır.

Bu veri türü bir GNU oluşumudur ve `sched.h` dosyasında tanımlanmıştır.

Bit kümesiyle çalışmak, bitleri belirtmek ve sıfırlamak için bazı makrolar tanımlanmıştır. Makroların bazıları parametre olarak bir işlemci numarası alır. Burada önemli olan bit kümesi genişliğinin aşılmamasıdır. Bu makro **cpu_set_t** bit kümesindeki bitlerin sayısını belirler:

`int` **CPU_SETSIZE** makro

Bir **cpu_set_t** nesnesi ile elde edilebilecek işlemci sayısıdır.

cpu_set_t veri türü şeffaf olmadığından bu veri türü ile ilgili çalışmalar aşağıdaki dört makro ile yürütülmelidir.

`void` **CPU_ZERO**`(cpu_set_t *`*küme*`)` makro

Bu makro *küme* işlemci kümesini bir boş küme olarak ilklendirir.

Bu makro bir GNU oluşumudur ve `sched.h` dosyasında tanımlanmıştır.

```
void CPU_SET(int          işlemci,                                  makro
             cpu_set_t *küme)
```

işlemci işlemcisini *küme* işlemci kümesine ekler.

Defalarca işleme sokulacağından *işlemci* parametresinin yan etkilere sebep olmayacak şekilde belirtilmesi önemlidir.

Bu makro bir GNU oluşumudur ve `sched.h` dosyasında tanımlanmıştır.

```
void CPU_CLR(int          işlemci,                                  makro
             cpu_set_t *küme)
```

işlemci işlemcisini *küme* işlemci kümesinden kaldırır.

Defalarca işleme sokulacağından *işlemci* parametresinin yan etkilere sebep olmayacak şekilde belirtilmesi önemlidir.

Bu makro bir GNU oluşumudur ve `sched.h` dosyasında tanımlanmıştır.

```
int CPU_ISSET(int          işlemci,                                  makro
              cpu_set_t *küme)
```

Eğer *işlemci* işlemcisi *küme* işlemci kümesinin bir üyesi ise bu makro sıfırdan farklı bir değerle (doğru), değilse sıfırla (yanlış) döner.

Defalarca işleme sokulacağından *işlemci* parametresinin yan etkilere sebep olmayacak şekilde belirtilmesi önemlidir.

Bu makro bir GNU oluşumudur ve `sched.h` dosyasında tanımlanmıştır.

İşlemci bit kümeleri ya sıfırdan oluşturulur ya da o an kurulu bir akrabalık maskesi olarak sistemden alınır.

```
int sched_getaffinity(pid_t       pid,                                    işlev
                      size_t      kümegenişliği,
                      cpu_set_t *küme)
```

Bu işlev işlemci akrabalık maskesini kimliği *pid* ile belirtilen süreç ya da evre için genişliği *kümegenişliği* bayt olarak belirtilen ve *küme* ile gösterilen bir nesne olarak döndürür. İşlev başarılı olduğunda daima **cpu_set_t** nesnesindeki tüm bitleri ilklendirir ve sıfırla döner.

Eğer, *pid* bir süreç ya da evreye karşılık değilse ya da işlev bir sebeple başarısız olmuşsa **-1** ile döner ve **errno** değişkenine hata durumu atanır.

ESRCH
Belirtilen kimliğe sahip bir süreç ya da evre yok.

EFAULT
küme göstericisi geçerli bir nesneyi göstermiyor.

Bu işlev bir GNU oluşumudur ve `sched.h` dosyasında bildirilmiştir.

Bunun, farklı POSIX evreleri için bilgi almak amacıyla kullanılmasının büyük ihtimalle taşınabilir olmayacağına dikkat çekmek gerekir. Bu durum için başka bir arayüz sağlanmalıdır.

```
int sched_setaffinity(pid_t             pid,                              işlev
                      size_t            kümegenişliği,
                      const cpu_set_t *küme)
```

Bu işlev işlemci akrabalık maskesini kimliği *pid* ile belirtilen süreç ya da evre için genişliği *kümegenişliği* bayt olarak belirtilen ve *küme* ile gösterilen bir nesneye göre belirler. İşlev başarılı olduğunda sıfırla dönecek ve zamanlayıcı bu akrabalık bilgisini gelecekte hesaba katacaktır.

İşlev bir sebeple başarısız olmuşsa **-1** ile döner ve **errno** değişkenine hata durumu atanır.

ESRCH
Belirtilen kimliğe sahip bir süreç ya da evre yok.

EFAULT
küme göstericisi geçerli bir nesneyi göstermiyor.

EINVAL
Bit kümesi geçersiz. Bu, akrabalık kümesindeki bir işlemcinin süreç ya da evre için bırakılmamış olabileceği anlamına gelebilir.

Bu işlev bir GNU oluşumudur ve `sched.h` dosyasında bildirilmiştir.

4. Bellek Özkaynakları

Sistemdeki mevcut belleğin miktarı ve belleğin düzenlenmesi sıklıkla yazılımların yapabileceklerine ve çalışabilmelerine bağlı olarak belirlenir. **mmap** gibi işlevler açısından her bir bellek sayfasının boyutunun bilinmesi ve bir yazılımın ne kadar belleği arabellekler gibi seçimler için kullanabileceğinin bilinmesi gereklidir. Bu tür ayrıntılara dalmadan önce geleneksel Unix sistemlerindeki bellek altsisteminden biraz bahsetmek gerekir.

4.1. Bellek Altsistemi

Unix sistemleri genelde süreçlere sanal adres alanları sunarlar. Bunun anlamı, verinin saklandığı adres alanlarının aslında fiziksel bellek adresleri değil, bunları kapsayan ama bu adreslere doğrudan erişim sağlamayan bellek bölgelerinin adresleri olduğudur. Sanal adreslerin fiziksel adreslere dönüştürüldüğü ek bir dolaylı işlem katmanı vardır ve bu işlemler normalde işlemcinin donanımı tarafından yapılır.

Bir sanal adres alanı kullanmanın çeşitli yararları vardır. En önemlisi süreç yalıtımıdır. Sistemde çalışmakta olan farklı süreçlerin birbirleriyle doğrudan etkileşmemeleri gerekir. Bir sürecin adres alanına başka hiçbir süreç yazamaz (Paylaşımlı bellek kullanımı hariç. Ancak bu da isteğe bağlıdır ve denetim altında yapılır).

Sanal bellek kullanımının diğer bir yararı da süreçlerin adres alanının mevcut fiziksel bellekten daha büyük bir bellek alanı olarak görünmesidir. Fiziksel bellek dış saklama ortamları ile genişletilebilir ve o an kullanılmayan bellek bölgeleri bu ortamlara aktarılabilir. Adres dönüşümü bu bellek bölgelerine erişimi engeller ve hemen bu içeriği fiziksel belleğe geri yükleyerek kullanıma hazır duruma getirir. Kullanılabilir fiziksel bellekle kullanılabilir sanal adres alanı arasındaki farkın bilindiği durumda bu işlem yazılımların belleği kullanabilmesi için gerekli hale gelir. Sistemde çalışmakta olan tüm süreçlerin çalışmalarını sürdürebilmeleri için fiziksel belleğin yetersiz kaldığı ve dış saklama ortamlarının hemen hemen dolduğu durumlarda bu iki ortam arsındaki takaslama işlemi küçük miktarlarda olmaya başlar ve bu sistemin belirgin biçimde yavaşlamasına sebep olur. Buna ***atıştırma*** (thrashing) denir (argosu: çöplenme).

Sanal bellek hakkında söylenebilecek son şey, önceki paragrafta bahsedilen sanal belleğin takaslanma büyüklüğü ile ilgilidir. Bu takaslama işlemi bayt bayt yapılmaz. Yönetsel karar organı bunun olmasına izin vermez (işlemci donanımı naparsan yap deyip bırakılmaz). Bunun yerine birkaç bin baytlık ***sayfa*** olarak nitelenen belirli miktarlarla bu işlem yapılır. Her sayfanın genişliği ikinin üstel katları olarak bayt cinsinden belirlenir. Günümüzdeki en küçük sayfa genişliği 4096 bayt olup, 8192, 16384 ve 65536 baytlık sayfa genişlikleri de görülmektedir.

4.2. Bellek Parametrelerinin Sorgulanması

Sürecin sanal bellek sayfa genişliğini bilmesi bazı durumlarda zorunludur. Bazı yazılım arayüzleri (**mmap** gibi, bkz. *Bellek Eşlemli G/Ç* (sayfa: 319)) kullanıcının sayfa genişliğine ayarlanmış bilgi vermesini gerektirir. **mmap**, sayfa genişliğinin katları olarak bir uzunluk argümanı gerektirir. Sayfa genişliğinin bilinmesinin faydalı olduğu diğer bir yer de bellek ayırmadır. Uygulama tarafından bölünerek kullanılmak üzere büyükçe bir tomar halinde bir ayırma yapılırsa, geniş blokların boyutlarının sayfa genişliğine ayarlanması yararlı olur. Çekirdek bellekle çalışırken sadece tamamı kullanılan bellek sayfalarını ayırmak zorunda olduğundan, ayrılmak istenen blok boyutunun sayfa genişliğinin katlarına yakın (daha büyük değil) olması çekirdeğin bellek ayırma ile ilgili olarak daha verimli çalışabilmesini sağlar. (Bu eniyilemeyi yapmak için bellek ayırıcının her blok için belleğin bir biti için bile nasıl davrandığı hakkında biraz birşeyler bilinmesi ve sayfa genişliğinin katlarını aşan bir toplam boyut talep edilmemesi gerekir.)

Sayfa genişliği geleneksel olarak bir derleme zamanı sabitidir. Fakat son zamanlarda geliştirilen işlemcilerle bu durum değişmiştir. İşlemciler artık farklı sayfa genişliklerini desteklemekte ve aynı sistem üzerinde farklı süreçler arasında bunun değişiklik göstermesine bile olanak verebilmektedirler. Bu nedenle, çalışma anında sistem o anki sayfa genişliği hakkında sorgulanmalı ve sayfa genişliği ile ilgili hiçbir önkabul yapılmamalıdır (sayfa genişliğinin ikinin üstel katları olması gerekliliği hariç).

Sayfa genişiğini sorgulamak için kullanılacak doğru arayüz **_SC_PAGESIZE** parametresi ile **sysconf**'tur (bkz. *Sysconf Tanımı* (sayfa: 787)). Ayrıca daha eski bir arayüz de vardır.

```
int getpagesize(void)                                                    işlev
```

getpagesize işlevi sürecin sayfa genişliği ile döner. Bu değer sürecin çalışması süresince sabittir ama aynı yazılımın farklı süreçlerinde farklı değerler olabilir.

Bu işlev `unistd.h` dosyasında bildirilmiştir.

Sistemin fiziksel belleği hakkında System V'den türetilmiş sistemlerde geniş olarak kullanılan bir bilgi alma yöntemi vardır.

```
sysconf (_SC_PHYS_PAGES)
```

çağrısı sistemin sahip olduğu belleğin fiziksel sayfalarının toplam sayısı ile döner. Bu, bu belleğin tümünün kullanılabilir olduğu anlamına gelmez. Bu bilgi,

```
sysconf (_SC_AVPHYS_PAGES)
```

çağrısı ile edinilebilir. Bu iki değer uygulamaların eniyilenmesine yardımcı olur. **_SC_AVPHYS_PAGES** için döndürülen değer uygulamanın başka bir süreç tarafından engellenmeksizin kullanabileceği bellek miktarıdır (başka hiçbir sürecin kendi bellek kullanımını, diğerlerinin hilafına arttıramayacağını belirtir). **_SC_PHYS_PAGES** için dönen değer ise az ya da çok çalışma birliği için donanımsal bir sınırdır. Eğer tüm uygulamalar birlikte bundan fazlasını kullanmaya çalışırsa, sistemin bellek miktarı ile başı dertte demektir.

GNU C kütüphanesi bu iki yönteme ek olarak bu bilgiyi almak için iki işlev daha içerir. Bu işlevler `sys/sysinfo.h` dosyasında bildirilmiştir. Yazılımcılar yukarıda açıklanan **sysconf** yöntemini tercih etmelidir.

```
long int get_phys_pages(void)                                    işlev
```

get_phys_pages işlevi sistemin sahip olduğu fiziksel sayfaların sayısı ile döner. Belleğin toplam miktarını bulmak için bu değer sayfa genişliği ile çarpılır.

Bu işlev bir GNU oluşumudur.

```
long int get_avphys_pages(void)                                  işlev
```

get_phys_pages işlevi sistemin sahip olduğu fiziksel sayfalardan kullanılabilir olanlarının sayısı ile döner. Belleğin kullanılabilir toplam miktarını bulmak için bu değer sayfa genişliği ile çarpılır.

Bu işlev bir GNU oluşumudur.

5. İşlemci Özkaynakları

Evrelerin ve süreçlerin paylaşımlı bellekle kullanımı bir uygulamanın bir sistemin sağlayabildiği tüm işlem gücünün getirilerinden yararlanmasına imkan verir. Eğer görev paralelleştirilebiliyorsa, bir uygulama yazmanın en uygun yolu aynı anda çok sayıda işlemci varmış gibi çok sayıda sürecin çalışmasının mümkün olması ile ilgilidir. Sistemdeki mevcut işlemcilerin sayısını saptamak için şöyle bir çağrı yapabilirsiniz:

```
sysconf (_SC_NPROCESSORS_CONF)
```

İşletim sisteminin yapılandırıldığı işlemcilerin sayısı ile döner. Ancak işletim sisteminin bazı işlemcileri iptal etmesi mümkün olduğundan,

```
sysconf (_SC_NPROCESSORS_ONLN)
```

çağrısı ile o an kullanılabilecek işlemcilerin sayısını öğrenebilirsiniz.

Bu ikisine ek olarak GNU C kütüphanesi bilgiyi doğrudan almayı mümkün kılan işlevler de içerir. Bu işlevler `sys/sysinfo.h` dosyasında bildirilmiştir.

```
int get_nprocs_conf(void)                                        işlev
```

get_nprocs_conf işlevi işletim sisteminin yapılandırıldığı işlemcilerin sayısı ile döner.

Bu işlev bir GNU oluşumudur.

```
int get_nprocs(void)                                                    işlev
```

get_nprocs işlevi o an kullanılabilecek işlemcilerin sayısı ile döner.

Bu işlev bir GNU oluşumudur.

Daha fazla evre başlatmadan önce işlemcilerin tamamen kullanımda olup olmadığına bakılmalıdır. Unix sistemi ***yük ortalaması*** adı verilen bir hesaplama yapar. Bu aynı anda kaç sürecin çalışmakta olduğunu gösteren bir sayıdır. Bu sayı farklı sürelerde alınan bir ortalamadır (normalde 1, 5 ve 15 dakikalık).

```
int getloadavg(double yükortalamaları[],                                işlev
               int    elem_sayısı)
```

İşlev sistemin yük ortalamasını 1, 5 ve 15 dakikalık ortalamalar olarak döndürür. Değerler *yükortalamaları* dizisine yerleştirilir. Diziye kaç ortalama yerleştirileceği *elem_sayısı* argümanına konur, ancak bu değer üçten fazla olmaz. İşlevin normal dönüş değeri *yükortalamaları* dizisine yazılan değerlerin sayısıdır, –1 dönmüşse bir hata oluşmuş demektir.

Bu işlev `stdlib.h` dosyasında bildirilmiştir.

XXIII. Yerel Olmayan Çıkışlar

İçindekiler

Bazan yazılımınızda çok iç içe işlev çağrıları yaptığınızda ve derinlerde bir yerde olumsuz bir durum oluştuğunda denetimi daha dış bir düzeye aktarmak ihtiyacı duyarsınız. Bu oylumda **setjmp** ve **longjmp** çağrılarıyla böyle ***yerel olmayan çıkışlar***ın nasıl yapılacağından bahsedilecektir.

1. Yerel Olmayan Çıkışlar Hakkında

Yerel olmayan çıkışların ne zaman faydalı olabileceğini bir örnekle açıklamaya çalışmak daha iyi olacak. Bir ana döngüyle kullanıcıdan komutları alıp bunları çalıştıran etkileşimli bir yazılımımız olduğunu varsayalım. Komutları bir dosyadaki girdilerden okuduğunu ve girdideki komutu işleme sokmadan önce bazı metin çözümleme işlemleriyle girdiyi analiz ettiğini farzedelim. Bir düşük seviyeli hata saptandığında, metin çözümleme, ayrımsama ve işleme fazlarının her yapılışında iç içe çağrılarda saptanan hatalarla yerinde uğraşmaktansa hemen ana döngüye dönebilmek iyi olurdu.

(Diğer taraftan, her fazdan çıkışta önemli miktarda temizlik işlemleri yapılması gerekli olabilir—örneğin, dosyaların kapatılması, tamponların ve veri yapılarının serbest bırakılması ve benzerleri—ancak bundan sonra normal dönüş yapmak daha uygun olabilir. Bir yerel olmayan çıkış ara fazların ve onlarla ilgili temizlik kodunun atlanmasına neden olacağından her fazın kendine özgü temizlik kodu olması da daha uygun olabilir. Bundan başka, temizliği ana döngüye dönmeden önce ya da döndükten sonra yapacak şekilde de bir yerel olmayan çıkış kullanabilirdiniz.)

Bazı bakımlardan, bir yerel olmayan çıkış, bir işlevden çıkış deyimi olan **return** kullanımına benzer. Fakat **return** sadece tek bir işlev çağrısından çıkıp denetimi geriye işlev çağrısının yapıldığı noktaya taşırken, bir yerel olmayan çıkış denetimin potansiyel olarak çok iç içe pek çok işlev çağrısının dışındaki bir noktaya taşınmasını sağlar.

Yerel olmayan çıkışlarda denetimin döndürüleceği noktayı **setjmp** işlevini çağırarak belirtebilirsiniz. Bu işlev, **setjmp** işlev çağrısının göründüğü icra ortamı hakkındaki bilgiyi **jmp_buf** türündeki bir nesneye kaydeder. Yazılımın çalışması **setjmp** çağrısından sonra normal olarak devam eder, fakat daha sonra bu dönüş noktası için kaydedilen **jmp_buf** türündeki nesne ile yapılan bir **longjmp** çağrısı ile bu dönüş noktasına bir çıkış yapılırsa, denetim **setjmp** çağrısının yapıldığı noktaya aktarılmış olur. **setjmp** çağrısından dönen değer ile sıradan bir dönüş ve **longjmp** çağrısı tarafından yapılan bir dönüş arasında ayrım yapılabilmesi için **setjmp** çağrısının bir **if** deyimi içinde görünmesi gerekir.

Bunun nasıl yapıldığına bir örnek:

```
#include <setjmp.h>
#include <stdlib.h>
#include <stdio.h>

jmp_buf main_loop;

void
abort_to_main_loop (int status)
```

```
{
  longjmp (main_loop, status);
}

int
main (void)
{
  while (1)
    if (setjmp (main_loop))
      puts ("Ana döngüye geri dönüldü....");
    else
      do_command ();
}


void
do_command (void)
{
  char buffer[128];
  if (fgets (buffer, 128, stdin) == NULL)
    abort_to_main_loop (-1);
  else
    exit (EXIT_SUCCESS);
}
```

abort_to_main_loop işlevi nereden çağrıldığına bakılmaksızın, denetimin, yazılımın ana döngüsüne geri dönmesine sebep olur.

main işlevinin içindeki akış denetimi başta biraz esrarlı görünebilir. Bir normal **setjmp** çağrısı sıfırla döner, böylece "else" sözcüğündeki kod çalışır. **do_command** içinde bir yerlerde **abort_to_main_loop** yapılırsa, hemen ardından **main** içinde **-1** değeri döndüren ikici bir **setjmp** çağrısı yapılmış gibi görünür.

setjmp kullanımını genel kalıbı şöyle görünür:

```
if (setjmp (tampon))
  /* Erken dönüş sonrası temizleme kodu. */
  …
else
  /* Dönüş noktası ayarlandıktan sonra
     normal olarak çalıştırılacak kod. */
  …
```

2. Yerel Olmayan Çıkışların Ayrıntıları

Burada yerel olmayan çıkışları gerçekleştirmekte kullanılan veri yapıları ve işlevler ayrıntılı olarak incelenecektir. Bu oluşumlar `setjmp.h` dosyasında bildirilmiştir.

jmp_buf veri türü

jmp_buf türündeki nesneler bir yerel olmayan çıkış tarafından eski durumuna getirilecek durum bilgisini tutar. Bir **jmp_buf** içeriği dönülecek yeri belirtir.

`int` **setjmp**`(jmp_buf` *durum*`)` makro

Normal olarak çağrıldığında, **setjmp** işlevi yazılımın icra durumu hakkında bilgiyi *durum* nesnesine kaydeder ve sıfırla döner. Daha sonra bir yerel olmayan çıkış gerçekleştirmek için bu *durum* bilgisi ile bir **longjmp** çağrısı yapılırsa, **setjmp** sıfırdan farklı bir değerle döner.

```
void longjmp(jmp_buf durum,                                          işlev
             int     değer)
```

Bu işlev o anki çalışma durumunu *durum* nesnesinde kayıtlı durum ile değiştirerek icranın dönüş noktasını oluşturan **setjmp** çağrısından devam etmesini sağlar. **setjmp**'ın dönüş durumu **0** yerine, **longjmp** çağrısının *değer* argümanında belirtilen değer olur. (Ancak, eğer *değer* olarak sıfır verilmişse, **setjmp** işlevi **1** ile döner).

setjmp ve **longjmp** kullanımında önemli sınırlamalar getirmeyen bazı karanlık noktalar vardır. Bu sınırlamaların çoğu hala vardır, çünkü yerel olmayan çıkışlar C derleyicisinin bir kısmının biraz sihirli olmasını ve dilin diğer parçaları ile tuhaf bir şekilde etkileşmesini gerektirir.

setjmp işlevi aslında bir işlev tanımı olmaksızın bir makrodur, yani onu **#undef** yapamazsınız ve adresini alamazsınız. Ek olarak, **setjmp** çağrıları sadece aşağıdaki bağlamla güvenlidir.

- Bir seçim ya da yineleme deyiminin sınama ifadesi olarak. (örneğin, **if**, **switch** veya **while**).
- Bir seçim ya da yineleme deyiminin sınama ifadesinde görünen bir eşitlik ya da karşılaştırma işlecinin terimi olarak. Diğer terim bir tamsayı sabit ifadesi olmalıdır.
- Bir seçim ya da yineleme deyiminin sınama ifadesinde görünen bir tek terimli **!** işlecinin terimi olarak.
- Bir ifade deyimi olarak kendisi tarafından.

Geri dönüş noktaları, sadece bu dönüş noktalarını oluşturan **setjmp**'ı çağıran işlevin (örnekteki, **main** işlevi) çalışması boyunca geçerlidirler. Eğer zaten dönmüş bir işlevde kurulu bir dönüş noktasına **longjmp** yapılırsa bunun sonuçları öngörülemez ve yıkıcı etkileri olabilir.

longjmp çağrısında *değer* argümanına sıfırdan farklı bir değer belirtmelisiniz. **longjmp**'ın, **setjmp**'dan dönecek değer olarak, **setjmp**'a sıfır değerini aktarmayı reddetmesi, aslında kaza ile oluşacak bir kötü kullanıma karşı oluşturulmuş bir güvencedir.

Bir yerel olmayan çıkış yaptığınızda, erişilebilir nesneler genellikle **longjmp** çağrısı yapıldığı sıradaki değerlerinde kalırlar. Bunun istisnası, **setjmp** çağrısını içeren işlevin yerel özdevinimli değişkenlerinin değerlerinin, **setjmp** çağrısının belirsizliğinden dolayı değişmesidir. Bunun olmaması için onları **volatile** olarak bildirebilirsiniz.

3. Yerel Olmayan Çıkışlarda Sinyaller

BSD Unix sistemlerinde, **setjmp** ve **longjmp** işlevleri *engellenen sinyalleri* (sayfa: 631) kaydedebilir ve daha sonra onları eski durumuna getirebilir. Ancak, POSIX.1 standardı **setjmp** ve **longjmp** gerçeklemesinin engellenen sinyalleri değiştirmemesini gerektirir ve BSD davranışını elde etmek için ek bir işlev çifti (**sigsetjmp** ve **siglongjmp**) sağlar.

setjmp ve **longjmp** işlevlerinin davranışı GNU C kütüphanesinde *özellik sınama makroları* (sayfa: 25) ile denetlenir. GNU sisteminde öntanımlı olan BSD davranışı değil POSIX.1 davranışıdır.

Bu kısımdaki oluşumlar `setjmp.h` başlık dosyasında bildirilmiştir.

```
sigjmp_buf                                                           veri türü
```

Bu veri türü, ayrıca engellenen sinyallerin durum bilgilerinin de kaydedilmesini sağlaması dışında **jmp_buf**'a benzer.

```
int sigsetjmp(sigjmp_buf durum,                                      işlev
              int        sinyal_kaydet)
```

setjmp işlevine benzer. Eğer *sinyal_kaydet* argümanının değeri sıfırdan farklıysa, engellenen sinyaller daha sonraki bir **siglongjmp** çağrısı ile yerine konmak üzere *durum* nesnesine kaydedilir.

```
void siglongjmp(sigjmp_buf durum,                                    işlev
                int         değer)
```

durum argümanının türü dışında **longjmp** işlevine benzer. Eğer **sigsetjmp** çağrısında sıfırdan farklı bir *sinyal_kaydet* argümanı belirtilerek engellenen sinyaller bu *durum* nesnesine kaydedilmişse, bu çağrı ayrıca onları da eski durumlarına getirir.

4. Bütünsel Bağlam Denetimi

Unix standardında var olan, yürütme yolunu denetleyen bir diğer işlev grubu. Bu işlevler bu bölümde buraya kadar anlatılanlardan daha güçlü olup en baştan beri System V API içinde bulunmaktadırlar ve buradan Unix API'sine aktarılmışlardır. Markalı Unix gerçekleştirimleri dışında bu arabirimlere fazla rastlanmaz. GNU C'nin bulunduğu bütün platformlar ve mimarilerde de yer almazlar. Var olup olmadıklarını **configure** kullanarak anlayabilirsiniz.

longjmp işlevinin durumunu içeren değişkenler için kullanılan **jmp_buf** ve **sigjmp_buf** türlerine benzer şekilde, burada anlatılacak olan arabirimler için de uygun türler tanımlanmıştır. Bu türteki nesneler daha fazla bilgi içerdiklerinden dolayı daha büyüktürler. Bu tür ileride göreceğimiz birkaç yerde daha kullanılmıştır. Bu bölümde anlatılan veri türleri ve işlevlerin tümü `ucontext.h` başlık dosyasında tanımlanmış ve bildirilmiştir.

```
ucontext_t                                                           veri türü
```

ucontext_t yapısı en azından aşağıdaki üyeleri içermelidir:

`ucontext_t *`**uc_link**
: Sonraki bağlam yapısına göstericidir. Eğer mevcut yapıda tanımlanan bağlamdan çıkılmışsa kullanılır.

`sigset_t` **uc_sigmask**
: Bu bağlam kullanıldığında engellenen sinyalleri içerir.

`stack_t` **uc_stack**
: Bu bağlam için kullanılan yığıt. Değeri bir yığıt göstericisi olmak zorunda değildir ve normalde de değildir. Bkz. *Sinyal Yığıtı* (sayfa: 639).

`mcontext_t` **uc_mcontext**
: Bu üye sürecin mevcut durumunu içerir. **mcontext_t** türü de bu başlıkta tanımlanır fakat bu tanımın geçirimsiz olduğu düşünülmelidir. Bu tür bilinerek geliştirilmiş uygulamaların taşınırlığı daha düşüktür.

Bu türden nesneler kullanıcı tarafından oluşturulmalıdır. İlklendirme ve değişiklik işlemleri için aşağıdaki işlevlerden biri kullanılır:

```
int getcontext(ucontext_t *bağlam)                                   işlev
```

getcontext işlevi, kendisini çağıran evre bağlamında *bağlam* ile gösterilen nesneyi ilklendirir. Buradaki bağlam, yazmaçlar, sinyal maskesi ve mevcut yığıta ait içerikleri barındırır. İçeriklerin icrası **getcontext** çağrısı döndüğü anda başlar.

İşlev eğer başarılıysa **0** değerini döndürür. Değilse **-1** döndürür ve *errno*'ya uygun değeri atar.

getcontext işlevi **setjmp**'a benzer ancak işlevin ilk kez mi döndüğü yoksa ilklendirilmiş bağlam kullanılıp yürütmeye o noktadan mı geri dönüldüğü konusunda bilgi vermez. Eğer bu ayrımın yapılması gerekliyse, bunu kullanıcı kendisi ortaya çıkarmalıdır. Ancak bunu dikkatlice yapmak gerekir çünkü bağlam içerisinde yazmaç değişkenlerini içeren yazmaçlar bulunabilir. Bu durumda değişkenleri **volatile** olarak tanımlamak iyi olur.

bağlam nesnesine bir kez ilk değer atandıktan sonra ya olduğu gibi ya da değiştirilerek kullanılır. Değiştirme normal olarak eş–yordamları veya benzeri yapıları gerçekleştirmek için yapılır. **makecontext** bunu yapmak kullanılır.

```
void makecontext(ucontext_t *bağlam,                                    işlev
                 void       (*işlev) (void),
                 int          argc,
                 …)
```

İşleve aktarılan *bağlam* parametresi **getcontext** tarafından ilklendirilmiş olmalıdır. Sonuçta elde edilen bağlama geri dönüldüğünde yapılan ilk işlem, *argc* tane tamsayı argüman ile *işlev* işlevinin çağrılmasıdır. **makecontext** çağrısında tamsayı argümanlar *argc* parametresinden sonra verilmelidir.

Bu işlev çağrılmadan önce *bağlam* yapısına ait **uc_stack** ve **uc_link** üyeleri ilklendirilmelidir. **uc_link** üyesi bu bağlam için kullanılan yığıtı tanımlar. Aynı anda kullanılan bağlamların her biri yığıt için ayrı bellek bölgeleri kullanmalıdır.

bağlam'ın gösterdiği nesnenin **uc_link** üyesi, *işlev* işlevinin geri dönüş noktasında yürütülecek bağlamı göstermeli veya bir boş gösterici olmalıdır. Kullanımı hakkında daha fazla bilgi için **setcontext**'e bakınız.

Yığıt için bellek ayırırken dikkatli olunmalıdır. Günümüzde işlemcilerin çoğunluğu bir bellek bölgesinin çalıştırılabilir kod içerip içeremeyeceği konusunda ayrım yapar. Veri bölütleri ve özdevimli ayırma yapılan bellekte bu tür bir yaftalama yoktur. Sonuçta yazılımlar başarısız olur. Bu tür çalıştırılabilir koda bir örnek, iç içe işlev çağrıları için GNU C derleyicisinin ürettiği çağrı dizilimleridir. Yığıtlar için güvenli bir şekilde bellek ayırmak için örneğin özgün evre yığıtı üzerindeki bellek kullanılabilir veya çalıştırılmaya uygun şekilde yaftalanmış bir bellek bölgesi ayırılabilir (bkz. *Bellek Eşlemli G/Ç* (sayfa: 319)).

Uyumluluk Bilgisi

Mevcut Unix standardı yığıt için bellek ayırma konusunda hemen hemen hiç belirleyici değildir. Bütün gerçekleştirimler **uc_stack** üyesinin kullanımında anlaşmıştır ama **stack_t** değerinin üyelerinin içerebileceği değerler konusunu açık bırakmıştır. GNU C kütüphanesi ve diğer Unix gerçekleştirimlerinin çoğunluğunda **uc_stack** üyesinin **ss_sp** değeri, yığıt için ayrılmış bellek bölgesinin tabanını göstermelir ve **ss_size** değeri de bu bölgenin büyüklüğünü içermelidir. Bazı gerçekleştirimlerde ise **ss_sp** değeri yığıt göstericisinin değerini içerir (yığıtın gelişme yönüne göre bu değer farklıdır). Bu farklılık **makecontext** işlevinin kullanımını zorlaştırır ve derleme sırasında platformn bilgisinin elde edilip kullanılmasını zorunlu kılar.

```
int setcontext(const ucontext_t *bağlam)                                işlev
```

setcontext işlevi, *bağlam* tarafından tanımlanmış bağlamı yeniden geçerli kılar. Bağlamda değişiklik yapılmaz ve istendiği kadar sık yeniden kullanılabilir.

Eğer bağlam **setcontext** ile oluşturulmuşsa yürütme sonunda yazmaçlar, **setcontext** sanki şimdi geri dönüş yapmış gibi aynı değerleri içerir.

Eğer **makecontext**'in çağrılmasıyla bağlamda değişiklik yapılmışsa **makecontext**'in çağrıldığı işlevle yürütmeye devam edilir ve bu işleve aynı çağrı sırasında verilen paramatreler aktarılır. Bu işlev geri dönüş yaptığında **makecontext** çağrılırken verilen bağlam yapısının **uc_link** üyesi tarafından gösterilen bağlamla yürütmeye devam edilir. Eğer **uc_link** bir boş gösterici ise, bu durumda uygulama sonlanır.

Bağlam yığıt hakkında bilgi içerdiğinden aynı bağlamı aynı anda iki evre kullanmamalıdır. Aksi takdirde, sonuç çoğu durumda felaket olurdu.

setcontext işlevi bir hata oluşmadıkça dönmeyecekir, hata oluşmuşsa **-1** değeri ile döner.

setcontext işlevi mevcut bağlamı basitçe *bağlam* parametresi ile tanımlayarak değiştirir. Mevcut bağlamın korunmasını gerektiren durumlar da olmasına rağmen bu çoğunlukla kullanışlıdır.

```
int swapcontext(ucontext_t *restrict       diğer–bağlam,                    işlev
                const ucontext_t *restrict bağlam)
```

swapcontext işlevi **setcontext** işlevine benzer, fakat geçerli bağlam olan *bağlam* bağlamını bir **getcontext** çağrısından dönmüş olan *diğer–bağlam* ile değiştirir. **swapcontext** çağrısında sonra yürütme bu bağlamla devam eder.

Eğer **swapcontext** başarılıysa işlev dönmez. *diğer–bağlam* bağlamı evvelki bir **makecontext** çağrısı ile değişiklik yapılmaksızın kullanışmışsa dönüş değeri **0**'dır. Eğer işlev başarısız olmuşsa **-1** döner ve *errno* değişkenine hata durumu atanır.

4.1. SVID Bağlam Denetimi Örneği

Bağlam işleme işlevlerini kullanmanın en kolay yolu **setjmp** ve **longjmp** işlevlerinin yerine bunları kullanmaktır.Bağlamın çoğu platformda daha az sürprizle sonuçlanan daha fazla bilgi içermesine rağmen bu işlevlerin kullanımı daha masraflıdır (daha az taşınabilir olması cabası).

```
int
random_search (int n, int (*fp) (int, ucontext_t *))
{
  volatile int cnt = 0;
  ucontext_t uc;

  /* Geçerli bağlamı güvene alalım.  */
  if (getcontext (&uc) < 0)
    return -1;

  /* Henüz n deneme yapmamışsak, tekrar deneyelim. */
  if (cnt++ < n)
    /* İşlevi yeni bir rasgele sayı ve bağlamla çağıralım. */
    if (fp (rand (), &uc) != 0)
      /* Aradığımızı bulduk.  */
      return 1;

  /* Bulamadık.  */
  return 0;
}
```

Bağlamların böyle bir yolla kullanımı olağandışılıkların elde edilme benzeşimini etkinleştirir. *fp* parametresi ile aktarılan arama işlevleri çok büyük, iç içe ve çağrıcıya aktarılan bir hata değeri ile işlevin bırakılması onu karmaşıklaştıracağından (veya en azından biraz daha kod gerekeceğinden), çok karmaşık olabilir. Bağlamı kullanarak arama işlevini tek bir adımda bırakmak ve ayrıca belirgin biçimde daha hızlı olabilen bir yan etkiyle aramanın yeniden başlatılmasına izin vermek mümkündür.

Geçici olarak farklı bir yürütme noktasına geçmek ve sonra yürütmenin durduğu yerden devam etmek gibi bazı şeylerin **setjmp** ve **longjmp** ile gerçeklenmesi daha zordur.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include <sys/time.h>

/* Buna sinyal eylemci atama yapacak. */
static volatile int expired;

/* Bağlamlar. */
static ucontext_t uc[3];

/* Biz belli sayıda geçiş yapacağız. */
static int switches;


/* Bu işi yapan işlev.  Sadece bir iskelet,
   gerçek kod sonra yerleştirilecek. */
static void
f (int n)
{
  int m = 0;
  while (1)
    {
      /* İşin yapıldığı yer. */
      if (++m % 100 == 0)
        {
          putchar ('.');
          fflush (stdout);
        }

      /* Değişkenin zaman aşımına uğrayıp uğramadığına
         düzenli olarak bakmak lazım. */
      if (expired)
        {
          /* Kodun daha fazla çalışmasını istemiyoruz. */
          if (++switches == 20)
            return;

          printf ("\n%d. bağlamdan %d. bağlama geçiliyor\n", n, 3 - n);
          expired = 0;
          /* Diğer bağlama geçip şimdikini kaydedelim. */
          swapcontext (&uc[n], &uc[3 - n]);
        }
    }
}

/* Sadece değişkene değer atayan bir sinyal eylemci bu. */
void
handler (int signal)
{
  expired = 1;
}
```

```
int
main (void)
{
  struct sigaction sa;
  struct itimerval it;
  char st1[8192];
  char st2[8192];

  /* Zamanlayıcının kullanacağı veri yapılarını ilklendirelim. */
  sa.sa_flags = SA_RESTART;
  sigfillset (&sa.sa_mask);
  sa.sa_handler = handler;
  it.it_interval.tv_sec = 0;
  it.it_interval.tv_usec = 1;
  it.it_value = it.it_interval;

  /* Zamanlayıcıyı kuralım ve çalışacağımız bağlamı alalım. */
  if (sigaction (SIGPROF, &sa, NULL) < 0
      || setitimer (ITIMER_PROF, &it, NULL) < 0
      || getcontext (&uc[1]) == -1
      || getcontext (&uc[2]) == -1)
    abort ();

  /* Bağlamı, f işlevinin 1 parametresi ile çağrılmasına
     sebep olan ayrı bir yığıt ile oluşturalım.
     uc_link'in, işlev döndüğü anda yazılımın sonlanmasına
     sebep olan ana bağlamı gösterdiğine dikkat edin. */
  uc[1].uc_link = &uc[0];
  uc[1].uc_stack.ss_sp = st1;
  uc[1].uc_stack.ss_size = sizeof st1;
  makecontext (&uc[1], (void (*) (void)) f, 1, 1);

  /* Aynı ama f'ye parametre olarak 2 aktarılıyor. */
  uc[2].uc_link = &uc[0];
  uc[2].uc_stack.ss_sp = st2;
  uc[2].uc_stack.ss_size = sizeof st2;
  makecontext (&uc[2], (void (*) (void)) f, 1, 2);

  /* İşbaşı! */
  swapcontext (&uc[0], &uc[1]);
  putchar ('\n');

  return 0;
}
```

Bu kod, bağlam işlevlerinin eş–yordamları veya çok evreli işbirliğini gerçekleştirmekte kullanılabilirliğine bir örnektir. Burada yapılan **swapcontext** kullanarak yürütmenin her seferinde farklı bir bağlamdan devam ettirilmesidir. Sinyal eylemci içinden ne **setcontext** ne de **swapcontext** çağrısı yapıldığından, bağlam değiştirme doğrudan sinyal eylemci tarafından yapılmamaktadır. Bunun yerine sinyal eylemci içinde bir değişkene değer atanıp, işlev içinden bu değişkene bakarak bu işlem gerçekleştirilmektedir. **swapcontext** geçerli bağlamı kaydettiğinden kod içinde farklı zamanlama noktaları olabilir. Yürütme daima kaldığı yerden devam edecektir.

XXIV. Sinyal İşleme

İçindekiler

Bir ***sinyal*** bir sürece gönderilen bir yazılım kesmesidir. İşletim sistemi, sinyalleri, çalışan bir yazılıma olağandışı durumları raporlamakta kullanır. Bazı sinyaller geçersiz bellek adreslerine erişim gibi durumlarda hata raporlamakta, bazıları da bir telefon hattının kapanması gibi rasgele olayları raporlamakta kullanılır.

GNU C kütüphanesi her biri başka bir çeşit olaya karşılık olmak üzere çeşitli sinyal türleri tanımlar. Bazı olaylar, bir yazılımı çalışmasını imkansız kılabilir, bu tür olayları raporlayan sinyallar yazılımın çalışmasını durdurmasına sebep olur. Diğer sinyal çeşitleri zararsız olayları raporlar ve bunlar öntanımlı olarak yoksayılır.

Bir olayın sinyallere sebep olacağını umuyorsanız, sinyalle tetiklenen bir işlev tanımlayıp, işletim sistemine böyle sinyaller geldiğinde bu işlevi çalıştırmasını belirtebilirsiniz.

Son olarak, bir süreç başka bir sürece sinyal gönderebilir; bu bir sürecin kendi alt sürecini durdurması gerektiğinde ya da birbiriyle haberleşerek eşzamanlı çalışması gereken süreçler arasında kullanılabilir.

1. Sinyallerle İlgili Temel Kavramlar

Bu kısımda sinyallerin nasıl üretildiği, bir sinyal alındıktan sonra neler olduğu ve yazılımlarda sinyallerin nasıl işlendiği gibi konularla ilgili kavramlara değinilecektir.

1.1. Bazı Sinyal Çeşitleri

Bir sinyal olağandışı bir olayın varlığını raporlar. Bir sinyale sebep olan (üreten ya da ortaya çıkaran) olayların bazıları:

- Sıfırla bölme ya da geçerli bir aralık dışında bir adres gösterme gibi yazılım hataları.
- Kullanıcı tarafından yazılımın durdurulmak ya da sonlandırılmak istenmesi. Çoğu ortam kullanıcıya **C-z** tuşlayarak uygulamayı durdurabilme veya **C-c** tuşlayarak uygulamayı sonlandırabilme imkanı sağlar. Bu tuş vuruşları algılandığında işletim sistemi sürece bu isteği belirten bir sinyal gönderir.
- Bir alt sürecin sonlanması.
- Alarm veya zamanlayıcının zamanaşımına uğraması.
- Aynı süreç tarafından yapılan bir **kill** veya **raise** çağrısı.
- Başka bir süreç tarafından yapılan bir **kill** çağrısı; sinyallerin süreçler arası iletişim için sınırlı ama kullanışlı biçimidir.
- Yapılamayacak bir G/Ç işlemini yapmaya çalışma. Örneğin, bir ucuna yazma yapılmayan bir boruyu okumaya çalışmak (bkz, *Borular ve FIFOlar* (sayfa: 393)), bazı durumlarda bir uçbirime yazmaya ya da okumaya çalışmak (bkz, *İş Denetimi* (sayfa: 716)).

Bu olayların her biri (açıkça yapılan **kill** ve **raise** çağrıları dışında) kendine özel bir sinyal üretir. Sinyal çeşitleri *Standart Sinyaller* (sayfa: 604) bölümünde listelenmiş ve açıklanmıştır.

1.2. Sinyal Üretimi İle İlgili Kavramlar

Genellikle, sinyalleri üreten olaylar üç ana sınıf altında incelenir: hatalar, dış olaylar, doğrudan yapılan istekler.

Bir hata, bir uygulamanın bazı şeyleri yanlış yaptığını ve çalışmasını sürdüremeyeceği bir durumu anlatır. Fakat, hata çeşitlerinin hepsi sinyal üretmez (aslında bu çoğu için geçerlidir). Örneğin, mevcut olmayan bir dosya bir hatadır ama bir sinyal üretmez, sadece **open** işlevi **-1** ile döner. Genelde hatalar kütüphane işlevleri ile ilişkilidir ve işlevler bir hata oluştuğunda hatayı bir değerle raporlarlar. Sinyalleri ortaya çıkaran hatalar sadece kütüphane çağrılarında değil yazılımda herhangi bir yerde oluşabilir. Bunlar sıfırla bölme, geçersiz bir bellek adresi olabilir.

Bir dış olay genelde G/Ç işlemleri ya da başka süreçlerle ilgilidir. Bunlar, bir girdinin alınması, bir zamanlayıcının zamanaşımına uğraması ve bir alt sürecin sonlanması olabilir.

Doğrudan yapılan istekler, amacı özellikle sinyal üretmek olan **kill** gibi bir kütüphane işlevinin kullanılmasıyla yapılır.

Sinyaller *eşzamanlı olarak* ya da *herhangi bir anda* üretilebilir. Bir eşzamanlı sinyal, yazılım içindeki belirli bir eylemle ilişkilidir ve (engellenmedikçe) bu eylem sırasında oluşur. Çoğu hatalar sinyalleri eşzamanlı üretir, öyle ki, kendisi için sinyal üretecek bir süreç bazan bunu bilinçli olarak yapar. Bazı makinalarda belli bir takım donanım hataları (genellikle gerçek sayılarla ilgili olağandışılıklar) tamamen eşzamanlı üretilirler, fakat ardından bir kaç makina komutu gelmelidir.

Herhangibir anda üretilen sinyaller onları alan sürecin denetimi dışındaki olaylardan kaynaklanır. Bu sinyaller icra sırasında hiç umulmadık zamanlarda gelir. Harici olaylar sinyalleri eşzamansız olarak üretir ve diğer süreçlere yapılacak istekler için kullanılır.

Bir sinyal ya özellikle eşzamanlı ya da özellikle eşzamansızdır. Örneğin, hatalar için gönderilen sinyaller özellikle eşzamanlıdır, çünkü hatalar sinyalleri eşzamanlı üretir. Ancak ister eşzamanlı olsun ister eşzamansız, sinyaller açıkça bir isteğin sonucu olarak üretilir.

1.3. Sinyallerin Gönderilmesi

Bir sinyal üretidiğinde *beklemeye* alınır. Normalde kısa bir süre için beklemede kaldıktan sonra sinyalleyeceği sürece gönderilir. Eğer sinyal engellenen türde ise sonsuza kadar beklemede kalır—sinyal engellenemeyecek duruma gelininceye kadar. Sinyalin engellenemeyeceği durum oluştuğunda anında gönderilecektir. Bkz. *Sinyallerin Engellenmesi* (sayfa: 631).

Bir sinyal gönderildiğinde, hemen ya da uzun bir beklemenin ardından bu sinyal için belirlenmiş eylem alınır. **SIGKILL** ve **SIGSTOP** gibi sinyaller için eylem bellidir, ama diğer sinyaller için yazılım bazı seçimler yapabilir: sinyali yoksayabilir, bir *sinyalle tetiklenen işlev* belirtebilir ya da bu sinyal için geçerli olan *öntanımlı eylem*i kabul eder. Yazılım seçimini **signal** veya **sigaction** (*Sinyal Eylemlerinin Belirtilmesi* (sayfa: 611)) gibi bir işlev ile belirtir. Bazan sinyalle tetiklenen işlevlerden belge içinde kimi zaman ***sinyal yakalayıcı*** kimi zaman da ***eylemci işlev*** olarak da söz edeceğiz. Bu işlev çalışırken buna ilişkin sinyal engellenir.

Bir sinyal için belirlenmiş eylem onun yoksayılması ise, böyle bir sinyal üretildiği anda iptal edilir. Bu, sinyal zamanında engellendiğinde de böyle olur. Bu yolla iptal edilmiş bir sinyal asla gönderilmez; yazılımda hemen ardından böyle bir sinyal için farklı bir eylem belirtilse hatta engellenmeyeceği belirtilse bile.

Bir sinyali ne işleme sokacağını ne de yoksayacağını belirtmemişse bu sinyal geldiğinde yazılım, sinyalin *öntanımlı eylem*ini almış olur. Her sinyal türü kendine özgü bir öntanımlı eyleme sahiptir. Bunlar *Standart Sinyaller* (sayfa: 604) bölümünde açıklanmıştır. Sinyallerin çoğu için öntanımlı eylem sürecin sonlandırılmasıdır. "Zararsız" olaylar için gönderilen sinyaller için öntanımlı eylem ise hiçbir şey yapılmamasıdır.

Bir sinyal bir süreci sonlandırdığında onu çalıştıran süreç sonlanma sebebini **wait** veya **waitpid** işlevleri kullanarak onlardan dönen sonlanma durum koduna bakarak saptayabilir. (Bu *Süreç Tamamlama* (sayfa: 690) bölümünde ayrıntılı olarak açıklanmıştır.) Alınan bilgi, sonlanmaya bir sinyalin mi sebep olduğunu ve ne çeşit sinyal alındığını içerir.

Normalde yazılım hatalarını gösteren sinyaller özel bir niteliğe sahiptir: bu sinyallerden biri süreci sonlandırdığında, sonlanma sırasında sürecin durumunu gösteren *core* isimli bir döküm dosyası çıktılanır. Bu

dosyayı bir hata ayıklayıcı ile inceleyip hatanın sebebini saptayabilirsiniz.

Yazılımınızın oluşturduğu bir "yazılım hatası"nın sonucu olarak bir sinyal alınıp bunun sonucu olarak süreç sonlandığında tıpkı bir hatanın sonucunda olduğu gibi `core` dosyası çıktılanır.

2. Standart Sinyaller

Bu bölümde standart sinyallerin isimleri ve bu sinyallerin hangi olayların karşılığı olduğu açıklanmaktadır. Her sinyal ismi kendini bir ***sinyal numarası*** ile ilişkilendiren bir makrodur. Burada dikkat etmeniz gereken nokta; yazılımınız bir sinyali bir numara olarak kabul etmeyip daima burada tanımlanmış isimlerini kullanmalıdır. Çünkü, isimlerin anlamları standarttır ama numaraları sistemden sisteme değişiklik gösterebilir.

Bu kısımdaki sinyal isimleri `signal.h` başlık dosyasında tanımlanmıştır.

`int` **`NSIG`** makro

Bu sembolik sabitin değeri tanımlı sinyallerin toplam sayısıdır. Sinyaller artan numaralar aldıklarından **`NSIG`** en büyük numaralı sinyalin numarasıdır.

2.1. Yazılım Hatalarının Sinyalleri

Aşağıdaki sinyaller, bilgisayarın kendi ya da işletim sistemi tarafından ciddi bir yazılım hatası saptandığında üretilir. Genelde bu sinyallerin tümü yazılımınızın bir şeyleri önemli ölçüde bozacağı ya da sistemin bütünü açısından yazılımın çalışmasının sorun olacağı durumları belirtir.

Bazı yazılımlar yazılım hatası sinyallerini sonlanmadan önceki düzenlemeler sırasında işleme sokar; örneğin, uçbirim girdisinin yansılamasını kapatan bir yazılım yansılamayı tekrar açacağı sırada yazılım hatasını işleme sokmalıdır. Sinyalin işlenmesi, öntanımlı eylemin bitirilmesinin ardından bu sinyalin tekrar yayınlanması şeklinde yapılır. Bu, yazılımın bir sinyal eylemci yokmuşçasına bu sinyalle sonlanmasını sağlayacaktır. (Bkz. *Süreci Sonlandıran Eylemciler* (sayfa: 619).)

Sonlanma, çoğu yazılım için bir yazılım hatasının nihai sonucudur. Buna rağmen Lisp gibi bazı yazılım geliştirme sistemleri bir hataya maruz kalsa bile derlenmiş kullanıcı yazılımını çalışır halde tutabilir. Bu sistemler denetimi komut seviyesine döndürmek için **`longjmp`** kullanılan sinyal eylemcilere sahiptir.

Bu sinyallerin tümü için öntanımlı eylem sürecin sonlandırılmasına sebep olmaktır. Bir gerçek hata yerine **`raise`** veya **`kill`** tarafından üretilmedikçe oluşan sinyalleri engeller, yoksayar ya da bir sinyal yakalayıcı kurup normale çevirirseniz, büyük olasılıkla yazılımınız dehşet verici şekilde bozulacaktır.

Bu yazılım hatası sinyalleri bir süreci sonlandırırken, sonlanma sırasında sürecin durum kaydı olarak o an içinde bulunulan dizine `core` isimli bir dosya olarak bellek dökümü çıktılar. (GNU sistemlerinde dosyanın ismini **`COREFILE`** ortam değişkeni ile belirtebilirsiniz.) Bu dosyanın çıktılanmasının amacı, dosyanın bir hata ayıklayıcı ile incelenerek hatanın sebebinin bulunmasını sağlamaktır.

`int` **`SIGFPE`** makro

`SIGFPE` sinyali bir ölümcül aritmetik hata raporlar. Hatanın isminin "floating–point exception" kısaltması olarak oluşturlmasına rağmen sinyal aslında sıfırla bölme ve taşma dahil tüm aritmetik hataları kapsar. Eğer bir yazılım, tamsayı veri sakladığı bir alanı daha sonra bir gerçek sayı işleminde kullanmaya çalışırsa bu, bir "geçersiz işlem" olağandışılığına sebep olur, çünkü işlemci veriyi bir gerçek sayı olarak ele alamaz.

Aslında gerçek sayı olağandışılıkları oldukça karmaşık bir konudur, çünkü çok farklı anlamlara gelen çözümü zor çok çeşitli olağandışılıklar vardır ve **`SIGFPE`** sinyali onları ayıramaz. İkilik Kayan Noktalı Aritmetik için IEEE standardında (ANSI/IEEE Std 754–1985 and ANSI/IEEE Std 854–1987) çeşitli olağandışılıklar tanımlanmıştır ve onların oluşumlarını raporlayacak bilgisayar sistemleri arasında uyumluluk gerektirir. Ancak, bu standart olağandışılıkların nasıl raporlanacağını ya da işletim sisteminin yazılımcıya ne çeşit bir işleme ve denetim imkanı vereceğini belirtmez.

BSD sistemleri **SIGFPE** makrosunun yanında olağandışılığın çeşitli sebeplerini ayrımsamak için bir ek argüman sağlar. Bu argümana erişim sırasında, sinyal eylemciyi iki argüman kabul edecek şekilde tanımlamalısınız. Eylemci kurulurken de tek argümanlı işlev türüne dönüştürmelisiniz. GNU kütüphanesi bu ek argümanı sağlar. Ancak argümanın değeri sadece bu bilgiyi sağlayan sistemler (GNU ve BSD) için anlamlıdır.

`FPE_INTOVF_TRAP`
: Tamsayı taşması (C yazılımlarında donanıma özel biçimde taşma yakalayıcıyı etkinleştirmedikçe imkansızdır).

`FPE_INTDIV_TRAP`
: Sıfırla tamsayı bölme.

`FPE_SUBRNG_TRAP`
: İndisleme aralığı (C yazılımlarında bazı şeyler hiç denetlenmez).

`FPE_FLTOVF_TRAP`
: Gerçek sayı taşması.

`FPE_FLTDIV_TRAP`
: Gerçek/tam sayılarda sıfırla bölme.

`FPE_FLTUND_TRAP`
: Gerçek sayılarda alttan taşma (Normalde etkin değildir).

`FPE_DECOVF_TRAP`
: Ondalık taşma (Sadece bir kaç makina ondalık aritmetiğe sahiptir, C hiç kullanmaz).

`int` **SIGILL** makro

Sinyalin ismi "illegal instruction" sözcüklerinden türetilmiştir; yazılımınızın bozuk ya da ayrıcalıklı bir makina kodu komutunu çalıştırmayı denediğini belirtir. C derleyicileri sadece geçerli makina kodu komutlar ürettiğinden **SIGILL** sinyali genellikle çalıştırılabilir dosyanın zarar görmüş olabileceğini ya da çalıştırılabilir olmayan bir kodu çalıştırmayı denediğinizi belirtir. İkinci durumun ortaya çıktığı çok bilinen durumlar şunlardır: bir işlev olarak ele alınacağı umulan bir göstericiyle geçersiz nir nesnenin aktarılması; bir özdevinimli dizinin sonundan sonrasına yazma denemesi (benzer durum özdevinimli değişkenlere göstericilerde de ortaya çıkabilir); yığıt üzerinde, yığıt çerçevesine dönüş adresi gibi bir takım verilerin bozulması.

SIGILL sinyali bunlardan başka, yığıt taşmalarında ya da sistemde çalışan sorunlu bir sinyal eylemcinin varlığında da üretilebilir.

`int` **SIGSEGV** makro

Bu sinyal, bir yazılımın kendine ayrılan bellek bölgesinin dışında okuma veya yazma denemesi yaptığında ya da salt okunur belleğe yazma denemesinde oluşur. (Aslında bu sinyal sadece sistemin bellek koruma mekanizması tarafından saptanabilen, yazılımın kendi alanının dışında çok uzak bölgelere yazmaya çalıştığı durumlarda ortaya çıkar.) Sinyalin ismi "segmentation violation" sözcüklerinden türetilmiştir.

SIGSEGV sinyalinin alındığı bilinen sorunlar: göstericinin bir boş ya da ilklendirilmemiş göstericiye dönüştürülmesi (dereferencing – dizi olmayan bir değişkene gösterici üzerinde gösterici aritmetiği uygulanması ya da ilklendirilmemiş bir yapı elemanını göstermek için bir –> işleci ile sol taraf değeri olarak kullanılması); bir dizinin sonunu kontrol etmeden dizi üzerinde gösterici aritmetiği ile işlem yapılması. Bir göstericinin bir boş göstericiye dönüşmesini durumunu çeşitli sistemler **SIGSEGV** ya da **SIGBUS** sinyali ile belirtir.

`int` **SIGBUS** makro

Bu sinyal geçersiz duruma gelmiş göstericiler kullanılmaya çalışıldığında ortaya çıkar. **SIGSEGV** sinyalindeki gibi bu sinyalde geçersiz bir gösterici kullanımıyla ilgili olarak üretilir. İkisi arasındaki fark, **SIGSEGV** sinyalinin geçerli belleğe geçersiz erişimi belirtmesi, **SIGBUS** sinyalinin ise geçersiz bir adrese erişimi belirtmesidir. Bazan **SIGBUS** sinyali göstericinin hatalı hizalama ile kullanıldığı durumlarda da üretilir; örneğin dört sözcüklük bir tamsayı değerinin saklandığı adreste alanın dörde bölünememesi gibi (her bilgisayarın kendine özgü adres hizalaması vardır).

Sinyalin ismi "bus error" sözcüklerinden türetilmiştir.

`int` **SIGABRT** makro

Bu sinyal yazılımın kendisi tarafından saptanan bir hatayı belirtir ve **abort** çağrısı ile raporlanır. Bkz, *Anormal Sonlanma* (sayfa: 683).

`int` **SIGIOT** makro

PDP–11 "iot" komutu tarafından üretilir. Çoğu makinada, **SIGABRT** sinyali olarak yer alır.

`int` **SIGTRAP** makro

Makinanın "breakpoint" komutu ve bazı diğer yakalama komutları tarafından üretilir. Bu sinyal hata ayıklayıcılar tarafından kullanılır. Yazılımınız büyük ihtimalle bazı hatalı makina komutları dolayısıyla sadece **SIGTRAP** sinyalini görecektir.

`int` **SIGEMT** makro

Öykünme tuzağı; bu sinyal henüz gerçeklenmemiş ama yazılım tarafından taklit edilen makina komutlarından ya da onların olması gerektiği gibi taklit edilememesinden dolayı işletim sistemi tarafından üretilir.

`int` **SIGSYS** makro

Hatalı sistem çağrısı; İşletim sisteminden çalıştırılması istenen ancak çağrı için belirtilen kod numarası makina komutlarında geçersiz olan çağrılarda üretilir.

2.2. Sonlandırma Sinyalleri

Bu sinyaller bir sürece şu veya bu şekilde sonlandırılacağını söyler. Farklı amaçlarla yazılımlar onları farklı algılama isteklerine uygun olarak farklı isimlere sahiptirler.

Bu sinyallerin işlenme sebebi yazılımın gerçek sonlandırmayı yapmadan önce bazı hazırlıklar yapmasına imkan vermektir. Örneğin, yazılımı sonlandırmadan önce son duruma ilişkin bilgileri bir yerlere kaydetmek, geçici dosyaları silmek, önceki uçbirim kipine dönmek isteyebilirsiniz. Bunu yapmak için önce sinyali engeller, bu işlemleri yaptıktan sonra asıl sonlandırmayı gerçekleştirmek için sinyali tekrar üretirsiniz. Bu işlem, yazılımınız sanki sinyali elde edemeyen bir yazılım gibi sonlanmasını sağlar. (Bakınız, *Süreci Sonlandıran Eylemciler* (sayfa: 619).)

Bu sinyaller için öntanımlı eylem sürecin sonlanmasına sebep olmaktır.

`int` **SIGTERM** makro

SIGTERM, yazılımın sonlanmasına sebep olan en temel sinyallerden biridir. **SIGKILL** sinyalinin tersine bu sinyal engellenebilir, işleme sokulabilir ya da yoksayılabilir. Normal yöntem yazılımı sonlandırmadan önce isteği kullanıcıya doğrulatmaktır.

kill kabuk komutu öntanımlı olarak (seçeneksiz kullanımda) **SIGTERM** sinyali üretir.

`int` **SIGINT** makro

SIGINT ("program interrupt" sözcüklerinden türetilmiştir) sinyali, kullanıcı tarafından INTR karakteri (normalde **C-c** tuşları) tuşlandığında üretilir. **C-c** sürücü desteği hakkında daha fazla bilgi için *Özel Karakterler* (sayfa: 454) bölümüne bakınız.

`int` **SIGQUIT** makro

SIGQUIT sinyali QUIT karakteri ile (normalde **C-**) üretilmesi dışında **SIGINT** sinyali gibidir. Süreci sonlandırırken bir yazılım hatası sinyalinin yaptığı gibi son bellek dökümü olarak `core` dosyası çıktılar. Bunu kullanıcı tarafından "saptanan" bir hata durumu olarak düşünebilirsiniz.

Hata durumndaki bellek dökümleri hakkında daha fazla bilgi için *Yazılım Hatalarının Sinyalleri* (sayfa: 604) bölümüne bakınız. Uçbirim sürücüsü desteği ile ilgili olarak da *Özel Karakterler* (sayfa: 454) bölümüne bakınız.

Bazı temizlik işlemleri yapmadan çıkmanın en iyi yolu **SIGQUIT** sinyalinin elde edilmesidir. Örneğin yazılımınız geçici dosyalar oluşturuyorsa ve diğer sonlandırma isteklerinde bu dosyaları siliyorsa onların silinmemesi için **SIGQUIT** sinyalini ürettirmek daha iyidir. Böylece bellek dökümü yanında bu geçici dosyalara da bakarak birşeyler daha iyi saptanabilir.

`int` **SIGKILL** makro

SIGKILL sinyali bir uygulamanın anında sonlandırılmasında kullanılır. Bu sinyal engellenemez ve yoksayılamaz.

Bu sinyal genellikle açıkça bir isteğin sonucunda üretilir. Yakalanamadığından **C-c** veya **SIGTERM** denedikten sonra sadece bir son çare olarak üretmelisiniz. Eğer süreç başka herhangi bir sonlandırma sinyaline yanıt vermezse, ona bir **SIGKILL** sinyali göndererek hemen hemen daima sonlanmasını sağlayabilirsiniz.

Ancak, eğer **SIGKILL** sinyali bir süreci sonlandıramazsa, bu bir işletim sistemi hatasıdır ve mutlaka rapor edilmelidir.

Ayrıca, bir sürecin çalışmasını sürdürmesinin imkansız olduğu durumlarda da süreç bir sinyal eylemci kullanıyor olsa bile sistem **SIGKILL** sinyali göndererek bu süreci sonlandırabilir.

`int` **SIGHUP** makro

SIGHUP ("hang–up" sözcüklerinden türetilmiştir) sinyali kullanıcının uçbiriminin bağlantısı kesildiğinde durumu bildirmek için üretilir. Bu genellikle bir ağ ya da telefon bağlantısı kesildiğinde olur. Bu durumla ilgili daha fazla bilgi için *Denetim Kipleri* (sayfa: 449) bölümüne bakınız.

Bu sinyal ayrıca, bir uçbirim üzerinde o oturuma ilişkin işler yürütülen bir sürecin sonlandırılmasında da kullanılır; bu sonlandırma oturumdaki tüm süreçlerin çalıştırıldığı uçbirimle bağlantısının kopmasına yol açar. Daha fazla bilgi için *Sonlandırmanın İçyapısı* (sayfa: 684) bölümüne bakınız.

2.3. Alarm Sinyalleri

Bu sinyaller zamanlayıcıların zaman aşımına uğradığını bildirmekte kullanılır. Bu sinyallerin gönderilmesine sebep olan işlevler *Bir Alarmın Ayarlanması* (sayfa: 568) bölümünde bulunabilir.

Bu sinyallerin öntanımlı davranışı o sürecin sonlandırılmasına sebep olmasıdır. Bu öntanımlı davranış geniş çapta kullanışlıdır; ancak, bu sinyallerin kullanıldığı yöntemler her durumda bir eylemci işlev kullanımını gerekli kılar.

`int` **SIGALRM** makro

Bu sinyal özellikle bir zamanlayıcının gerçek ya da saat tikleri cinsinden ölçülen değeri için saptanan sınırın aşıldığını belirtir. **alarm** işlevi gibi işlevlerle üretilir.

`int` **SIGVTALRM** makro

Bu sinyal, özellikle süreç tarafından kullanılan işlemci zamanı cinsinden zamanlayıcı değerinin zamanaşımına uğradığını belirtir. "virtual time alarm" sözcüklerinden türetilmiştir.

`int` **SIGPROF** makro

Bu sinyal, özellikle süreç tarafından kullanılan hem işlemci zamanı cinsinden hem de sistem tarafından süreç lehine kullanılan işlemci zamanı cinsinden zamanlayıcı değerinin zamanaşımına uğradığını belirtir. Bu tür zamanlayıcılar kod profili oluşumlarının gerçeklemelerinde kullanılır, zaten sinyalin ismi de buradan gelir.

2.4. Eşzamansız G/Ç Sinyalleri

Bu bölümde açıklanan sinyaller eşzamansız G/Ç oluşumlarıyla ilgilidir. Bu sinyalleri üretecek dosya tanımlayıcıları etkinleştirecek **fcntl** çağrısı ile eylemi (*Sinyallerle Sürülen Girdi* (sayfa: 349)) doğrudan elde edebilirsiniz. Bu sinyaller için öntanımlı eylem onların yoksayılmasıdır.

`int` **SIGIO** makro

Bu sinyal bir dosya tanımlayıcı girdi veya çıktı işlemleri yapmaya hazır olduğunda üretilir.

Çoğu işletim sisteminde, **SIGIO** üretebilen dosya çeşitleri sadece uçbirimler ve soketlerdir. Sıradan dosyalarında dahil olduğu diğer dosya çeşitleri, isteseniz bile **SIGIO** sinyalini asla üretmez.

GNU sisteminde **SIGIO** sinyali daima **fcntl** işleviyle eşzamansız kipe girildiğinde üretilir.

`int` **SIGURG** makro

Bu sinyal bir soket üzerinden "acil" ya da bantdışı veri geldiğinde üretilir. Bkz, *Bantdışı Veri Aktarımı* (sayfa: 431).

`int` **SIGPOLL** makro

Bu bir System V sinyal ismidir, az çok **SIGIO** sinyaline benzer. Sadece uyumluluk için vardır.

2.5. İş Denetim Sinyalleri

Bu sinyaller iş denetimine (job control) destek için kullanılır. Sisteminizde iş denetimi desteği yoksa, bu makrolar tanımlanmış bile olsa, sinyaller üretilemez ve yakalanamaz.

İş denetiminin nasıl çalıştığı hakkında bir fikriniz yoksa bu sinyalleri unutun. Daha fazla bilgi için *İş Denetimi* (sayfa: 716) bölümüne bakınız.

`int` **SIGCHLD** makro

Bu sinyal, bir alt süreci çalıştıran sürece alt süreç durdurulduğunda ya da sonlandırıldığında gönderilir.

Bu sinyal için öntanımlı eylem yoksayılmaktır. Sonlandırılmış alt süreçlerinizin olup bunların durumlarının **wait** veya **waitpid** (*Süreç Tamamlama* (sayfa: 690)) ile bildirilmediği durumlar için ya da bir işletim sistemine bağımlı olmamak için bu sinyal için bir eylemci oluşturabilirsiniz.

`int` **SIGCLD** makro

SIGCHLD için atıl olmuş bir sinyal ismidir.

`int` **SIGCONT** makro

SIGCONT sinyalini bir sürecin devam etmesini istemek için üretebilirsiniz. Bu sinyal alınmadan önce durmuş olan bir sürecin çalışmasını sürdürmesi istendiğinde gönderilir. Öntanımlı davranış başka bir şey yapılmamasıdır. Bu sinyal engellenemez. Bir eylemci belirtseniz bile **SIGCONT** sürecin çalışmaya devam etmesini sağlar.

Çoğu yazılım için **SIGCONT** sinyalini yakalamanın bir anlamı yoktur; basitçe hiç durdurulmamışlar gibi çalışmaya kaldıkları yerden devam edeceklerdir. Bu sinyal için bir eylemci sadece durdurulduktan sonra çalıştırldığında yazılıma özel bazı işlemler yapmanız gerekiyorsa anlamlıdır. Örneğin, durdurma öncesi uçbirim çıktılaması kapalıyken açtıysanız, çalışmaya devam edileceğinde bunu tekrar kapatmak isteyebilirsiniz.

`int` **SIGSTOP** makro

SIGSTOP sinyali süreci durdurur. Yakalanamaz, engellenemez, yoksayılamaz.

`int` **SIGTSTP** makro

SIGTSTP sinyali bir etkileşimli durdurma sinyalidir. **SIGSTOP** sinyalinin aksine yakalanabilir ve yoksayılabilir.

Bu sinyalle bir durdurma isteği geldiğinde dosyalarınızı ve sistem tablolarınızı güvenli durumda bırakmak isterseniz bu sinyal için bir eylemci oluşturmalısınız. Örneğin uçbirimde çıktılamayı kapatmışsanız, durdurma sırasında bunu açmanız gerekir.

Bu sinyal kullanıcı tarafından `SUSP` karakteri (normalde **C-z**) tuşlandığında üretilir. Uçbirim sürücü desteği ile ilgili daha fazla bilgi için *Özel Karakterler* (sayfa: 454) bölümüne bakınız.

`int` **SIGTTIN** makro

Bir süreç bir artalan işi olarak çalışıyorsa kullanıcı uçbirimini okuyamaz. Bir artalan işindeki herhangi bir süreç uçbirimden okuma yapmak istediğinde işteki tüm süreçlere bir **SIGTTIN** sinyali gönderilir. Bu sinyal için öntanımlı eylem sürecin durdurulmasıdır. Uçbirimle girilen bu etkileşimle ilgili daha fazla bilgi almak için *Denetim Uçbirimine Erişim* (sayfa: 717) bölümüne bakınız.

`int` **SIGTTOU** makro

SIGTTIN sinyaline benzer, farklı olarak artalandaki iş içindeki süreç uçbirime yazmaya ya da kipi değiştirmeye çalıştığında üretilir. Burada da öntanımlı eylem sürecin durdurulmasıdır. **SIGTTOU** sinyali sadece **TOSTOP** çıktı kipi belirtilerek uçbirime yazmaya çalışıldığında üretilir; bkz, *Çıktı Kipleri* (sayfa: 449).

Bir süreç durdurulduğunda **SIGKILL** ve **SIGCONT** sinyalleri dışında hiçbir sinyali alamaz, süreç devam ettirilene kadar askıya alınırlar. **SIGKILL** sinyali daima süreci sonlandırır ve engellenemez, yakalanamaz ve yoksayılamaz. **SIGCONT** sinyali yoksayılabilir ama daima durdurulmuş bir sürecin kaldığı yerden çalışmasına devam etmesini sağlar. Bir sürece **SIGCONT** sinyalinin gönderilmesi askıda bekleyen bir durdurma sinyali varsa iptal edilmesine sebep olur. Benzer şekilde, askıya alınmış bir **SIGCONT** sinyali bir durdurma sinyali alındığında iptal edilir.

Bir *öksüz süreç grubundaki* (sayfa: 718) bir süreç **SIGTSTP**, **SIGTTIN** veya **SIGTTOU** sinyallerinden birini alırsa ve o sinyali yakalamıyorsa, süreç durmaz. Şüphesiz böyle bir sürecin durdurulması pek kullanışlı değildir. Çünkü böyle bir süreci durması için uyaracak bir kabuk ya da devam etmesine izin verecek bir kullanıcı olmayacaktır. Bazı sistemler hiçbir şey yapmayabilir; bazıları da bunun yerine **SIGKILL** veya **SIGHUP** gibi bir sinyal alabilir. GNU sistemlerinde süreç **SIGKILL** ile öldürülür; bu sistemde durmuş ya da öksüz kalmış süreçlerle ilgili sorunları da çözer.

2.6. İşlemsel Hata Sinyalleri

Bu sinyaller yazılım tarafından yapılan bir işlemin ürettiği çeşitli hataları raporlamakta kullanılır. Bunlar her zaman yazılımdaki bir yazılım geliştirme hatasını belirtmezler, bir işletim sistemi çağrısının tamamlanmasına engel olan bir hatayı da belirtebilir. Bunların hepsi için öntanımlı eylem sürecin sonlanmasına sebep olmaktır.

`int` **`SIGPIPE`** makro

Kırık boru (Broken pipe). Boruları ya da FIFO'ları kullanıyorsanız, yazılımınızı, bir süreç bir borunun ucundan yazmaya başlamasından önce başka bir sürecin diğer uçtan okumaya başlamasını sağlayacak şekilde tasarlamak zorundasınız. Eğer okuyan süreç başlamazsa ya da beklenmedik şekilde sonlanırsa, boruya ya da FIFO'ya yazan süreç bir **`SIGPIPE`** sinyali üretir. Eğer **`SIGPIPE`** engellenir, işleme sokulur ya da yoksayılırsa etkilenen çağrı **`EPIPE`** ile başarısız olur.

Borular ve FIFOlar özel dosyalardır ve *Borular ve FIFOlar* (sayfa: 393) bölümünde ayrıntılı olarak açıklanmıştır.

`SIGPIPE` sinyalinin başka bir sebebi de bağlı olmayan bir sokete yazma denemesidir. Bkz, *Veri Gönderimi* (sayfa: 426).

`int` **`SIGLOST`** makro

Özkaynak kaybı. Bir NFS dosyası üzerinde tavsiye niteliğinde bir kilit varsa ve NFS sunucusu yeniden başlatıldığında sizin kilidi hazırladığınızı unutacağından bu sinyal üretilir.

GNU sisteminde herhangi bir sunucu beklenmedik şekilde ölürse, **`SIGLOST`** sinyali üretilir. Genellikle sinyal yoksayılabilir; ancak ölmüş bir sunucuya yapılan bir çağrı sadece hata döndürür.

`int` **`SIGXCPU`** makro

İşlemci zaman sınırı aşıldı. Bu sinyal bir sürecin işlemci zamanı üzerindeki sanal özkaynak sınırı aşıldığında üretilir. Bkz, *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

`int` **`SIGXFSZ`** makro

Dosya boyu sınırı aşıldı. Bu sinyal, dosya boyu üzerindeki sürecin sanal özkaynak sınırı aşılacak şekilde dosya büyütülmeye çalışıldığında üretilir. *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

2.7. Çeşitli Sinyaller

Bu sinyaller başka başka amaçlar için kullanılır. Onları birşeyler için özellikle kullanmıyorsanız hiçbir etkileri yoktur.

`int` **`SIGUSR1`** makro
`int` **`SIGUSR2`** makro

`SIGUSR1` ve **`SIGUSR2`** sinyalleri istediğiniz herhangi bir amaçla kullanabilmeniz için vardır. Onlar için bir sinyal eylemci yazarsanız, basit süreçler arası iletişim için kullanışlıdırlar.

Başka Bir Sürece Sinyal Gönderme (sayfa: 628) bölümünde **`SIGUSR1`** ve **`SIGUSR2`** sinyallerinin kullanımına bir örnek bulabilirsiniz.

Öntanımlı eylem sürecin sonlanmasıdır.

`int` **`SIGWINCH`** makro

Pencere boyutu değişti. Bu sinyal, bazı sistemlerde (GNU dahil), uçbirim sürücüsünün ekranın satır ve sütun sayılarını tutan kaydı değiştiğinde üretilir. Öntanımlı eylem sinyalin yoksayılmasıdır.

Bir yazılım alanını tam ekrana genişletilirse **SIGWINCH** sinyalini yakalamalıdır. Sinyal geldiğinde, yazılım yeni boyutlara göre kendini ayarlamalıdır.

`int` **SIGINFO** makro

Bilgi isteği. 4.4 BSD ve GNU sisteminde, bu sinyal kullanıcı tarafından meşru kipte STATUS karakterini tuşladığında denetçi uçbirimin önalan süreç grubundaki tüm süreçlere gönderilir; bkz, *Sinyal Gönderen Karakterler* (sayfa: 456).

Eğer süreç, süreç grubunun lideri ise öntanımlı eylem, sürecin ne yaptığı ve sistem hakkındaki bazı durum bilgilerinin basılmasıdır. Aksi takdirde öntanımlı olarak hiçbir şey yapılmaz.

2.8. Sinyal İletileri

Daha önce bahsettiğimiz gibi, bir alt süreç sinyal ile sonlandığında, kabuk bu sinyali açıklayan bir ileti basar. Bir sinyali açıklayan bir ileti basmanın en temiz yolu **strsignal** ve **psignal** işlevlerini kullanmaktır. Bu işlevler hangi sinyal çeşidini açıklayan iletinin basılacağını belirtmek için bir sinyal numarası kabul ederler. Sinyal numarası bir alt sürecin sonlanma durumundan (*Süreç Tamamlama* (sayfa: 690)) ya da aynı sürecin sinyal eylemcisinden gelebilir.

`char *`**strsignal**`(int` *sinyalnum*`)` işlev

sinyalnum numaralı sinyali açıklayan bir iletinin durağan olarak ayrılmış dizgesine bir gösterici ile döner. Bu dizgenin içeriğinde değişiklik yapmamalısınız; ayrıca daha sonraki çağrılar bu dizgenin yeniden yazılmasına sebep olacağından dizgeyi hemen kullanmayacaksanız bir kopyasını saklamalısınız.

Bu işlev bir GNU oluşumudur ve `string.h` başlık dosyasında bildirilmiştir.

```
void psignal(int           sinyalnum,                                    işlev
             const char *ileti)
```

sinyalnum numaralı sinyali açıklayan bir iletiyi **stderr** standart hata çıktılama akımına basar. Bkz, *Standart Akımlar* (sayfa: 237).

psignal işlevini bir boş gösterici ya da bir boş dizge olarak bir *ileti* ile çağırırsanız *sinyalnum*'un karşılığı olan iletinin sonuna bir satırsonu karakteri ile bir boşluk yerleştirir.

Bu işlev bir BSD oluşumudur ve `signal.h` başlık dosyasında bildirilmiştir.

Bunlardan başka çeşitli sinyal kodları için iletiler içeren **sys_siglist** dizisi vardır. Bu dizi, **strsignal**'in aksine BSD sistemlerinde bulunur.

3. Sinyal Eylemlerinin Belirtilmesi

Bir sinyalin oluşturacağı eylemi değiştirmenin en basit yolu **signal** işlevini kullanmaktır. Yerleşik eylemlerden birini belirtebileceğiniz gibi bir ***sinyal yakalayıcı*** da oluşturabilirsiniz.

GNU kütüphanesi ayrıca, daha yetenekli olan **sigaction** oluşumunu da içerir. Bu kısım her iki oluşum ve kullanımlarına ilişkin önerileri içerir.

3.1. Basit Sinyal İşleme

signal işlevi, belirli bir sinyal için bir eylem oluşturmayı sağlayan basit bir arayüzdür. İşlevin ve bununla ilgili makroların bildirimleri `signal.h` başlık dosyasında bulunur.

sighandler_t veri türü

Bu, sinyal yakalama işlevlerinin veri türüdür. Sinyal yakalama işlevleri sinyal numarasının belirtildiği tek bir argüman alırlar ve dönüş türleri **void**'dir. Böyle bir sinyal yakalama işlevi şöyle tanımlanmalıdır:

```
void eylemci (int sinyalnum) { … }
```

Bu veri türünün ismi olan **sighandler_t** bir GNU oluşumudur.

```
sighandler_t signal(int sinyalnum,                                    işlev
                    sighandler_t eylem)
```

signal işlevi *sinyalnum* sinyali için eylem olarak *eylem* eylemini oluşturur.

İlk argüman olan *sinyalnum*, denetlenecek davranışın karşılığı olan sinyaldir ve bir sinyal numarası olarak belirtilmelidir. Bir sinyal numarasını belirtirken sembolik sinyal isimlerini kullanmanız gerekir (*Standart Sinyaller* (sayfa: 604)). Doğrudan doğruya numarasını belirtmeyin, çünkü sinyallerin numaraları işletim sistemleri arasında değişiklik gösterebilir.

İkinci argüman olan *eylem* ise, *sinyalnum* sinyali için kullanılacak eylemi belirtmek için kullanılır. Bu şağıdakilerden biri olabilir:

`SIG_DFL`
: **SIG_DFL**, belli bir sinyal için öntenımlı olan eylemi belirtir. Çeşitli sinyaller için öntanımlı olan eylemler *Standart Sinyaller* (sayfa: 604) bölümünde bulunabilir.

`SIG_IGN`
: **SIG_IGN**, sinyalin yoksayılacağını belirtmek için kullanılır.

  Normalde yazılımınız birbiri ardınca gelen eylemlere ait sinyalleri ya da sonlandırma isteği olarak kullanılan sinyalleri yoksaymamalıdır. **SIGKILL** veya **SIGSTOP** sinyalini ne yaparsanız yapın yoksayamazsınız. **SIGSEGV** benzeri bir sinyali yoksayabilirsiniz, ama bir hatanın yoksayılması ve yazılımın çalışmasını sürdürmesi anlamlı olmaz. **SIGINT**, **SIGQUIT** ve **SIGTSTP** gibi kullanıcı isteğini belirten bir sinyali yoksaymak pek dostça sayılmaz.

  Yazılımınızın belli bir bölümünde sinyallerin alınmasını istemiyorsanız onları yoksaymayın, *onları engelleyebilirsiniz* (sayfa: 631).

eylemci
: Sinyal alındığında yapılacak eylemi gerçekleştirecek işlevin adresi belirtilir.

  Sinyalle tetiklenen işlevler hakkında daha ayrıntılı bilgi edinmek için *Sinyal Yakalayıcıların Tanımlanması* (sayfa: 617) bölümüne bakınız.

Bir sinyal için **SIG_IGN** veya **SIG_DFL** belirtirseniz ve öntanımlı eylem sinyalin yoksayılması ise, bekleyen sinyallerden bu türde olanları (engellenmeye çalışılsa bile) iptal edilir. Bekleyen bir sinyalin iptal edilmesi, hemen ardından başka bir eylem belirtilmedikçe ve bu tür sinyallerin engellenmemesi istenmedikçe, bunların asla alınmayacağı anlamına gelir.

signal işlevi, *sinyalnum* sinyali için evvelce belirtilmiş olan eylemle döner. Böylece, bu değeri saklayabilir ve daha sonra **signal** işlevini tekrar çağırarak bu eylemin tekrar etkin olmasını sağlayabilirsiniz.

Eğer **signal** işlevi kendinden isteneni yerine getiremezse **SIG_ERR** ile döner. Bu işlev için tanımlanmış **errno** değerleri:

`EINVAL`
: Geçersiz bir *sinyalnum* belirttiniz; ya da **SIGKILL** veya **SIGSTOP** için sinyal yakalayıcı oluşturmaya ya da bunları yoksaymaya çalıştınız.

Uyumluluk Bilgisi

signal işlevi ile çalışırken saptanmış bir sorun, BSD ve SVID sistemlerdeki davranış farkıdır. SVID sistemlerde sinyal yakalayıcı sinyal alındıktan sonra kendiliğinden tekrar kurulur. BSD sistemlerde ise yakalayıcı tekrar kurulmak zorundadır. GNU C kütüphanesinde öntanımlı olarak BSD sürümünü kullanıyoruz. SVID sürümünü kullanmak isterseniz, aşağıda anlatılan **sysv_signal** işlevini ya da bir makro seçici olan **_XOPEN_SOURCE**'u kullanabilirsiniz (*Özellik Sınama Makroları* (sayfa: 25)). Uyumluluk sorunlarından kaçınmak için normalde bu işlevler kullanılmamalıdır. Bunlar yerine uyumluluk açısından bir sorun çıkarmayan **sigaction**'ı kullanmak daha iyidir.

Aşağıda, bazı ölümcül sinyaller alındığında geçici dosyaları silen basit bir yakalayıcı örneği yeralmaktadır:

```
#include <signal.h>

void
termination_handler (int signum)
{
  struct temp_file *p;

  for (p = temp_file_list; p; p = p->next)
    unlink (p->name);
}

int
main (void)
{
  ...
  if (signal (SIGINT, termination_handler) == SIG_IGN)
    signal (SIGINT, SIG_IGN);
  if (signal (SIGHUP, termination_handler) == SIG_IGN)
    signal (SIGHUP, SIG_IGN);
  if (signal (SIGTERM, termination_handler) == SIG_IGN)
    signal (SIGTERM, SIG_IGN);
  ...
}
```

Eğer belirtilen bir sinyal evvelce yoksayılmaya ayarlanmışsa bu kodun bu ayarı değiştirmediğine dikkat edin. Bu, iş denetimi yapmayan kabukların alt süreçleri başlatırken bazı sinyalleri çoğunlukla yoksaymasından ve alt süreçler için buna riayet edilmesi önemli olduğundan dolayıdır.

Yazılım hata sinyallerini veya **SIGQUIT** sinyalini, hata ayıklamada bilgi sağlamak için (bellek dökümü almak için) tasarlandıklarından ve geçici dosyalar hata ayıklamak için faydalı bilgiler sağlayabileceğinden bu örnekte işleme sokmadık.

```
sighandler_t sysv_signal(int           sinyalnum,                    işlev
                         sighandler_t  eylem)
```

sysv_signal işlevi, SVID sistemlerindeki standart **signal** işlevinin davranışını gereçekleştirmek için tasarlanmıştır. Bunun BSD sistemlerinden farkı, bir sinyalin alınmasının ardından yakalayıcının kendiliğinden tekrar kurulmasıdır.

Uyumluluk Bilgisi

signal işlevi için yukarıda bahsedildiği gibi, bu işlevin kullanılmasından kaçınılmalı ve bunun yerine mümkünse **sigaction** tercih edilmelidir.

```
sighandler_t ssignal(int           sinyalnum,                                  işlev
                     sighandler_t eylem)
```

ssignal işlevi **signal** işlevinin yaptığını yapar; sadece SVID ile uyum için vardır.

```
sighandler_t SIG_ERR                                                           makro
```

Bu makronun değeri, **signal** işlevinin dönen bir hata değeri olarak kullanılır.

3.2. Gelişmiş Sinyal İşleme

sigaction işlevi **signal** işlevi ile aynı temel etkiye sahiptir: bir sinyalin süreç tarafından nasıl işleneceği belirtilir. Farklı olarak, sinyalin üretilmesi ve eylemcinin çağrılması ile ilgili çeşitli denetim seçenekleri belirtebilirsiniz.

sigaction işlevi `signal.h` başlık dosyasında bildirilmiştir.

```
struct sigaction                                                               veri türü
```

struct sigaction türündeki yapılar, **sigaction** işlevinde belli bir sinyalin nasıl işleneceği hakkındaki bilgilerin belirtilmesi için kullanılır. Bu yapı en azından aşağıdaki üyeleri içerir:

`sighandler_t` **sa_handler**
: **signal** işlevindeki *eylem* argümanının yerine geçer. Değer olarak, **SIG_DFL**, **SIG_IGN** veya bir işlev göstericisi alır. Bkz. *Basit Sinyal İşleme* (sayfa: 611).

`sigset_t` **sa_mask**
: Eylemci çalışırken engellenecek sinyalleri belirtmek içindir. Sinyallerin engellenmesi *Eylemci Çalışırken Sinyallerin Engellenmesi* (sayfa: 634) bölümünde anlatılmıştır. Alınan bir sinyal, eylemcisi başlatılmadan önce öntanımlı olarak özdevinimli engellenir; bu, **sa_mask**'ın değerine bakılmaksızın böyledir. Bir sinyalin eylemcisi nedeniyle engellenmemesini istiyorsanız eylemci içindeki kodu sinyalin engellenmemesini sağlayacak şekilde yazmalısınız.

`int` **sa_flags**
: Burada, sinyalin davranışını etkileyebilen çeşitli seçenekler belirtilebilir. Bunlar *sigaction Seçenekleri* (sayfa: 616) bölümünde daha ayrıntılı olarak açıklanmıştır.

```
int sigaction(int                                 sinyalnum,                   işlev
              const struct sigaction *restrict    eylem,
              struct sigaction *restrict          eski–eylem)
```

eylem argümanı ile *sinyalnum* sinyali için yeni bir eylem belirtilirken, *eski–eylem* argümanı, bu sembolle ilişkili evvelki eylem hakkında bilgi döndürmek için kullanılır. (başka bir deyişle, *eski–eylem* argümanı **signal** işlevinin dönüş değeri gibi kullanılmıştır. Bununla eski eylemin ne olduğuna bakabilir ve isterseniz bu eylemi tekrar yerinde bırakmak anlamında etkinleştirebilirsiniz.)

Hem *eylem* hem de *eski–eylem* birer boş gösterici olabilir. *eski–eylem* bir boş gösterici ise, *sinyalnum* sinyali ile ilişkili eylem değişmez; bu, bir sinyalin işlenme şeklini değiştirilmeksizin o sinyalin işlenmesi ile ilgili bilgi edinmenizi mümkün kılar.

sigaction başarılı olduğunda sıfır ile aksi takdirde **-1** ile döner. Aşağıdaki bu işlev ile ilişkili **errno** değerleri bulunmaktadır:

`EINVAL`
: *sinyalnum* argümanı geçersiz; ya da **SIGKILL** veya **SIGSTOP** sinyali yoksayılmaya ya da yakalanmaya çalışılıyor.

3.3. `signal` ve `sigaction` arasındaki etkileşim

`signal` ve **`sigaction`** işlevlerini aynı yazılım içinde kullanmak mümkündür. Ancak tuhaf bir yolla bu iki işlev birbirinden etkilenir, bu nedenle bu ikisini aynı yazılım içinde kullanıyorsanız dikkatli olmanız gerekir.

`sigaction` işlevi **`signal`** işlevinden daha fazla bilgi içerir. Yani, **`signal`** işlevinin dönüş değeri **`sigaction`** işlevinin döndürdüğünden daha az bilgi döndürür. Diğer taraftan, bir eylemi kaydedip daha sonra etkinleştirmek için **`signal`** işlevini kullanırsanız, tekrar kurulan eylemci **`sigaction`** tarafından yeniden kurulan eylemci kadar düzgün oluşmayacaktır.

Sonuç olarak, sorunlardan kaçınmak için, yazılımınızda her yerde **`sigaction`** kullanmışsanız, bir eylemi kaydetmek ve yeniden oluşturmak için yine **`sigaction`** işlevini kullanın. Hatta, **`sigaction`** daha genel olduğundan, bir eylem hangi işlev ile kurulmuş olursa olsun, bir eylemi orjinal haliyle saklamak ve yeniden oluşturmak için daima **`sigaction`** işlevini kullanın.

Bazı sistemlerde, eğer bir eylemi **`signal`** ile oluşturup daha sonra **`sigaction`** ile incelerseniz eylemci işlevin adresinin **`signal`** işlevinin argümanı olarak belirtilen adresle aynı olmadığını görebilirsiniz. Hatta **`signal`** işlevinin bir argümanı olarak kullanmak için bile uygun olmayabilir. Ama **`sigaction`** işlevine bir argüman olarak kullanabilirsiniz. Bu sorun GNU sistemlerinde asla görülmez.

Bu durumda, en iyisi bir yazılım içinde sürekli olarak bu mekanizmalardan sadece birini kullanmaktır.

Taşınabilirlik Bilgisi

`sigaction` işlevi POSIX.1'in parçası olduğu halde, **`signal`** işlevi bir ISO C oluşumudur. Yazılımınızın POSIX olmayan sistemlere taşınabilirliği bakımından kaygınız varsa, **`sigaction`** yerine **`signal`** işlevini tercih etmelisiniz.

3.4. `sigaction` Örneği

Basit Sinyal İşleme (sayfa: 611) bölümünde sonlandırma sinyalleri için **`signal`** işlevi kullanılan basit bir eylemci örneği verilmişti. Burada bu örneğin **`sigaction`** eşdeğerini bulacaksınız:

```
#include <signal.h>

void
termination_handler (int signum)
{
  struct temp_file *p;

  for (p = temp_file_list; p; p = p->next)
    unlink (p->name);
}

int
main (void)
{
  ...
  struct sigaction yeni_eylem, eski_eylem;

  /* Yeni eylemi içeren yapıyı hazırlayalım. */
  yeni_eylem.sa_handler = termination_handler;
  sigemptyset (&yeni_eylem.sa_mask);
  yeni_eylem.sa_flags = 0;

  sigaction (SIGINT, NULL, &eski_eylem);
```

```
  if (eski_eylem.sa_handler != SIG_IGN)
    sigaction (SIGINT, &yeni_eylem, NULL);
  sigaction (SIGHUP, NULL, &eski_eylem);
  if (eski_eylem.sa_handler != SIG_IGN)
    sigaction (SIGHUP, &yeni_eylem, NULL);
  sigaction (SIGTERM, NULL, &eski_eylem);
  if (eski_eylem.sa_handler != SIG_IGN)
    sigaction (SIGTERM, &yeni_eylem, NULL);
  ...
}
```

Yazılım, **yeni_eylem** veri yapısını istenen parametrelerle yükler ve onu **sigaction** çağrısına aktarır. **sigemptyset** işlevinin kullanımı *Sinyallerin Engellenmesi* (sayfa: 631) bölümünde açıklanmıştır.

signal işlevinin kullanıldığı örnekte, evvelce yoksayılmaya ayarlanmış sinyalleri işlemekten kaçınmıştık. Bu örnekte ise yeni eylemi etkin kılmadan önce evvelki eylemi **sigaction** oluşumu sayesinde inceleme şansımız var. Böylece anlık bile olsa yoksayılmaya ayarlanmış bir sinyal eylemciyi değiştirmemiş oluyoruz.

Aşağıda başka bir örnek var. **SIGINT** sinyali için eylemi değiştirmeksizin mevcut eylem hakkında bilgi alıyoruz:

```
struct sigaction query_action;

if (sigaction (SIGINT, NULL, &query_action) < 0)
  /* sigaction hata durumunda -1 döndürüyor. */
else if (query_action.sa_handler == SIG_DFL)
  /* SIGINT öntanımlı olarak yakalanıyor, ölümcül durum. */
else if (query_action.sa_handler == SIG_IGN)
  /* SIGINT yoksayılıyor. */
else
  /* Tanımlanan sinyal yakalayıcı etkinleştiriliyor. */
```

3.5. sigaction Seçenekleri

sigaction veri yapısının **sa_flags** üyesi özel durumları belirtmek içindir. Çoğu durumda, **SA_RESTART** bu alanda kullanmak için iyi bir değerdir.

sa_flags üyesinin değeri bir bit maskesi olarak yorumlanır. Böylece çok sayıda seçenek belirtilebilir.

Her sinyalin kendine has seçenekleri vardır. her **sigaction** çağrısı belli bir sinyal için yapılır ve belirtilen seçenekler de sadece bu sinyale uygulanır.

GNU C kütüphanesinde, **signal** işlevi ile kurulan eylemci için bu seçenekler, değeri **siginterrupt** kullanımına bağlı olan **SA_RESTART** haricinde sıfıra ayarlanır. Bu durumla ilgili bilgiyi *Sinyallerle Kesilen İlkeller* (sayfa: 626) bölümünde bulabilirsiniz.

Bu makrolar `signal.h` başlık dosyasında tanımlanmıştır.

`int` **SA_NOCLDSTOP** makro

Bu seçenek sadece **SIGCHLD** sinyali için anlamlıdır. Bu seçenek etkin olduğunda sistem, durdurulan değil, sonlandırılan bir alt süreç olduğunda bu sinyali alır. Öntanımlı olarak, **SIGCHLD** sinyali hem durdurulan hem de sonlandırılan bir alt süreç olduğunda alınır.

Bu seçenek **SIGCHLD** dışında bir sinyal için belirtildiğinde etkisizdir.

`int` **SA_ONSTACK** makro

Bu seçenek belli bir sinyal için etkin olduğunda, sistem bu çeşit sinyalleri aldığında *sinyal yığıtını* (sayfa: 639) kullanır. Bu seçeneğin etkin olduğu bir sinyal alınırsa ve siz bir sinyal yığıtı oluşturmamışsanız, sistem yazılımınızı **SIGILL** sinyali ile sonlandırır.

`int` **SA_RESTART** makro

Bu seçenek, **open**, **read** ve **write** gibi ilkellerin bir sinyal aldıklarında nasıl davranacaklarını belirler ve sinyal yakalayıcı normal olarak döner. İki durum söz konusu olabilir: kütüphane işlevi ya çalışmasını sürdürür ya da **EINTR** hata kodu ile başarısız olur.

Seçimi belirleyen, sinyal alındığında **SA_RESTART**'ın etkin olup olmadığıdır. Etkinse, kütüphane işlevi çalışmasını sürdürür, değilse sinyal işlevin başarısız olmasına sebep olur. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

3.6. Sinyal Eylemlerinin İlk Durumu

Yeni bir süreç oluşturulduğunda (sayfa: 687), sinyal yakalayıcılar onu oluşturan süreçten miras alınır. Bununla birlikte, yeni süreci **exec** işlevi ile yüklediğinizde (*Bir Dosyanın Çalıştırılması* (sayfa: 688)), her sinyali **SIG_DFL** eylemine döndürecek kendi eylemcinizi tanımlamış olursunuz. (Burada durup biraz düşüneceksiniz, bu farklı birşey; eski yazılımın yakalama işlevleri ona özeldir ve yeni yazılımın adres alanında bunlar mevcut değildir.) Şüphesiz, yeni yazılım kendi eylemcilerini oluşturabilir.

Bir yazılım bir kabukta çalıştırıldığında, normalde kabuk, oluşturduğu alt sürecin eylemlerini duruma göre **SIG_DFL** ya da **SIG_IGN**'e ayarlar. Kendi sinyal yakalayıcınızı oluşturmadan önce, kabuğun alt sürece başlangıç olarak **SIG_IGN** eylemini belirtmediğinden emin olmanız için onu kontrol etmeniz iyi olur.

Bu örnekte, eğer yoksayılmıyorsa **SIGHUP** sinyali için bir eylemcinin nasıl kurulacağı gösterilmiştir:

```
...
struct sigaction temp;

sigaction (SIGHUP, NULL, &temp);

if (temp.sa_handler != SIG_IGN)
  {
    temp.sa_handler = handle_sighup;
    sigemptyset (&temp.sa_mask);
    sigaction (SIGHUP, &temp, NULL);
  }
```

4. Sinyal Yakalayıcıların Tanımlanması

Bu bölümde **signal** veya **sigaction** işlevi kullanılarak oluşturulan bir sinyal yakalama işlevinin nasıl yazılacağı anlatılmıştır.

Bir sinyal yakalama işlevi, yazılımınızın içinde derlenen bir işlevdir. Tek farkla, bu işlevi doğrudan siz çağırmazsınız, **signal** veya **sigaction** işlevini kullanarak bir sinyal geldiğinde işletim sisteminin bu işlevi çağırmasını sağlarsınız. Buna ***eylemci oluşturmak*** diyoruz. Bkz. *Sinyal Eylemlerinin Belirtilmesi* (sayfa: 611).

Bir eylemci işlevde kullanabileceğiniz iki strateji vardır:

- Bazı genel veri yapılarıyla oynarken sinyal alındığında çalışan bir eylemciniz olabilir ve bu normal olarak döner.
- Eylemci işleviniz süreci sonlandırabilir ya da denetimi, sinyali oluşturan durumu ortadan kaldıran bir yere taşımanızı sağlayabilir.

Eylemci işlevleri yazmak için özellikle yardıma ihtiyacınız olacak, çünkü bu işlevlerin ne zaman çağrılacağı hiç belli olmaz. Hatta çok kısa aralıkla iki sinyal birden alabilirsiniz ve bu durumda bir eylemcinin başka bir eylemciyi çalıştırması gerekebilir. Bu kısımda eylemci işlevi yazarken neleri yapmanız neleri yapmamanız gerektiği açıklanmıştır.

4.1. Dönen Sinyal Yakalayıcılar

Normal olarak dönen eylemciler genellikle, G/Ç ve süreçler arası iletişim sinyalleri ile **SIGALRM** benzeri sinyaller için kullanılır. Ancak, **SIGINT** sinyali için de bir eylemci dönebilir. Bir farkla, sürece uygun bir zamanda sonlanmasını söyleyen bir seçeneği etkinleştirerek döner.

Bir yazılım hata sinyali için normal olarak dönen bir işlev yazmak doğru olmayacaktır. Çünkü yazılımın davranışının bir yazılım hatası sinyali alındıktan sonra ne olacağı belli değildir. Bkz. *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

Normalde dönen eylemciler bir etki yaratması umulan bir genel değişkene değer atamalıdır. Bu değişken yazılımın çalışma anında belirli aralıklarla baktığı bir değişken olmalıdır. *Atomik Veri Erişimi ve Sinyal İşleme* (sayfa: 624) bölümünde açıklanan sebeplerle bu değişkenin veri türü **sig_atomic_t** olmalıdır.

Aşağıda, böyle bir yazılım örneği vardır. Bir **SIGALRM** sinyali alınıncaya kadar bir döngüyü çalıştırmaktadır. Bu teknik, döngü tamamlanmadan bir sinyal alınıncaya kadar yinelenen işlemler için yararlıdır.

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

/* Bu değişken döngünün denetimi içindir. */
volatile sig_atomic_t keep_going = 1;

/* Sinyal yakalayıcı değişkeni sıfırlar ve kendini tekrar etkinleştirir. */
void
catch_alarm (int sig)
{
  keep_going = 0;
  signal (sig, catch_alarm);
}

void
do_stuff (void)
{
  puts ("Uyarı gelene kadar birşeyler yapılıyor....");
}

int
main (void)
{
  /* SIGALRM sinyalleri için bir eylemci oluşturalım. */
  signal (SIGALRM, catch_alarm);

  /* kısa süreli bir uyarıyı etkinleştirelim. */
  alarm (2);

  /* Her çevrimde değişkenin değerine bakılsın. */
  while (keep_going)
    do_stuff ();

  return EXIT_SUCCESS;
```

```
}
```

4.2. Süreci Sonlandıran Eylemciler

Süreci sonlandıran eylemci işlevler genellikle düzenlenmiş bir durdurma için ya da yazılım hata sinyalleri ile etkileşimli kesmelerden kurtulmak amacıyla kullanılırlar.

Süreci sonlandıran bir eylemci için en iyi yöntem aynı sinyali eylemci çalıştığı anda tekrar yayınlamaktır. Bunun yapılışına bir örnek:

```
volatile sig_atomic_t fatal_error_in_progress = 0;

void
fatal_error_signal (int sig)
{
  /* Bu eylemci çok sayıda sinyal çeşidi için kurulduğundan,
     diğer sinyaller için de defalarca çağrılabilecektir. Bu
     durumu izlemek için bir durağan değişken kullanacağız. */
  if (fatal_error_in_progress)
    raise (sig);
  fatal_error_in_progress = 1;

  /* Şimdi biraz temizlik yapalım:
     - uçbirim kipleri sıfırlansın
     - alt süreçler ölsün
     - kilit dosyaları silinsin */
  …

  /* Şimdi sinyali tekrar yayınlayalım. Süreci sonlandırması için
     sinyalin öntanımlı eylemini etkin kılıyoruz.
     Tam bu anda exit veya abort
     çağrısı yapabilmeli ve sürecin çıkış durumunun doğru ayarlanması
     için sinyali yeniden yayınlamalıyız. */
  signal (sig, SIG_DFL);
  raise (sig);
}
```

4.3. Eylemci İşlevlerde Denetimin Aktarımı

Bir sinyal yakalayıcı için **setjmp** ve **longjmp** oluşumları kullanılarak denetim başka bir yere aktarılabilir. Bkz. *Yerel Olmayan Çıkışlar* (sayfa: 593).

Bir eylemci denetimi dışarı aktardığı zaman, çalışmakta olan yazılım kalan işlemi biteremez. Örneğin yazılım o anda önemli bir veri yapısını güncelliyorsa, veri yapısı belirsiz bir durumda kalacaktır. Süreç sonlandırıldığından belirsizlik daha sonra benzer şekilde bildirilecektir.

Bu sorundan kaçınmanın iki yolu vardır. Biri, önemli veri yapısının güncellenmesi bitene kadar sinyalin engellenmesidir. Bkz. *Sinyallerin Engellenmesi* (sayfa: 631).

Diğer yol ise, önemli veri yapısını eylemci içinde yeniden ilklendirmek ve değerlerini belirli yapmaktır.

Örnekte, bir genel değişkenin yeniden ilklendirilmesi gösterilmiştir:

```
#include <signal.h>
#include <setjmp.h>

jmp_buf return_to_top_level;
```

```
volatile sig_atomic_t waiting_for_input;

void
handle_sigint (int signum)
{
  /* Sinyal alındığında girdi için beklemeliyiz, ama denetimi
     aktaracağımızdan artık bekleyemeyiz.  */
  waiting_for_input = 0;
  longjmp (return_to_top_level, 1);
}

int
main (void)
{
  ...
  signal (SIGINT, sigint_handler);
  ...
  while (1) {
    prepare_for_command ();
    if (setjmp (return_to_top_level) == 0)
      read_and_execute_command ();
  }
}

/* Bunun çeşitli komutlar için kullanılan bir yordam
   olduğunu hayal edin. */
char *
read_data ()
{
  if (input_from_terminal) {
    waiting_for_input = 1;
    ...
    waiting_for_input = 0;
  } else {
    ...
  }
}
```

4.4. Eylemci Çalışırken Sinyal Alınması

Sinyal yakalama işlevi çalışırken başka bir sinyal alındığıda ne olacak?

Belli bir sinyal için eylemci işlev çağrıldığında eylemci işlemlerini bitirene kadar sinyal özdevinimli olarak engellenir. Bu, aynı iki sinyal peşpeşe alınırsa, biri için işlem tamamlanana kadar diğeri engellenecek demektir. (Bu türden birden fazla sinyal geldiğinde bunlara da izin vermek isterseniz, **sigprocmask** kullanarak sinyalin engellenmemesini sağlayabilirsiniz; bkz. *Sürecin Sinyal Maskesi* (sayfa: 633).)

Bununla birlikte, eylemci işleviniz başka çeşit bir sinyal ile hala durdurulabilir durumdadır. Bundan kaçınmak için, **sigaction** ile kullanılan veri yapısının **sa_mask** üyesinde eylemci çalışırken hangi sinyallerin engelleneğini belirtebilirsiniz. Bunlar eylemcinin çağrılmasına sebep olan sinyale ek olarak belirtilir ve diğer sinyaller normal olarak süreç tarafından engellenir. Bkz. *Eylemci Çalışırken Sinyallerin Engellenmesi* (sayfa: 634).

Eylemci işini bitirdiğinde engellenen sinyaller eylemcinin çalıştırılmadan önceki durumlarına dönerler. Bu durumda, eylemci işlev içinde **sigprocmask** kullanılması sadece eylemcinin çalışması sırasında gelen sinyalleri etkiler, eylemci işlev döndükten sonra gelen sinyalleri etkilemez.

Taşınabilirlik Bilgisi

Yazılımınızın System V Unix üzerinde gerektiği gibi çalışmasını istiyorsanız ve eşzamansız bir sinyal alacağınızı umuyorsanız, bu sinyalin eylemcisini oluşturmak için daima **sigaction** kullanın. Bu sistem üzerinde **signal** ile oluşturulan bir eylemci ile sinyalin yakalanması özdevinimli olarak sinyal eyleminin öntanımlı eyleme yani **SIG_DFL**'a ayarlanmasına sebep olur. Böyle bir eylemci her çalıştığında kendini tekrar kurmalıdır. Bu uygulama, rahatsız edici olsa da, eylemci çalışırken sinyaller işleme alınamadığında da çalışır. Şöyle ki, hemen ardından başka bir sinyal gelebilir ve eylemci kendini daha kurmadan sinyal alınabilir. Bu durumda ikinci sinyal öntanımlı eylem ile karşılanacak ve süreç sonlanabilecektir.

4.5. Eylemci Çalışmadan İkinci Bir Sinyalin Alınması

Süreciniz bir sinyal yakalayıcının çalışmasından hemen önce aynı türden çok sayıda sinyal alırsa, yakalayıcı tek bir sinyal alınmış gibi sadece bir kere çalıştırılabilir. Gerçekte ise, sinyaller tek bir sinyal içine katıştırılmış olur. Bu durum, sinyal engellendiğinde ya da çok süreçli bir ortam da sistemin başka bir süreçle meşgul olduğu bir sırada ortaya çıkabilir. Bu, örneğin, bir sinyal yakalayıcıyı bir sinyal sayacı olarak kullanamayacağınız anlamına gelir. Ayırdına varacağınız tek şey, bir kerede en azından bir sinyal almış olduğunuz olacaktır.

Aşağıda, alt süreç tarafından üretilen sinyallerin alınan sinyallerin sayısına eşit olmayışını telafi eden bir **SIGCHLD** yakalayıcısı örneği verilmiştir. Burada yazılımın, alt süreçlerin izini sürmek için bir yapı zinciri kullandığı varsayılmıştır:

```
struct process
{
  struct process *next;
  /* Bu sürecin süreç kimliği.  */
  int pid;
  /* Bu süreçten gelen çıktıların yönlendirileceği
     uçbirim ya da boru tanımlayıcı. */
  int input_descriptor;
  /* Bu süreç durdurulursa ya da sonlandırılırsa
     değişkenin değeri sıfırdan farklı olacak.  */
  sig_atomic_t have_status;
  /* Bu sürecin durumu; çalışıyorsa sıfırdır,
     aksi takdirde waitpid'deki durum değeridir. */
  int status;
};

struct process *process_list;
```

Bu örnek, hemen öncesinde bazı sinyallerin alınmasına bağlı olarak bir değişken de kullanıyor. Her seferinde yazılımın sonunda bu sıfırlanıyor.

```
/* Sıfırdan farklı bir değer alt sürecin durumunun
   değiştiği anlamına gelir. Bu durumla ilgili ayrıntılar
   için process_list'e bakılmalı. */
int process_status_change;
```

Eylemci:

```
void
sigchld_handler (int signo)
{
  int old_errno = errno;
```

```
  while (1) {
    register int pid;
    int w;
    struct process *p;

    /* Tanımlanabilir bir sonuç alana kadar sorguyu sürdürüyoruz.  */
    do
      {
        errno = 0;
        pid = waitpid (WAIT_ANY, &w, WNOHANG | WUNTRACED);
      }
    while (pid <= 0 && errno == EINTR);

    if (pid <= 0) {
      /* Bir alt süreç kalmamışsa çıkabiliriz.  */
      errno = old_errno;
      return;
    }

    /* Sinyal gönderen süreci bulalım ve durumunu kaydedelim.  */

    for (p = process_list; p; p = p->next)
      if (p->pid == pid) {
        p->status = w;
        /* Durum alanınnı bir veri içerdiğini belirtelim.
           Bunu onu sakladıktan sonra yapıyoruz.  */
        p->have_status = 1;

        /* Süreç sonlandırılmışsa çıktısını beklemekten vazgeçiyoruz.  */
        if (WIFSIGNALED (w) || WIFEXITED (w))
          if (p->input_descriptor)
            FD_CLR (p->input_descriptor, &input_wait_mask);

        /* Yazılım arasıra bu değişkene süreç listesinde yeni bir
           süreç var mı diye bakmalı.  */
        ++process_status_change;
      }

    /* Bize söyleyecek birşeyleri var mı diye tekrar dönüp tüm
       süreçlere bakalım.  */
  }
}
```

process_status_change değişkenini denetlemek için bir yöntem:

```
if (process_status_change) {
  struct process *p;
  process_status_change = 0;
  for (p = process_list; p; p = p->next)
    if (p->have_status) {
      … p->status incelemesi …
    }
}
```

Listeyi incelemeye başlamadan önce seçeneğin temizlenmesi hayati önemdedir; aksi takdirde, seçeneğin temizlenmesinden önce bir sinyal alınırsa ve süreç listesinin ilgili elemanı etkinse, bu seçenekle ilgili olarak sinyal hakkında bir uyarı alınamayacaktır. Bu sorundan kaçınmak için listeyi taramaya başlamadan önce seçeneği

temizlemeniz gerekir; bazı işlemleri doğru sırada yapmak önemlidir.

Döngüde **p->status** alanı incelenerek, sürecin durumu hakkında bilgi edinilmeye çalışılır. **p->have_status** etkin bir değere sahipse, süreç durdurulmuş ya da sonlandırılmıştır; değilse, yazılım tekrar uyarı alana kadar durdurulamamış ya da sonlandırılamamıştır. Bir değişkene erişim sırasındaki kesmelerin kaydedilmesi hakkında daha fazla bilgi için *Atomsal Kullanım Şekilleri* (sayfa: 626) bölümüne bakınız.

Bşaka bir yol da, sinyal yakalayıcının son sınamadan beri çalıştırılıp çalıştırılmadığına bakmaktır. Bu teknikte, sinyal yakalayıcının dışında değiştirilmeyen bir sayaç kulanılır. Sayacı sıfırlamak yerine, yazılım sayacın son değerini hatırlayarak önceki sınamadan beri bir değişiklik olup olmadığına bakar. Bu yöntemin bir faydası da, yazılımın parçalarının birbirinden bağımsız olarak denetlenebilmesidir.

```
sig_atomic_t process_status_change;

sig_atomic_t last_process_status_change;


…
{
  sig_atomic_t prev = last_process_status_change;
  last_process_status_change = process_status_change;
  if (last_process_status_change != prev) {
    struct process *p;
    for (p = process_list; p; p = p->next)
      if (p->have_status) {
        … p->status incelemesi…
      }
  }
}
```

4.6. Sinyal İşleme ve Evresel Olmayan İşlevler

Sinyal yakalama işlevleri genelde çok zor değildir. En iyi çözüm, hiçbir şey yapmayan ama yazılımın sürekli sınadığı bir harici değişkene değer atayan ve bu değişkenle ilgili işlemleri yazılıma bırakan bir kod yazmaktır. En iyisi budur çünkü, eylemci işlev rasgele, umulmadık bir zamanda, basit bir işlevin ortasında ve hatta çok sayıda makina kodu komut gerektiren C işleçlerinin başlangıcı ile sonu arasında çağrılabilir. Üzerinde çalışılan veri yapıları eylemci işlevler çağrıldığında kararsız durumda bile olabilirler. Bir **int** türünden değişkenin diğerine kopyalanması çoğu makinada iki makina komutundan oluşur.

Bu, bir sinyal yakalayıcı ile ne yapmaya hazırlandığınıza bağlı olarak çok dikkatli olmak zorunda olduğunuz anlamına gelir.

- Eğer eylemci işlevin yazılımınızdan herhangi bir genel değişkene erişmesi gerekiyorsa bu değişkenleri **volatile** olarak bildirin. Bu, derleyiciye bu değişkenin herhangi bir anda değişebileceğini ve bu tür değişikliklere göre değerlendirilmiş eniyilemelerin yapılmamasını söyler.

- Eylemci içinden bir işlev çağırıyorsanız, onun sinyallerle ilgili olarak ***evresel*** olduğundan, değilse, başka bir ilgili işlev tarafından sinyalin engellenmediğinden emin olmalısınız.

Bir işlev yığıtı değil de belleği kullanıyorsa evresel olmayabilir.

- Bir işlev bir durağan veya bir genel ya da sadece kendinin erişebildiği bir özdevimli ayrılmış nesne kullanıyorsa, evresel olmayabilir ve işlevin herhangi iki çağrısı bir diğeri ile etkileşebilir.

  Örneğin sinyal yakalayıcının **gethostbyname** işlevini kullandığını varsayalım. Bu işlev değerini bir durağan nesne içinde döndürür. Eğer **gethostbyname** çağrısı sırasında ya da çağrının ardından (yazılım hala bu değeri kullanıyorken) bir sinyal gelirse, yazılımın istediği değerin taşmasına sebep olabilir.

Diğer yandan, yazılım **gethostbyname** işlevini ya da aynı nesnedeki bilgiyi döndüren herhangi bir başka işlev kullanmıyorsa ya da her kullanımda daima sinyaller engelleniyorsa güvendesiniz demektir.

Değeri bir sabit nesne içinde döndüren ve bu şekilde aynı nesnenin daima yeniden kullanılabilir olmasını sağlayan çok sayıda kütüphane işlevi vardır ve bunların tamamı aynı soruna yol açabilir. Bu kılavuzdaki işlev açıklamalarında bu davranış daima açıklanmıştır.

- Eğer bir işlev, sizin tanımladığınız bir nesneyi kullanıyor ve değiştiriyorsa, büyük ihtimalle işlev evresel değildir; bu işlevlerin aynı nesneyi kullanması halinde birbirlerini etkileyebilirler.

  Akımlarla G/Ç işlemleri yaptığınızda bu durum ortaya çıkabilir. Bir sinyal yakalayıcını **fprintf** işlevi ile bir ileti bastığını varsayalım. Tam da aynı akımı kullanan bir **fprintf** çağrısının ortasında yazılımın bir sinyal aldığını varsayalım. Hem sinyal eylemcinin hem de yazılımın verisi bozulurdu, çünkü her iki çağrı kendi akımı üzerinde aynı veri yapısıyla çalışıyor olacaklardı.

  Bununla birlikte, eylemcinin kullandığı akımın sinyal geldiğinde yazılım tarafından kullanılması mümkün olmayabilir ki, bu durumda güvendesiniz demektir. Yazılım başka bir akımı kullanıyorsa sorun yoktur.

- Çoğu sistemde, hangi bellek bloklarının serbest bırakılacağını kaydettikleri bir durağan veri yapısı kullandıklarından **malloc** ve **free** evresel değildir. Sonuç olarak, bellek ayıran ve serbest bırakan kütüphane işlevleri evresel olmayacaktır. Bu, bir sonucu saklayacağı alanı ayıran işlevleri de kapsar.

  Bir yakalama işlevi içinde bellek ayırma ihtiyacından kaçınmanın en iyi yolu, işlev için kullanılacak alanı önceden ayırmaktır.

  Bir yakalama işlevi içinde belleği serbest bırakma ihtiyacından kaçınmanın en iyi yolu ise, serbest bırakılacak nesneleri kaydetmek ya da imlemek ve yazılımda zaman zaman bu türde serbest bırakılmayı bekleyen nesneler olup olmadığına bakmaktır. Fakat bu dikkatli yapılmalıdır çünkü bir nesnenin bir zincire yerleştirilmesi atomik değildir ve başka bir sinyal yakalayıcı ile işlem kesmeye uğratılırsa nesnelerden birinin kaybedilmesi gibi şeyler olabilir.

- **errno** değişkenini değiştiren işlevler evresel değildir, fakat bunu düzeltebilirsiniz: eylemcide **errno** değişkeninin orjinal değerini kaydedip normal olarak dönmeden önce eski yerine koyabilirsiniz. Bu, sistemin bir sinyalle tetiklediği eylemcinin çalışmaya başlamasıyla alınması engellenen hataların eylemcinin işi bittiğinde elde edilebilmesini sağlar.

  Bu teknik genellikle uygulanabilirdir; belli bir nesneyi bellekte değiştiren bir işlevi bir sinyal yakalayıcı içinden çağırmak isterseniz, bunu nesneyi kaydederek ve sonra eski değeri yerine koyarak rahatça yapabilirsiniz.

- Sadece, bir sinyal alındığında bir bellek nesnesinin okunması güvenli olabilir. Ancak, bazı veri türlerinde atama işleminin birden fazla makina komutuna malolduğunu unutmayın, bir sinyal yakalayıcı çalışmaya başladığı anda eğer *atomik değilse bir değişkene yapılan atama* (sayfa: 624) işleminin arasına girebilir.

- Bir sinyal alındığında, bir bellek nesnesine yazma işlemi sadece değerin olabildiğince anlık bir değişikliğinde güvenli olabilir. Bu durumda eylemcinin çalışması birşeyi bozmaz.

4.7. Atomik Veri Erişimi ve Sinyal İşleme

Uygulama verinizin atomsal ya da sırt metin olup olmamasına bağlı olarak, atomik olmayan tek bir veriye erişirken dikkatli olmalısınız. Bir nesnenin okunması ya da yazılması tek bir makina kodu ile gerçekleşemeyebilir, böyle bir durumda bir sinyal yakalayıcı işlemin arasına girebilir ve işlem yarıda kalabilir.

Bu sorunla ilgili olarak uygulanabilecek üç yöntem vardır. Veri türlerine erişimi atomsal yapabilirsiniz; işlemi dikkatlice yapılandırabilir böylece erişimin kesilmesi sözkonusu ise böyle bir veri erişimini hiç yapmazsınız ya da veriye erişirken işlem sırasında sinyallerin tamamını *engellersiniz* (sayfa: 631).

4.7.1. Atomsal Olmayan Veriye Erişimle İlgili Sorunlar

Bu örnekte, bir değişkenin değerinin değiştirilmesi sırasında bir sinyal yakalama işlevinin araya girdiği durum gösterilmiştir. (Bir değişkenin okunması sırasında kesintiye uğratılması mantıksız sonuçlara sebep olabilir, ama burada sadece yazma olayı gösterilmiştir.)

```
#include <signal.h>
#include <stdio.h>

struct two_words { int a, b; } memory;

void
handler(int signum)
{
    printf ("%d,%d\n", memory.a, memory.b);
    alarm (1);
}

int
main (void)
{
    static struct two_words zeros = { 0, 0 }, ones = { 1, 1 };
    signal (SIGALRM, handler);
    memory = zeros;
    alarm (1);
    while (1)
      {
        memory = zeros;
        memory = ones;
      }
}
```

Yazılım, **memory** değişkenini sırayla sıfırlarla ve birlerle doldurmaktadır. Bu sırada her saniyede bir, alarm sinyalinin yakalayıcısı o anki içeriği basmaktadır. (Eylemci içinde **printf** kullanımı, sinyal oluştuğu sırada **printf** eylemci dışında çağrılmadığından bu kod için sorun çıkarmaz.)

Başka bir deyişle yazılım sıfır ya da bir çiftlerinde birini basar. Ama bu iş tamamen bizim istediğimiz gibi olmaz! Çoğu makinada, **memory** değişkeninde yeni değerin saklanması bir makina komutundan fazlasına ihiyaç duyar. Bu komutların çağrılması sırasında bir sinyal alınırsa, eylemci **memory.a** için 0, **memory.b** için 1 ya da tersini basabilir (Normalde ikisi de aynı olmalıydı).

Bazı makinalar, işlemi tek bir makina kodu ile yapabilir ve bu durumda eylemci daima sıfır ya da bir çiftleri basar.

4.7.2. Atomsal Türler

Bir değişkene erişimin ne olursa olsun kesintiye uğratılamamasını sağlam için erişimin daima atomsal olduğu veri türünü kullanabilirsiniz: **sig_atomic_t**. Bu veri türündeki bir değişkene yapılan bir okuma veya yazma işleminin tek bir makina kodu ile gerçekleştirilmesi garanti edilmiştir. Böylece bir sinyal yakalayıcının işlemin arasına girmesi engellenmiş olur.

sig_atomic_t türü daima bir tamsayı veri türüdür, ama hangisi olduğu yani kaç bit genişlikte olduğu makinadan makinaya değişebilir.

sig_atomic_t veri türü

Bu bir tamsayı veri türüdür. Bu türdeki nesnelere erişim daima atomsaldır.

Uygulamada, **int** türünü atomsal olduğunu varsayabilirsiniz. Ayrıca çok elverişli olmasa da gösterici türlerininde atomsal olduğunu varsayabilirsiniz. GNU C kütüphanesini destekleyen makinalarda ve bilinen tüm POSIX sistemlerinde her iki kabulde geçerlidir.

4.7.3. Atomsal Kullanım Şekilleri

Bir erişimin kesmeye uğratılmasından kaçınılmasını sağlayan belli erişim şekikilleri vardır. Örneğin, sinyal yakalama işlevi tarafından bir seçenek etkinleştirilebilir ve ana yazılım tarafından bu değişken zaman zaman 1 ya da 0 yapılabilir. Böylece, bir veriye erişim için iki komut gerekiyorsa bu işlem güvenceye alınmış olur. Bunun böyle olduğunu göstermek için, veriye her erişimin kesmeye uğratılmaya çalışıldığını kabul edeceğiz, ancak kesmeye uğratılsa bile bunun bir sorun oluşturmadığını göstereceğiz.

Seçeneğin sınanması sırasında bir kesme sorun çıkarmayacaktır, çünkü ya değerin kesinliğinin önemi olmadığı duruma karşılık değer sıfırdan farklı olacak ya da bir sonraki sınama için sıfırdan farklı görünecektir.

Seçeneğin sıfırlanması sırasında da bir kesme sorun çıkarmayacaktır, çünkü seçenek sıfırlanmadan önce bir sinyal geldiğinde ne olacağına bağlı olarak ya değer sıfır olacak ya da sıfırdan farklı olacaktır.Kod her iki durumu da olması gerektiği gibi mümkün olduğunca iyi elde eder, ayrıca seçeneğin sıfırlanması sırasında bir sinyal yakalama işlemi de yapılabilir.

Bazan, bir nesneye erişimin kesintiye uğratılmamasını, nesnenin başka bir nesne tarafından korunmasını sağlayarak sağlama alabilirsiniz, bu da atomsallığı garanti etmenin yollarından biridir. Örnek için *Eylemci Çalışmadan İkinci Bir Sinyalin Alınması* (sayfa: 621) bölümüne bakınız.

5. Sinyallerle Kesilen İlkeller

open veya **read** benzeri bir G/Ç ilkeli bir G/Ç aygıtını beklerken bir sinyal alabilir ve bu sinyal işleme sokulabilir. Sinyalin eylemcisi işlemlerini bitirdikten sonra sistem bir sorunla başbaşa kalır: şimdi ne olacak?

POSIX bir yaklaşım belirtir: bir ilkel başarısız olduğunda ne yapacaksan beklemeden yap. Bu çeşit başarısızlıklar için hata kodu **EINTR**'dir. Genellikle, sinyal eylemciler kullanılan POSIX uygulamaları bu hatayı döndüren her işlev çağrısının dönüş durumuna mutlaka bakmalıdır. Çoğunlukla yazılımcılar bu genel hata kaynağına bakmayı unuturlar.

GNU kütüphanesi, geçici bir başarısızlığın ardından çağrının yinelenmesini sağlayan oldukça kullanışlı bir yöntem olan **TEMP_FAILURE_RETRY** makrosunu içerir:

TEMP_FAILURE_RETRY (*ifade*) makro

Bu makro *ifade*'yi bir kere değerlendirir ve **long int** türünde döner. Eğer değer **-1** ise bu bir başarısızlık gösterir ve **errno** değişkenine hata durumu atanır. Eğer başarısız olursa ve **EINTR** hata kodunu raporlarsa, **TEMP_FAILURE_RETRY** onu tekrar değerlendirmeye tabi tutar ve bu işlem geçici başarısızlık durumu ortadan kalkana dek yinelenir.

TEMP_FAILURE_RETRY makrosunun dönüş değeri *ifade*'nin dönüş değeri neyse odur.

BSD, **EINTR** hata kodunu hiç göstermez ve daha iyi bir yaklaşım yapar: kesintiye uğratılan ilkeli başarısız olarak döndürmez ve hep yeniden başlatır. Bu yaklaşımı benimserseniz **EINTR** ile ilgilenmeniz gerekmez.

GNU kütüphanesindeki yaklaşımı seçebilirsiniz. Bir sinyal yakalayıcıyı **sigaction** ile kuruyorsanız, eylemcinin nasıl davranacağını belirtebilirsiniz. **SA_RESTART** seçeneğini belirtirseniz eylemci döndükten sonra ilkel kaldığı yerden işine devam eder, belirtmezseniz eylemci **EINTR** hatasının dönmesine sebep olacaktır. Bkz. *sigaction Seçenekleri* (sayfa: 616).

Seçiminizi belirtmenin diğer bir yolu da **siginterrupt** işlevidir. Bkz. *BSD Eylemciler* (sayfa: 641).

Bir sinyal yakalayıcının ne yapacağını **sigaction** veya **siginterrupt** ile belirtmezseniz, öntanımlı seçim kullanılır. GNU kütüphanesinde öntanımlı seçim sizin tanımladığınız sınama makrolarına bağlıdır. **signal** işlevini çağırmadan önce **_BSD_SOURCE** veya **_GNU_SOURCE** makrosunu tanımlarsanız, öntanımlı davranış ilkelin işlemi kaldığı yerden devam ettirmesidir; aksi takdirde, öntanımlı davranış **EINTR** ile başarısızlık olacaktır. (Kütüphane **signal** işlevinin diğer sürümlerini de içerir ve özellik sınama makroları gerçekte hangisinin kullanılacağının saptanmasını sağlar.) Bkz. *Özellik Sınama Makroları* (sayfa: 25).

Bu kısımda bahsedilen her ilkel, hata kodu olarak **EINTR** döndürebilen ilkellerden biridir.

Seçimizden etkilenmeyen ve işlemin kaldığı yerden devamına konu olmayan tek bir durum vardır: **read** veya **write** gibi bir veri aktarım işlevi verinin bir parçasını aktardıktan sonra bir sinyal aldığında. Bu durumda işlev zaten, kısmi başarıyı belirtmek üzere aktarılan baytların sayısı ile dönecektir.

Bunun en başta kayıt yönlenimli aygıtlarda (datagram soketleri gibi; bkz. *Datagram Soket İşlemleri* (sayfa: 433)) beklenmeyen davranışlara sebep olduğu görülür; bir **read** ya da **write** işleminin ikiye bölünmesi iki okuma ya da yazmaya sebep olur. Aslında, bir sorun yoktur, çünkü böyle aygıtlarda bir kısmi aktarım sonrası kesme oluşamaz; bunlar bir kaydın tümünü veri aktarımı bir kez başladı mı beklemeksizin tek bir seferde aktarırlar.

6. Sinyallerin Üretilmesi

Sinyallerin bir donanım kapanı ya da kesmesinin sonucu olarak üretilmesine ilaveten, yazılımınız da kendisine ya da başka süreçlere doğrudan doğruya sinyal gönderebilir.

6.1. Kendine Sinyal Gönderme

Bir sürecin kendine sinyal göndermesi için **raise** işlevi kullanılır. Bu işlev `signal.h` başlık dosyasında bildirilmiştir.

`int` **raise**`(int` *sinyalnum*`)` işlev

işlev çağrıldığı sürece *sinyalnum* sinyalini gönderir. İşlem başarılı olursa sıfırla, aksi takdirde sıfırdan farklı bir değerle döner. Başarısızlığın tek sebebi *sinyalnum* değerinin geçersiz olmasıdır.

`int` **gsignal**`(int` *sinyalnum*`)` işlev

raise ile aynı işi yapar; sadece SVID uyumluluğu için vardır.

raise işlevinin genellikle kullanıldığı tek yer yakalanan sinyalin öntanımlı davranışını yeniden üretmektir. Örneğin, yazılımınızın bir kullanıcısının SUSP karakterini (**C-z**; bkz. *Özel Karakterler* (sayfa: 454)) tuşladığını varsayalım. Bu işlem yazılıma etkileşimli durdurma sinyalinin (**SIGTSTP**) gönderilmesine sebep olur. Bu durumda siz de durdurma öncesi bazı tamponları temizlemek istersiniz. Bunu da şöyle yaparsınız:

```
#include <signal.h>

/* Bir durdurma sinyali geldiğinde, eylemi önce öntanımlı eyleme
   ayarlayalım, temizliği yaptıktan sonra da sinyali yeniden
   gönderelim. */

void
tstp_handler (int sig)
{
  signal (SIGTSTP, SIG_DFL);
  /* Temizlik işlemleri */
  ...
```

```
  raise (SIGTSTP);
}

/* Süreç çalışmaya kaldığı yerden devam edeceği zaman
   sinyal eylemciyi yeniden kuralım. */

void
cont_handler (int sig)
{
  signal (SIGCONT, cont_handler);
  signal (SIGTSTP, tstp_handler);
}

/* Her iki eylemciyi de yazılım başlatıldığında etkinleştirelim. */

int
main (void)
{
  signal (SIGCONT, cont_handler);
  signal (SIGTSTP, tstp_handler);
  ...
}
```

Taşınabilirlik Bilgisi

raise işlevi ISO C komisyonu tarafından tasarlanmıştır. Daha eski sistemler işlevi desteklemeyebilir, bu bakımdan **kill** işlevinin kullanılması daha taşınabilir olacaktır. Bkz. *Başka Bir Sürece Sinyal Gönderme* (sayfa: 628).

6.2. Başka Bir Sürece Sinyal Gönderme

kill işlevi bir sinyalin başka bir sürece gönderilmesi için kullanılır. İsmine rağmen, bir sürecin sonlandırılmasına sebep olmaktan farklı birşeyler yapmak için de kullanılabilir. Süreçler arasında sinyal gönderilmesini gerektiren durumlara ilişkin bazı örnekler:

- Bir süreç bir işlemi yerine getirmesi için kendini bir alt süreç olarak çalıştırabilir — bir altsüreç çalıştırılması bir kısır döngü oluşturabilir — ve işlem yerine getirildiğinde alt sürecini sonlandırabilir.
- Bir hata oluştuğunda ya da bir olay gerçekleştiğinde, bir süreç, bir grubun parçası olarak çalıştırılıp gruptaki başka bir süreci uyarmak ya da sonlandırmak için kullanılabilir.
- Birlikte çalışan iki sürecin eşzamanlanması gerekir.

Bu bölümde *bir sürecin nasıl çalıştığını* (sayfa: 685) bildiğinizi varsayacağız.

kill işlevi `signal.h` dosyasında bildirilmiştir.

```
int kill(pid_t pid,                                                    işlev
         int   sinyalnum)
```

kill işlevi *pid* ile belirtilen süreç ya da süreç grubuna *sinyalnum* sinyalini gönderir. *Standart Sinyaller* (sayfa: 604) bölümünde listelenen sinyallere ilaveten, ayrıca *pid* süreç kimliğini doğrulamak için sıfır değerini de kullanabilirsiniz.

Sinyal alacak süreç veya süreç grubunu belirten *pid* değerleri ve anlamları:

pid> 0
: Belirteci *pid* olan süreç.

pid== 0
: Gönderen ile aynı gruptaki süreçlerin tümü.

pid< -1
: Belirteci *–pid* olan süreç grubu.

pid== -1
: Eğer süreç ayrıcalıklı ise, sinyal, bazı özel sistem süreçleri dışında kalan tüm süreçlere gönderilir. Aksi takdirde, sinyal, aynı etkin kullanıcı kimlikli tüm süreçlere gönderilir.

Bir süreç **kill (getpid(), *sinyal*)** gibi bir çağrı ile kendisine bir sinyal gönderebilir ve sinyal engellenmez, sonrasında **kill** dönmeden önce sürece en az bir sinyal (*sinyalnum* yerine beklemede olan engellenmeyen sinyaller gidebilir) gönderir.

Sinyal gönderme başarılı olduğunda **kill** sıfır ile döner. Aksi takdirde sinyal gönderilmemiş demektir ve **-1** ile döner. Eğer *pid* bir sinyalin birden fazla sürece gönderilmesini belirtiyorsa, en azından bir sürece sinyal gönderilebilmişse **kill** sıfır ile dönecektir. Sinyali alan ve alamayan süreçlerin hangileri olduğunu saptayacak bir yöntem yoktur.

Bu işlev için tanımlanmış olan **errno** değerleri:

EINVAL
: *sinyalnum* geçersiz ya da desteklenmeyen bir numara.

EPERM
: *pid* ile belirtilen sürece ya da süreç grubundaki herhangi bir sürece bir sinyal göndermeye yetkili değilsiniz.

ESCRH
: *pid* mevcut bir süreci veya grubu betimlemiyor.

```
int killpg(int pgid,                                                    işlev
           int sinyalnum)
```

sinyalnum sinyalini *pgid* ile belirtilen süreç grubuna göndermesi dışında **kill** gibidir. Bu işlev BSD uyumluluğu için vardır. Bunun yerine **kill** kullanmak yazılımınızı daha taşınabilir yapar.

kill kullanımına basit bir örnek olan
kill (getpid (), *sinyal*) çağrısı
raise (*sinyal*) çağrısı ile aynı etkiye sahiptir.

6.3. kill ile İlgili Sınırlamalar

Herhangi bir sürece **kill** kullanarak sinyal göndermenizi engelleyen bazı sınırlamalar vardır. Bunlar, başka bir kullanıcının bir süreci kendi kararının bir sonucu olarak öldürmesi gibi asosyal davranışlara karşı tasarlanmıştır. **kill** genellikle bir sürecin kendi altsüreçlerine ve kardeş süreçler arasında sinyal gönderilmesi için kullanılır ve bu çeşit kullanımda yetkileriniz yeterli olur. Alışılmışın dışında olan tek durum bir alt süreç olarak bir setuid yazılımın çalıştığı durumdur; eğer yazılım sürecin hem gerçek hem de etkin kullanıcı kimliğini değiştiriyorsa, sürece sinyal gönderecek yetkiniz olmayabilir. **su** komutu bunu yapar.

Bir sürece sinyal gönderme yetkiniz olup olmadığına iki sürecin kullanıcı kimliklerine bakarak karar verilir. Bu kavram *Bir Sürecin Aidiyeti* (sayfa: 743) bölümünde ayrıntılı olarak açıklanmıştır.

Genelde, bir sürecin başka bir sürece sinyal gönderebilmesi için ya gönderen süreç ayrıcalıklı kullanıcıya ait olmalı (**root** gibi) ya da gönderen sürecin gerçek ve etkin kullanıcı kimliği alan sürecinki ile aynı olmalıdır. Eğer alan süreç kendi süreç imge dosyasında set–user–ID bitiyle etkin kullanıcı kimliğini değiştirirse, etkin kullanıcı kimlik yerine süreç imge dosyasının sahibi kullanılır. Bazı gerçeklemelerde, bir süreç kendi alt sürecine kullanıcı kimlikleri aynı olmasa bir sinyal gönderebilir olmalıyken bazıları da başka sınırlamalar getirebilir.

SIGCONT sinyali özel bir durumdur. Eğer gönderici ve alıcı aynı oturumun parçaları iseler kullanıcı kimliklerine bakılmaksızın bu sinyali gönderilebilir.

6.4. `kill` Örneği

Burada süreçler arası iletişim için sinyallerin kullanıldığı daha kapsamlı bir örneğe yer verilmiştir. **SIGUSR1** ve **SIGUSR2** sinyalleri ile neler yapılabileceği gösterilmiştir. Bu sinyaller öntanımlı olarak ölümcül olduklarından, sürecin bu sinyalleri **signal** veya **sigaction** ile yakalayacakları varsayılır.

Bu örnekte, bir süreç bir alt süreci çatalladıktan sonra alt sürecin kendini ilklendirmesini beklemekte, alt süreç ise **kill** işlevini bir **SIGUSR1** sinyalini göndermek için kullanarak hazır olduğunu bildirmektedir.

```
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

/* SIGUSR1 sinyali alındığında bu değişken 1 olacak. */
volatile sig_atomic_t usr_interrupt = 0;

void
synch_signal (int sig)
{
  usr_interrupt = 1;
}

/* Bu işlevi alt süreç çalıştıracak. */
void
child_function (void)
{
  /* İlklendirme bitmiş demektir. */
  printf ("Hazirim!!!  Surec kimligim: %d.\n", (int) getpid ());

  /* Üst sürece de bildirmek lazım. */
  kill (getppid (), SIGUSR1);

  /* Bilgilendirme bitti, işbaşı!. */
  puts ("Simdilik hoscakalin....");
  exit (0);
}

int
main (void)
{
  struct sigaction usr_action;
  sigset_t block_mask;
  pid_t child_id;

  /* Sinyal eylemciyi kurgulayalım. */
  sigfillset (&block_mask);
```

```
  usr_action.sa_handler = synch_signal;
  usr_action.sa_mask = block_mask;
  usr_action.sa_flags = 0;
  sigaction (SIGUSR1, &usr_action, NULL);

  /* Alt süreci oluşturalım. */
  child_id = fork ();
  if (child_id == 0)
    child_function ();          /* Birşey dönmeyecek. */

  /* Alt sürecin bir sinyal göndermesini bekleyelim */
  while (!usr_interrupt)
    ;

  /* Alt süreç hazır, işbaşı!. */
  puts ("Bu kadar!");

  return 0;
}
```

Bu örnekte beklerken işlemci meşgul ediliyor, bu iyi değil, işlemciyi kullanabilecek başka süreçler engelleniyor. Sinyalin gelip gelmediğini sisteme sormak daha iyidir. Böyle bir örneği *Sinyalin Beklenmesi* (sayfa: 637) bölümünde bulabilirsiniz.

7. Sinyallerin Engellenmesi

Sinyal engelleme işlemi daha sonra alınmak üzere işletim sisteminin onu tutmasını sağlamaktır. Genelde, bir yazılım sinyalleri sonsuza kadar engelleyemez ama onların eylemlerini **SIG_IGN** ile yoksayabilir. Ancak, önemli işlemleri yapabilecek kadar kısa bir süre için sinyalleri engellemek daha iyidir. Örneğin,

- Bu sinyallerin eylemcileri tarafından değiştirilen genel değişkenleri ayarlayana kadar sinyalleri engellemek için **sigprocmask** işlevini kullanabilirsiniz.
- Belli bir sinyal eylemci çalışırken sinyallerin engellenmesi için **sigaction** veri yapısının **sa_mask** üyesini kullanabilirsiniz. Böylece sinyal eylemcinin başka sinyallerle kesintiye uğratılması engellenmiş olur.

7.1. Sinyalleri Engellemenin Amaçları

Sinyalleri **sigprocmask** ile engelleyerek yazılımınızın kritik kısımlarının kesmelerden korunmasını sağlayabilirsiniz. Yazılımınızın bu kısımları etkinken bir sinyal gelirse bir sinyalin engellenmesini kaldırınca o sinyal tekrar alınabilir.

Buna kullanışlı bir örnek, sinyal eylemci ile yazılımın diğer kısımları arasında veri paylaşımıdır. Veri türü **sig_atomic_t** (bkz. *Atomik Veri Erişimi ve Sinyal İşleme* (sayfa: 624)) değilse yazılımınız bu veriyi okur ya da yazarken işlemin ortasında sinyal eylemci çalışmaya başlarsa işlem yarıda kalacağından sorunlara yol açabilir.

Yazılımın düzgün çalışabilmesi için verinin işlenmesi sırasında sinyallerin alınmasını engelleyerek sinyal eylemcinin çalışmamasını sağlayabilirsiniz.

Ayrıca, belli bir eylemi sadece bir sinyal gelmediğinde gerçekleştirmeniz gerekiyorsa sinyallerin engellenmesi gereklidir. Bir sinyal eylemcisinin **sig_atomic_t** türünde bir değişkene değer atadığını, sizin de eyleminizi gerçekleştirmeden önce bu değişkenin değerine baktığınızı varsayalım. Bu yöntem güvenilir değildir. Eyleminizi gerçekleştirmeden, bu değişkenin değerine baktıktan hemen sonra da bir sinyal gelebilir.

Bir sinyalin alınıp alınmadığına bakmanın en güvenilir yolu sinyali engelleyerek değişkenin değerine bakmaktır.

7.2. Sinyal Kümeleri

Sinyal engeleme işlevlerinin tümü ***sinyal kümesi*** adı verilen bir veri yapısını kullanırlar. Bu işlem iki kademede yapılır: sinyal kümesi oluşturulur ve bir argüman olarak bir kütüphane işlevine aktarılır.

Bu oluşumlar `signal.h` başlık dosyasında bildirilmiştir.

```
sigset_t(int pgid,                                                    veri türü
         int sinyalnum)
```

sigset_t veri türü bir sinyal kümesi oluşturmak için kullanılır. Kütüphane içinde bir tamsayı tür ya da bir yapı olarak gerçekleştirilmiş olabilir.

Taşınabilirlik açısından, **sigset_t** yapısı içindeki veriyi okuma ve değiştirme işlemlerini doğrudan değil, bu bölümde açıklanan işlevleri kullanarak yapmalısınız.

Bir sinyal kümesini oluşturmanın iki yolu vardır. **sigemptyset** ile boş olduğunu belirtip sinyalleri tek tek eklersiniz. Ya da, **sigfillset** ile tüm sinyalleri içerdiğini belirtip sinyalleri tek tek silersiniz.

Herhangi bir işlem yapmadan önce sinyal kümesini bu iki işlevden birini kullanarak ilklendirmelisiniz. Sinyal kümesinde sinyalleri bu işlevleri kullanmadan eklemeye ya da silmeye çalışmayın, çünkü **sigset_t** nesnesinde ilklendirilmesi gereken başka alanlar da (sürüm alanı gibi) olabilir. (Ek olarak, sistemin sizin bildiklerinizden başka sinyalleri içermediği kabulünü yapmanız pek akıllıca olmayacaktır.)

```
int sigemptyset(sigset_t *küme)                                       işlev
```

Bu işlev *küme* sinyal kümesini tanımlı hiçbir sinyali içermediği biçimde ilklendirir. Daima **0** ile döner.

```
int sigfillset(sigset_t *küme)                                        işlev
```

Bu işlev *küme* sinyal kümesini tanımlı tüm sinyalleri içerdiği biçimde ilklendirir. Daima **0** ile döner.

```
int sigaddset(sigset_t *küme,                                         işlev
              int       sinyalnum)
```

Bu işlev *sinyalnum* sinyalini *küme* sinyal kümesine ekler. **sigaddset**'ler sadece *küme*'yi değiştirir, sinyal engelleme/engellememe yapmaz.

Başarılı olursa **0**, aksi takdirde **-1** ile döner. Aşağıdaki **errno** değeri bu işlev için tanımlanmıştır:

EINVAL
: *sinyalnum* argümanı geçerli bir sinyal belirtmiyor.

```
int sigdelset(sigset_t *küme,                                         işlev
              int       sinyalnum)
```

Bu işlev *sinyalnum* sinyalini *küme* sinyal kümesinden çıkarır. **sigdelset**'ler sadece *küme*'yi değiştirir, sinyal engelleme/engellememe yapmaz. Dönüş değeri ve hata durumları **sigaddset** işlevindeki gibidir.

Son olarak, bir sinyalin sinyal kümesinde olup olmadığına bakmak için kullanılan bir işlev vardır:

```
int sigismember(sigset_t *küme,                                       işlev
                int       sinyalnum)
```

sigismember işlevi *sinyalnum* sinyalinin *küme* sinyal kümesinin bir üyesi olup olmadığına bakmak için kullanılır. Sinyal, kümenin bir elemanı ise **1** ile, değilse **0** ile, bir hata oluşmuşsa **-1** ile döner.

Aşağıdaki **errno** değeri bu işlev için tanımlanmıştır:

EINVAL
: *sinyalnum* argümanı geçerli bir sinyal belirtmiyor.

7.3. Sürecin Sinyal Maskesi

Engellenen sinyallerden oluşan küme ***sinyal maskesi*** olarak da adlandırılır. Her sürecin kendine özgü bir sinyal maskesi vardır. Yeni bir *süreç oluşturduğunuzda* (sayfa: 687), süreç sinyal maskesini onu çalıştıran süreçten miras alır. Bu sinyal maskesini değiştirerek istediğiniz sinyalleri engelleyebilir ya da da engellemeyebilirsiniz.

sigprocmask işlevinin prototipi **signal.h** dosyasındadır.

Her evre kendi sinyal maskesine sahip olduğundan ve dolayısıyla tek bir sinyal maskesi olmadığından çok evreli süreçlerde **sigprocmask** işlevini kullanmamalısınız. POSIX'e göre, çok evreli bir süreçte **sigprocmask** davranışı "belirsizdir". Yerine **pthread_sigmask** kullanılmalıdır.

```
int sigprocmask(int                         nasıl,                    işlev
                const sigset_t *restrict    küme,
                sigset_t *restrict          eski_küme)
```

sigprocmask işlevi çağrıldığı sürecin sinyal maskesini değiştirmek ya da okumak için kullanılır. *nasıl* argümanı ile sinyal maskesinin nasıl değiştirileceği aşağıdaki değerlerden biri ile belirtilmelidir:

SIG_BLOCK
: *küme* içindeki sinyaller engellensin—mevcut maskeye eklensin. Başka bir deyişle, yeni maske, mevcut maske ile *küme*'nin birleşimi olur.

SIG_UNBLOCK
: *küme* içindeki sinyaller engellenmesin—mevcut maskeden kaldırılsın.

SIG_SETMASK
: Sinyal maskesi için *küme* kullanılsın; önceki maske yoksayılsın.

Son argüman olan *eski_küme* sürecin eski maskesi hakkında bilgi döndürmek için kullanılır. Eski maskeye bakmaksızın maskeyi değiştirmek isterseniz *eski_küme* argümanı ile boş gösterici aktarmalısınız. Benzer şekilde, makeyi değiştirmeksizin mevcut maske hakkında bilgi almak için *küme* argümanı ile boş gösterici aktarmalısınız (bu durumda *nasıl* argümanının önemi yoktur). *eski_küme* argümanı çoğunlukla, sinyal maskesini daha sonra eski durumuna getirmek için mevcut durum bilgisini almak gerektiğinde kullanılır. (Sinyal maskesi **fork** ve **exec** çağrıları üzerinden miras alındığından, yazılımınız çalışmaya başlamadan içeriği hakkında bilgi edinmeniz mümkün değildir.)

sigprocmask çağrısı bekleyen sinyallerin alınmasına sebep olacağından **sigprocmask** dönmeden önce bu sinyallerden en az biri alınmış olacaktır. Bekleyen sinyallerin hangisinin önce alınacağı belli olmaz, ancak her sinyal için ayrı ayrı **sigprocmask** çağrısı yaparak hangi sinyalin önce alınacağını kendiniz belirleyebilirsiniz.

sigprocmask işlevi başarılı olduğunda **0** ile döner. **-1** ile dönmüşse bir hata var demektir. Aşağıdaki **errno** değeri bu işlev için tanımlanmıştır:

EINVAL
: *nasıl* argümanı geçersiz.

SIGKILL ve **SIGSTOP** sinyallerini engelleyemezsiniz, ama bir sinyal kümesi bunları içerebilir. Bu durumda **sigprocmask** hata döndürmez basitçe onları yoksayar.

Bir yazılım hatası (**raise** veya **kill** ile gönderilen sinyallerin aksine) sonucu olarak üretilen **SIGFPE** gibi yazılım hata sinyallerinin engellenmesinin tahmin edilmesi mümkün olmayan sonuçlara yol açacağı unutulmamalıdır. Böyle bir durumda yazılım bozulmuş olacağından sinyallerin engellenmesi kaldırıldığında yazılım isteneni yapmayabilecektir. Bkz. *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

7.4. Sinyal Alımının Sınanması

Basit bir örnekle açıklayalım. **SIGALRM** sinyali için bir eylemci oluşturduğunuzu, bununla bir değişkene değer atadığınızı ve zaman zaman azılım içinde bu değişkene baktığınızı ve onu sıfırladığınızı varsayalım. Kodun kritik parçaları icra edilirken **SIGALRM** sinyallerinin alınmasından **sigprocmask** çağrıları ile aşağıdaki gibi kaçınabilirsiniz:

```
/* Bu değişkene SIGALRM sinyal eylemcisi değer atayacak. */
volatile sig_atomic_t flag = 0;

int
main (void)
{
  sigset_t block_alarm;

  ...

  /* Sinyal maskesini ilklendirelim. */
  sigemptyset (&block_alarm);
  sigaddset (&block_alarm, SIGALRM);

  while (1)
    {
      /* Bir sinyal gelmiş mi bakalım. Gelmişse değişkeni sıfırlayalım. */
      sigprocmask (SIG_BLOCK, &block_alarm, NULL);
      if (flag)
        {
          sinyal alınmamışsa yapılacak işlemler
          flag = 0;
        }
      sigprocmask (SIG_UNBLOCK, &block_alarm, NULL);

      ...
    }
}
```

7.5. Eylemci Çalışırken Sinyallerin Engellenmesi

Bir sinyal eylemci çalışmaya başladığında çalışmasının başka bir sinyal ile kesilmesini istemezsiniz. Eylemci çalışmaya başlayıp, işini bitirene kadar çalışmasının ya da verisinin bozulmaması için sinyalleri engellemelisiniz.

Bir eylemci işlev bir sinyal ile çalıştırıldığında bu sinyal (ve sürecin sinyal maskesindeki diğer sinyaller) özdevinimli olarak engellenir. Örneğin, **SIGTSTP** sinyali için bir eylemciniz varsa, bu eylemci çalışmaya başlayınca daha sonra gelen **SIGTSTP** sinyalleri eylemcinin çalışması süresince bekletilecektir.

Bununla birlikte, öntanımlı olarak, diğer çeşit sinyaller engellenmez ve eylemcinin çalışması sırasında gelebilirler.

Eylemcinin çalışması sırasında gelen farklı sinyalleri engellemenin en güvenilir yolu **sigaction** yapısının **sa_mask** üyesini kullanmaktır.

Bir örnek:

```
#include <signal.h>
#include <stddef.h>

void catch_stop ();

void
install_handler (void)
{
  struct sigaction setup_action;
  sigset_t block_mask;

  sigemptyset (&block_mask);
  /* Eyelmci çalışırken uçbirimden kaynaklanan sinyaller engellensin */
  sigaddset (&block_mask, SIGINT);
  sigaddset (&block_mask, SIGQUIT);
  setup_action.sa_handler = catch_stop;
  setup_action.sa_mask = block_mask;
  setup_action.sa_flags = 0;
  sigaction (SIGTSTP, &setup_action, NULL);
}
```

Bu, diğer sinyallerin eylemciyi kesintiye uğratmasının engelleminin daha güvenilir bir yoludur. Ancak, sinyalleri eylemci içinden doğrudan engellerseniz, eylemcinin başlangıcında kısa bir süre için bu engelleme etkin olmayacaktır.

Bu mekanizma ile sürecin sinyal maskesinden sinyalleri kaldıramazsınız. Bununla birlikte, eylemci içinden yapacağınız **sigprocmask** çağrıları ile bazı sinyalleri engelemeyi ya da engellememeyi tercih edebilirsiniz.

Her durumda, eylemci işini bitirdiğinde sistem, eylemcinin çalışmaya başladığı andaki duruma geri dönecek ve bekletilen sinyaller varsa bu sinyaller gönderilecektir.

7.6. Bekleyen Sinyallerin Sınanması

Bekleyen sinyallerin hangileri olduğunu herhangi bir anda yapacağınız **sigpending** çağrıları ile öğrenebilirsiniz. Bu işlev `signal.h` dosyasında bildirilmiştir.

`int` **sigpending**`(sigset_t *`*küme*`)` işlev

> **sigpending** işlevi bekleyen sinyallere ilişkin bilgileri *küme* içinde saklar. Alınması engellenmiş bir sinyal varsa, bu sinyal dönen kümenin bir üyesidir. (Bir sinyalin bu kümenin üyesi olup olmadığını *Sinyal Kümeleri* (sayfa: 632) bölümünde açıklanan **sigismember** işlevini kullanarak öğrenebilirsiniz.)
>
> Dönen değer bir hata oluşmamışsa **0**, aksi takdirde **-1**'dir.

Bekleyen bir sinyalin olup olmadığına bakmak her zaman kullanışlı olmaz. Engellenmemiş bir sinyalin varlığına bakmak ise daima kötü bir tasarım olur.

Bir örnek:

```
#include <signal.h>
#include <stddef.h>

sigset_t base_mask, waiting_mask;

sigemptyset (&base_mask);
sigaddset (&base_mask, SIGINT);
```

```
sigaddset (&base_mask, SIGTSTP);

/* Diğer işlemler yapılırken kullanıcı kesmeleri engellensin. */
sigprocmask (SIG_SETMASK, &base_mask, NULL);
...

/* Bir süre sonra, bekleyen bir sinyal var mı bakalım. */
sigpending (&waiting_mask);
if (sigismember (&waiting_mask, SIGINT)) {
  /* Kullanıcı süreci sonlandırmayı denedi. */
}
else if (sigismember (&waiting_mask, SIGTSTP)) {
  /* Kullanıcı süreci durdurmayı denedi. */
}
```

Süreciniz için bekleyen belli bir sinyal varsa, sonradan gelen aynı türdeki sinyallerin iptal edilebileceğini unutmayın. Örneğin, bir **SIGINT** sinyali askıya alınmışsa başka bir **SIGINT** sinyali geldiğinde yazılımınız bu sinyali engellenmesi durdurulduktan sonra alacaktır.

Taşınabilirlik Bilgisi

sigpending işlevi POSIX.1 ile gelmiştir. Daha eski sistemler eşdeğeri bir oluşuma sahip değildir.

7.7. Bir Sinyalin Eyleminin Sonradan Hatırlanması

Bir sinyalin engellenmesini, kütüphane oluşumlarını kullanmak yerine daha sonra sinyalin engellenmediği bir sırada sınanacak bir değişkene sinyal eylemci içinde bir değer atamak suretiyle yapabilirsiniz. Bir örnek:

```
/* Bu değişkenin değeri sıfırdan farklıysa, o sinyal bekletiliyordur. */
volatile sig_atomic_t signal_pending;

/* Bir sinyal geldiğinde işleme sokulmayacaksa bu değişkenin değeri
   sıfırdan farklı olacaktır. */
volatile sig_atomic_t defer_signal;

void
handler (int signum)
{
  if (defer_signal)
    signal_pending = signum;
  else
    ... /* "Gerçekten" sinyal yakalanmış. */
}

...

void
update_mumble (int frob)
{
  /* Sinyallerin hemen etki etmesini engelleyelim. */
  defer_signal++;
  /* Şimdi kesmelerden korkmadan mumble'ı güncelleyelim. */
  mumble.a = 1;
  mumble.b = hack ();
  mumble.c = frob;
  /* mumble güncellendi.  Gelen herhangi bir sinyali işleme sokabiliriz. */
```

```
  defer_signal--;
  if (defer_signal == 0 && signal_pending != 0)
    raise (signal_pending);
}
```

Belli bir sinyalin geldiğinde **signal_pending** içinde nasıl saklandığına dikkat edin. Bu yolla, aynı mekanizmayı kullanarak farklı türdeki sinyalleri yakalayabiliriz.

Kodun önemli yerlerinde **defer_signal** değişkeninin değerini önce arttırıp sonra azaltıyoruz; böylece **signal_pending** sıfırdan farklıyken **update_mumble** çağrıldığında sadece **update_mumble** içinde değil ayrıca çağrıcı içinde de ertelenecektir. Bu nedenle, **defer_signal** sıfırdan farklı olduğunda **signal_pending**'e bakmıyoruz.

defer_signal'in arttırılması ve eksiltilmesi bir makina komutundan fazlasını gerektirir; bu nedenle işlemin tamamlanmamışken sinyal alınması ihtimal dahilindedir. Fakat bu herhangi bir soruna yolaçmaz. Arttırma ya da eksiltmenin başlamasından hemen önce bir sinyal gelirse bile çalışma bozulmayacaktır.

defer_signal'in **signal_pending** sınanmadan önce arttırılması ölümcül öneme sahiptir. Bu çözümü zor bir yazılım hatasından kaçınmayı sağlar. Eğer bu işlemleri aşağıdaki gibi farklı bir sırada yapmış olsaydık,

```
  if (defer_signal == 1 && signal_pending != 0)
    raise (signal_pending);
  defer_signal--;
```

if deyimi ile arttırım arasında gelen bir sinyal sonsuza kadar kaybolurdu. **defer_signal**'e sadece sinyal eylemci değer atasa ve yazılım bu değişkene baksa eylemci bu değişkene bakmazdı.

Bu çeşit yazılım hatalarına ***zamanlama hataları*** adı verilir. Yaygın olarak oluştukları halde yeniden üretilmeleri imkansız olduklarından özellikle kötü hatalardır. Bu hatayı üretemeyeceğinizden bir hata ayıklayıcı ile bu hataları bulamazsınız. Bu nedenle onlardan kaçınmak için özellikle büyük dikkat sarfetmelisiniz.

(**defer_signal** değişkenini sayaç olarak kullanıyorsanız kodu bu şekilde yazmak size daha kolay gelebilir. Normalde bu değişkenin **signal_pending** değişkeni ile birlikte sınanması gerekir. Her durumda bu sayacı sıfır değeri için sınamak, bir değeri için sınamaktan daha basittir. Eğer **defer_signal** değişkeni sayaç olarak kullanılmıyor ve sadece sıfır veya bir değerini alabiliyorsa, sınama sırası sınama işleminin karmaşıklığını pek etkilemez. Bu da **defer_signal** değişkenini sayaç olarak kullanmanın bir diğer getirisidir: Kodu yanlış sırada yazmak suretiyle bulunması zor olacak bir hata yapma şansınız azalır.)

8. Sinyalin Beklenmesi

Yazılımınız harici olaylarla tetikleniyorsa ya da eşzamanlama için sinyalleri kullanıyorsa ve yapacak başka işi yoksa bir sinyal gelene kadar beklemesi gerekir.

8.1. **pause** Kullanımı

Sinyal gelene kadar beklemenin en basit yolu **pause** çağrısı yapmaktır. Kullanmadan önce lütfen aşağıda açıklanan olumsuzluklarını okuyun.

```
int pause()                                                    işlev
```

pause işlevi, bir sinyal eylemciyi çalıştırmak ya da süreci sonlandırmak gibi bir eylemi yerine getirmek üzere bir sinyalin gelmesini beklemek için kullanılır.

Sinyal, bir eylemci işlevi tetikliyorsa **pause** döner. Bu başarısız bir dönüş olarak ele alınır (başarılı davranış, süreci sonsuza dek bekletmektir) ve işlev **-1** değeri ile döner. Hatta bir sinyal

eylemci döndüğünde diğer ilkellerin çalışmalarına kaldıkları yerden devam edeceklerini belirtseniz bile (*Sinyallerle Kesilen İlkeller* (sayfa: 626)), bunun **pause** üzerinde bir etkisi yoktur; bir sinyal geldiğinde daima başarısız olarak dönecektir.

Aşağıdaki **errno** değeri bu işlev içindir:

EINTR
: İşlev bir sinyal alarak kesintiye uğradı.

Sinyal sürecin sonlanmasına sebep oluyorsa **pause** dönmeyecektir (ister istemez).

Bu işlev çok evreli yazılımlar için bir iptal noktasıdır. Eğer evre **pause** çağrısı sırasında bazı özkaynakları (bellek, dosya tanımlayıcılar, semaforlar, v.s.) ayırıyorsa sorun çıkar. Evre iptal aldığı andan itibaren süreç sonlanana kadar bu özkaynaklar ayrılmış olarak kalacaktır. Bu tür **pause** çağrılarından kaçınmak için iptal eylemcileri kullanarak korunulmalıdır.

pause işlevi `unistd.h` dosyasında bildirilmiştir.

8.2. **pause** Sorunları

pause basitleştirmesi, yazılımın sihirli bir şekilde çökmesine sebep olan bir sürü zamanlama hatasını gizleyebilir.

Eğer herşeyi yazılımınız yapıyorsa, yani kendi sinyal eylemcilerini kullanıyor ve yazılım **pause** çağırmaktan başka birşey yapmıyorsa **pause** kullanmak güvenilirdir. Her sinyal alınışında sinyal eylemci bir sonraki işi yapar ve döner, böylece yazılımın ana döngüsü tekrar bir **pause** çağrısı yapabilir.

Birden fazla sinyalin işlenmesi için **pause** kullanarak beklemek ve sonra çalışmayı sürdürmek mümkün olmayabilir. Sinyal eylemci çalışmaya başladığında bir değişken ile bunu belirliyorsanız **pause** kullanmak artık güvenilir olmaz; örnek:

```
/* usr_interrupt'a sinyal eylemci değer atıyor.  */
if (!usr_interrupt)
  pause ();

/* Sinyal geldikten sonra yapılacaklar.  */
…
```

Bu bir yazılım hatasıdır: sinyal, *usr_interrupt*'a bakıldıktan sonra ve **pause** çağrılmadan önce gelmelidir. Eğer böyle bir sinyal gelmezse, süreç bu kod parçasını bir kez daha asla çalıştırmayacaktır.

pause kullanmak yerine döngü içinde beklemeyi **sleep** kullanarak sınırlayabilirsiniz. **sleep** hakkında daha ayrıntılı bilgiyi *Uyku* (sayfa: 570) bölümünde bulabilirsiniz. Bu durumda kod şöyle olurdu:

```
/* usr_interrupt'a sinyal eylemci değer atıyor.  */
while (!usr_interrupt)
  sleep (1);

/* Sinyal geldikten sonra yapılacaklar.  */
…
```

Bazı amaçlar için bu yeterli olur. Ama biraz daha karmaşık olmakla beraber belli bir sinyal eylemcisinin çalıştırılmasını **sigsuspend** kullanarak sağlayabilirsiniz.

8.3. **sigsuspend** Kullanımı

Bir sinyali beklemenin en temiz ve güvenilir yolu sinyali engelleyip **sigsuspend** kullanmaktır. Bir döngü içinde **sigsuspend** kullanarak, her sinyal için farklı bir sinyal eylemci oluşturarak sinyalleri bekleyebilirsiniz.

`int` **sigsuspend**`(const sigset_t *`*küme*`)` işlev

Bu işlev sürecin sinyal maskesini *küme* ile değiştirerek bir sinyal eylemciyi çalıştıracak ya da süreci sonlandıracak bir sinyal alıncaya kadar süreci bekletir. Başka bir deyişle, süreç, *küme*'nin üyesi olmayan sinyallerden biri gelene dek süreci bekletir.

Eğer süreç bir sinyal eylemciyi çalıştıracak bir sinyalin alınması ve eylemci işlevin dönmesiyle işlemi sürdürüyorsa **sigsuspend** ayrıca dönecektir.

küme ile belirtilen maske **sigsuspend** etkin olduğu sürece geçerli olur. İşlev döndüğünde eski maske tekrar etkin olur.

Dönüş değeri ve hata durumu **pause** ile aynıdır.

Önceki bölümdeki **pause** ve **sleep** ile ilgili örnekleri **sigsuspend** tamamen güvenilir duruma getirebilirsiniz:

```
sigset_t mask, oldmask;

...

/* Geçici olarak engellenecek sinyallerin maskesini oluşturalım. */
sigemptyset (&mask);
sigaddset (&mask, SIGUSR1);

...

/* Sinyal gelmesini bekleyelim. */
sigprocmask (SIG_BLOCK, &mask, &oldmask);
while (!usr_interrupt)
  sigsuspend (&oldmask);
sigprocmask (SIG_UNBLOCK, &mask, NULL);
```

Kodun son parçası biraz dikkat gerektiriyor. Burada hatırlanması gereken nokta, **sigsuspend** döndüğünde sinyal maskesini çağrılmadan önceki değere ayarlamasıdır. Bu durumda **SIGUSR1** sinyali bir kere daha engellenir. İkinci **sigprocmask** çağrısı bu sinyalin engellenmesini kaldırmak için gereklidir.

Bir diğer nokta: **while** döngüsünün neden gerekli olduğunu düşünebilirsiniz. Öyle ya, yazılım sadece tek bir **SIGUSR1** sinyali bekliyor. Yanıt: **sigsuspend**'e aktarılan maskeye bakarsanız yazılımın ayrıca başka sinyalleri de beklediğini görürsünüz—örneğin, iş denetim sinyalleri. Eğer **usr_interrupt** değişkenine dokunmayan sinyal eylemciler de varsa, değişkeni değiştiren bir sinyal eylemci çalıştırılana kadar döngü sürecektir.

Bu teknik bir kaç satır daha gerektirir ama bu kullanım amacınıza uygun kriterlere bağlı olacaktır. Koda aslında sadece bu dört satırda bekler.

9. Sinyal Yığıtı

Bir sinyal yığıtı, sinyal eylemcilerin icra yığıtı olarak kullanılan özel bir bellek alanıdır. Taşmalardan kaçınmak için oldukça büyük olmalıdır; boyutunu belirlemek amacıyla **SIGSTKSZ** makrosu tanımlanmıştır. Yığıt için gereken alanı **malloc** ile ayırabilirsiniz. Bundan sonra yapacağınız bir **sigaltstack** veya **sigstack** çağrısı ile sisteme bu alanın sinyal yığıtı olarak kullanılacağını belirtebilirsiniz.

Sinyal yığıtını kullanmaları için sinyal eylemcileri oluştururken farklı bir şey yapmanız gerekmez. Bir yığıttan diğerine geçilmesi özdevinimli gerçekleşir. (GNU hata ayıklayıcıları dışındaki bazı hata ayıklayıcılar bazı makinelerde bir sinyal eylemcisinin sinyal yığıtını kullandığı durumda yığıt izlemesini başarıyla yapamayabilir.)

Sisteme ayrı bir sinyal yığıtı kullanmasını belirtmenin iki yolu vardır. 4.2 BSD ile gelen **sigstack** biraz eski bir arayüzdür. 4.4 BSD ile gelen **sigaltstack** daha yenidir. **sigaltstack** işlevinin bir getirisi vardır; makinaya ve işletim sistemine bağlı olan yığıt büyümesinin yönü ile ilgilenmek zorunda kalmazsınız.

stack_t veri türü

Bu veri yapısı bir sinyal yığıtı hakkında bilgi içerir. Aşağıdaki üyelere sahiptir:

`void *`**ss_sp**
Sinyal yığıtının taban adresidir.

`size_t` **ss_size**
ss_sp'den başlayan sinyal yığıtının boyutudur. Yığıt için ne kadar yer ayrılacağını bu üye ile belirteceksiniz.

Yığıt için gereken alanı hesaplamak için `signal.h` dosyasında tanımlanmış iki makro vardır:

`SIGSTKSZ`
Bu bir sinyal yığıtı için olması gereken boyuttur. Normal kullanımlar için yeterli olabilecek kadardır.

`MINSIGSTKSZ`
İşletim sisteminin sinyal göndermeyi gerçekleştirebileceği sinyal yığıtı boyutudur. Ayıracağınız sinyal yığıtı boyutu bu değerden *BÜYÜK* olmalıdır.

Çoğu durumda, **ss_size** için **SIGSTKSZ** kullanmak yeterlidir. Ama yazılımınızın sinyal eylemcilerinin ne kadar yığıt alanı kullanacaklarını biliyorsanız, bundan farklı bir değer de kullanabilirsiniz. Bu durumda en az **MINSIGSTKSZ** yer ayırmanız gerekir, **ss_size** ile bu değerden büyük herhangi bir değeri atayabilirsiniz.

`int` **ss_flags**
Bu üye şu seçeneklerin bit bit veyalanmış değerini içerir:

`SS_DISABLE`
Sistemin sinyal yığıtını kullanmaMAsını belirtir.

`SS_ONSTACK`
Sistem tarafından atanır ve sinyal yığıtının o an kullanılmakta olduğunu belirtir. Bu bit etkin değilse sinyaller, normal kullanıcı yığıtını kullanıyor demektir.

```
int sigaltstack(const stack_t *restrict yığıt,                    işlev
                stack_t *restrict       eski_yığıt)
```

sigaltstack işlevi, sinyal eylemcilerin ayrı bir sinyal yığıtı kullanacağını belirtir. Süreç tarafından bir sinyal alındığında sinyalin eyleminin sinyal yığıtı üzerinden işlenecekse, sistem, sinyal eylemcinin çalışırken bu sinyal yığıtının kullanılması için gerekli düzenlemeyi yapar.

eski_yığıt bir boş gösterici değilse, kurulu sinyal yığıtının bilgileri bu adreste döndürülür. *yığıt* bir boş gösterici değilse, bu, sinyal eylemciler tarafından kullanılacak yeni yığıtı belirtir.

İşlev başarılı olduğunda **0** ile döner, aksi takdirde **−1** ile döner. Aşağıdaki **errno** değerleri bu işlev için tanımlanmıştır:

`EINVAL`
: Kullanımda olan bir yığıtı iptal etmeye çalıştınız.

`ENOMEM`
: Yeni yığıtın boyu **`MINSIGSTKSZ`** değerinden küçük.

Burada eski arayüz olan **`sigstack`** işlevini de anlatacağız ancak, sistem bu işlevi içerse bile bunun yerine **`sigaltstack`** işlevini kullanmalısınız.

`struct` **`sigstack`** — veri türü

Bu veri yapısı bir sinyal yığıtı hakkında bilgi içerir. Şu üyelere sahiptir:

`void *`**`ss_sp`**
: Yığıt göstericisidir. Yığıt aşağı doğru büyüyorsa bu değer yığıtın tepesini, yukarı doğru büyüyorsa yığıtın altını gösterir.

`int` **`ss_onstack`**
: Sistem o an bu yığıtı kullanıyorsa bu alanın değeri "doğru"dur.

```
int sigstack(const struct sigstack *yığıt,
             struct sigstack       *eski_yığıt)
```
— işlev

`sigstack` işlevi, sinyal eylemcilerin ayrı bir sinyal yığıtı kullanacağını belirtir. Süreç tarafından bir sinyal alındığında sinyalin eyleminin sinyal yığıtı üzerinden işlenecekse, sistem, sinyal eylemcinin çalışırken bu sinyal yığıtının kullanılması için gerekli düzenlemeyi yapar.

eski_yığıt bir boş gösterici değilse, kurulu sinyal yığıtının bilgileri bu adreste döndürülür. *yığıt* bir boş gösterici değilse, bu, sinyal eylemciler tarafından kullanılacak yeni yığıtı belirtir.

İşlev başarılı olduğunda **`0`** ile döner, aksi takdirde **`-1`** ile döner.

10. BSD Usulü Sinyal İşleme

Bu bölümde sinyal işleme işlevlerinin BSD Unix'de gerçeklenmiş benzerleri açıklanmıştır. Bu oluşumlar zamanı için ileri düzeydeydiler; günümüzde ise tamamen atıl olmuşlardır ve sadece BSD Unix uyumluluğu için kütüphaneye konmuşlardır.

BSD ve POSIX sinyal işleme oluşumları arasında bir çok benzerlik vardır, çünkü POSIX oluşumları tasarlanırken BSD oluşumlarından ilham alınmıştır. İsim karışıklıklarından kaçınmak için isimlerinin farklılaştırılmalarının yanında aralarında iki temel fark bulunur:

- BSD Unix sinyal maskeleri, birer **`sigset_t`** nesnesi olarak değil birer **`int`** bit maskesi olarak tasarlanmıştır.
- Kesme alan ilkellerin başarısız mı kabul edileceği yoksa işlemlerine kaldıkları yerden devam mı edeceklerini belirten öntanımlama BSD oluşumlarında farklıdır. POSIX oluşumları aksi belirtilmedikçe ilkellerin başarısız olacaklarını, BSD oluşumları ise aksi belirtilmedikçe bu ilkellerin işlemlerine kaldıkları yerden devam edecekleri kabulüne dayanır. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

BSD oluşumları `signal.h` dosyasında bildirilmiştir.

10.1. BSD Eylemciler

`struct` **`sigvec`** — veri türü

Bu veri türü **struct sigaction** yapısının BSD eşdeğeridir. (Bkz. *Gelişmiş Sinyal İşleme* (sayfa: 614)); **sigvec** işlevine sinyal eylemlerini belirtmek için kullanılır. Üyeleri şunlardır:

`sighandler_t` **sv_handler**
: Eylemci işlevdir.

`int` **sv_mask**
: Eylemci çalışırken engellenecek sinyallerden oluşan maske.

`int` **sv_flags**
: Sinyalin davranışını etkileyen seçeneklerden oluşan maske. Bu alana ayrıca **sv_onstack** ile de erişebilirsiniz.

Bu sembolik sabitler bir **sigvec** yapısının **sv_flags** alanının değerleri olarak kullanılabilirler. Bu alan bir bit maskesi olduğundan bu değerleri bit bit veyalayarak birarada belirtebilirsiniz.

`int` **SV_ONSTACK** makro

Bu bit, **sigvec** yapısının **sv_flags** alanında etkinse, sinyal alındığında sinyal yığıtı kullanılır.

`int` **SV_INTERRUPT** veri türü

Bu bit, **sigvec** yapısının **sv_flags** alanında etkinse, sistem çağrılarını kesmeye uğratan sinyaller alındığında, sinyal eylemci işini bitirdiğinde bu çağrılar yeniden yapılmaz; bunun yerine sistem çağrıları **EINTR** hata durumu ile döner. Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

`int` **SV_RESETHAND** veri türü

Bu bit, **sigvec** yapısının **sv_flags** alanında etkinse, sinyal alındığında sinyalin eylemi **SIG_DFL** yapılır.

```
int sigvec(int                    sinyalnum,
           const struct sigvec *eylem,
           struct sigvec       *eski_eylem)
```
işlev

sigaction işlevinin eşdeğeridir (Bkz. *Gelişmiş Sinyal İşleme* (sayfa: 614)); *sinyalnum* sinyali için *eylem* eylemini kurar. Önceki eylem *eski_eylem* argümanında döner.

```
int siginterrupt(int sinyalnum,
                 int seçenek)
```
işlev

sinyalnum sinyali arafından kesintiye uğrayan ilkellerin hangi yaklaşımı kullanacağını belirtmekte kullanılır. *seçenek* yanlışsa, *sinyalnum* sinyali ilkelleri yeniden başlatır; doğruysa, ilkeller **EINTR** hata kodu ile başarısız olur Bkz. *Sinyallerle Kesilen İlkeller* (sayfa: 626).

10.2. BSD'de Sinyal Engelleme

```
int sigmask(int sinyalnum)
```
işlev

Bu makro, *sinyalnum* sinyalinin bir sinyal maskesine dahil edilmesi için kullanılır. Birden fazla sinyali her sinyal için yapılan **sigmask** çağrılarını veyalayarak maskeye dahil edebilirsiniz. Örnek:

```
(sigmask (SIGTSTP) | sigmask (SIGSTOP)
 | sigmask (SIGTTIN) | sigmask (SIGTTOU))
```

Bu kod, tamamı iş denetim sinyallerinden oluşan bir maske belirtir.

```
int sigblock(int maske)
```
işlev

sigprocmask (bkz. *Sürecin Sinyal Maskesi* (sayfa: 633)) işlevinin *nasıl* argümanına **SIG_BLOCK** atanmış bir eşdeğeridir: *maske* ile belirtilen sinyalleri sürecin engellenen sinyaller kümesine ekleyerek önceki sinyal kümesi ile döner.

`int` **sigsetmask**`(int` *maske*`)` işlev

sigprocmask (bkz. *Sürecin Sinyal Maskesi* (sayfa: 633)) işlevinin *nasıl* argümanına **SIG_SETMASK** atanmış bir eşdeğeridir: *maske* ile belirtilen sinyalleri sürecin engellenen sinyaller kümesi yaparak önceki sinyal kümesi ile döner.

`int` **sigpause**`(int` *maske*`)` işlev

sigsuspend (bkz. *Sinyalin Beklenmesi* (sayfa: 637)) işlevinin eşdeğeridir: *maske* ile belirtilen sinyalleri sürecin engellenen sinyaller kümesi yaparak bir sinyal gelmesini bekler. İşlev dönerken eski sinyal kümesi etkinleştirilir.

XXV. Temel Yazılım ve Sistem Arayüzü

İçindekiler

Süreçler sistem kaynaklarının ayrılması için kullanılan ilkel birimlerdir. Her sürecin kendi adres uzayı ve (dolayısıyla) bir denetim evresi vardır. Bir süreç bir yazılımı çalıştırır; aynı yazılımı çalıştıran çok sayıda süreç

oluşturabilirsiniz fakat her sürecin kendi adres uzayında kendi yazılım kopyası bulunur ve onu diğer kopyalardan bağımsız olarak çalıştırır. Bir süreç aynı yazılım içinde çok sayıda denetim evresine sahip olabildiği ve bir yazılım çok sayıda mantıksal olarak ayrı modüllerin birleşimi olabildiği halde bir süreç daima sadece bir yazılımı çalıştırır.

Biz bu kılavuzun amaçlarına uygun olarak ve Unix sistemi bağlamında bir ortak tanımın karşılığı olan "program" için "yazılım" karşılığını kullanıyoruz. Popüler kullanımda, "program" daha geniş bir tanıma sahiptir; örneğin, bir sistemin çekirdeği, bir metin düzenleyici makrosu, karmaşık bir yazılım paketi veya bir süreç içinde çalıştırılan ayrı bir kod parçası olabilir.

Yazılımın geliştirilmesi bu kılavuzun tamamında anlatıldığı gibi yapılır. Bu oylumda yazılımınız ile onun çalıştığı sistem arasındaki çok temel arayüz açıklanacaktır. Bu, sistem için parametrelerin (argümanlar ve ortam) aktarılması, sistemden temel hizmetlerin istenmesi ve sisteme yazılımın ne yaptığının söylenmesini içerir.

Bir yazılım başka bir yazılımı **exec** ailesinden bir sistem çağrısı ile başlatır. Bu oylumda olaya yazılımı çalıştıran açısından değil çalışan yazılım açısından bakacağız. Olaya bir yazılımı çalıştıranın açısından bakmak için *Bir Dosyanın Çalıştırılması* (sayfa: 688) bölümüne bakınız. (Ç.N.: Aslında bu oylumda olaya yazılımın içinden bakacağız. Olaya yazılımın dışından yani bir dış uygulama açısından bakmak için *Bir Dosyanın Çalıştırılması* (sayfa: 688) bölümüne bakın demek daha doğru olacak.)

1. Yazılım Argümanları

Sistem bir C yazılımını **main** işlevini çağırarak başlatır. Bu aslında sizin yazdığınız, ismi **main** olan bir işlevdir; bu işlevi yazmamışsanız, yazılımınızı hatasız derlemeniz mümkün olamaz.

ISO C'ye göre **main** işlevini ya argümansız ya da yazılımın komut satırı argümanlarını ifade eden iki argümanla bu örnekteki gibi tanımlayabilirsiniz:

```
int main (int argc, char *argv[])
```

Komut satırı argümanları kabukta yazılımı çağırırken kullanılan boşluklarla ayrılmış bir takım sözcüklerdir; örneğin, **cat foo bar** gibi bir komut satırında **foo** ve **bar** argümanlardır. Bir yazılımın komut satırı argümanlarına bakabileceğiniz tek yer **main** işlevidir ve bunu işlevin argümanları sağlar. Eğer **main** işlevini argümansız olarak tanımlamışsanız, komut satırı argümanlarını elde edemezsiniz.

argc argümanının değeri komut satırı argümanlarının sayısıdır. *argv* argümanı ise bir C dizgeleri vektörüdür; yani elemanları komut satırı argüman dizgeleri olan bir dizidir. Yazılımın dosya ismi de bu vektör içinde vektörün ilk elemanı olarak yer alır; *argc* ise bu dizideki elemanların sayısıdır. Son eleman daima bir boş göstericidir: *argv*[*argc*] bir boş göstericidir.

cat foo bar komutu için, *argc* argümanının değeri 3'tür ve *argv*, elemanları **"cat"**, **"foo"** ve **"bar"** olan bir dizidir.

Unix sistemlerinde **main** işlevini üç argüman kullanılan üçüncü bir yöntemle tanımlayabilirsiniz:

```
int main (int argc, char *argv[], char *envp[])
```

İlk iki argüman aynıdır. Üçüncü argüman olan *envp* yazılımın ortamını verir; **environ** değeri ile aynıdır. Bkz. *Ortam Değişkenleri* (sayfa: 676). POSIX.1 bu üçüncü argümana izin vermez, dolayısıyla taşınabilirlik açısından en iyisi **main** işlevini iki argümanlı olarak yazmak ve **environ** değerini kullanmaktır.

1.1. Yazılım Argümanları için Sözdizimi Uzlaşımları

POSIX komut satırı argümanları için şu uzlaşımları tavsiye eder. **getopt** (*getopt* (sayfa: 646)) ve **argp_parse** (*Argp* (sayfa: 653)) işlevleri ile bunların gerçeklenmesi kolaylaştırılmıştır.

- Tire (**-**) ile başlayan argümanlar seçeneklerdir.
- Argüman almayan seçenekler, tek bir tire işaretinden sonra aralarında boşluk bırakmaksızın tek bir sözcük olarak belirtilebilir. Yani, **-abc** ile **-a -b -c** eşdeğerdir.
- Seçenek isimleri tek bir alfanümerik karakterden (**isalnum** kapsamındaki karakterler) oluşur; bkz. *Karakterlerin Sınıflandırılması* (sayfa: 82).
- Bazı seçenekler bir argüman gerektirebilir. Örneğin, **ld** komutunun **-o** seçeneği bir argüman gerektirir: bir çıktı dosyası ismi.
- Bir seçenek ile argümanı arasında bir ayraç olabilir de olmayabilir de. (Başka bir deyişle aralarında bir boşluk bırakılması isteğe bağlıdır.) Yani **-o foo** ile **-ofoo** eşdeğerdir.
- Seçenekler genellikle seçenek olmayan argümanlardan önce gelir.

  GNU C kütüphanesindeki **getopt** ve **argp_parse** gerçeklemeleri, kullanıcı seçeneklerle seçenek olmayan argümanları karışık olarak vermiş olsa bile, çözümlemenin doğası gereği onları seçenekler önce görünecek duruma getirir. Bu davranış standartdışıdır; bunu istemiyorsanız **_POSIX_OPTION_ORDER** ortam değişkenini tanımlayın. Bkz. *Standart Ortam Değişkenleri* (sayfa: 678).

- **--** argümanı tüm seçeneklerin sonunu belirtir; bu argümandan sonra gelen argümanlar tire işareti ile başlasalar bile seçenek olarak ele alınmazlar.
- Tek başına bir tire işareti sıradan bir seçenek–olmayan–argüman olarak değerlendirilir. Teamülen (uzlaşımsal olarak), standart girdi ve standart çıktı için girdi ve çıktı belirtmekte kullanılır.
- Seçenekler herhangi bir sırada verilebilir ya da defalarca belirtilebilir. Yorumlanması yazılımın yeteneğine bırakılır.

GNU bu uzlaşımlara ***uzun seçenekler***i ekler. Uzun seçenekler iki tire işareti ile başlayan ve çok sayıda alfanümerik karakter ve tire işaretinden oluşabilen bir dizgedir. Uzun seçeneklerin isimleri genellikle en çok üç sözcük uzunlukta olurlar ve bu sözcükler arasında bir tire işareti bulunur. Kullanıcılar seçenek isimlerini eşsizliğini koruyarak kısaltılmış olarak kullanabilirler.

Bir uzun seçenek için bir argüman belirtmek gerekirse, **--***isim*=*değer* yazılır. Bu sözdizimi bir uzun seçeneğin isteğe bağlı bir argüman kabul etmesini sağlar.

Neticede, GNU sistemi kabukta uzun seçeneklerin tamamlanmasını sağlayacaktır.

1.2. Yazılım Argümanlarının Çözümlenmesi

Yazılımınızın komut satırı argümanlarının sözdizimi yeterince basitse, *argv*'den argümanları kendiniz ayıklayabilirsiniz. Yazılımınız sabit sayıda argüman almıyorsa ve tüm argümanlar aynı yöntemle (örneğin dosya isimleri) elde edilemiyorsa, argümanları çözümlemek için genellikle **getopt** (*getopt* (sayfa: 646)) veya **argp_parse** (*Argp* (sayfa: 653)) işlevlerini kullanmak daha iyidir.

getopt işlevi daha standarttır (sadece kısa seçenek kabul eden sürümü POSIX standardının bir parçasıdır), ama **argp_parse** işlevinin kullanımı hem çok basit hem de çok karmaşık seçenek yapıları için daha kolaydır.

2. getopt

getopt ve **getopt_long** işlevleri tipik unix komut satırı argümanları çözümlemesinin biraz zevksiz olan yanını sizin için hallederler.

2.1. getopt Kullanımı

Bu bölümde **getopt** işlev çağrısının ayrıntıları üzerinde durulacaktır. Bu oluşumları kullanacaksanız, yazılımınıza `unistd.h` başlık dosyasını dahil etmelisiniz.

`int` **opterr**	değişken

Bu değişkenin değeri sıfırdan farklıysa, **getopt** işlevi bilinmeyen bir seçenek karakteri veya argüman gerektiren bir seçenek için argüman belirtilmediğini saptarsa standart hataya bir hata iletisi basar. Bu değişkene sıfır değerini atarsanız, **getopt** işlevi standart hataya birşey basmaz ama bir hatayı belirtmek üzere **?** karakteri ile döner.

`int` **optopt** değişken

getopt işlevi bilinmeyen bir seçenek karakteri veya argüman gerektiren bir seçenek için argüman belirtilmediğini saptadığında seçenek karakterini bu değişkende saklar. Bunu kendi tanı iletileriniz için kullanabilirsiniz.

`int` **optind** değişken

Bu değişkene **getopt** tarafından *argv* dizisinin işlenen elemanından sonraki elemanın indisini koyar. **getopt** tüm seçenekleri bulduktan sonra, bu değişkeni seçenek olmayan ilk argümanın indisini saptamakta kullanabilirsiniz. Bu değişkenin başlangıç değeri **1**'dir.

`char *`**optarg** değişken

Bu değişkene **getopt** tarafından argüman kabul eden seçeneklerin argümanına gösterici yerleştirilir.

```
int getopt(int          argc,                                    işlev
           char       **argv,
           const char *seçenekler)
```

argv and *argc* argümanları ile belirtilen argüman listesindeki sonraki seçenek argümanı ile döner. Normalde bu değerler **main** işlevinden alınan argümanlardan gelir.

seçenekler argümanı yazılım için geçerli seçenek karakterlerinin belirtildiği bir dizgedir. Bu dizgedeki bir seçenek karakterinden sonra bir iki nokta üstüste (`:`) varsa bu, o seçeneğin bir argüman gerektirdiğini belirtir. Seçenek karakterinden sonra iki tane iki nokta üstüste (`::`) varsa bu, o seçeneğin argümanının isteğe bağlı olduğunu belirtir; bu bir GNU oluşumudur.

getopt işlevi seçenek olmayan *argv* elemanlarından sonraki seçenekler için üç yöntem kullanır. Özel **--** argümanı her durumda seçeneklerin sonunu belirtir.

- Neticede tüm seçenek olmayan argümanların sonda olacağından hareketle, taranan *argv* içeriğinin kendi aralarında yer değiştirilmesi öntanımlıdır. Bu, seçeneklerin herhangi bir sırada verilebilmesini, bu durumun olabileceği varsayılmamış yazılımlarda bile sağlar.
- Eğer *seçenekler* dizgesi bir tire işareti (**-**) ile başlıyorsa, bu özel olarak ele alınır. Bu durumda, **\1** seçenek karakteri ile ilişkiliymiş gibi döndürülecek seçenek olmayan argümanlara izin verilir.
- POSIX şu davranışı talep eder: İlk seçenek olmayan argüman işlemi durdurur. Bu kip, **POSIXLY_CORRECT** ortam değişkeni atanarak ya da *seçenekler* dizgesini artı (**+**) işareti ile başlatarak seçilebilir.

getopt işlevi sonraki komut satırı seçeneği olan seçenek karakteri ile döner. Artık seçenek argümanı kalmadığında **-1** döndürür. Hala seçenek olmayan argümanlar olabilir; bunu sınamak için **optind** harici değişkenini *argc* parametresi ile karşılaştırabilirsiniz.

Seçenek bir argümana sahipse, **getopt** argümanı *optarg* değişkenine saklayıp döner. *optarg* değişkeninin değerini sırası geldikçe kopyalamanız gerekmez, çünkü değişkenin değeri üzerine yazılabilen durağan alana gösterici değil, özgün *argv* dizisine bir göstericidir.

getopt işlevi *argv* içinde *seçenekler* ile belirtilmemiş bir seçenek karakteri bulursa ya da bir seçenek argümanı eksikse **?** ile döner ve **optopt** harici değişkenine seçenek karakterini atar. *seçenekler* dizisinin ilk argümanı bir ikinokta üstüste (**:**) ise, **getopt :** yerine eksik seçenek argümanını belirten **?** ile döner. Ek olarak, **opterr** harici değişkeni sıfırdan farklıysa (öntanımlı olarak sıfırdan farklıdır), **getopt** bir hata iletisi basar.

2.2. getopt Örneği

Bu bölümde **getopt** işlevinin tipik kullanımını gösteren bir örneğe yer verilmiştir. Önemli noktalar şunlardır:

- Normalde, **getopt** bir döngü içinde çağrılır. **getopt** işlevi **-1** ile dönerse, artık seçenek kalmamış demektir, dolayısıyla döngüden çıkılır.
- **getopt** işlevinden dönen değeri ayıklamak için bir **switch** deyimi kullanılmıştır. Tipik kullanımda, her **case** deyiminde daha sonra yazılımda kullanılmak üzere bir değişkene değer atanır.
- İkinci bir döngü kalan seçenek olmayan argümanlar içindir.

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main (int argc, char **argv)
{
  int aflag = 0;
  int bflag = 0;
  char *cvalue = NULL;
  int index;
  int c;

  opterr = 0;

  while ((c = getopt (argc, argv, "abc:")) != -1)
    switch (c)
      {
      case 'a':
        aflag = 1;
        break;
      case 'b':
        bflag = 1;
        break;
      case 'c':
        cvalue = optarg;
        break;
      case '?':
        if (optopt == 'c')
          fprintf (stderr, "-%c seçeneği bir argüman gerektirir.\n", optopt);
        else if (isprint (optopt))
          fprintf (stderr, "`-%c' seçeneği bilinmiyor.\n", optopt);
        else
          fprintf (stderr,
                   "Seçenek karakteri `\\x%x' bilinmiyor.\n",
                   optopt);
        return 1;
      default:
        abort ();
```

```
      }

  printf ("aflag = %d, bflag = %d, cvalue = %s\n",
          aflag, bflag, cvalue);

  for (index = optind; index < argc; index++)
    printf ("Seçenek olmayan argüman: %s\n", argv[index]);
  return 0;
}
```

Yazılımın değişik komut satırı seçenekleriyle çalıştırılmasıyla alınan bazı sonuçlar:

```
$ testopt
aflag = 0, bflag = 0, cvalue = (null)

$ testopt -a -b
aflag = 1, bflag = 1, cvalue = (null)

$ testopt -ab
aflag = 1, bflag = 1, cvalue = (null)

$ testopt -c foo
aflag = 0, bflag = 0, cvalue = foo

$ testopt -cfoo
aflag = 0, bflag = 0, cvalue = foo

$ testopt arg1
aflag = 0, bflag = 0, cvalue = (null)
Seçenek olmayan argüman: arg1

$ testopt -a arg1
aflag = 1, bflag = 0, cvalue = (null)
Seçenek olmayan argüman: arg1

$ testopt -c foo arg1
aflag = 0, bflag = 0, cvalue = foo
Seçenek olmayan argüman: arg1

$ testopt -a -- -b
aflag = 1, bflag = 0, cvalue = (null)
Seçenek olmayan argüman: -b

$ testopt -a -
aflag = 1, bflag = 0, cvalue = (null)
Seçenek olmayan argüman: -
```

2.3. `getopt_long` ile Uzun Seçeneklerin Çözümlenmesi

Tek karakterlik seçeneklerin yanında GNU tarzı uzun seçeneklerinde kabul edilmesi için **`getopt`** yerine **`getopt_long`** işlevini kullanabilirsiniz. Bu işlev `unistd.h` değil, `getopt.h` başlık dosyasında bildirilmiştir. Her ne kadar ek bir çalışma gerektirse de yazılımı kullanan acemilerin yazılım kullanımını hatırlamalarına yardımcı olacağından[12] yazılımınız her tek karakterlik seçenek için bir uzun seçenek kabul etmelidir.

`struct option`	veri türü

Bu yapı **getopt_long** işlevinin hatırı için tek bir uzun seçenek ismini açıklar. İşlevin *uzun–seçenekler* argümanı her elemanı bir uzun seçenek içeren bu yapıların bir dizisi olmalıdır. Dizi tamamı sıfır içeren bir elemanla sonlandırılır.

`struct` **option** yapısı şu alanlara sahiptir:

`const char *`**name**
: Seçeneğin isimini içerir. Bir dizgedir.

`int` **has_arg**
: Seçeneğin bir argüman alıp almadığı belirtilir. Bir tamsayıdır ve üç meşru değerden birini içerir: **no_argument** (argümansız), **required_argument** (argüman gerekli) ve **optional_argument** (argüman isteğe bağlı).

`int *`**flag**
`int` **val**
: Bu alanlar bu seçeneğe rastlandığında nasıl raporlanacağı ve rölünü denetler.

  flag bir boş gösterici ise, **val** bu seçeneği kimliklendiren bir değerdir. Bu değerler çoğunlukla, belli bir uzun seçeneği eşsiz olarak kimliklendirecek şekilde seçilirler.

  flag bir boş gösterici değilse, bu seçenek için bir bayrak olan **int** türünde bir değişkenin adresi olmalıdır. **val** içindeki değer de, seçeneğe rastlandığını belirten bayrakta saklanacak değerdir.

```
int getopt_long(int                   argc,                                  işlev
                char *const           *argv,
                const char            *kısa–seçenekler,
                const struct option   *uzun–seçenekler,
                int                   *indis–göstr)
```

argv vektöründen seçenekleri ayıklar. *kısa–seçenekler* argümanı **getopt** işlevindeki gibi kabul edilecek kısa seçenekleri açıklar. *uzun–seçenekler* argümanı ise yukarıda bahsedildiği gibi kabul edilen uzun seçenekleri açıklar.

getopt_long bir kısa seçeneğe rastlarsa **getopt** işlevinin yaptığını yapar: seçeneğin karakter kodu ile döner ve seçenek argümanını (eğer varsa) **optarg** içinde saklar.

getopt_long bir uzun seçeneğe rastlarsa, bu seçeneği tanımlayan **flag** ve **val** üzerine tabanlanmış eylemleri ele alır.

Eğer **flag** bir boş gösterici ise, **getopt_long** işlevi hangi seçeneğin bulunduğunu belirtmek için **val** içeriği ile döner. **val** alanındaki değerleri, farklı anlamlara gelen seçenekleri ayıklamak üzere düzenlemelisiniz, böylece işlev döndükten sonra bu değerleri çözümleyebilirsiniz. Uzun seçenek bir kısa seçeneğin eşdeğeri ise, **val** içinde kısa seçeneğin karakter kodunu kullanabilirsiniz.

Eğer **flag** bir boş gösterici değilse, bu, seçeneğin yazılımda bir bayrağı etkinleştirdiği anlamına gelir. Bayrak sizin tanımlayacağınız **int** türünde bir değişkendir. Bayrağın adresini **flag** alanına ve saklamasını istediğiniz değeri **val** alanına koyun. Bu durumda, **getopt_long** işlevi **0** ile döner.

Herhangi bir uzun seçenek için, **getopt_long** işlevi seçenek tanımlarını içeren *uzun–seçenekler* dizisinin seçeneği içeren elemanının indisini **indis–göstr* içinde saklayarak döndürür. Seçeneğin ismini *uzun–seçenekler*`[*indis-göstr].name` ile alabilirsiniz. Uzun seçenekleri, **val** alanlarındaki değerlerine göre ya da indislerine göre ayrımsayabilirsiniz. Ayrıca bir bayrak tanımlayan uzun seçenekleri bu yolla da ayrımsayabilirsiniz.

Bir uzun seçenek bir argümana sahipse, **getopt_long** işlevi dönmeden önce argüman değerini **optarg** değişkenine atar. Seçenek argümana sahip değilse, **optarg** değişkenindeki değer bir boş gösterici olacaktır. Bu durum bir isteğe bağlı argüman olup olmadığını size söyleyebilir.

getopt_long artık seçenek bulamazsa, **-1** değeri ile döner ve *argv* dizisindeki sonraki argümanın indisini **optind** değişkenine atar.

getopt_long işlevinin devreye girmesinden önce **--option value** gibi seçenekler yerine **-option value** gibi uzun seçenekleri tanıyan yazılım arayüzleri sayesinde uzun seçenek isimleri daha önce de kullanılmaktaydı. Bu yazılımların GNU'nun getopt işlevselliğini kullanabilmelerini sağlamak için bir işlev daha vardır.

```
int getopt_long_only(int                   argc,                    işlev
                     char *const           *argv,
                     const char            *kısa–seçenekler,
                     const struct option   *uzun–seçenekler,
                     int                   *indis–göstr)
```

getopt_long_only işlevi uygulamanın kullanıcısına **--** yerine **-** ile başlayan uzun seçenekler belirtme imkanı vermesi dışında **getopt_long** işlevine eşdeğerdir. **--** ile başlayan seçenekler yine tanınırken **-** ile başlayan seçenekler için işlev önce dizgeye karşılık bir uzun seçenek ismi var mı diye bakar, yoksa dizgenin karakterlerini kısa seçeneklerle eşlemeye çalışır.

getopt_long_only işlevinin şu komut satırıyla çalıştırılan bir yazılımın seçeneklerini çözümlemek için kullanıldığını varsayalım:

```
app -foo
```

getopt_long_only işlevi önce **foo** isimli bir uzun seçeneğin varlığına bakacaktır. Eğer bulamazsa, **f**, **o** ve **o** kısa seçenklerinin varlığına bakacaktır.

2.4. getopt_long Kullanım Örneği.

```
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

/* --verbose için tanımlanan bayrak. */
static int verbose_flag;

int
main (argc, argv)
      int argc;
      char **argv;
{
  int c;

  while (1)
    {
      static struct option long_options[] =
        {
        /* Bu seçenekler bir bayrak tanımlar. */
          {"verbose", no_argument,       &verbose_flag, 1},
          {"brief",   no_argument,       &verbose_flag, 0},
        /* Bu seçenekler bayraksız. Onları indislerine bakıp bulacağız. */
          {"add",     no_argument,       0, 'a'},
          {"append",  no_argument,       0, 'b'},
```

```
        {"delete",  required_argument, 0, 'd'},
        {"create",  required_argument, 0, 'c'},
        {"file",    required_argument, 0, 'f'},
        {0, 0, 0, 0}
      };
    /* getopt_long seçenek indisini burada saklar. */
    int option_index = 0;

    c = getopt_long (argc, argv, "abc:d:f:",
                     long_options, &option_index);

    /* Seçeneklerin sonu mu, bakalım. */
    if (c == -1)
      break;

    switch (c)
      {
      case 0:
        /* Bu seçenek bir bayrak tanımlamamışsa,
           şimdilik birşey yapmayacağız. */
        if (long_options[option_index].flag != 0)
          break;
        if (optarg)
          printf ("%s argümanı ile ", optarg);
        printf ("%s seçeneği", long_options[option_index].name);

        printf ("\n");
        break;

      case 'a':
        puts ("-a seçeneği\n");
        break;

      case 'b':
        puts ("-b seçeneği\n");
        break;

      case 'c':
        printf ("`%s' değeri ile -c seçeneği\n", optarg);
        break;

      case 'd':
        printf ("`%s' değeri ile -d seçeneği\n", optarg);
        break;

      case 'f':
        printf ("`%s' değeri ile -f seçeneği\n", optarg);
        break;

      case '?':
        /* getopt_long bir hata iletisi basmış oldu. */
        break;

      default:
        abort ();
      }
  }
```

```
  /* --verbose ve --brief seçeneklerine rastlandığında onları
     raporlamak yerine onların sonucu olan son durumu raporlayalım. */
  if (verbose_flag)
    puts ("ayrıntı bayrağı etkinleştirildi");

  /* Kalan komut satırı argümanlarını (seçenek olmayan) basalım. */
  if (optind < argc)
    {
      printf ("seçenek olmayan ARGV elemanları: ");
      while (optind < argc)
        printf ("%s ", argv[optind++]);
      putchar ('\n');
    }

  exit (0);
}
```

3. Argp

Argp unix tarzı argüman vektörlerini çözümlemek için bir arayüzdür. Bkz. *Yazılım Argümanları* (sayfa: 645).

Argp arayüzü, **getopt** arayüzü ile kullanılamayan özellikler içerir. Bu özellikler GNU kodlama standartlarında belirtildiği gibi **--help** and **--version** seçenekleri için özdevimli bir çıktı üretilmesini içerir. Argp kullanımı, yazılımcının bu ek seçeneklerin gerçeklenmesini ya da güncel tutulmasını boşvermesi olasılığını azaltır.

Argp ayrıca, birbirinden bağımsız tanımlanmış seçenek çözümleyicileri arasındaki çelişkilere bir orta yol bularak ve sonuçları tek bir çözümleyiciden geliyormuşçasına biraraya getirerek, birarada kullanma yeteneğine de sahiptir. Bir kütüphane, kullanıcı yazılımlarının kendi seçenek çözümleyicileri ile birlikte çalışarak ve kullanıcı yazılımları için daha az iş üreterek, bir argp çözümleyici içerebilir. Bazı yazılımlar sadece kütüphanelerin içerdiği çözümleyicileri kullanır, böylece ayrıntıya girmeden tutarlı ve verimli seçenek çözümlemesi kütüphaneler tarafından gerçekleştirilir.

Argp oluşumlarını kullanacaksanız yazılımınıza `argp.h` başlık dosyasını dahil etmelisiniz.

3.1. **argp_parse** İşlevi

Argp arayüzünün ana işlevi **argp_parse** işlevidir. Bir çok durumda, **argp_parse** çağrısı sadece **main** işlevinde argüman çözümlemesi gerekliyse yapılır. Bkz. *Yazılım Argümanları* (sayfa: 645).

```
error_t argp_parse(const struct argp *argp,                    işlev
                   int                argc,
                   char             **argv,
                   unsigned           bayraklar,
                   int               *arg_indisi,
                   void              *girdi)
```

argp_parse işlevi *argp* çözümleyicisini kullanarak *argc* uzunluğundaki *argv* içindeki argümanları çözümler.

Sıfır değeri **struct argp** yapısının üyelerinin hepsinin sıfır olduğu bir duruma denktir. *bayraklar* çözümlemenin gidişatını etkileyen *bayrak bitlerinden* (sayfa: 663) oluşur. *girdi* ile argp çözümleyiciye parametreleri belirtmek için kullanılan bir yapının göstericisi aktarılır, çözümleyici sonuçları bu yapıyla döndürür.

bayraklar içinde **ARGP_NO_EXIT** veya **ARGP_NO_HELP** bitleri yoksa **argp_parse** çağrısı sonuçlarını yazılım çıkarken verebilir. Bu davranış bir hata saptandığında ya da bilinmeyen bir seçeneğe rastlandığında gerçekleşir. Bkz. *Yazılımın Sonlandırılması* (sayfa: 681).

arg_indisi bir boş gösterici değilse, burada *argv* içindeki çözümlenmemiş ilk seçeneğin indisi değer olarak döner.

Çözümleme başarılı olursa işlev sıfırla döner, bir hata oluşmuşsa o *hatanın kodu* (sayfa: 32) ile döner. Farklı argp çözümleyicileri aynı hata için farklı hata kodları döndürebilirse de, standart hata kodları şunlardır: bir bellek ayırma hatası oluşmuşsa **ENOMEM**, bilinmeyen bir seçenek ya da seçenek argümanına rastlanmışsa **EINVAL**.

3.2. Argp Genel Değişkenleri

Bu değişkenler, **--help** çıktısında bir hata raporlama adresi sağlanması ve **--version** seçeneğinin gerçeklenmesini kullanıcı yazılımları açısından kolaylaştırır. Bunlar argp içinde öntanımlı olarak gerçeklenmiştir.

`const char *`**argp_program_version** — değişken

Tanımlanmışsa ve değeri sıfırdan farklıysa, **argp_parse** çözümlemesine, satırsonu karakteri ile biten bir sürüm bilgisi basıp yazılımın çıkmasını sağlayan bir **--version** seçeneği eklenir. Sürüm bilgisinin basılıp yazılımın çıkması istenmiyorsa, **ARGP_NO_EXIT** biti ile bu sağlanabilir.

`const char *`**argp_program_bug_address** — değişken

Tanımlanmışsa ve değeri bir boş gösterici değilse, **--help** seçeneği için standart çıktının sonuna basılacak `Report bugs to` *address*`.` cümlesinin *address* dizgesine bir göstericidir.

argp_program_version_hook — değişken

Tanımlanmışsa ve değeri sıfırdan farklıysa, **argp_parse** çözümlemesine, satırsonu karakteri ile biten bir sürüm bilgisi basıp yazılımın sıfır durumu ile çıkmasını sağlayan bir **--version** seçeneği eklenir. **ARGP_NO_HELP** biti etkinse bu yapılmaz. **ARGP_NO_EXIT** biti etkinse, argp başka yazılımlarca kullanılıyormuşçasına, yazılımın çıkış davranışı ya engellenir ya da değiştirilir.

Değişken, şöyle bir işlevi göstermelidir:

`void` *print–version* `(FILE *`*akım*`, struct argp_state *`*durum*`)`

durum için *Argp Çözümleme Durumu* (sayfa: 661) bölümüne bakınız.

argp_program_version değişkeni de atanmışsa, bu değişken önceliklidir ve yazılımın sürüm bilgisi basit bir dizge ile karşılanamıyorsa bu değişkeni kullanmak daha yararlıdır.

`error_t` **argp_err_exit_status** — değişken

Argp bir çözümleme hatası dolayısıyla çıkarken kullanılan çıkış durumudur. Yazılımda tanımlanmamışsa ya da bir değer atanmamışsa öntanımlı değeri, `sysexits.h` dosyasındaki **EX_USAGE**'dir.

3.3. Argp Çözümleyicisinin Belirtilmesi

argp_parse işlevinin ilk argümanı *argp çözümleyici* olarak bilinen bir **struct argp** yapısına bir göstericidir:

`struct` **argp** — veri türü

Bu yapı, belirtilen seçenek ve argümanların nasıl çözümleneceğini belirtir. Şüphesiz bu işlem diğer argp çözümleyiciler ile birlikte yapılır. Yapı şu üyelere sahiptir:

`const struct argp_option *`**options**

Argp çözümleyiciye seçenekleri belirtmek için kullanılan **argp_option** yapılarının bir vektörüne göstericidir; çözümlenecek bir seçenek yoksa sıfır olabilir. Bkz. *Seçenekler* (sayfa: 655).

`argp_parser_t` **parser**
Bu çözümleyici için eylemleri tanımlayan bir işleve göstericidir; her seçeneğin çözümlenmesinde ve çözümleme sürecindeki diğer iyi tanımlanmış noktalarda bu işlev çağrılır. Değer olarak sıfır belirtilmişse bu, daima **ARGP_ERR_UNKNOWN** döndüren bir işleve göstericiymiş gibi ele alınır. Bkz. *Argp Çözümleyici İşlevleri* (sayfa: 657).

`const char *` **args_doc**
Sıfırdan farklıysa, bu çözümleyici tarafından çağrılan seçenek olmayan argümanları açıklayan bir dizgedir. Bu sadece **Usage:** (Kullanımı:) iletilerini basmak için kullanılır. İçinde satırsonu karakterleri varsa, onların diğer kullanım iletileri oldukları varsayılarak ayrı satırlara basılır. İlk satırdan sonraki satırların başına **Usage:** yerine **or:** (veya:) getirilir.

`const char *` **doc**
Sıfırdan farklıysa bir uzun yardım iletisinden önce ve sonra basılacak ek metinleri içeren bir dizgedir. Dizge içinde bu metinler bir düşey sekme (**'\v'**, **'\013'**) karakteri ile ayrılır. Uzlaşımsal olarak, seçeneklerden önceki bilgiler yazılımın ne yaptığını açıklayan bir dizgedir. Seçeneklerden sonra da yazılımın davranışını daha ayrıntılı açıklayan bilgiler basılır.

`const struct argp_child *` **children**
argp_children yapılarının vektörüne göstericidir. Gösterici hangi ek argp çözümleyecilerin birlikte kullanılacağını belirtir. Bkz. *Çocuk Çözümleyiciler* (sayfa: 662).

`char *(*help_filter)(int` *anahtar*`, const char *`*metin*`, void *`*girdi*`)`
Sıfırdan farklıysa, yardım iletilerinin çıktısını süzen bir işleve göstericidir. Bkz. *Argp Yardım Çıktısının Özelleştirilmesi* (sayfa: 663).

`const char *` **argp_domain**
Sıfırdan farklıysa, bir dizgeye göstericidir. Argp kütüphanesi bu dizgeyi etki alanını değiştirmekte kullanır. Sıfırsa, öntanımlı etki alanı kullanılır.

Yukarıdaki grubun, **options**, **parser**, **args_doc** ve **doc** alanlarının hepsi gereklidir. Bir argp çözümleyici bir C değişkeni olarak tanımlanmışsa, değişken ilklendirilirken sadece bu alanların belirtilmesi yeterlidir. C yapılarının ilklendirilmeleri gereği olarak kalan üyelerin değerleri öntanımlı olarak sıfır olacaktır. Çoğu argp yapısında bu tasarım kullanılır; sık kullanılan alanlar bir arada gruplanır, kullanılmayanlar belirtilmeden bırakılır.

3.4. Seçenekler

struct argp yapısının **options** alanı, her birinde argp çözümleyicinin desteklemesi için bir seçenek belitilen **struct argp_option** yapılarının vektörünü gösterir. Farklı isimlere sahip tek bir seçenek için çok sayıda girdi kullanılabilir. Böyle bir girdi grubu tüm alanları sıfır olan bir girdi ile sonlandırılır. Böyle bir C dizisini ilklendirirken bu işlemi yapmak için **{ 0 }** yazmanın yeterli olacağını hatırlatalım.

`struct` **argp_option** veri türü

Bu yapı, argp çözümleyicinin arayacağı tek bir seçeneği belirtmek için kullanılmasının yanında, bu seçeneğin nasıl çözümleneceği ve yardım iletisinde bu seçenek için nasıl bir açıklama verileceğini belirtmek için de kullanılır. Şu alanlara sahiptir:

`const char *` **name**

Bu seçeneğin **--***isim* biçimindeki uzun ismi; seçenek sadece kısa isme sahipse bu üye sıfır olarak bırakılabilir. Bir seçeneğin çok sayıda isimle kullanılabilmesi durumunda, bu girdiyi ek girdilerin izlemesini istiyorsanız **OPTION_ALIAS** bitini etkinleştirmelisiniz. Bkz. *Bayraklar* (sayfa: 656).

`int` **key**
Seçenek çözümleyiciye bu seçeneği tanımlayan tamsayı anahtar. Eğer *anahtar* basılabilir bir ascii karakterin (yani, **isascii (***anahtar***)**) değeri ise, **-***karakter* biçimindeki bir kısa seçeneğin ismi olan karakteri de ifade eder.

`const char *` **arg**
Sıfırdan farklı ise, **--***isim***=***değer* veya **-***karakter değer* sözdizimlerindeki gibi bu seçenekle kullanılan argümanın ismidir. Bu argüman ismi **OPTION_ARG_OPTIONAL** biti etkin değilse anlamlıdır. Bkz. *Bayraklar* (sayfa: 656).

`int` **flags**
Bu seçenekle ilgili bayraklar (bir kısmından yukarıda bahsedilmişti). Bkz. *Bayraklar* (sayfa: 656).

`const char *` **doc**
Bu seçenek için bilgilendirme iletisi; yardım iletilerinde seçeneğin açıklaması olarak basılır.

name ve **key** alanlarının ikisi de sıfırsa, bu dizge grup başlığı yapılmak üzere normalde seçeneklerin bulunduğu sütununa basılır. Bu dizge kendi grubunun ilki olmalıdır. Kullanımda, bu dizge teamülen **:** ile biter.

`int` **group**
Bu seçenek için grup kimliği.

Uzun bir yardım iletisinde, her gruptaki seçenekler alfabetik olarak sıralanırlar ve gruplar da 0, 1, 2, …, *n*, –*m*, …, –2, –1 sırasıyla basılırlar.

Bir seçenekler dizisinin bu alanı 0 olan her girdisi grup numarasını, kendinden bir önceki girdiden miras alacaktır, doğal olarak ilki sıfırsa sıfır olacaktır. Grubunu başında **name** ve **key** alanlarının ikisi de sıfırsa, "önceki girdi + 1" öntanımlıdır. Argp arayüzü tarafından üretilen **--help** gibi seçeneklerin grubu –1'dir.

C yapılarının ilklendirme kurallarından dolayı, bu alan çoğunlukla belirtilmez, çünkü 0 geçerli bir değerdir.

3.4.1. Bayraklar

Aşağıdaki bayraklar VEYA'lanarak bir **struct argp_option** yapısının **flags** alanında kullanılır. Bu bayraklar, seçeneklerin nasıl çözümleneceğini veya yardım iletilerinde nasıl gösterileceğini belirlerler:

`OPTION_ARG_OPTIONAL`
Bu seçenekle ilgili argümanın belirtilmesi isteğe bağlıdır.

`OPTION_HIDDEN`
Bu seçenek hiçbir yardım iletisinde gösterilmez.

`OPTION_ALIAS`
Bu seçenek en yakın takma ad olmayan seçeneğin takma adıdır. Yani takma adı olduğu yardım girdisi ile aynı girdide gösterilir. Takma adı olduğu seçeneğin **name** ve **key** dışındaki üyelerinin değerlerini miras alacaktır.

`OPTION_DOC`

Bu seçenek aslında bir seçenek değildir ve seçenek çözümleyicide yoksayılır. Seçeneklerle aynı manada gösterilecek keyfi bir belgeleme bölümüdür. *Belgeleme seçeneği* olarak da bilinir.

Bu bit etkinse, seçeneğin **name** alanı değiştirilmeksizin gösterilir (yani önüne hiç **-** eklenmez). Dizge kısa seçeneklerin olduğu yerde gösterilir. Sıralama amacına uygun olarak dizgenin başında **-** olmadıkça başlangıçtaki boşluklar ve noktalama işaretleri yoksayılır. Bu girdi tüm seçeneklerden sonra, **-** ile başlayan **OPTION_DOC** girdilerinden sonra aynı grupta gösterilir.

`OPTION_NO_USAGE`

Bu seçenek "uzun" kullanım iletisine konmaz, diğer yardım iletilerine konur. Bu bit, argp arayüzünün **args_doc** alanında tamamen belgelenmiş seçenekler için tasarlanmıştır. Bkz. *Argp Çözümleyicisinin Belirtilmesi* (sayfa: 654). Bu bitin soysal kullanım listesinde bulunması gereksiz olurdu, bundan kaçınmak gerekir.

Örneğin, **args_doc** alanında `"FOO BAR\n-x BLAH"` varsa ve **-x** seçeneğinin amacı bu iki durumu ayırmaksa, **-x** şüphesiz **OPTION_NO_USAGE** olarak imlenecekti.

3.5. Argp Çözümleyici İşlevleri

Bir **struct argp** (*Argp Çözümleyicisinin Belirtilmesi* (sayfa: 654)) yapısının **parser** alanı ile gösterilen işlev, çözümlenen her seçenek ve argümana verilen yanıt içinde yer alan eylemi tanımlar. Ayrıca, çözümleme sırasında belirli başka noktalarda uygulanacak işlemleri mümkün kılan bir kanca işlev olarak da kullanılır.

Argp çözümleyici işlevleri şöyle bir şeydir:

`error_t` *çözümleyici* `(int` *anahtar*`, char *`*argüman*`, struct argp_state *`*durum*`)`

Buradaki argümanlar:

anahtar

Çözümlenen her seçenek için, *çözümleyici* işlevi *seçenek vektöründeki* (sayfa: 655) seçeneğin **key** alanındaki *anahtar* değeri ile çağrılır. *çözümleyici* işlevi ayrıca, seçenek olmayan argümanlar için **ARGP_KEY_ARG** gibi *özel anahtarlarla* (sayfa: 658) da çağrılır.

argüman

Eğer *anahtar* bir seçenek belirtiyorsa, *argüman* onun için belirtilmiş değerdir. Hiçbir değer belirtilmemişse öntanımlı değeri sıfırdır. Sadece *argüman* alanı sıfırdan farklı olan seçenekler bir değer alabilir. **OPTION_ARG_OPTIONAL** bayrağı belirtilmedikçe bunlar *daima* bir değer alırlar. Eğer bir değere izin vermeyen bir seçenek için bir değer belirtilmiş bir girdi çözümleniyorsa, *çözümleyici* çağrısından önce bir hata oluşur.

Eğer *anahtar* değeri **ARGP_KEY_ARG** ise, *argüman* bir seçeneği olmayan argümandır. Diğer özel anahtarlar daima sıfır *argüman* değerine sahiptir.

durum

durum argümanı, *çözümleyici* tarafından kullanılmak için o anki çözümleme durumu hakkında bilgi içeren **struct argp_state** için bir gösterici içerir. Bkz. *Argp Çözümleme Durumu* (sayfa: 661).

çözümleyici çağrıldığında, *anahtar* için uygun eylemi uygular ve başarılı olursa **0** ile döner. *anahtar* değeri işlev tarafından elde edilemezse, işlev **ARGP_ERR_UNKNOWN** ile, gerçekten bir hata oluşmuşsa bir unix hata kodu ile döner. Bkz. *Hata Kodları* (sayfa: 32).

`int` **ARGP_ERR_UNKNOWN** makro

Argp çözümleyici işlevi *anahtar* değeri olarak belirtilen değeri tanımıyorsa ya da seçeneği olmayan argümanları (*anahtar*== ARGP_KEY_ARG) elde etmek için çağrılmamışsa bu argümanlar için **ARGP_ERR_UNKNOWN** ile döner.

Tipik bir çözümleyici işlev *anahtar* üzerinde bir switch deyimi kullanır:

```
error_t
parse_opt (int anahtar, char *arg, struct argp_state *durum)
{
  switch (anahtar)
    {
    case seçenek–anahtarı:
      eylem
      break;
    …
    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}
```

3.5.1. Argp Çözümleyici İşlevleri için Özel Anahtarlar

Kullanıcı seçeneklerine karşılık olan anahtar değerlerine ek olarak argp çözümleyici işlevlerinin *anahtar* argümanında bazı özel değerler de kullanılabilir. Aşağıdaki örnekte *argüman* ve *durum*, çözümleyici işlevin argümanlarını ifade eder. Bkz. *Argp Çözümleyici İşlevleri* (sayfa: 657).

ARGP_KEY_ARG
: Seçeneği olmayan bir komut satırı argümanını belirtmek için *argüman* bu değere bir gösterici olur.

  Çok sayıda argp çözümleyicinin bulunmasından dolayı çok sayıda çözümleyici işlev varsa, belli bir argümanın hangisi tarafından çözümleneceğini bilmek mümkün olmaz. Bu durumda sıfır ya da **ARGP_ERR_UNKNOWN** dışında bir hata döndürünceye kadar herbiri çağrılır; yine de bir argüman elde edilememişse, **argp_parse** işlevi başka bir argüman çözümlemesi yapmaksızın başarılı olarak döner.

  Bu anahtar için bir çözümleyici işlev başarılı olmuşsa, bu kaydedilir ve **ARGP_KEY_NO_ARGS** durumu kullanılmaz. Ancak, bir çözümleyici işlev bir argümanı işlerken, *durum* argümanının **next** alanını azaltıyorsa, seçenek işlenmemiş varsayılacaktır; bu durumda hala bir seçenek içinde argümanı değiştirme ve onu tekrar işleme sokma imkanı olacaktır.

ARGP_KEY_ARGS
: Bir çözümleyici işlev, **ARGP_KEY_ARG** için **ARGP_ERR_UNKNOWN** hatası ile dönerse, bu anahtarla benzer anlama sahip ama kalan tüm argümanlar üzerinde etkili olan **ARGP_KEY_ARGS** anahtarı ile çağrı hemen yinelenir. *argüman* 0'dır ve argüman vektörünün ucu *durum*->argv + *durum*->next ile bulunur. Bu anahtar için işlev başarılı ise ve *durum*->next değişmemişse, kalan tüm argümanlar tüketilmiş varsayılır. Aksi takdirde, *durum*->next ile belirtilen miktar kullanılmış olanların sayısına ayarlanır. Örnekte farklı argümannlar için her iki durum da kullanılmıştır:

```
…
  case ARGP_KEY_ARG:
    if (durum->arg_num == 0)
      /* İlk argüman */
      ilk_arg = arguman;
    else
      /* Sonra çözümlenecek.  */
```

```
      return ARGP_KEY_UNKNOWN;
    break;
  case ARGP_KEY_ARGS:
    kalan_argumanlar = durum->argv + durum->next;
    kalan_arg_sayisi = durum->argc - durum->next;
    break;
```

ARGP_KEY_END
: Bu anahtar başka komut satırı kalmadığını belirtir. Çözümleyici işlevler farklı bir sırada (önce **children**) çağrılır. Bu, her çözümleyici işlevin çağrıcısı için kendi durumunu temizleme imkanı verir.

ARGP_KEY_NO_ARGS
: Seçeneği olmayan argümanların yokluğunda bazı özel işlemler ortaktır. Bundan dolayı, eğer çözümleyici işlevde seçeneği olmayan argümanları başarıyla işleme yeteneği yoksa bu işlevler bu anahtarla çağrılır. Bu çağrı, önceden çözümlenmiş argümanlar üzerinde daha genel doğrulama sınamalarının yapılabilmesi için **ARGP_KEY_END**'li bir çağrıdan önce yapılır.

ARGP_KEY_INIT
: Herhangi bir çözümleme yapılmadan önce kullanılır. Bunun ardından, *durum* yapısının **child_input** alanının her elemanının değeri, çocuk çözümleyiciler çağrılırken **input** üyesini ilklendirmek üzere herbirinin durumuna kopyalanır.

ARGP_KEY_SUCCESS
: Bazı argümanlar kalsa bile, istenen çözümleme başarıyla tamamlandığında kullanılır.

ARGP_KEY_ERROR
: Bir hata oluştuğunda ya da çözümleme tamamlandığında kullanılır. Bu takdirde **ARGP_KEY_SUCCESS** anahtarlı bir çağrı asla yapılmamalıdır.

ARGP_KEY_FINI
: **ARGP_KEY_SUCCESS** ve **ARGP_KEY_ERROR** anahtarlı çağrılardan bile sonra kullanılan son anahtar. **ARGP_KEY_INIT** anahtarlı bir çağrı ile ilklendirilen özkaynaklar bu anahtar kullanılarak yapılan bir çağrı ile serbest bırakılır. Bu sırada, bir başarılı çözümleme sonrası çağrıcıya döndürülen özkaynaklar ayrılmış olarak kalır. Bu durumda, bu özkaynaklar **ARGP_KEY_ERROR** durumuyla serbest bırakılabilir.

Tüm durumlarda, **ARGP_KEY_INIT** anahtarı çözümleyici işlev tarafından görülen ilk anahtar; **ARGP_KEY_INIT** için çözümleyiciden bir hata döndürülmedikçe, **ARGP_KEY_FINI** ise son anahtardır. Diğer anahtarlar aşağıdaki sıralamalarla görünürler. *sçn–anh* keyfi bir seçenek anahtarını ifade eder:

sçn–anh... **ARGP_KEY_NO_ARGS ARGP_KEY_END ARGP_KEY_SUCCESS**
: Çözümlenen argümanlar seçeneği olmayan argümanları içermiyorsa bu sıralama kullanılır.

(*sçn–anh* | **ARGP_KEY_ARG**)... **ARGP_KEY_END ARGP_KEY_SUCCESS**
: Seçeneği olamayan argümanları işleme yeteneğine sahip işlev(ler) varsa bu sıralama kullanılır. Çok sayıda argp çözümleyici birlikte kullanılıyorsa çok sayıda çözümleyici işlev olabilir.

(*sçn–anh* | **ARGP_KEY_ARG**)... **ARGP_KEY_SUCCESS**
: Seçeneği olmayan argümanlardan bilinmeyenler varsa bu sıralama kullanılır.

  Çözümleyici işlevlerin hepsi bir argüman için **ARGP_KEY_UNKNOWN** ile döndüğünde, eğer *arg_indisi* bir boş gösterici ise çözümleme bu argümanda durdurulur. Aksi takdirde bir hata oluşur.

Tüm durumlarda, **argp_parse**'a aktarılan boş gösterici olmayan bir *arg_indisi* için çözümlenmemiş ilk komut satırı argümanı bu gösterici ile döndürülür.

Gerek argp tarafından gerekse bir hata değeri döndüren bir çözümleyici işlev nedeniyle bir hata değeri dönmüşse, her çözümleyici **ARGP_KEY_ERROR** ile çağrılır. Son çağrı olan **ARGP_KEY_FINI** anahtarlı çağrı dışında bir çağrı yapılmaz.

3.5.2. Argp Çözümleyicilere Yardımcı İşlevler

Argp arayüzü, çoğunlukla hata iletileri üretmek için *argp kullanıcısında* (sayfa: 657) kullanmak için bazı işlevler içerir. İlk argüman olarak çözümleyici işlevin *durum* argümanını alırlar. Bkz. *Argp Çözümleme Durumu* (sayfa: 661).

```
void argp_usage(const struct argp_state *durum)                    işlev
```

Çözümleyici tarafından *durum* ile belirtilen standart kullanım iletisini *durum->err_stream*'e çıktılar ve yazılımı **exit (argp_err_exit_status)** çağrısı ile sonlandırır. Bkz. *Argp Genel Değişkenleri* (sayfa: 654).

```
void argp_error(const struct argp_state *durum,                    işlev
                const char              *biçim,
                …)
```

Yazılım isminden sonra bir iki nokta üstüste koyup, ardından *biçim* ile belirtilen printf biçim dizgesi ve argümanlarını bastıktan sonra buna `Try … --help` iletisini ekler ve yazılımı **argp_err_exit_status** durumu ile sonlandırır. Bkz. *Argp Genel Değişkenleri* (sayfa: 654).

```
void argp_failure(const struct argp_state *durum,                  işlev
                  int                      çıkış–durumu,
                  int                      hatanum,
                  const char              *biçim,
                  …)
```

Standart GNU hata raporlama işlevi olan **error** işlevine benzer. Yazılım isminden sonra bir iki nokta üstüste koyup, ardından *biçim* ile belirtilen printf biçim dizgesi ve argümanlarını basar. *hatanum* sıfırdan farklıysa, bu hata durumu ile ilgili standart Unix hata metnini basar. *çıkış–durumu* sıfırdan farklıysa bu değeri çıkış durumu olarak kullanarak yazılımı sonlandırır.

argp_failure ile **argp_error** arasındaki fark, **argp_error** hataların çözümlenmesi için iken, **argp_failure** çözümleme sırasında oluşan diğer sorunlar içindir, ama kuraldışı değer verilmiş seçenekler, ayın yanlış evrede olması gibi girdi ile ilgili sözdizimsel sorunları ifade etmek için değildir.

```
void argp_state_help(const struct argp_state *durum,               işlev
                     FILE                    *akım,
                     unsigned                 bayraklar)
```

Çözümleyici tarafından *durum* ile belirtilen bir yardım iletisini *akım*'a çıktılar. *bayraklar* argümanı ile yardım iletisinin hangi sıra ile üretileceği belirtilir. Bkz. *argp_help Bayrakları* (sayfa: 664).

Hata çıktısı *durum->err_stream*'e gönderilir ve basılan yazılım ismi *durum->name*'dir.

Bu işlevlerin çıktıları ya da sonlandırma davranışları, **argp_parse** işlevine **ARGP_NO_EXIT** veya **ARGP_NO_ERRS** bayrağı aktarılarak baskılanabilir. Bkz. *argp_parse Bayrakları* (sayfa: 663).

Bu davranış, argp çözümleyici başka yazılımlar (örn, bir kütüphane) tarafından kullanılmak içinse yararlıdır ve çözümleme hatalarına yanıt olarak yazılımın sonlanmasının istenmediği bir bağlamda kullanılabilir. Bu

tür kullanımlar için tasarlanmış argp çözümleyicilerde ve yazılımın sonlanmayacağı durum için bu işlevlerin çağrılarından sonra ilgili hata kodunu döndüren kodlar olmalıdır:

```
if (argüman sözdizimi hatalı)
  {
    argp_usage (durum);
    return EINVAL;
  }
```

Eğer bir çözümleyici işlev sadece **ARGP_NO_EXIT** etkin iken kullanılacaksa, dönmeyebilir.

3.5.3. Argp Çözümleme Durumu

Argp çözümleyici işlevlerinin (sayfa: 657) üçüncü argümanı seçenek çözümleme durumu hakkında bilgi içeren **struct argp_state** yapısına bir göstericidir.

`struct` **argp_state** veri türü

Bu yapının üyeleri şunlardır (değişiklik yapılabilecek üyeler belirtilmiştir):

`const struct argp *const` **root_argp**
: Çözümleme için kullanılan en üst seviye argp çözümleyici. Bu çoğunlukla, yazılım tarafından çağrılan **argp_parse** işlevine aktarılan **struct argp** ile aynı değildir. Bkz. *Argp* (sayfa: 653). Bu, **--help** gibi **argp_parse** tarafından gerçeklenmiş seçenekleri içeren dahili argp çözümleyicidir.

`int` **argc**
`char **`**argv**
: Çözümlenecek argüman vektörü. Bu üyenin değeri değiştirilebilir.

`int` **next**
: Çözümlenecek sonraki argümanın **argv** içindeki indisi. Bu üyenin değeri değiştirilebilir.

  Girdide kalan tüm argümanları tüketmenin tek yolu **next** alanındaki değeri kaydettikten sonra *durum*`->next = `*durum*`->argc` ataması yapmaktır. Aynı seçenek bu alanın değeri azaltılarak yeniden çözümlenebilir ve bundan sonra çözümlenecek seçenek *durum*`->argv[`*durum*`->next]` ile belirtilebilir.

`unsigned` **flags**
: **argp_parse**'a aktarılacak bayraklar. Bazı bayraklar sadece **argp_parse** ilk çağrıldığında etkili olabileceğinden bu üyenin değeri değiştirilebilir. Bkz. *argp_parse Bayrakları* (sayfa: 663).

`unsigned` **arg_num**
: Çözümleyici işlev *anahtar* argümanında **ARGP_KEY_ARG** belirtilerek çağrıldığında, bu üye, ilkinin indisi 0 olmak üzere o anki argümanın indisini gösterir. Her **ARGP_KEY_ARG**'lı çağrıdan sonra değeri bir artar. Bunun dışında, işlenen **ARGP_KEY_ARG** argümanlarının sayısını içerir.

`int` **quoted**
: Sıfırdan farklıysa değeri, özel **--** argümanından sonraki ilk **argv** argümanının indisidir. Bu indisten itibaren hiçbir argüman seçenek olarak yorumlanmaz. Bu değer sadece, bu özel seçenekten önce çözümlenmemiş seçenek kalmadığında atanır.

`void *`**input**
: **argp_parse** işlevine *girdi* argümanı ile aktarılan keyfi bir gösterici.

`void **`**child_inputs**

Çocuk çözümleyicilere aktarılacak değerleri içerir. Bu vektörün eleman sayısı o anki çözümleyicideki çocukların sayısı ile aynı olacaktır. *i* bu çözümleyicinin **children** alanındaki çocuk çözümleyicinin indisi olmak üzere, *durum*->child_inputs[*i*] değeri her çocuk çözümleyicinin *durum*->input alanının değeri olacaktır. Bkz. *Çocuk Çözümleyiciler* (sayfa: 662).

void ***hook**
Çözümleyici işlevin kullanması içindir. 0 ile ilklendirilir, başka bir değer verilse bile argp tarafından bu değer yok sayılır.

char ***name**
İletileri basarken kullanılacak isim. Bu üye normalde **argv[0]** ile ilklendirilir. **argv[0]** mevcut değilse, **program_invocation_name** ile ilklendirilir.

FILE ***err_stream**
FILE ***out_stream**
Argp arayüzünün iletileri basarken kullandığı standart G/Ç akımları. Hata iletileri **err_stream**'e, tüm diğer çıktılar (örn, **--help** çıktısı) **out_stream**'e çıktılanır. Bunlar sırasıyla **stderr** ve **stdout** olarak ilklendirilir. Bkz. *Standart Akımlar* (sayfa: 237).

void ***pstate**
Argp gerçekleniminin kullanımına özeldir.

3.6. Çocuk Çözümleyiciler

Bir **struct argp** yapısının **children** alanı aynı argüman kümesinin çözümlenmesi için birlikte kullanılacak diğer argp çözümleyicilerle ilgili bilgi içerir. Bu alan, bir **struct argp_child** vektörüne göstericidir. Vektör, **argp** alanı sıfır olan bir yapı ile sonlanır.

Çözümleyiciler arasında aynı isimli iki seçeneği belirtilmesi nedeniyle bir çatışma ortaya çıkarsa, çatışmalar ata argp çözümleyici(ler) ya da çocuk çözümleyiciler listesindeki daha önceki argp çözümleyiciler lehine çözümlenir.

struct **argp_child** veri türü

Bir **struct argp** yapısının **children** alanı tarafından gösterilen yardımcı argp çözümleyici listesindeki bir girdinin veri türü. Yapı şu üyelere sahiptir:

const struct argp ***argp**
Çocuk argp çözümleyici; yapı, listenin son elemanıysa sıfır.

int **flags**
Bu çocuk çözümleyici için bayraklar.

const char ***header**
Sıfırdan farklıysa, çocuk seçeneklerden önce yardım çıktısına basılacak isteğe bağlı başlık. Bir yan etki olarak, sıfırdan farklı bir değer çocuk seçeneklerin birlikte gruplanmasına sebep olur. Bir başlık basmadan bu yan etkiyi elde etmek isterseniz **""** değerini kullanın. Başlık dizgesi basılırken son karakter teamülen **:** olur. Bkz. *Seçenekler* (sayfa: 655).

int **group**
Çocuk seçeneklerin, ata argp çözümleyicinin seçenekleri ile birlikte basılırken, bu seçenekler arasında grup olarak hangi sırada basılacağı bu üye ile belirtilir. **struct argp_option** yapısının **group** alanındaki değer ile aynıdır. Bkz. *Seçenekler* (sayfa: 655). Tüm çocuk seçenek grupları, ata seçenekler arasında belli bir gruplama seviyesinde basılır. Eğer bu alan ve **header** alanı her ikisi de sıfırsa, çocuk seçenekler gruplanmaz, ata seçeneklerle aynı seviyede basılırlar.

3.7. `argp_parse` Bayrakları

argp_parse işlevinin öntanımlı davranışı yazılımın komut satırı argümanlarının teamülen çok bilinen bir duruma göre çözümleme yapması için tasarlanmıştır. Bu davranışı değiştirmek için **argp_parse** işlevinin *bayraklar* argümanında aşağıdaki bayraklar VEYAlanarak belirtilebilir:

ARGP_PARSE_ARGV0
: **argp_parse** işlevinin *argv* argümanının ilk elemanı yoksayılmaz. **ARGP_NO_ERRS** etkin olmadıkça argüman vektörünün komut satırında yazılım ismine denk düşen ilk elemanı seçenek çözümleme amaçlarına uygun olarak yoksayılır.

ARGP_NO_ERRS
: Bilinmeyen seçenekler için **stderr** akımına hata iletileri basılmaz. Bu bayrak etkin olmadıkça, **ARGP_PARSE_ARGV0** yoksayılır ve hata iletilerinde **argv[0]** yazılım ismi olarak basılır. Bu bayrak ayrıca **ARGP_NO_EXIT** uygular. Hata olduğunda hiçbir bilgi vermeksizin yazılımı sonlandırmanın kötü bir davranış olacağından hareketle bu davranış isteğe bağlı yapılmıştır.

ARGP_NO_ARGS
: Seçenek olmayan argümanlar çözümlenmez. Normalde bunlar çözümleme işlevleri **ARGP_KEY_ARG** anahtarı ile çağrılarak çözümlenirler. Bir argümanın çözümlenmesi başarısız olduğunda çözümleme normalde durduğundan bu bayrağın kullanılmasına gerek kalmaz. Bkz. *Argp Çözümleyici İşlevleri* (sayfa: 657).

ARGP_IN_ORDER
: Seçenekler ve argümanları komut satırında verildikleri sırada çözümlenir. Normalde seçenekler önce çözümlenecek şekilde düzenleme yapılır.

ARGP_NO_HELP
: Normalde standart **--help** seçeneği ile seçeneklerin kullanım açıklamalarını içeren yardım iletisinin basılıp **exit (0)** ile çıkılır. Bu bayrak bu davranışı iptal eder.

ARGP_NO_EXIT
: Bir hata iletisi ile sonuçlansa bile hatalarda çıkış yapılmaz.

ARGP_LONG_ONLY
: Argümanların çözümlenmesinde GNU getopts uzun seçenek kuralları kullanılır. Bu bayrak uzun seçenekleri tek **-** ile (**-help** biçiminde) belirtilebilmesini mümkün kılar. Bu, daha az kullanışlı bir arayüz ile sonuçlanır ve hem GNU kodlama standartları hem de çoğu GNU yazılımı ile bu davranış uyumsuz olacağından kullanılması önerilmez.

ARGP_SILENT
: İleti basma ve çıkma seçeneklerini, özellikle **ARGP_NO_EXIT**, **ARGP_NO_ERRS** ve **ARGP_NO_HELP** ile ilgili olarak iptal eder.

3.8. Argp Yardım Çıktısının Özelleştirilmesi

Bir **struct argp** yapısının **help_filter** alanı yardım iletilerinin metninin özelleştirilmesini mümkün kılmak için bir işleve gösterici içerebilir. Böyle bir işlevin prototipi şuna benzer:

```
char *yardım–süzgeci (int anahtar, const char *metin, void *girdi)
```

Burada *anahtar* bir seçenekteki bir anahtar olduğunda *metin* bu seçeneğin yardım metnidir. Bkz. *Seçenekler* (sayfa: 655). *anahtar* olarak **ARGP_KEY_HELP_** ile başlayan özel anahtarlardan biri de kullanılabilir; bu durumda, *metin* bu duruma ilişkin yardım metnini içerecektir. Bkz. *Argp Yardım Özelleştirme Anahtarları* (sayfa: 664).

İşlev ya olduğu gibi bırakılmış olarak ya da **malloc** kullanılarak ayrılmış başka bir dizgeyi içeren *metin* ile dönmelidir. Metnin oluşturulmasına bağlı olarak ya argp tarafından serbest bırakılarak ya da sıfır yapılarak hiçbir şey basılmaması sağlanabilir. *metin* değeri bir dönüşümün sonunda oluşur. Yani metnin çevirisi gerekiyorsa bu işlem bu işlev tarafından yapılmalıdır. *girdi*, **argp_parse**'a verilen girdi olabileceği gibi **argp_help** doğrudan kullanıcı tarafından çağrılmışsa sıfır olabilir.

3.8.1. Argp Yardım Özelleştirme Anahtarları

Aşağıdaki özel değerler, kullanıcı seçeneklerinin anahtar değerlerine ek olarak bir argp yardım iletisi özelleştirme işlevinin ilk argümanında değer olarak kullanılabilir. Bunlar işlevin *metin* argümanında hangi metnin içerileceğini belirlerler:

`ARGP_KEY_HELP_PRE_DOC`
: Seçeneklerden önce basılacak yardım metni.

`ARGP_KEY_HELP_POST_DOC`
: Seçeneklerden sonra basılacak yardım metni.

`ARGP_KEY_HELP_HEADER`
: Seçenek başlık dizgesi.

`ARGP_KEY_HELP_EXTRA`
: Tüm diğer açılamalardan sonra kullanılır; bu anahtar için *metin* sıfırdır.

`ARGP_KEY_HELP_DUP_ARGS_NOTE`
: Tekrarlanan seçenek argümanları engellendiğinde basılacak açıklayıcı bilgi.

`ARGP_KEY_HELP_ARGS_DOC`
: Argüman açıklama dizgesi; usulen argp çözümleyicideki **args_doc** alanıdır. Bkz. *Argp Çözümleyicisinin Belirtilmesi* (sayfa: 654).

3.9. argp_help İşlevi

Normalde argp arayüzünü kullanan yazılımlarda argüman kullanım iletilerinin basılması için, bu işlem argp tarafından standart **--help** seçeneği ile özdevinimli olarak yapıldığından, ayrıca kod yazılması gerekmez. Hata durumlarında ise **argp_usage** ve **argp_error** kullanılır. Bkz. *Argp Çözümleyicilere Yardımcı İşlevler* (sayfa: 660). Ancak, bir yardım iletisinde yazılımın seçenek çözümlemesi dışında bazı bilgilerin basılması istenebilir. Argp arayüzü bu tür istekleri karşılamak üzere **argp_help** arayüzünü içerir.

```
void argp_help(const struct argp *argp,                    işlev
               FILE              *akım,
               unsigned           bayraklar,
               char              *isim)
```

argp çözümleyici için bir yardım iletisini *akım* akımına çıktılar. Basılacak iletinin türü *bayraklar* ile belirtilir.

Argp arayüzü tarafından **--help** gibi özdevinimli gerçeklenen seçeneklere ilişkin yardım çıktısı bu çıktıda yer almaz. Bu nedenle çağrıyı bir argp çözümleyici işlev içinden yapıyorsanız en iyisi **argp_state_help** işlevini kullanmaktır. Bkz. *Argp Çözümleyicilere Yardımcı İşlevler* (sayfa: 660).

3.10. argp_help Bayrakları

argp_help (bkz. *argp_help İşlevi* (sayfa: 664)) veya **argp_state_help** (bkz. *Argp Çözümleyicilere Yardımcı İşlevler* (sayfa: 660)) çağrısı yapıldığında çıktı *bayraklar* argümanında belirtilen değerden etkilenir. Bu değer, aşağıdaki değerler VEYAlanarak oluşturulabilir:

`ARGP_HELP_USAGE`
: Tüm seçeneklerin listelendiği Unix `Usage:` iletisi.

`ARGP_HELP_SHORT_USAGE`
: Seçeneklerin açıkça değil, seçeneklerin sadece yer belirtilerek çıktılandığı Unix `Usage:` iletisi; seçenek olmayan argümanların sözdizimi gösterilirken yararlıdır.

`ARGP_HELP_SEE`
: `Try ... for more help` iletisi; burada **...** yazılımın ismini ve **`--help`** seçeneğini içerir.

`ARGP_HELP_LONG`
: Her seçeneğin kendi bilgilendirme metni bulunan ayrıntılı yardım iletisi.

`ARGP_HELP_PRE_DOC`
: Ayrıntılı yardım iletisinden önceki argp çözümleyici açıklama dizgesi.

`ARGP_HELP_POST_DOC`
: Ayrıntılı yardım iletisinden sonraki argp çözümleyici açıklama dizgesi.

`ARGP_HELP_DOC`
: **`(ARGP_HELP_PRE_DOC | ARGP_HELP_POST_DOC)`**

`ARGP_HELP_BUG_ADDR`
: **`argp_program_bug_address`** değişkeninde atanmışsa, bu yazılımla ilgili yazılım hatalarının raporlanacağı yeri belirten ileti.

`ARGP_HELP_LONG_ONLY`
: Çıktı, **`ARGP_LONG_ONLY`** kipine göre değiştirilir.

Aşağıdaki bayraklar sadece **`argp_state_help`** işlevinde kullanıldığında anlamlıdır. İleti basıldıktan sonra ya yazılım sonlandırılır ya da işlev döner. Bu seçenekler bunlardan birini seçer:

`ARGP_HELP_EXIT_ERR`
: Yazılımın **`exit (argp_err_exit_status)`** ile sonlanmasını sağlar.

`ARGP_HELP_EXIT_OK`
: Yazılımın **`exit (0)`** ile sonlanmasını sağlar.

Aşağıdaki bayraklar, standart iletilerin basılmasında kullanılan temel bayrakların birleşiminden oluşur:

`ARGP_HELP_STD_ERR`
: Hata iletisinin bir çözümleme hatası içerdiği varsayımıyla, nasıl yardım alınacağını belirten bir ileti basılır ve yazılım bir hata ile sonlandırılır.

`ARGP_HELP_STD_USAGE`
: Bir standart kullanım iletisi basılır ve yazılım bir hata ile sonlandırılır. Bu duruma özel bir hata iletisisin yokluğunda kullanılır.

`ARGP_HELP_STD_HELP`
: **`--help`** seçeneğinin standart sonucu olan ileti basılır ve yazılım başarılı olarak sonlandırılır.

3.11. Argp Örnekleri

Bu örnek yazılımlarla argp arayüzünün temel kullanımı örneklenmeye çalışılmıştır.

3.11.1. 1. Örnek

Bu örnekte, argp arayüzünü kullanan olası en küçük yazılımın nasıl olacağı gösterilmiştir. Komut satırında mevcut olmayan bir seçenek belirtildiğinde bir hata iletisi basıp çıkmak dışında birşey yapmaz. **--help** seçeneği ile ise argp arayüzünde gerçeklenmiş seçeneklerin yardım iletisini basar.

```
/* 1. Argp örneği -- argp kullanılan en küçük yazılım */

/* Bu, argp kullanılan (olası) en küçük yazılımdır.
   --help ve --usage ile yardım ve kısa kullanım iletisi
   basmak dışında, sadece tanımsız bir komut satırı seçeneği
   ya da argümanı için bir hata iletisi basar. */

#include <argp.h>

int main (int argc, char **argv)
{
  argp_parse (0, argc, argv, 0, 0, 0);
  exit (0);
}
```

Çıktısı şöyle birşey oluyor:

```
~/deneme$ gcc deneme.c
~/deneme$ ./a.out
~/deneme$ ./a.out --help
Usage: a.out [OPTION...]

  -?, --help                 Give this help list
      --usage                Give a short usage message
~/deneme$ ./a.out --usage
Usage: a.out [-?] [--help] [--usage]
~/deneme$ ./a.out --version
./a.out: unrecognized option '--version'
Try 'a.out --help' or 'a.out --usage' for more information.
~/deneme$ ./a.out alooo
a.out: Too many arguments
Try 'a.out --help' or 'a.out --usage' for more information.
```

3.11.2. 2. Örnek

Bu yazılımda GNU standart komut satırı biçimi ile uyumlu argp kullanımı dışında herhangi bir seçenek ya da argüman tanımlanmamıştır.

--help ve **--usage** seçeneklerine ek olarak GNU standartlarına uygun olarak bir de **--version** seçeneğine sahiptir. GNU standardındaki gibi **--help** çıktısında açıklayıcı bir dizge ile hata bildirme adresi basar.

argp değişkeni argüman çözümleyici belirtimini içerir. **argp_parse** işlevine parametreler bu yapının alanları üzerinden aktarılır. Normalde ilk üç alan kullanılır ama bu küçük yazılımda kullanılmamıştır. Argp arayüzünün kullandığı iki genel değişken bu yazılımda kullanılmıştır: **argp_program_version** ve

argp_program_bug_address. Bunlar, hemen hemen her yazılımda çeşitli görevler için farklı argüman çözümleyiciler kullanılıyor olsa bile, daima birer sabit olarak verildiğinden genel değişkenler olacağı varsayılmıştır.

```
/* 2. Argp Örneği - Argp kullanılan az küçük bir yazılım */

/*  Bu yazılımda GNU standart komut satırı biçimi ile uyumlu argp
    kullanımı dışında herhangi bir seçenek ya da argüman
    tanımlanmamıştır.

    --help ve --usage seçeneklerine ek olarak GNU standartlarına uygun
    olarak bir de --version seçeneğine sahiptir. GNU standardındaki gibi
    --help çıktısında açıklayıcı bir dizge ile hata bildirme adresi basar.

    argp değişkeni argüman çözümleyici belirtimini içerir. argp_parse
    işlevine parametreler bu yapının alanları üzerinden aktarılır. Normalde
    ilk üç alan kullanılır ama bu küçük yazılımda kullanılmamıştır. Argp
    arayüzünün kullandığı iki genel değişken bu yazılımda kullanılmıştır:
    argp_program_version ve  argp_program_bug_address.
    Bunlar, hemen hemen her yazılımda çeşitli görevler için farklı
    argüman çözümleyiciler kullanılıyor olsa bile, daima birer sabit
    olarak verildiğinden genel değişkenler olacağı varsayılmıştır. */

#include <argp.h>

const char *argp_program_version =
  "argp-ex2 1.0";
const char *argp_program_bug_address =
  "<bug-gnu-utils@gnu.org>";

/* Yazılım açıklaması. */
static char doc[] =
  "Argp example #2 -- a pretty minimal program using argp";

/*  Argüman çözümleyicimiz. options, parser, ve
    args_doc alanları sıfırdır, çünkü bizim seçenek ve
    argümanımız yok. --help seçeneğinin çıktısında doc ve
    argp_program_bug_address, --version seçeneğinin çıktısında ise
    argp_program_version kullanılacak.  */
static struct argp argp = { 0, 0, 0, doc };

int main (int argc, char **argv)
{
  argp_parse (&argp, argc, argv, 0, 0, 0);
  exit (0);
}
```

Çıktısı şöyle birşey oluyor:

```
~/deneme$ gcc -o argp-ex2 deneme.c
~/deneme$ ./argp-ex2 --version
argp-ex2 1.0
~/deneme$ ./argp-ex2 --help
Usage: argp-ex2 [OPTION...]
Argp example #2 -- a pretty minimal program using argp

  -?, --help                 Give this help list
```

```
      --usage                Give a short usage message
  -V, --version              Print program version

Report bugs to <bug-gnu-utils@gnu.org>.
~/deneme$ ./argp-ex2 --usage
Usage: argp-ex2 [-?V] [--help] [--usage] [--version]
```

3.11.3. 3. örnek

Bu yazılımda 2. örneğe ek olarak bazı kullanıcı seçenekleri ve argümanları kullanılmıştır.

Bu örnekte **argp**'nin ilk dört alanını kullandık (*Argp Çözümleyicisinin Belirtilmesi* (sayfa: 654)) ve çözümleyici işlev olarak **parse_opt** işlevini belirttik. Bkz. *Argp Çözümleyici İşlevleri* (sayfa: 657).

Bu örnekte, **main** işlevinde **parse_opt** ile iletişim için bir yapı kullanıldığına dikkat edin. Bu yapı, bir gösterici olarak **argp_parse** tarafından *input* argümanında aktarılır. Bkz. *Argp* (sayfa: 653). **parse_opt** işlevi tarafından *state* argümanının **input** alanı ile alınır. Bkz. *Argp Çözümleme Durumu* (sayfa: 661). Şüphesiz bunun yerine genel değişkenler kullanmak mümkündür ama böyle bir yapı kullanmak biraz daha esnek ve temizdir.

```
/* 3. Argp Örneği -- Argp arayüzünü ek seçenek ve argümanlarla
   kullanan bir yazılım örneği
*/

/*  Bu yazılımda 2. örneğe ek olarak bazı kullanıcı seçenekleri ve
    argümanları kullanılmıştır.

    Bu örnekte argp'nin ilk dört alanını kullandık:
      options  - argp_option vektörüne bir gösterici (aşağıya bakın)
      parser   - argp tarafından çağrılan ve tek bir seçeneği çözümleyen işlev
      args_doc - seçenek olmayan argümanların kullanımını açıklayan bir dizge
      doc      - bu yazılımın açıklamasını içeren dizge; bir düşey sekme (\v)
                 içeriyorsa, bundan sonraki parça seçeneklerden sonra basılır

    parser işlevi şu argümanları alır:
      key   - Seçeneğin türünü (argp_option yapısının KEY alanından alınarak)
              ya da bunun dışında birşeyi belirten özel bir anahtar; burada
              kullandığımız tek özel anahtar bir seçenek olmayan argüman
              belirten ARGP_KEY_ARG anahtarıdır. ARGP_KEY_END anahtarı ise
              tüm argümanların çözümlendiğini belirtir.
      arg   - bir dizge olarak seçenek argümanı; argümansızsa NULL
      state - argp_state yapısına bir gösterici; çözümleme durumu ile ilgili
              faydalı bilgiler içerir. Burada kullanılan, argp_parse işlevinin
              girdi argümanı olan input alanı ile çözümlenen
              seçenek olmayan argümanın numarasını içeren arg_num alanıdır.
    İşlev başarılı ise 0 ile belirtilen anahtar bilinmiyorsa ARGP_ERR_UNKNOWN
    ile ya da başka bir hatayı belirten bir hata kodu ile dönmelidir.

    Bu örnekte, main işlevinde parse_opt ile iletişim için bir yapı
    kullanıldığına dikkat edin. Bu yapı, bir gösterici olarak argp_parse
    tarafından input argümanında aktarılır. parse_opt işlevi tarafından
    state argümanının input alanı ile alınır. Şüphesiz bunun yerine genel
    değişkenler kullanmak mümkündür ama böyle bir yapı kullanmak biraz daha
    esnek ve temizdir.

    options alanı bir argp_option vektörüne bir gösterici içerir; bu yapı
    aşağıdaki alanlara sahiptir (bu örnekteki gibi dizi ilklendirmesiyle
```

```
    seçenek yapılarınıza atama yapıyorsanız, belirtilmeyen alanlar öntanımlı
    olarak 0 olacak ve belirtilmeleri gerekmeyecektir):
      name   - seçeneğin uzun seçenek ismi (sıfır olabilir)
      key    - bu seçenek ve seçeneğin kısa seçenek ismi (basılabilen bir ascii
               karakterse) çözümlenirken çözümleyici işleve aktarılacak anahtar.
      arg    - varsa, bu seçeneğinin argümanının ismi
      flags  - bu seçeneği açıklayan bayraklar; bazıları:
               OPTION_ARG_OPTIONAL - bu seçeneğin argümanı isteğe bağlıdır
               OPTION_ALIAS        - bu seçenek önceki seçeneğe bir takma addır.
               OPTION_HIDDEN       - --help çıktısında bu seçenek gösterilmez.
      doc    - --help çıktısında bu seçeneğin açıklamasını içeren dizge

    Bir seçenek vektörü tüm alanları sıfır değeri içeren bir yapı ile
    sonlanmalıdır.
*/

#include <argp.h>

const char *argp_yazılım_version =
  "argp-ex3 1.0";
const char *argp_yazılım_bug_address =
  "<bug-gnu-utils@gnu.org>";

/* Yazılım açıklaması. */
static char doc[] =
  "Argp example #3 -- a program with options and arguments using argp";

/* Kabul ettiğimiz argümanlar için bir açıklama. */
static char args_doc[] = "ARG1 ARG2";

/* Kabul ettiğimiz seçenekler. */
static struct argp_option options[] = {
  {"verbose",  'v', 0,      0,  "Produce verbose output" },
  {"quiet",    'q', 0,      0,  "Don't produce any output" },
  {"silent",   's', 0,      OPTION_ALIAS },
  {"output",   'o', "FILE", 0,
    "Output to FILE instead of standard output" },
  { 0 }
};

/* parse_opt ile main iletişiminde kullanılır. */
struct arguments
{
  char *args[2];                /* arg1 ve arg2 */
  int silent, verbose;
  char *output_file;
};

/* Tek bir seçeneği çözümlemek için. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  /* argp_parse'daki girdi argümanında bizim arguments
     yapısına bir gösterci olduğunu biliyoruz. */
  struct arguments *arguments = state->input;

  switch (key)
```

```
    {
    case 'q': case 's':
      arguments->silent = 1;
      break;
    case 'v':
      arguments->verbose = 1;
      break;
    case 'o':
      arguments->output_file = arg;
      break;

    case ARGP_KEY_ARG:
      if (state->arg_num >= 2)
        /* Argümanlar fazla geldi. */
        argp_usage (state);

      arguments->args[state->arg_num] = arg;

      break;

    case ARGP_KEY_END:
      if (state->arg_num & 2)
        /* Argümanlar yetersiz. */
        argp_usage (state);
      break;

    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}

/* Argp çözümleyicimiz. */
static struct argp argp = { options, parse_opt, args_doc, doc };

int main (int argc, char **argv)
{
  struct arguments arguments;

  /* Öntanımlı değerler. */
  arguments.silent = 0;
  arguments.verbose = 0;
  arguments.output_file = "-";

  /* Argümanlarımız çözümlensin;  parse_opt tarafından
     görülen her seçenek arguments içine yansıyacak. */
  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  printf ("ARG1 = %s\nARG2 = %s\nOUTPUT_FILE = %s\n"
          "VERBOSE = %s\nSILENT = %s\n",
          arguments.args[0], arguments.args[1],
          arguments.output_file,
          arguments.verbose ? "yes" : "no",
          arguments.silent ? "yes" : "no");

  exit (0);
}
```

3.11.4. 4. Örnek

Bu yazılım, 3. örnekteki özelliklerden fazla olarak daha fazla seçenek içerir ve **--help** çıktısı için daha fazla yapı kullanılmıştır. Ayrıca, bir öğe listesi kabul eden yazılımlar için belli bir noktadan sonraki girdi argümanlarının nasıl 'çalınabileceği' gösterilmiştir. Bundan başka, yazılıma seçenek olmayan argümanların belirtilmediği durumda *key* argümanında **ARGP_KEY_NO_ARGS** anahtarının kullanımı gösterilmiştir. Bkz. *Argp Çözümleyici İşlevleri için Özel Anahtarlar* (sayfa: 658).

Yardım çıktısının yapılanması için iki özellik kullanılmıştır: *başlıklar* ve iki parçalı seçenek dizgesi. *başlıklar* seçenekler vektöründeki ilk dört alanı 0 olan girdilerdir. Bkz. *Seçenekler* (sayfa: 655). İki parçalı açıklama dizgesi **doc** değişkeninde belirtilmiştir. Açıklama dizgesinin düşey sekme karakterine (**'\v'** veya **'\013'**) kadar olan kısmı seçeneklerden önce, kalan kısmı da seçeneklerden sonra basılır. Teamülen, seçeneklerden önce basılan kısım yazılımın ne iş yaptığını kısaca açıklamak içindir. Seçeneklerden sonra basılan kısım ise, yazılımın davranışını daha ayrıntılı açıklayan daha uzun bir dizgedir. Açıklama dizgesinin her iki parçası da çıktıya özdevinimli olarak sığdırılır, belli noktalarda satırları sonlandırmak için satırsonu karakterleri kullanılabilir. Ek olarak, açıklama dizgeleri o anki yerele uygun olarak çevrilmesi için **gettext** işlevine aktarılır.

```
/* 4. Argp Örneği - Biraz daha karmaşık seçenekli bir yazılım */

/*  Bu yazılım, 3. örnekteki özelliklerden fazla olarak daha fazla seçenek
    içerir ve --help çıktısı için daha fazla yapı kullanılmıştır.
    Ayrıca, bir öğe listesi kabul eden yazılımlar için belli bir noktadan
    sonraki girdi argümanlarının nasıl 'çalınabileceği' gösterilmiştir.
    Bundan başka, yazılıma seçenek olmayan argümanların belirtilmediği durumda
    key argümanında ARGP_KEY_NO_ARGS anahtarının kullanımı gösterilmiştir.

    Yardım çıktısının yapılanması için iki özellik kullanılmıştır:
    başlıklar ve iki parçalı seçenek dizgesi.
    başlıklar, seçenekler vektöründeki ilk dört alanı 0 olan girdilerdir.
    İki parçalı açıklama dizgesi doc değişkeninde belirtilmiştir. Açıklama
    dizgesinin düşey sekme karakterine ('\v' veya '\013') kadar olan kısmı
    seçeneklerden önce, kalan kısmı da seçeneklerden sonra basılır. Teamülen,
    seçeneklerden önce basılan kısım yazılımın ne iş yaptığını kısaca
    açıklamak içindir. Seçeneklerden sonra basılan kısım ise, yazılımın
    davranışını daha ayrıntılı açıklayan daha uzun bir dizgedir. Açıklama
    dizgesinin her iki parçası da çıktıya özdevinimli olarak sığdırılır,
    belli noktalarda satırları sonlandırmak için satırsonu karakterleri
    kullanılabilir. Ek olarak, açıklama dizgeleri o anki yerele uygun olarak
    çevrilmesi için gettext işlevine aktarılır.
*/

#include <stdlib.h>
#include <error.h>
#include <argp.h>

const char *argp_program_version =
  "argp-ex4 1.0";
const char *argp_program_bug_address =
  "<bug-gnu-utils@prep.ai.mit.edu>";

/* Yazılım açıklaması. */
static char doc[] =
  "Argp example #4 -- a yazılım with somewhat more complicated\
options\
\vThis part of the documentation comes *after* the options;\
  note that the text is automatically filled, but it's possible\
```

```
  to force a line-break, e.g.\n<-- here.";

/* Kabul ettiğimiz argümanlar için açıklama. */
static char args_doc[] = "ARG1 [STRING...]";

/* Kısa seçeneksiz seçenekler için anahtarlar. */
#define OPT_ABORT  1            /* -abort */

/* Kabul ettiğimiz seçenekler. */
static struct argp_option options[] = {
  {"verbose",  'v', 0,       0, "Produce verbose output" },
  {"quiet",    'q', 0,       0, "Don't produce any output" },
  {"silent",   's', 0,       OPTION_ALIAS },
  {"output",   'o', "FILE",  0,
    "Output to FILE instead of standard output" },

  {0,0,0,0, "The following options should be grouped together:" },
  {"repeat",   'r', "COUNT", OPTION_ARG_OPTIONAL,
    "Repeat the output COUNT (default 10) times"},
  {"abort",    OPT_ABORT, 0, 0, "Abort before showing any output"},

  { 0 }
};

/* main ile parse_opt'un iletişimi için kullanılır. */
struct arguments
{
  char *arg1;                   /* arg1 */
  char **strings;               /* [string...] */
  int silent, verbose, abort;   /* -s, -v, --abort */
  char *output_file;            /* --output için dosya ismi*/
  int repeat_count;             /* --repeat için argüman sayısı*/
};

/* Tek bir seçeneği çözümlemek için. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
  /* argp_parse'daki girdi argümanında bizim arguments
     yapısına bir gösterci olduğunu biliyoruz. */
  struct arguments *arguments = state->input;

  switch (key)
    {
    case 'q': case 's':
      arguments->silent = 1;
      break;
    case 'v':
      arguments->verbose = 1;
      break;
    case 'o':
      arguments->output_file = arg;
      break;
    case 'r':
      arguments->repeat_count = arg ? atoi (arg) : 10;
      break;
    case OPT_ABORT:
```

```
      arguments->abort = 1;
      break;

    case ARGP_KEY_NO_ARGS:
      argp_usage (state);

    case ARGP_KEY_ARG:
      /*  Burada daha fazla argüman alabilecekken çözümlemeyi
          sonlandırdığımız için state->arg_num == 0 olduğunu biliyoruz. */
      arguments->arg1 = arg;

      /*  Artık kalan tüm argümanları tüketebiliriz.
          state->next ilgilendiğimiz ilk dizge olarak çözümlenecek sonraki
          argümanın state->argv içindeki indisidir.
          Yani, arguments->strings için değer olarak
          &state->argv[state->next] kullanabiliriz.

          Buna ek olarak, state->next'e argümanların sonunu atayarak,
          argp'nin çözümlemeyi burada sonlandırıp dönmesini sağlayabiliriz. */
      arguments->strings = &state->argv[state->next];
      state->next = state->argc;

      break;

    default:
      return ARGP_ERR_UNKNOWN;
    }
  return 0;
}

/* Argp çözümleyicimiz. */
static struct argp argp = { options, parse_opt, args_doc, doc };

int main (int argc, char **argv)
{
  int i, j;
  struct arguments arguments;

  /* Öntanımlı değerler. */
  arguments.silent = 0;
  arguments.verbose = 0;
  arguments.output_file = "-";
  arguments.repeat_count = 1;
  arguments.abort = 0;

  /* Argümanlarımız çözümlensin;  parse_opt tarafından
     görülen her seçenek arguments içine yansıyacak. */
  argp_parse (&argp, argc, argv, 0, 0, &arguments);

  if (arguments.abort)
    error (10, 0, "ABORTED");

  for (i = 0; i < arguments.repeat_count; i++)
    {
      printf ("ARG1 = %s\n", arguments.arg1);
      printf ("STRINGS = ");
      for (j = 0; arguments.strings[j]; j++)
```

```
        printf (j == 0 ? "%s": ", %s", arguments.strings[j]);
      printf ("\n");
      printf ("OUTPUT_FILE = %s\nVERBOSE = %s\nSILENT = %s\n",
              arguments.output_file,
              arguments.verbose ? "yes": "no",
              arguments.silent ? "yes": "no");
    }

  exit (0);
}
```

3.12. Argp Arayüzünün Kişiselleştirmesi

Argp **`--help`** çıktısının biçimi bazı bakımlardan yazılımın kullanıcıları tarafından belirlenebilir. Bu işlem **`ARGP_HELP_FMT`** ortam değişkenine virgül ayraçlı bir takım sözcükler belirterek yapılır. Boşluklar yoksayılır:

`dup-args`
`no-dup-args`
: Yinelenen argüman kipini açar/kapar. Eğer bir seçenek aynı argümanı farklı seçenek isimleriyle kabul ediyorsa, yinelenen argüman kipinde, argüman her seçenek ismiyle ayrı ayrı gösterilir. Aksi takdirde, argüman sadece ilk uzun seçenekle birlikte gösterilir. Ardarda gösterilen farklı seçenek isimlerinden birinde belirtilen argümanın, diğer seçenek isimleriyle de kullanılacağını kullanıcı bilir. Öntanımlı olan **`no-dup-args`**'dır, yani argüman bir defa gösterilir.

`dup-args-note`
`no-dup-args-note`
: Seçenek argümanı yinelemesi yapılmadığında, kullanıcıya bilgilendirme iletisi çıktılanmasını açar/kapar. Öntanımlı olan **`dup-args-note`**'dur.

`short-opt-col=`*n*
: Kısa seçeneğin basılacağı sütun. Öntanımlı değeri 2'dir.

`long-opt-col=`*n*
: Uzun seçeneğin basılacağı sütun. Öntanımlı değeri 6'dır.

`doc-opt-col=`*n*
: Yazılım açıklamasının basılacağı sütun (bkz. *Bayraklar* (sayfa: 656)). Öntanımlı değeri 2'dir.

`opt-doc-col=`*n*
: Seçenek açıklamalarının basılacağı sütun. Öntanımlı değeri 29'dur.

`header-col=`*n*
: Grup başlıklarının basılacağı sütun. Öntanımlı değeri 1'dir.

`usage-indent=`*n*
: **`Usage:`**'den sonraki iletinin basılacağı sütun. Öntanımlı değeri 12'dir.

`rmargin=`*n*
: Satır sarmalamasının yapılacağı sütun. Öntanımlı değeri 79'dur.

3.13. Alt Seçeneklerin Çözümlenmesi

Bazan tek seviyeli seçenekler yetersiz olur. Ya çok fazla seçenek olur ya da birbiriyle ilişkili seçenekler olur.

Bu durumda yazılımlar alt seçenekler kullanır. Bu tür yazılımlara bilinen en iyi örnek **mount**(8)[B965]'dur. **-o** seçeneği, virgül ayraçlı seçenek listesi olarak tek bir argüman alır. Böyle bir kodun geliştirilmesini kolaylaştırmak için **getsubopt** işlevi vardır.

```
int getsubopt(char              **altseçenekler,                    işlev
              const char* const *isimler,
              char              **değerler)
```

altseçenekler parametresi işlenecek dizgenin adresini içeren bir gösterici olmalıdır. İşlev, bir alt seçeneği çözümledikten sonra sonraki alt seçeneğin adresini, tüm altseçenekler işlenmişse sonlandırıcı boş karakterin (**\0**) adresini bu argümana yerleştirir.

isimler parametresi bilinen altseçenek isimlerini içeren bir dizge dizisidir. Tüm dizgeler boş karakterle, dizge dizisi ise boş gösterici ile sonlandırılmalıdır. İşlev, geçerli altseçeneği bulmak için *isimler* dizisindeki altseçenek isimleriyle karşılaştırma yapar ve bulduğu ismin dizideki indisi ile döner.

Altseçeneğin **=** karakteri ile bir değerle ilişkilendirlmesi durumunda, değerin göstericisi *değerler* içinde döndürülür. Değer boş karakter sonlandırmalıdır. Bir değer belirtilmemişse boş gösterici kullanılır. Böylece çağrıcı gerekli değerin verilip verilmediğini ya da umulmadık bir değer mi verildi acaba diye sınama yapabilir.

Bir altseçeneğin *isimler* dizisinde olmaması durumunda, alt seçeneğin başlangıç adresi olası değerini de içererek *değerler* argümanına konur ve işlev **-1** değeriyle döner.

3.14. Alt Seçenek Çözümleme Örneği

mount(8)[B966] yazılımının kodu **getsubopt** kullanımı için en iyi örnektir:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int do_all;
const char *type;
int read_size;
int write_size;
int read_only;

enum
{
  RO_OPTION = 0,
  RW_OPTION,
  READ_SIZE_OPTION,
  WRITE_SIZE_OPTION,
  THE_END
};

const char *mount_opts[] =
{
  [RO_OPTION] = "ro",
  [RW_OPTION] = "rw",
  [READ_SIZE_OPTION] = "rsize",
  [WRITE_SIZE_OPTION] = "wsize",
  [THE_END] = NULL
};
```

```
int
main (int argc, char *argv[])
{
  char *subopts, *value;
  int opt;

  while ((opt = getopt (argc, argv, "at:o:")) != -1)
    switch (opt)
      {
      case 'a':
        do_all = 1;
        break;
      case 't':
        type = optarg;
        break;
      case 'o':
        subopts = optarg;
        while (*subopts != '\0')
          switch (getsubopt (&subopts, mount_opts, &value))
            {
            case RO_OPTION:
              read_only = 1;
              break;
            case RW_OPTION:
              read_only = 0;
              break;
            case READ_SIZE_OPTION:
              if (value == NULL)
                abort ();
              read_size = atoi (value);
              break;
            case WRITE_SIZE_OPTION:
              if (value == NULL)
                abort ();
              write_size = atoi (value);
              break;
            default:
              /* Unknown suboption. */
              printf ("Unknown suboption `%s'\n", value);
              break;
            }
        break;
      default:
        abort ();
      }

  /* Do the real work. */

  return 0;
}
```

4. Ortam Değişkenleri

Bir yazılım çalıştırıldığında, bağlamı hakkında bilgiyi iki yolla alabilir. İlk yöntemde, *Yazılım Argümanları* (sayfa: 645) kısmında açıklanan **main** işlevinin argümanları olan *argv* ve *argc* parametreleri kullanılırken, ikinci yöntemde bu kısımda açıklanacak olan ***ortam değişkenleri*** kullanılır.

argv mekanizması, yazılımı çalıştırmak için komut satırına yazılan komut satırı argümanlarını kullanır. Ortam ise, çoğu yazılımın ortaklaşa kullandığı, sıkça değişen ama daha az sıklıkla kullanılan bilgilerden oluşur.

Bu kısımda bahsedilecek ortam değişkenleri kabuğun **export** komutu kullanılarak atanan ortam değişkenlerinden başka birşey değildir. Kabukta çalıştırılan bütün yazılımlar ortam değişkenlerini kabuktan miras alırlar.

Standart ortam değişkenleri, kullanıcının ev dizini, uçbirim türü, geçerli yerel, vb. hakkında bilgileri kullanır; bunlara ek olarak kendi amaçlarınıza uygun ortam değişkenleri de tanımlayabilirsiniz. Tüm ortam değişkenlerinin ve değerlerinin hepsine birden ***ortam*** adı verilir.

Ortam değişkenlerinin isimleri harf büyüklüğüne duyarlıdır ve isimler **=** karakterini içermemelidir. Sistemce tanımlanmış ortam değişkenlerinin isimlerinin tamamı büyük harflerden oluşur.

Ortam değişkenlerinin değerleri bir dizge olarak ifade edilebilecek herhangi bir değer olabilir. Böyle bir değerin içinde, dizgeyi sonlandırması nedeniyle, bir boş karakter olmamalıdır.

4.1. Ortama Erişim

Bir ortam değişkeninin değerine **getenv** işlevi ile erişebilirsiniz. Bu işlev `stdlib.h` başlık dosyasında bildirilmiştir. Bu bölümdeki işlevlerin hepsini çok evreli yazılımlarda güvenle kullanabilirsiniz. Ortamın böyle rasgele değişmesi hatalara yol açmaz.

```
char *getenv(const char *isim)                                    işlev
```

Bu işlev *isim* ile ismi belirtilen ortam değişkeninin değeri olan dizgeyi döndürür. Bu dizgeyi değiştirmemelisiniz. GNU kütüphanesinin kullanılmadığı bazı Unix sistemlerinde **getenv** çağrısının sonraki çağrıları bu değerin üzerine yazar (ama başka bir kütüphane işlevi bunu yapmaz). Eğer *isim* isimli bir ortam değişkeni yoksa, işlev bir boş gösterici ile döner.

```
int putenv(char *dizge)                                           işlev
```

putenv işlevi ortam değişkenlerini tanımlamak ya da kaldırmak için kullanılır. Ortamda bir değişken tanımlamak için *dizge*, *isim=değer* biçiminde verilmelidir. Aksi takdirde *dizge*, mevcut bir ortam değişkeninin ismi olarak yorumlanıp, bu değişken ortamdan kaldırılır.

setenv işlevinden farkı, *dizge* parametresi olarak belirtilen dizgenin ortama konulmasıdır. Eğer **putenv** çağrısından sonra kullanıcı değişkenin değerini değiştirmezse bu dizge ortamda aynen böyle görünecektir. Ayrıca, *dizge* olarak belirtilen isim, değişken ortamdan kaldırıldığında da yazılımın etki alanı içinde varlığını sürdüren bir özdevinimli değişkenin ismi olmamalıdır. Aynısı normal olarak, daha sonra serbest bırakılan özdevimli ayrılmış değişkenler için de geçerlidir.

Bu işlev genişletilmiş Unix arayüzünün bir parçasıdır. Eski SVID kütüphanelerinde de kullanılabildiğinden bunu sağlamak için yazılım içinde herhangi bir başlık dosyasından önce *_XOPEN_SOURCE* veya *_SVID_SOURCE* tanımlamalısınız.

```
int setenv(const char *isim,                                      işlev
           const char *değer,
           int         değiştir)
```

setenv işlevi ortama yeni bir tanım eklemek için kullanılabilir. *isim* isimli girdi *isim=değer* değeri ile değiştirilir. Bunun, *değer* bir boş dizge olarak verildiğinde de böyle olacağını unutmayın. Bunu yapmak için yeni dizge oluşturulur ve *isim* ve *değer* dizgelerine kopyalanır. *değer* dizgesi olarak boş gösterici kuraldışıdır (boş dizge, boş gösterici değildir). Ortam zaten *isim* isimli bir değişken içeriyorsa bu durumda ne yapılacağı *değiştir* ile belirtilir. *değiştir* değeri sıfırsa hiçbir şey yapılmaz. Aksi takdirde eski girdi yenisi ile değiştirilir.

Bir girdiyi ortamdan bu işlev ile kaldıramayacağınızı lütfen unutmayın.

Bu işlev bir zamanlar BSD kütüphanesinin bir parçasıyken şimdi Unix standardının da parçasıdır.

`int unsetenv(const char *isim)` işlev

Bu işlevi kullanarak bir değişkeni ortamdan tamamen kaldırabilirsiniz. Eğer ortamda ismi *isim* olan bir ortam değişkeni varsa, bu girdi ortamdan tamamen kaldırılır. Bu işlevin çağrısı, **putenv** işlevinin *değer* argümanına boş dizge belirtilerek çağrılmasına eşdeğerdir.

isim bir boş gösterici ise, bir boş dizge ise ya da dizge bir **=** karakeri içeriyorsa işlev **-1** ile döner. Çağrı başarılı olduğunda **0** döner.

Bu işlev bir zamanlar BSD kütüphanesinin bir parçasıyken şimdi Unix standardının da parçasıdır. Ne var ki, BSD sürümü bir değer döndürmüyordu.

Ortamda değişiklik yapan bir işlev daha vardır. Bu işlevin POSIX.9 [Fortran 77 ile bağlantılı POSIX] içinde olduğundan ve onun POSIX.1'e dahi edilmesi gerektiğinden bahsedilir. Fakat bu şimdiye kadar olmadı. Ama biz bu işlevi bir GNU oluşumu olarak, standart Fortran ortamlarına uyumlu yazılım geliştirilebilmesini sağlamak için bulunduruyoruz.

`int clearenv(void)` işlev

clearenv işlevi ortamdan tüm girdileri kaldırır. **putenv** ve **setenv** çağrılarıyla ortama tekrar yeni girdiler eklenebilir.

İşlev başarılı olursa **0** ile döner. Aksi takdirde sıfırdan farklı bir değer ile döner.

Ortama değişken eklemek için ortam nesnelerinin bellibaşlı gösterimleri ile doğrudan çalışabilirsiniz (örneğin, çalıştıracağınız başka bir yazılımla haberleşmek için; bkz. *Bir Dosyanın Çalıştırılması* (sayfa: 688)).

`char **environ` değişken

Ortamı bir dizge dizisi olarak içerir. Her dizge *isim=değer* biçimindedir. Dizgelerin ortmda göründüğü sıra önemsizdir, fakat aynı isim birden fazla görünmez. Dizinin son elemanı bir boş göstericidir.

Bu değişken `unistd.h` başlık dosyasında bildirilmiştir.

Sadece bir ortam değişkeninin değeri ile ilgileniyosanız **getenv** işlevini kullanın.

Unix sistemleri ve GNU sisteminde **environ** değişkenin değeri **main** işlevinde üçüncü bir argüman olarak belirtilebilir. Bkz. *Yazılım Argümanları* (sayfa: 645).

4.2. Standart Ortam Değişkenleri

Bu ortam değişkenlerinin anlamları standarttır. Bu onları daima ortam değişkenleri olarak belirtilebileceği anlamına gelmez; sadece bu değişkenlerle belirtildiklerinde hep aynı anlama gelirler. Bu ortam değişkenlerinin isimlerini başka amaçlarla kullanmayı denememelisiniz.

`HOME`

Bu dizge kullanıcının ***ev dizini*** ya da oturum açtığında içine düştüğü öntanımlı çalışma dizinidir.

Kullanıcı **HOME** değişkenine herkangi bir değer atayabilir. Bu bakımdan, belli bir kullanıcının ev dizinin yerini öğrenmek için bu değişkene bakmamalısınız, bunu yerine *kullanıcı veritabanında* (sayfa: 760) kullanıcının ismine bakmalısınız.

Kullanıcı buraya istediği değeri atayabildiğinden, **HOME** değişkenini başka amaçlar için kullanmak daha iyidir.

LOGNAME
: Bu kullanıcının sisteme oturum açarken kullandığı isimdir. Ortamdaki değerler keyfi olarak değiştirilebildiğinden bir yazılımı çalıştıran kullanıcının kim olduğuna bakmak için bu değişkenin kullanılması doğru bir yöntem olmayacaktır. En iyisi **getlogin** (*Oturumu Açan Kim?* (sayfa: 752)) gibi bir işlev kullanmaktır.

  Kullanıcı buraya istediği değeri atayabildiğinden, **LOGNAME** değişkenini başka amaçlar için kullanmak daha iyidir.

PATH
: Bir dosya yolu (path), bir dosyanın hangi dizinlerde aranacağını belirtmek için kulanılır. **PATH** ortam değişkeni ise çalıştırılacak bir yazılımın aranacağı dizinleri belirtmek için kullanılır.

  execlp ve **execvp** işlevleri (*Bir Dosyanın Çalıştırılması* (sayfa: 688)) bu ortam değişkenini kullanır, dolayısıyla bu işlevlerle gerçeklenmiş uygulamalar ve kabuk da bu değişkeni kullanır.

  Değişkenin değeri, dizin isimlerinin iki nokta üstüstelerle ayrılmasıyla oluşturulan bir dizgedir. Bir dizin olarak belirtilmiş boş bir dizge *çalışılan dizini* (sayfa: 351) belirtir.

  Örneğin, bu ortam değişkeni için değer olarak belirtilen bir dizge:

```
:/bin:/etc:/usr/bin:/usr/new/X11:/usr/new:/usr/local/bin
```

  ise ve kullanıcı **foo** isimli bir yazılımı çalıştırmak isterse, kabuk sırayla **./foo**, **/bin/foo**, **/etc/foo**, ... dosyalarını arayacak ve önce hangisini bulursa onu çalıştıracaktır.

TERM
: Yazılım çıktısını alan uçbirimin çeşidini belirler. Bazı uygulamalar bu değeri özel önceleme dizilimlerinden ya da belli başlı uçbirim çeşitleri ile desteklenen uçbirim kiplerinden yararlanmak için kullanır. Örneğin, termcap kütüphanesini[B974] kullanan çoğu yazılım **TERM** ortam değişkenini kullanır.

TZ
: Zaman dilimini belrtir. Bu dizgenin biçimi ve nasıl kullanıldığı hakkında daha ayrıntılı bilgi edinmek için *Zaman Diliminin TZ ile Belirtilmesi* (sayfa: 565) bölümüne bakınız.

LANG
: Ne **LC_ALL** değişkeni ne de belli bir kategori için tanımlanan bir ortam değişkeni ile tanımlanmış yerel kategorileri için öntanımlı yereli belirtir. Yereller ile ilgili daha ayrıntılı bilgi için *Yereller ve Uluslararasılaştırma* (sayfa: 164) bölümüne bakınız.

LC_ALL
: Bu ortam değişkeni tanımlanmışsa, bunun değeri atanmış diğer tüm **LC_*** ortam değişkenlerinin değerlerine göre öncelik kazanır. Yani bu değişken ortamda tanımlıysa diğer tüm **LC_*** ortam değişkenleri yoksayılır.

LC_COLLATE
: Dizge sıralaması için kullanılacak yereli belirtir.

LC_CTYPE
: Karakter kümeleri ve karakter sınıflaması için kullanılacak yereli belirtir.

LC_MESSAGES

Basılan iletilerin dili ve bunlara yanıtların çözümlenmesi için kullanılacak yereli belirtir.

`LC_MONETARY`
Parasal değerleri biçimlemek için kullanılacak yereli belirtir.

`LC_NUMERIC`
Sayıları biçimlemek için kullanılacak yereli belirtir.

`LC_TIME`
Tarih/saat değerlerini biçimlemek için kullanılacak yereli belirtir.

`NLSPATH`
İleti çeviri kataloglarının bulunduğu dizinleri **`catopen`** işlevine belirtmek için kullanılır.

`_POSIX_OPTION_ORDER`
Bu ortam değişkeni tanımlıysa, **`getopt`** ve **`argp_parse`** işlevleri tarafından komut satırı argümanlarının yeniden sıralanması engellenir. Bkz. *Yazılım Argümanları için Sözdizimi Uzlaşımları* (sayfa: 645).

5. Sistem Çağrıları

Bir ***sistem çağrısı*** bir yazılımın çekirdekten bir hizmet isteği yapması için kullanılır. Hizmet genellikle, G/Ç işlemleri gibi sadece çekirdeğin ayrıcalığında olan şeylerdir. Yazılım geliştirenlerin genellikle bu sistem çağrılarını bilmeye ihtiyacı olmaz. Çünkü GNU C kütüphanesi sistem çağrılarının yaptığı hemen herşeyi sanal olarak sağlayan işlevler içerir. Örneğin, bir dosyanın erişim izinlerini değiştiren bir sistem çağrısı vardır, ama GNU C kütüphanesinin **`chmod`** işlevi zaten bu işlemi yaptığından bu sistem çağrısını kullanma ihtiyacı ortaya çıkmaz.

Sistem çağrılarına bazan ***çekirdek çağrıları*** dendiği de olur.

Her ne kadar doğrudan sistem çağrıları yapma ihtiyacı duyulmasa da GNU C kütüphanesi bunu yapabilmeniz için **`syscall`** işlevini içerir. **`syscall`** kullanımı zordur ve **`chmod`** gibi işlevleri kullanmak daha taşınabilirdir, ama sistem çağrılarını makina kodu komutları ile kodlamaktan daha kolay ve daha taşınabilirdir.

`syscall` çağrıları, henüz GNU C kütüphanesinde bulunmayan özel sistem çağrıları ile çalışacağınız zaman oldukça kulllanışlı olacaktır. **`syscall`** tamamen soysal bir yöntemle gerçeklenmiştir; işlev belli bir sistem çağrısının ne yaptığıyla, hatta geçerli olup olmadığıyla bile ilgilenmez.

Bu kısımdaki **`syscall`** işlevi ile ilgili açıklama, GNU C kütüphanesinin çalıştığı çeşitli platformlardaki sistem çağrıları için belli bir protokolün varlığı kabulüne dayanır. Bu protokol herhangi bir otorite tarafından tanımlanmamıştır, ancak onu burada açıklamayacağız.

`syscall` işlevi `unistd.h` başlık dosyasında bildirilmiştir.

```
long int syscall(long int sysno, ...)                                    işlev
```

`syscall` temel bir sistem çağrısını uygular.

sysno, sistem çağrı numarasıdır. Sistem çağrılarının her biri bir numara ile yapılır. Olası tüm sistem çağrılarının numaralarını içeren makrolar `sys/syscall.h` başlık dosyasında tanımlanmıştır.

Kalan argümanlar o sistem çağrısına özel argümanlardır. Her çeşit sistem çağrısı kendine özgü sayıda, birden beşe kadar argümana sahiptir. Eğer kodunuz sistem çağrısının aldığından daha fazla argüman içeriyorsa bunlar basitçe yoksayılacaktır.

İşlevin dönüş değeri sistem çağrısı başarısız olmadıkça, sistem çağrısının dönüş değeri olacaktır. Sistem çağrısı başarısız olursa işlev **`-1`** ile döner ve **`errno`** değişkenine sistem çağrısından dönen hata kodu atanır. Sistem çağrıları başarılı olduklarında **`-1`** döndürmezler.

Geçersiz bir *sysno* belirtirseniz, **syscall** işlevi **-1** ile döner ve **errno**= **ENOSYS** olur.

Örnek:

```
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

...

int rc;

rc = syscall(SYS_chmod, "/etc/passwd", 0444);

if (rc == -1)
   fprintf(stderr, "chmod, errno = %d ile başarısız oldu\n", errno);
```

Uyumluluk bakımından yıldızları barışmışsa, bu kod şu koda eşdeğerdir:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

...

int rc;

rc = chmod("/etc/passwd", 0444);
if (rc == -1)
   fprintf(stderr, "chmod, errno = %d ile başarısız oldu\n", errno);
```

6. Yazılımın Sonlandırılması

Bir yazılımı sonlandırmanın uygun yolu **main** işlevinden dönmektir. **main** işlevinden dönen ***çıkış durum değeri*** sürecin üst sürecine ya da kabuğa bilgi vermek için kullanılır.

Bir yazılım bundan başka **exit** işlevini çağırarak da kendini sonlandırabilir.

Bunlara ek olarak yazılımlar sinyallerle sonlandırılabilir; bu ayrıntılı olarak *Sinyal İşleme* (sayfa: 601) bölümünde anlatılmıştır. **abort** işlevi bir yazılımı öldüren bir sinyale sebep olur.

6.1. Normal Sonlandırma

Bir süreç, kendi yazılımı tarafından yapılan bir **exit** çağrısıyla normal olarak sonlanır. **main** işlevinden dönmek ile **exit** çağrısı eşdeğerdir, **main** işlevinin return deyiminde kullanılan değer, **exit** işlevinde argüman olur.

```
void exit(int durum)                                                    işlev
```

exit işlevi sisteme yazılımın işinin bittiğini söyleyen ve sürecin sonlanmasına sebep olan işlevdir.

durum yazılımın çıkış durumudur ve sürecin sonlandırma durumu haline gelir. Bu işlev dönmez.

Normal sonlandırma şu eylemlere yolaçar:

1. **atexit** veya **on_exit** işlevi ile kaydedilen işlevler, kaydedildikleri sıranın tersine bir sıralamayla çağrılırlar. Bu mekanizma sonlanma sırasında bazı temizlik işlemleri (örn, yazılımın sonlanma durum bilgisinin bir dosyaya kaydedlimesi, veritabanlarından kilitlerin kaldırılması gibi) yapabilmenizi sağlar.
2. Tüm açık akımlar, tamponlanmış verileri yazılarak kapatılır. Bkz. *Akımların Kapatılması* (sayfa: 241). Ek olarak, **tmpfile** işlevi ile açılmış geçici dosyalar silinir; bkz. *Geçici Dosyalar* (sayfa: 389).

3. **_exit** çağrılarak yazılım sonlandırılır. Bkz. *Sonlandırmanın İçyapısı* (sayfa: 684).

6.2. Çıkış Durumu

Bir yazılım çıkarken, ***çıkış durumu***nu kullanarak, sürecini başlatan sürece sonlanmasının sebebi ile ilgili küçük bir bilgi verir. Bu değer 0 ile 255 arasındadır ve yazılım tarafından **exit** işlevinde argüman olarak belirtilerek sürece aktarılır.

Normalde, başarı ya da başarısızlık hakkında bilgi vermek için çıkış durumunu kullanmalısınız. Başarısızlık durumunda sebebi için daha fazla bilgi sağlayamazsınız ve zaten çoğu üst süreç de çok fazla ayrıntı istemez.

Yazılımların dönüş değerleri ile ilgili bazı uzlaşımlar vardır. En bilinen uzlaşım başarı durumunda 0, başarısızlık durumunda 1 döndürmektir. Karşılaştırma işlemleri yapan uygulamalar biraz daha farklı bir uzlaşım kullanır: eşleşmeme durumunda 1, karşılaştırmanın yapılamaması durumunda 2 döndürürler. Sizin yazılımınızın da çıkış durumu uzlaşımlarına uygun davranması için mevcut uzlaşımlardan size uygun olanını kullanmalısınız.

Genel bir uzlaşımda, çıkış durumu 128 özel amaçlar için ayrılır. Kısmen, 128 değeri bir alt süreç olarak çalıştırılan başka bir yazılımın çalıştırılmasında başarısızlığı gösterir. Bu uzlaşım evrensel değildir ama yazılımlarınızda buna uyarsanız iyi olur.

Uyarı

Hata sayısını çıkış durumu olarak kullanmayın. Bu aslında hiç de kullanışlı değildir; bir üst süreç genelde kaç tane hata oluştuğu ile ilgilenmez. Dahası, bu çoğunlukla çalışmaz, çünkü çıkış değeri sekiz bitle sınırlıdır. Bu nedenle eğer yazılımınız 256 hata raporlamak isterse üst süreç 0 hatalık bir rapor alacaktır, normal olarak da bu, bir başarı göstergesidir.

Aynı sebeple **errno** değeri de çıkış durumu olarak kullanılmamalıdır, çünkü hata kodları 255'i aşar.

Taşınabilirlik Bilgisi

POSIX olmayan bazı sistemlerde çıkış durumu değerleri için farklı uzlaşımlar kullanılır. Daha yüksek taşınabilirlik açısından başarı ve başarısızlık durumlarının uzlaşımsal durum değerleri olarak **EXIT_SUCCESS** ve **EXIT_FAILURE** makrolarını kullanabilirsiniz. Bu makrolar `stdlib.h` başlık dosyasında bildirilmiştir.

`int` **EXIT_SUCCESS** makro

Bu makro **exit** işlevinde yazılımın başarıyla tamamlandığını belirtmek için kullanılabilir.

POSIX sistemlerinde, bu makronun değeri **0**'dır. Başka sistemlerde (sabit olmaması olası) bir tamsayı ifadesi olabilir.

`int` **EXIT_FAILURE** makro

Bu makro **exit** işlevinde yazılımın başarısızlıkla sonlandığını belirtmek için kullanılabilir.

POSIX sistemlerinde, bu makronun değeri **1**'dir. Başka sistemlerde (sabit olmaması olası) bir tamsayı ifadesi olabilir. Sıfırdan farklı diğer değerler de ayrıca birer başarısızlık gösterirler. Belli başlı bazı uygulamalar başarısısızlık çeşidini belirten farklı çıkış durum değerleri kullanırlar. Örneğin **diff** dosyaların farklı olduğunu **1** ile, dosyaların açılışındaki zorlukları ise **2** ve üstü değerlerle ifade eder.

Bir yazılımın çıkış durumu ile bir sürecin sonlanma durumunu birbirine karıştırmayın. Bir sürecin yazılımının bitişinin yanında sonlanmasının bir çok sebebi olabilir. Sürecin sonlanması sırasında sebep yazılımının sonlanması (yani **exit**) ise, yazılımın çıkış durumu, sürecin sonlanma durumunun bir parçası haline gelir.

6.3. Çıkışta Temizlik

Yazılımınız normal sonlandırma sırasında kendi temizlik işlevlerini çalıştıracak düzenlemeyi yapabilir. Çeşitli uygulama yazılımlarında kullanılan bir kütüphane yazıyorsanız, tüm uygulamaların çıkış sırasında kütüphanenin temizlik işlevlerini çağırmasında ısrar etmek güvenilir olmazdı. Temizlik işlevlerinin uygulamaya görünmez yapmanın kesin yolu **atexit** veya **on_exit** işlevlerini kullanarak bir temizlik işlevini belirtmektir.

```
int atexit(void (*işlev) (void))                                    işlev
```

atexit işlevi *işlev* işlevini normal yazılım sonlanması sırasında çağrılmak üzere kaydeder. *işlev* argümansız çağrılır.

atexit işlevinin normal dönüş değeri sıfırdır, eğer işlev kaydedilemezse sıfırdan farklı bir değerle döner.

```
int on_exit(void (*işlev)(int durum, void  *arg),                   işlev
            void  *arg)
```

Bu işlev **atexit** işlevinin biraz daha güçlü bir sürümüdür. İki argüman kabul eder: *işlev* işlevi ve bir argümana gösterici. Normal yazılım sonlanması sırasında işlev iki argümanlar çağrılır: **exit** işlevine aktarılan *durum* ve bir *arg* argümanı.

Bu işlev GNU kütüphanesine sadece SunOS uyumluluğu için dahil edilmiştir. Diğer gerçeklemeler tarafından desteklenmeyebilir.

Burada, **exit** ve **atexit** işlevlerinin kullanımını gösteren göstermelik bir yazılıma yer verilmiştir:

```
#include <stdio.h>
#include <stdlib.h>

void
elveda (void)
{
  puts ("Elveda, zalim Dünya....");
}

int
main (void)
{
  atexit (elveda);
  exit (EXIT_SUCCESS);
}
```

Bu yazılım çalıştırıldığında bir ileti basar ve çıkar.

6.4. Anormal Sonlanma

abort işlevini kullanarak normal olmayan durumlarda yazılımınızdan çıkabilirsiniz. Bu işlevin prototipi `stdlib.h` başlık dosyasında bulunur.

```
void abort(void)                                                    işlev
```

abort işlevi anormal yazılım sonlanmasına sebep olur. Bu işlev, **atexit** veya **on_exit** işlevi gibi temizlik işlevleri kaydetmez.

Bu işlev aslında bir **SIGABRT** sinyali göndererek süreci sonlandırır. Yazılımınız da bu sinyal için bir eylemci içerebilir; bkz. *Sinyal İşleme* (sayfa: 601).

Gelecekte Değişiklik Uyarısı

Federal sansür düzenlemelerinin isteği ile bu işlevin çağrılma olasılığı hakkında size bilgi vermek bize yasaklanabilir. Bir yazılımın bu yolla sonlanmasının kabul edilebilir bir yöntem olmadığını söylemek ihtiyacı duyabilirdik.

6.5. Sonlandırmanın İçyapısı

Sürecin **exit** ile sonlanmasında **_exit** ilkeli kullanılır. İşlev `unistd.h` başlık dosyasında bildirilmiştir.

`void` **_exit**`(int` *durum*`)` işlev

_exit işlevi bir sürecin *durum* durumu ile sonlanmasını sağlayan ilkel işlevdir. Bu işlev yapılan bir çağrı **atexit** veya **on_exit** işlevi ile kaydedilen temizlik işlevlerini çalıştırmaz.

`void` **_Exit**`(int` *durum*`)` işlev

_Exit işlevi **_exit** işlevinin ISO C eşdeğeridir. ISO C komitesinin üyeleri **_exit** ve **_Exit** tanımlarının uyumluluğundan emin olamadıklarından işlevin POSIX ismini kullanmamışlardır.

Bu işlev ISO C99 standardının parçasıdır ve `stdlib.h` başlık dosyasında bildirilmiştir.

Yazılımın sonlanması ya da bir sinyalin sonucu gibi bir sebeple bir süreç sonlanırken şunlar olur:

- Süreçteki tüm açık dosya tanıtıcılar kapatılır. Bkz. *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305). Süreç sonlanırken akımların özdevinimli olarak boşaltılmayacağını unutmayın; bkz. *Akımlar Üzerinde Giriş/Çıkış* (sayfa: 236).
- Bir sürecin çıkış durumu **wait** veya **waitpid** üzerinden üst sürece raporlanmak üzere kaydedilir; bkz. *Süreç Tamamlama* (sayfa: 690). Yazılım çıkmışsa bu değer onun sekiz bitlik yazılım çıkış durumunu içerir.
- Sürecin sonlanmış olan herhangi bir alt süreci yeni bir üst sürece atanır. (GNU dahil çoğu sistemde bu, süreç kimliği 1 olan **init** sürecidir.)
- Bir **SIGCHLD** sinyali üst sürece gönderilir.
- Eğer süreç, bir denetim uçbirimi olan bir oturum lideri ise, onun önalan işindeki her sürece bir **SIGHUP** sinyali gönderir ve denetim uçbiriminin bu oturumla ilişkisi kesilir. Bkz. *İş Denetimi* (sayfa: 716).
- Bir sürecin sonlanması bir süreç grubunun öksüz kalmasına sebep olmuşsa ve bu süreç grubundaki bir üye durmuşsa bu üyeye bir **SIGCONT** sinyali ve gruptaki her üyeye bir **SIGHUP** sinyali gönderilir. Bkz. *İş Denetimi* (sayfa: 716).

XXVI. Süreçler

İçindekiler

Süreçler sistem kaynaklarının ayrılmasını sağlayan temel birimlerdir. Her süreç kendi adres alanına ve (genellikle) bir kontrol evresine sahiptir. Bir süreç bir yazılımı çalıştırır; aynı yazılımı çalıştıran farklı süreçleriniz olabilir, fakat her süreç kendi adres alanında yazılımın kendi kopyasına sahiptir ve bunu diğer kopyalardan bağımsız olarak çalıştırır.

Süreçler hiyerarşik olarak düzenlenmiştir. Her süreç, kendisini yaratan bir ***üst süreç***e sahiptir. Üst bir süreç tarafından yaratılan süreçler ***alt süreçler*** olarak anılırlar. Bir alt süreç bir çok özelliğini üst sürecinden alır.

Bu kısım bir yazılımın alt süreçleri nasıl yaratabileceğini, sonlandırabileceğini ve kontrol edebileceğini anlatır. Aslında, üç farklı işlem içerilmiştir: yeni bir alt sürecin oluşturulması, yeni sürecin bir yazılımı çalıştırmasına neden olmak ve alt sürecin tamamlanmasını ana yazılımla eşgüdümlemek.

system işlevi başka bir yazılımın çalıştırılması için basit, taşınabilir bir mekanizma sunmaktadır; üç adımı da özdevinimli olarak yapar. Bunun yapılışı ile ilgili ayrıntılar üzerinde daha fazla denetim sahibi olmak istiyorsanız, temel işlevleri kullanarak her adımı tek tek gerçekleştirebilirsiniz.

1. Bir Komutun Çalıştırması

Başka bir yazılımı çalıştırmanın kolay yolu **system** işlevinin kullanılmasıdır. Bu işlev bir alt yazılımı çalıştırmak için gerekli bütün işi yapar, fakat ayrıntılar üzerindeki denetimi size fazla vermez: başka bir şey yapmadan önce alt yazılım sonlanıncaya kadar beklemeniz gerekir.

```
int system(const char *komut)                                    işlev
```

Bu işlev *komut* komutunu bir kabuk komutu olarak çalıştırır. GNU C kütüphanesinde, bir komutu çalıştırmak için her zaman öntanımlı kabuk olan **sh** kullanılır. Özellikle **PATH** içinde belirtilen dizinleri arayarak çalıştırılacak yazılımı bulmaya çalışır. Kabuk sürecini oluşturamadıysa dönüş değeri **-1**, aksi takdirde kabuk sürecinin durumudur. Bu durum kodunun nasıl yorumlanacağı konusunda ayrıntılı bilgi için bkz. *Süreç Tamamlama* (sayfa: 690).

Eğer *komut* argümanı boş gösterici olarak verilirse, dönüş değerinin sıfır olması bir komut işlemcisi olmadığını belirtir.

Bu işlev çok evreli yazılımlar için iptal noktasıdır. Eğer **system** çağrıldığında bir evre bazı kaynakları (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) ayırırsa bu sorun olur. Evre iptal edilirse bu kaynaklar yazılım sonlanıncaya kadar ayrılmış durumda kalırlar. Bundan kaçınmak için **system**e yapılan çağrılar iptal işleyicilerini kullanarak korunmalıdır.

system işlevi `stdlib.h` başlık dosyası içinde tanımlıdır.

Uyumluluk Bilgisi

Bazı C gerçeklemeleri başka yazılımları çalıştıran bir komut işlemcisine sahip olmayabilir. Komut işlemcisinin olup olmadığını **system (NULL)** çalıştırarak anlayabilirsiniz; eğer dönüş değeri sıfır değilse, bir komut işlemcisi vardır.

popen ve **pclose** işlevleri (*Bir Alt Sürece Boru Hattı* (sayfa: 395)) **system** işleviyle yakından ilgilidir. Bunlar üst sürecin, çalıştırılan komutun standart girdi ve çıktı kanallarıyla haberleşmesini sağlarlar.

2. Süreç Oluşturma Kavramları

Bu bölüm süreçlere genel bakış, adım adım süreç oluşturma ve süreçlerin başka bir yazılımı çalıştırmasını içermektedir.

Her süreç bir ***süreç kimliği*** numarasıyla adlandırılır. Süreçler oluşturulurken her birine tek bir süreç kimliği ayrılır. Bir sürecin ***ömrü*** üst sürecine sonlandırıldığı bildirildiğinde biter; o zaman, süreç kimliği dahil bütün süreç kaynakları serbest bırakılır.

Süreçler **fork** sistem çağrısı ile oluşturulurlar (bu nedenle yeni süreç oluşturma bazen süreci ***çatallamak*** olarak anılır.) **fork** ile yaratılan bir***alt süreç*** orjinal ***üst süreç***in bir kopyasıdır, sadece kendisine ait süreç kimliği farklıdır.

Bir alt süreci çatalladıktan sonra, üst ve alt süreçler normal çalışmalarına devam ederler. Eğer yazılımınızın devam etmeden önce alt süreçleri çalışmalarını bitirinceye kadar beklemesini istiyorsanız, bunu çatallanma işleminden hemen sonra **wait** veya **waitpid** işlevlerini çağırarak açıkça yapmanız gerekir(*Süreç Tamamlama* (sayfa: 690)). Bu işlevler alt sürecin neden sonlandırıldığı hakkında sınırlı bilgi verirler—örneğin, çıkış durum kodu gibi.

Yeni çatallanan bir alt süreç aynı yazılımı, **fork** çağrısının döndüğü noktada, üst süreci olarak çalıştırmaya devam eder. **fork** işlevinin dönüş değerini yazılımın üst süreçte mi yoksa alt süreçte mi çalıştığını söylemek için kullanabilirsiniz.

Aynı yazılımı çalıştıran çeşitli süreçlerin olması ara sıra kullanışlıdır. Fakat alt süreç **exec** işlevlerinden birini kullanarak bir başka yazılımı da çalıştırabilir; bkz. *Bir Dosyanın Çalıştırılması* (sayfa: 688). Sürecin çalıştırdığı yazılıma ***süreç görüntüsü*** denir. Yeni yazılımın çalıştırılmasının başlatılması, sürecin, önceki süreç görüntüsü hakkındaki herşeyi unutmasına sebep olur; yeni yazılım sonlandığında, önceki süreç görüntüsüne dönülmez, süreç de sonlanır.

3. Süreç Kimliği

pid_t veri türü süreç kimlikleri için kullanılır. Bir sürecin süreç kimliğini **getpid** işlevini çağırarak alabilirsiniz. **getppid** işlevi geçerli sürecin üst sürecinin süreç kimliğini döndürür (aynı zamanda ***üst süreç kimliği*** olarak da bilinir). Bu işlevleri kullanmak için yazılımınız `unistd.h` ve `sys/types.h` başlık dosyalarını içermelidir.

pid_t veri türü

pid_t veri türü süreç kimliğini gösterebilen bir işaretli tamsayıdır. GNU kütüphanesinde, bu bir **int**'tir.

`pid_t` **getpid**`(void)` işlev

getpid işlevi geçerli sürecin süreç kimliğini döndürür.

`pid_t` **getppid**`(void)` işlev

getppid işlevi geçerli sürecin üst süreç kimliğini döndürür.

4. Bir Sürecin Oluşturulması

fork işlevi süreç oluşturmanın temelidir ve `unistd.h` başlık dosyası içinde bildirilmiştir.

`pid_t` **fork**`(void)` işlev

fork işlevi yeni bir süreç oluşturur.

Süreç oluşturma başarılıysa, hem üst hem de alt süreçler çalışır ve her ikisi de **fork** işlevinin dönüş değerini görür, ancak bu değerler farklıdır: **fork** işlevi, alt süreçte **0** değerini ve üst süreçte alt sürecinin süreç kimliğini döndürür.

Süreç oluşturma başarısızsa, **fork** işlevi üst süreçte **-1** değerini döndürür. Aşağıdaki **errno** hata durumları **fork** işlevi için tanımlanmıştır:

`EAGAIN`
: Başka süreç oluşturmak için yeterli sistem kaynağı yok ya da kullanıcının zaten çok fazla süreci çalışmakta. Bu **RLIMIT_NPROC** kaynak sınırının aşılmakta olduğu anlamına gelir, bu genellikle artırılabilir; bkz. *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

`ENOMEM`
: Süreç sistemin sağlayabileceğinden fazla yere ihtiyaç duymaktadır.

Alt süreci üstünden farklılaştıran özellikleri:

- Alt sürecin kendi süreç kimliği vardır.
- Alt sürecin üst süreç kimliği üst sürecinin süreç kimliğidir.
- Alt süreç, üst sürecin açık dosya tanımlayıcılarının kendine ait kopyalarını alır. Böylece üst süreçteki dosya tanımlayıcısının özelliklerinin değiştirilmesi alttaki dosya tanımlayıcıları etkilemez, bu tersi için de geçerlidir. Bkz. *Dosyalar Üzerindeki Denetim İşlemleri* (sayfa: 338). Ancak, her tanımlayıcıyla ilişkilendirilmiş olan dosya konumu her iki süreç tarafından paylaşılır; bkz. *Dosyada Konumlama* (sayfa: 232).
- Alt süreçler için biten işlemci süreleri sıfırlanır; bkz. *İşlemci Süresinin Sorgulanması* (sayfa: 541).
- Alt süreç üstü tarafından kurulmuş dosya kilitlerini miras almaz.*Dosyalar Üzerindeki Denetim İşlemleri* (sayfa: 338).
- Alt süreç üstü tarafından kurulmuş uyarıları miras almaz.*Bir Alarmın Ayarlanması* (sayfa: 568).
- Alt süreç için bekleyen sinyal kümesi (*Sinyallerin Gönderilmesi* (sayfa: 603)) temizlenir. (Alt süreç baskılanmış sinyallerin maskesini ve sinyal hareketlerini üst sürecinden miras alır.)

```
pid_t vfork(void)                                                    işlev
```

vfork işlevi **fork** gibidir fakat bazı sistemlerde daha verimlidir; fakat, güvenli kullanımı için bazı kısıtlar vardır.

forkçağıran sürecin adres alanının tam bir kopyasını alıp hem alt hem de üst sürecin bağımsız olarak çalışmasını sağlarken, **vfork** bu kopyayı yapmaz. Bunun yerine **vfork** ile oluşturulan alt süreç **_exit** veya **exec** işlevleri çağrılana kadar üst sürecin adres alanını paylaşır. Bu arada, üst süreç çalışmayı askıya alır.

vfork ile oluşturulan alt sürecin evrensel (global) verileri, hatta üstüyle paylaştığı yerel değişkenleri bile değiştirmesine izin vermemek konusunda dikkatli olmalısınız. Bundan başka, alt süreç **vfork**'u çağıran işlevden dönemez! Bu üst sürecin kontrol bilgisini karıştırabilir. Kuşkunuz varsa, **fork** kullanın.

Bazı işletim sistemleri gerçek anlamda **vfork**'u uygulamazlar. GNU C kütüphanesi **vfork**'u bütün sistemlerde kullanmanıza izin verir, ancak aslında **vfork** yoksa **fork** çalıştırır. Eğer **vfork** kullanımında önlemlerinizi alırsanız, yazılımınız, sistem onun yerine **fork**'u kullansa da çalışır.

5. Bir Dosyanın Çalıştırılması

Bu bölüm bir dosyayı bir süreç görüntüsü olarak çalıştırmak için kullanılan **exec** ailesi işlevlerini anlatmaktadır. Bu işlevler bir alt sürecin çatallandıktan sonra yeni bir yazılımı çalıştırmasını sağlamak için kullanılabilir.

exec işlevinin etkilerini çağrılan yazılımın bakış açısıyla görmek için, bkz. *Temel Yazılım ve Sistem Arayüzü* (sayfa: 644).

Bu ailedeki işlevler argümanlarının belirtiliş şekillerine göre farklılıklar gösterir, aksi takdirde hepsi aynı işi yapar. Bunlar `unistd.h` başlık dosyası içinde bildirimlidir.

```
int execv(const char *dosyaismi,                                     işlev
          char *const argv[])
```

execvişlevi *dosyaismi* adındaki dosyayı yeni bir süreç görüntüsü olarak çalıştırır.

argv argümanı boş karakter sonlandırmalı dizgelerden oluşan bir dizidir ve bu çalıştırılan yazılımın **main** işlevinin **argv** argümanına değer sağlamak için kullanılır. Bu dizinin son elemanı bir boş gösterici olmalıdır. Kural olarak, bu dizinin ilk elemanı yazılımın dosya ismidir (dizinsiz hali). Yazılımların bu argümanlara nasıl eriştiğini bütün ayrıntılarıyla incelemek için, bkz. *Yazılım Argümanları* (sayfa: 645),

Yeni süreç görüntüsünün ortamı geçerli süreç görüntüsünün **environ** değişkeninden alınır; ortam değişkenleri hakkında bilgi için *Ortam Değişkenleri* (sayfa: 676) bölümüne bakınız.

```
int execl(const char *dosyaismi,                                     işlev
          const char *arg0,
          ...)
```

Bu da **execv** gibidir, fakat *argv* dizgeleri bir dizi yerine tek tek belirtilir. Son argüman bir boş gösterici olmalıdır.

```
int execve(const char *dosyaismi,                                    işlev
           char *const argv[],
           char *const ortam[])
```

Bu da **execv** gibidir, fakat yeni yazılım için ortamı açıkça belirtmenize *ortam* argümanıyla izin verir. Bu **environ** değişkeniyle aynı biçimde, dizgelerden oluşan bir dizi olmalıdır; bkz. *Ortama Erişim* (sayfa: 677).

```
int execle(const char *dosyaismi,                                    işlev
           const char *arg0,
           char *const ortam[],
           …)
```

Bu da **execl** gibidir, fakat yeni yazılım için ortamı açıkça belirtmenize izin verir. *ortam* argümanı son *argv* argümanını olan boş göstericiden sonra gelmeli ve **environ** değişkeniyle aynı biçimde, dizgelerden oluşan bir dizi olmalıdır.

```
int execvp(const char *dosyaismi,                                    işlev
           char *const argv[])
```

execvp işlevi **execv** gibidir, ancak eğer *dosyaismi* bir **/** içermiyorsa *dosyaismi* isimli dosyanın tam ismini bulmak için **PATH** ortam değişkeninde (*Standart Ortam Değişkenleri* (sayfa: 678)) listelenen dizinleri de arar.

Bu işlev sisteme yardımcı yazılımların çalıştırılmasında kullanışlı olabilir, çünkü bu yazılımları bulmak için kullanıcının seçtiği yerlere bakar. Kabuklar bunu kullanıcının yazdığı komutları çalıştırmak için kullanırlar.

```
int execlp(const char *dosyaismi,                                    işlev
           const char *arg0,
           …)
```

Bu işlev **execl** gibidir, ancak **execvp** işlevi gibi dosya ismi arama işlemi uygular.

Argüman listesinin ve ortam değişkenleri listesinin toplam boyutu **ARG_MAX** bayttan büyük olmamalıdır. Bkz. *Genel Sınırlar* (sayfa: 784). GNU sisteminde, her dizge için bu boyut (**ARG_MAX** ile karşılaştırıldığında), "dizge içerisindeki karakter sayısı, artı **char *** türünün boyutu, artı bir" değerinin **char *** boyutunun katlarına yuvarlanması ile elde edilir. Diğer sistemler biraz farklı sayım kuralları uygulayabilir.

Bu işlevler normalde değer döndürmezler, çünkü yeni bir yazılımın çalıştırılması halen çalışan yazılımın tamamen terk edilmesine neden olur. Hata durumunda **-1** döndürülür. Olağan *dosya ismi hatalarına* (sayfa: 234) ek olarak, aşağıdaki **errno** hata durumları bu işlevler için tanımlanmıştır:

E2BIG
: Yeni yazılımın argüman listesinin uzunluğuyla ortam değişkenleri listesinin birleştirilmiş büyüklüğü **ARG_MAX** bayttan büyüktür. GNU sisteminde argüman listesi büyüklüğü için belirtilmiş bir sınır yoktur, böylece bu hata kodu oluşamaz, ancak eğer argümanlar kullanılabilir bellek için çok büyük ise bunun yerine **ENOMEM** alabilirsiniz.

ENOEXEC
: Belirtilen dosya doğru biçimde olmadığı için çalıştırılamamaktadır.

ENOMEM
: Belirtilen dosyanın çalıştırılması mevcut olandan daha fazla depolama alanı gerektirmektedir.

Yeni dosyanın çalıştırılması başarılı olursa, okumada olduğu gibi dosyanın erişim zamanı alanını günceller. Dosyaların erişim zamanları hakkında daha fazla bilgi için *Dosya Zamanları* (sayfa: 383) bölümüne bakınız.

Dosyanın tekrar kapandığı nokta belirtilmemiştir, ancak süreçten çıkmadan önceki bir nokta veya başka bir süreç görüntüsü çalıştırılmadan önceki nokta belirtilmiş olabilir.

Yeni bir süreç görüntüsünün çalıştırılması, sadece argüman ve ortam dizgelerini yeni yerlerine kopyalayarak bellek içeriğini tamamen değiştirir. Fakat sürecin diğer bir çok özelliği değişmez:

- Süreç kimliği ve üst süreç kimliği. Bkz. *Süreç Oluşturma Kavramları* (sayfa: 686).
- Oturum ve süreç grubu üyeliği. Bkz. *İş Denetimi Kavramları* (sayfa: 716).

- Gerçek kullanıcı kimliği ve grup kimliği ile ek grup kimlikleri. Bkz. *Bir Sürecin Aidiyeti* (sayfa: 743).
- Bekleyen uyarılar. Bkz. *Bir Alarmın Ayarlanması* (sayfa: 568).
- Geçerli çalışma dizini ve kök dizini. Bkz. *Çalışma dizini* (sayfa: 351). GNU sisteminde, kök dizini bir setuid yazılımı çalıştırılırken kopyalanmaz; bunun yerine yeni yazılım için sistemin öntanımlı kök dizini kullanılır.
- Dosya kipi oluşturma maskesi. Bkz. *Dosya İzinlerinin Atanması* (sayfa: 380).
- Süreç sinyal maskesi; bkz.*Sürecin Sinyal Maskesi* (sayfa: 633).
- Bekleyen sinyaller; bkz. *Sinyallerin Engellenmesi* (sayfa: 631).
- Sürece ilişkin biten işlemci süresi; bkz. *İşlemci Süresinin Sorgulanması* (sayfa: 541).

Eğer süreç görüntü dosyasının set–user–ID ve set–group–ID kip bitleri işaretlenmişse, bu sürecin etkin kullanıcı kimliği ve grup kimliği (sırasıyla) değerlerini etkiler. Bu kavramlara ayrıntılı bir şekilde *Bir Sürecin Aidiyeti* (sayfa: 743) içinde değinilmiştir.

Geçerli süreçte görmezden gelinecek sinyaller yeni süreçte de dikkate alınmayacak şekilde ayarlanmalıdır. Diğer bütün sinyaller yeni süreç görüntüsünde öntanımlı hareketlerine ayarlanırlar. Sinyaller hakkında daha fazla bilgi için, bkz. *Sinyal İşleme* (sayfa: 601).

Varolan süreç görüntüsünde açık kalan dosya tanımlayıcıları yeni süreç görüntüsünde de **FD_CLOEXEC** (close–on–exec) bayrak kümesine sahip değillerse açık kalırlar. Açık kalan dosyalar varolan süreç görüntüsünden açık dosya tanımlayıcısının dosya kilitleri dahil bütün özelliklerini miras alırlar. dosya tanımlayıcıları *Düşük Seviyeli Girdi ve Çıktı* (sayfa: 305) içinde incelenmiştir.

Akımlar, buna karşın, **exec** işlevleriyle varlıklarını sürdüremezler, çünkü bunlar sürecin kendi bellek alanına yerleşmektedirler. Yeni süreç görüntüsünün kendisinin yeniden oluşturdukları dışında akımları yoktur. **exec** öncesi süreç görüntüsünün içindeki akımların her birinin içinde bir tanımlayıcı vardır ve bunlar **exec** ile varlıklarını sürdürürler (**FD_CLOEXEC** kümesinin olmaması halinde). Yeni süreç görüntüsü bu yeni akımlara **fdopen** kullanarak tekrar bağlanabilir (bkz. *Tanıtıcılar ve Akımlar* (sayfa: 315)).

6. Süreç Tamamlama

Bu bölümde anlatılan işlevler bir alt sürecin sonlanmasını veya durmasını beklemek ve durumunu algılamak için kullanılırlar. Bu işlevler `sys/wait.h` başlık dosyası içinde bildirimlidir.

```
pid_t waitpid(pid_t pid,                                          işlev
              int  *durum–gstr,
              int   seçenekler)
```

waitpid işlevi, süreç kimliği *pid* olan alt sürecin durum bilgisini istemek için kullanılır. Normalde, çağıran süreç, alt süreç sonlanarak durum bilgisini verene kadar askıda kalır.

pid argümanı için verilecek diğer değerler farklı yorumlanır. **-1** veya **WAIT_ANY** değeri her hangi bir alt süreç için durum bilgisi ister; **0** veya **WAIT_MYPGRP** değeri çağıran süreçle aynı süreç grubundaki her hangi bir alt süreç için bilgi ister; ve *–sgkim* gibi başka herhangi bir negatif değer, süreç grup kimliği *sgkim* olan herhangi bir alt süreç için bilgi ister.

Eğer alt süreç durum bilgisi hemen mevcutsa, bu işlev beklemeden hemen döner. Eğer birden fazla seçilebilir alt sürecin durum bilgisi mevcutsa, rastgele birisi seçilir ve durum bilgisi hemen döndürülür. Diğer seçilebilir alt süreçlerden birinin durum bilgisini almak için, **waitpid** işlevini tekrar çağırmanız gerekir.

seçenekler argümanı bir bit maskesidir. Değeri sıfır veya daha çok sayıda **WNOHANG** ve **WUNTRACED** bayrağının bit bit VEYAlanmış (**|** işleci) hali olmalıdır. **WNOHANG** bayrağı üst sürecin beklememesi gerektiğini belirtmek için; ve **WUNTRACED** bayrağı duran ve hatta sonlanan süreçlerden durum bilgisi istemek için kullanılır.

Alt süreçten alınan durum bilgisi, *durum–gstr* boş gösterici değilse, *durum–gstr* 'nin gösterdiği nesnede saklanır.

Bu işlev çok–evreli yazılımlar için iptal noktasıdır. Eğer **waitpid** çağrıldığında bir evre bazı kaynakları (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) ayırırsa bu sorun olur. Evre iptal edilirse bu kaynaklar yazılım sonlanıncaya kadar ayrılmış durumda kalırlar. Bundan kaçınmak için **waitpid**'e yapılan çağrılar iptal işleyicileri kullanılarak korunmalıdır.

Dönüş değeri normalde durumu bildirilen alt sürecin süreç kimliğidir. Eğer alt süreçler var ancak hiçbiri uyarım için beklemiyorsa, **waitpid** birisi uyarılana kadar baskılanır. Fakat, **WNOHANG** seçeneği belirtildiyse, **waitpid** baskılanmadan sıfır döndürür.

Eğer **waitpid** belirli bir süreç kimliği için bekleyecekse, diğer hiçbir alt süreci (varsa) dikkate almaz. Bu nedenle eğer uyarım için bekleyen alt süreçler varsa, fakat beklenmesi için belirtilen süreç kimliğine sahip alt süreç bunlardan biri değilse, **waitpid** yukarıda açıklandığı gibi ya baskılanır ya da sıfır döndürür.

-1 değeri hata halinde döndürülür. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EINTR
: Çağıran sürece bir sinyal gelmesi nedeniyle işlev kesintiye uğradı. Bkz *Sinyallerle Kesilen İlkeller* (sayfa: 626).

ECHILD
: Bekleyen alt süreç yok veya belirtilen *pid* çağıran sürecin bir alt süreci değil.

EINVAL
: *seçenekler* argümanı için geçersiz bir değer verildi.

Bu sembolik sabitler **waitpid** işlevinin *pid* argümanının değerleri olarak tanımlanmıştır.

WAIT_ANY
: Bu sabit makro (değeri **-1**dir) **waitpid**'nin herhangi bir alt sürecin durum bilgisini döndürmesi için belirtilirr.

WAIT_MYPGRP
: Bu sabit (**0** değerli) **waitpid**'in çağıran süreçle aynı süreç grubundaki herhangi bir alt sürecin durum bilgisini döndürmesi için belirtilir.

Bu sembolik sabitler **waitpid** işlevinin *seçenekler* argümanı için bayrak olarak tanımlanmıştır. Bayrakları bit bit VEYAlayarak argümana değer olarak kullanabilirsiniz.

WNOHANG
: Bu bayrak, eğer uyarım için bekleyen alt süreç yoksa, **waitpid** işlevinin beklemeden hemen dönmesi gerektiğini belirtir.

WUNTRACED
: Bu bayrak, **waitpid** işlevinin durmuş veya sonlandırılmış alt süreçlerinin durumlarını bildirmesini belirtir.

```
pid_t wait(int  *durum–gstr)                                                    işlev
```

Bu **waitpid**'in basitleştirilmiş halidir ve herhangi bir alt süreç sonlanıncaya kadar beklemek için kullanılır. Aşağıdaki çağrı şekli:

```
wait (status)
```

aşağıdaki ile tamamen aynıdır:

```
waitpid (-1, status, 0)
```

Bu işlev çok–evreli yazılımlar için iptal noktasıdır. Eğer **wait** çağrıldığında bir evre bazı kaynakları (bellek, dosya tanımlayıcısı, semafor veya her hangi başka bir kaynak) ayırırsa bu sorun olur. Evre iptal edilirse bu kaynaklar yazılım sonlanıncaya kadar ayrılmış durumda kalırlar. Bundan kaçınmak için **wait**'e yapılan çağrılar iptal işleyicileri kullanılarak korunmalıdır.

```
pid_t wait4(pid_t           pid,                                                işlev
            int             *durum–gstr,
            int             seçenekler,
            struct rusage   *kullanım)
```

Eğer *kullanım* bir boş gösterici ise, **wait4** işlevi **waitpid (***pid***,** *durum–gstr***,** *seçenekler***)** ile eşdeğerdir.

Eğer *kullanım* bir boş gösterici değilse, **wait4** işlevi ***kullanım** içinde alt sürecin kullanım şekillerini saklar (alt süreç durduysa değil, yalnızca sonlandıysa). Bkz. *Özkaynak Kullanımı* (sayfa: 572).

Bu işlev bir BSD oluşumudur.

Burada sonlandırılmış bütün alt süreçlerin durumunu beklemeden almak için **waitpid** işlevinin kullanımını gösteren bir örnek görüyoruz. Bu işlev, en azından bir alt sürecin sonlandırılması gerektiğini belirten bir sinyal olan, **SIGCHLD** için bir işleyici olarak tasarlanmıştır.

```
void
sigchld_handler (int signum)
{
  int pid, status, serrno;
  serrno = errno;
  while (1)
    {
      pid = waitpid (WAIT_ANY, &status, WNOHANG);
      if (pid < 0)
        {
          perror ("waitpid");
          break;
        }
      if (pid == 0)
        break;
      notice_termination (pid, status);
    }
  errno = serrno;
}
```

7. Süreç Tamamlanma Durumu

Eğer alt sürecin *çıkış durum değeri* (sayfa: 681) sıfırsa, o zaman **waitpid** veya **wait** ile bildirilen durum değeri de sıfırdır. Aşağıdaki makroları kullanarak dönen durum değerlerinde kodlanmış diğer türlerdeki bilgileri sınayabilirsiniz. Bu makrolar `sys/wait.h` başlık dosyası içinde tanımlıdır.

`int` **WIFEXITED**`(int` *durum*`)` makro

Bu makro alt süreç normal olarak **exit** veya **_exit** ile sonlandırıldıysa sıfırdan farklı bir değer döndürür.

`int` **WEXITSTATUS**`(int` *durum*`)` makro

Eğer **WIFEXITED** *durum* için doğruysa, bu makro, alt süreçten *çıkış değerinin* (sayfa: 682) düşük–sıralı 8 bitini döndürür.

`int` **WIFSIGNALED**`(int` *durum*`)` makro

Bu makro alt süreç işlenemeyen bir sinyal aldıysa ve sonlandıysa sıfırdan farklı bir değer döndürür. Bkz. *Sinyal İşleme* (sayfa: 601).

`int` **WTERMSIG**`(int` *durum*`)` makro

Eğer **WIFSIGNALED** *durum* için doğruysa, bu makro alt süreci sonlandıran sinyalin sinyal numarasını döndürür.

`int` **WCOREDUMP**`(int` *durum*`)` makro

Bu makro alt süreç sonlandıysa ve bir çekirdek dökümü ürettiyse sıfırdan farklı bir değer döndürür.

`int` **WIFSTOPPED**`(int` *durum*`)` makro

Bu makro alt süreç durduysa sıfırdan farklı bir değer döndürür.

`int` **WSTOPSIG**`(int` *durum*`)` makro

Eğer **WIFSTOPPED** *durum* için doğruysa, bu makro alt sürecin durmasına neden olan sinyalin sinyal numarasını döndürür.

8. BSD Süreç Bekleme İşlevleri

GNU kütüphanesi aynı zamanda BSD Unix ile uyumluluk sağlayan bu oluşumları da sağlamaktadır. BSD, durum değerlerini göstermek için **int** yerine **union wait** veri türünü kullanmaktadır. İki gösterim bir birinin yerine kullanılabilir; aynı bit kalıplarını açıklamaktadırlar. GNU C Kütüphanesi **WEXITSTATUS** gibi makrolar tanımlar, böylece her iki türdeki projelerde de çalışır, ayrıca **wait** işlevi kendisinin *durum–gstr* argümanına her iki türde gösterici kabul etmek için tanımlanmıştır.

Bu işlevler `sys/wait.h`içinde tanımlıdır.

`union` **wait** veri türü

Bu veri türü yazılım sonlandırma durum değerini gösterir. Aşağıdaki üyelere sahiptir:

`int` **w_termsig**
: Bu üyenin değeri **WTERMSIG** makrosunun değeriyle aynıdır.

`int` **w_coredump**
: Bu üyenin değeri **WCOREDUMP** makrosunun değeriyle aynıdır.

`int` **w_retcode**
: Bu üyenin değeri **WEXITSTATUS** makrosunun değeriyle aynıdır.

`int` **`w_stopsig`**
: Bu üyenin değeri **`WSTOPSIG`** makrosunun değeriyle aynıdır.

Bu üyelere doğrudan erişmektense, eşdeğer makrolarını kullanmalısınız.

`wait3` işlevi daha esnek olan **`wait4`** işlevinin atasıdır. **`wait3`** artık atıldır.

işlev
```
pid_t wait3(union wait     *durum–gstr,
            int             seçenekler,
            struct rusage *kullanım)
```

Eğer *kullanım* bir boş gösterici ise, **`wait3`** işlevi **`waitpid (-1,`** *durum–gstr*`,` *seçenekler*`)`** işlevinin eşdeğeridir.

Eğer *kullanım* boş gösterici değilse, **`wait3`** alt sürecin kullanım şekillerini ***`*`***`kullanım`* içinde saklar (eğer alt süreç sonlandıysa; durduysa değil). *Özkaynak Kullanımı* (sayfa: 572).

9. Süreç Oluşturma Örneği

Burada yerleşik **`system`** işlevine benzer örnek bir yazılım görüyoruz. Kendi *komut* argümanını **`sh -c`** *komut* 'ın eşdeğerini kullanarak çalıştırır.

```
#include stddef.h
#include stdlib.h
#include unistd.h
#include sys/types.h
#include sys/wait.h

/* Komutu bu kabuk yazılımını kullanarak çalıştır.  */
#define SHELL "/bin/sh"

int
my_system (const char *komut)
{
  int status;
  pid_t pid;

  pid = fork ();
  if (pid == 0)
    {
      /* Bu alt süreçtir.  Kabuk komutunu çalıştırır. */
      execl (SHELL, SHELL, "-c", komut, NULL);
      _exit (EXIT_FAILURE);
    }
  else if (pid < 0)
    /* Çatallama başarılamadı.  Başarısızlığı bildir.  */
    status = -1;
  else
    /* Bu üst süreçtir.  Bitirmek için alt süreci bekle.  */
    if (waitpid (pid, &status, 0) != pid)
      status = -1;
  return status;
}
```

Bu örnekte dikkat etmeniz gereken bir kaç nokta var.

Unutmayınız ki yazılıma sağlanan ilk **argv** argümanı çalıştırılan yazılımın adıdır. Bu nedenle, **execl**'ye yapılan çağrıda, **SHELL** bir kere çalıştırılacak yazılım ismini sağlamak için, bir kere de **argv[0]**'a değer sağlamak için kullanılmıştır.

Alt süreçteki **execl** çağrısı başarılıysa değer döndürmez. Başarısız olursa, alt sürecin sonlanması için birşeyler yapmanız gerekir. Sadece **return** ile kötü durum kodu döndürülmesi, orjinal yazılımı çalıştıran iki süreci bırakabilir. Bunun yerine, doğru davranış üst sürece alt süreç ile ilgili başarısızlığın bildirilmesidir.

Bunu başarmak için **_exit** işlevini çağırın. **_exit** işlevini **exit** yerine kullanma nedeni **stdout** gibi tamamen tamponlanmış akımları boşaltmanın önüne geçmektir. Bu akımların tamponları büyük olasılıkla üst süreçten **fork** ile kopyalanmış veri içerir, sonunda bu veri üst süreç tarafından çıktı alınır. Alt süreçte **exit** çağrısı verinin iki kez çıktı vermesine neden olabilir. Bkz *Sonlandırmanın İçyapısı* (sayfa: 684).

10. POSIX Evreleri

Bu bölümde pthreads (POSIX evreleri) kütüphanesi anlatılmaktadır. Bu kütüphane çok–evreli programlar için destek işlevleri sağlamaktadır: evre ilkelleri, eşzamanlama nesneleri, vb. Aynı zamanda POSIX 1003.1b semaforlarını (System V semaforlarıyla karıştırılmamalıdır) gerçekler.

Evre işlemleri (**pthread_***) *errno* kullanmazlar. Bunun yerine hata kodunu doğrudan döndürürler. Semafor işlemleri ise *errno* kullanır.

10.1. Basit Evre İşlemleri

Bu işlevler **fork**, **exit** ve **wait** işlevlerinin evre eşdeğerleridir.

```
int pthread_create(pthread_t       *evre,                                    işlev
                   pthread_attr_t  *öznitelik,
                   void          *(*başlatma_işlevi)(void *),
                   void            *arg)
```

pthread_create çağıran evre ile aynı zamanda çalışan yeni bir kontrol evresi yaratır. Yeni evre ilk argümanına *arg* geçirerek *başlatma_işlevi* işlevini çağırır. Yeni evre **pthread_exit** işlevini çağırarak her ikisini de açıkça sonlandırır, veya *başlatma_işlevi* işlevinden dönerek örtük olarak sonlandırır. İkinci yaklaşım *başlatma_işlevi* işlevinin çıkış kodu olarak dönen sonuçla **pthread_exit** işlevinin çağrılmasına eşdeğerdir.

öznitelik argümanı yeni evreye uygulanacak evre özelliklerini belirler. Ayrıntılar için bkz. *Evre Öznitelikleri* (sayfa: 696). *öznitelik* argümanı **NULL** da olabilir, bu durumda öntanımlı özellikler kullanılır: oluşturulan evre birleşimcidir (ayrık değildir) ve sıradan (gerçek zamanlı değil) bir zamanlama ilkesine sahiptir.

Başarı halinde, yeni oluşturulan evrenin tanıtıcısı *evre* argümanıyla gösterilen yerde saklanır ve bir 0 döndürülür. Hata halinde, sıfırdan farklı bir hata kodu döndürülür.

İşlev aşağıdaki hataları döndürebilir:

EAGAIN
: Yeni evre için süreç oluşturacak yeterli sistem kaynağı yok veya **PTHREAD_THREADS_MAX** den fazla sayıda evre rtkin.

```
void pthread_exit(void *dönüş_değeri)                                        işlev
```

pthread_exit çağrılan evrenin çalıştırılmasını sonlandırır. Çağrılan evre için **pthread_cleanup_push** ile atanmış bütün *temizlik işleyicileri* (sayfa: 700) ters sırayla çalıştırılır (son

eklenen işleyici ilk çalıştırılır). Ardından evreye özgü veriler için kullanılan sonlandırma işlevleri, **NULL** olmayan değere sahip, çağıran evreyle ilişkili bütün anahtarlar için çağrılır (*Evreye Özgü Veri* (sayfa: 709)). Son olarak, çağıran evrenin çalıştırılması durdurulur.

dönüş_değeri argümanı evrenin dönüş değeridir. **pthread_join** kullanarak başka bir evreden elde edilebilir.

pthread_exit işlevi hiç bir zaman dönmez.

```
int pthread_cancel(pthread_t evre)                                    işlev
```

pthread_cancel işlevi *evre* argümanıyla belirtilen evreye bir iptal isteği gönderir. Eğer böyle bir evre yoksa, **pthread_cancel** başarısız olur ve **ESRCH** döndürür. Aksi takdirde 0 döndürür. Ayrıntılar için bkz. *İptaletme* (sayfa: 699).

```
int pthread_join(pthread_t evre,                                      işlev
                 void     **evre_dönüş)
```

pthread_join *evre* ile tanımlanan evre sonlanıncaya kadar çağıran evrenin çalıştırılmasını **pthread_exit** işlevini çağırarak veya iptal edilerek askıya alır.

Eğer *evre_dönüş* **NULL** değilse, *evre*nin dönüş değeri *evre_dönüş* ile gösterilen yerde saklanır. *evre*nin dönüş değeri ya **pthread_exit**e verdiği argümandır ya da eğer *evre* iptal edildiyse **PTHREAD_CANCELED** değeridir.

Birleşmiş evre *evre* birleşebilir durumda olmalıdır: **pthread_detach** ile ayrılmış olmamalıdır veya **PTHREAD_CREATE_DETACHED** özelliği **pthread_create** işlevine verilmemiş olmalıdır.

Birleşebilir bir evre sonlandığında, onun bellek özkaynakları (evre tanımlayıcısı ve yığıt) başka bir evre üzerinde **pthread_join** uygulayıncaya kadar serbest bırakılmazlar. Bu nedenle, bellek kaçağını önlemek için **pthread_join** işlevinin her birleşebilir evre için çağrılması gerekir.

Verilen bir evrenin sonlanması için en çok bir evre bekleyebilir. Üzerinde başka bir evrenin sonlanması için beklemekte olduğu, bir *evre* evresi üzerinde, **pthread_join** çağrısı hata döndürür.

pthread_join bir iptal noktasıdır. **pthread_join**de askıya alınmış bir evre iptal edilirse, evre hemen işletmeyi sürdürür ve *evre* evresinin sonlanması beklenmeden iptal işletilir. Eğer **pthread_join** süresince iptal yaşanırsa, *evre* evresi birleşmemiş kalır.

Başarı halinde, *evre*nin dönüş değeri *evre_dönüş* ile gösterilen yerde saklanır ve 0 döndürülür. Hata halinde, aşağıdaki değerlerden biri döndürülür:

ESRCH
: *evre* ile belirtilene uygun bir evre bulunamadı.

EINVAL
: *evre* evresi ayrılmış veya başka bir evre *evre*nin sonlanmasını beklemektedir.

EDEADLK
: *evre* argümanı çağıran evreyi belirtmektedir.

10.2. Evre Öznitelikleri

Evreler oluşturulmaları sırasında aldıkları bir miktar özelliğe sahip olabilir. Bu, **pthread_attr_t** türündeki bir *öznitelik* evre öznitelik nesnesinin doldurulması, ardından da **pthread_create** işlevine ikinci argüman

olarak aktarılmasıyla olur. **NULL** aktarmak bütün özniteliklerine öntanımlı değerler atanmış bir evre öznitelik nesnesi aktarmakla eşdeğerdir.

Öznitelik nesnelerine sadece yeni bir evre oluşturulacağı zaman başvurulur. Aynı öznitelik nesnesi bir çok evrenin oluşturulmasında kullanılabilir. **pthread_create** çağrıldıktan sonra bir öznitelik nesnesinin değiştirilmesi önceden oluşturulan bir evrenin özniteliklerini değiştirmez.

`int` **pthread_attr_init**`(pthread_attr_t *`*öznitelik*`)` işlev

pthread_attr_init *öznitelik* evre öznitelik nesnesini hazırlar ve özniteliklerini öntanımlı değerlerle doldurur. (Öntanımlı değerler her öznitelik için aşağıda listelenmiştir.)

Her *isim* özniteliği (bütün özniteliklerin listesi için aşağı bakınız) **pthread_attr_set***isim* işlevi ile tek tek belirlenebilir ve **pthread_attr_get***isim* işlevi ile değeri alınabilir.

`int` **pthread_attr_destroy**`(pthread_attr_t *`*öznitelik*`)` işlev

pthread_attr_destroy *öznitelik* ile gösterilen özellik nesnesini ilişkili bütün kaynakları serbest bırakarak yok edebilir. *öznitelik* tanımlanmamış bir durumda bırakılır ve tekrar hazırlanıncaya kadar herhangi bir POSIX evre işlevi ile kullanmamanız gerekir.

`int` **pthread_attr_setattr**`(pthread_attr_t *`*nesne*`,`
`int` *değer*`)` işlev

nesne ile gösterilen öznitelik nesnesinin içindeki *öznitelik* özelliğine *değer* değerini verir. Olası öznitelikler ve alabilecekleri değerler listesi için aşağıya bakınız.

Başarı halinde, bu işlevler 0 döndürür. Eğer *değer* değiştirilen *öznitelik* için anlamlı değilse, **EINVAL** hata kodunu döndürürler. Bazı işlevlerin başka hata kipleri vardır; aşağıya bakınız.

`int` **pthread_attr_getattr**`(pthread_attr_t *`*nesne*`,`
`int` *`değer`*`)` işlev

nesne içindeki *öznitelik* özelliğinin geçerli ayarlarını *değer* ile gösterilen değişken içinde saklar.

Bu işlevler her zaman 0 döndürür.

Aşağıdaki evre öznitelikleri desteklenmektedir:

`detachstate`
Evrenin birleşimci bir durumda mı (**PTHREAD_CREATE_JOINABLE** değeri) yoksa ayrık durumda mı (**PTHREAD_CREATE_DETACHED**) oluşturulacağını seçer. Öntanımlı olan **PTHREAD_CREATE_JOINABLE** değeridir.

Birleşimci durumunda, başka bir evre, evrenin sonlanışıyla eşzamanlanabilir ve **pthread_join** kullanarak kendi sonlanma kodunu kurtarabilir, fakat bazı evre özkaynakları evre sonlandıktan sonra ayrılmış kalır ve ancak başka bir evre o evre üzerinde **pthread_join** uygularsa geri alınabilir.

Ayrık durumda, evre sonlandığında özkaynakları anında serbest bırakılır, fakat evrenin sonlandırılışında eşzamanlamak için **pthread_join** kullanılamaz.

Birleşimci durumda oluşturulan bir evre daha sonra **pthread_detach** ile ayrık evreye konulabilir.

`schedpolicy`
Evre için zamanlama ilkesini seçilir: **SCHED_OTHER** (normal, gerçek zamanlı olmayan zamanlama), **SCHED_RR** (gerçek zamanlı, döner turnuva) veya **SCHED_FIFO** (gerçek zamanlı, ilk giren ilk çıkar). Öntanımlı olan **SCHED_OTHER** ilkesidir.

Gerçek zamanlı zamanlama ilkeleri olan **SCHED_RR** ve **SCHED_FIFO** ilkeleri sadece süper kullanıcı haklarına sahip süreçler için mevcuttur. Eğer yetkisizken gerçek zamanlı bir ilke kurmayı denerseniz **pthread_attr_setschedparam** işlevi başarısız olur ve **ENOTSUP** hatasını döndürür.

Bir evre oluşturulduktan sonra, zamanlama ilkesi **pthread_setschedparam** ile değiştirilebilir.

schedparam
: Evre için zamanlama parametresini (zamanlama önceliği) değiştirir. Öntanımlı değeri 0'dır.

  Bu özellik zamanlama ilkesi **SCHED_OTHER** ise anlamlı değildir; bu sadece gerçek zamanlı **SCHED_RR** ve **SCHED_FIFO** ilkelerini ilgilendirir.

  Bir evre oluşturulduktan sonra, zamanlama önceliği **pthread_setschedparam** ile değiştirilebilir.

inheritsched
: Yeni oluşturlan evre için zamanlama ilkesinin ve parametresinin *schedpolicy* ve *schedparam* özelliklerinin değerleriyle mi (**PTHREAD_EXPLICIT_SCHED** değeri) yoksa üst evreden miras alınarak mı (**PTHREAD_INHERIT_SCHED** değeri) belirleneceğini seçer. Öntanımlı değer **PTHREAD_EXPLICIT_SCHED** değeridir.

scope
: Oluşturulan evre için zamanlama çekişme kapsamını seçer. Öntanımlı değeri **PTHREAD_SCOPE_SYSTEM**'dir ve bu, evrelerin işlemci zamanı için makinada çalışan bütün süreçlerle çekişmesi anlamına gelir. Evre öncelikleri makinada çalışan diğer bütün süreçlere göre yorumlanır. Diğer bir olasılık, **PTHREAD_SCOPE_PROCESS** olup bu, evrelerin zamanlama çekişmesinin sadece çalışan sürecin evreleri arasında gerçekleştiği anlamına gelir: evre öncelikleri sürecin diğer evrelerine göre yorumlanır, diğer süreçlerin önceliklerine bakılmaz.

  PTHREAD_SCOPE_PROCESS LinuxThreads'de desteklenmemektedir. Eğer kapsamı bu değer olarak ayarlamaya çalışırsanız, **pthread_attr_setscope** başarısız olur ve **ENOTSUP** döndürür.

stackaddr
: Uygulama yönetimli yığıt için adres sağlar. Yığıtın büyüklüğü en az **PTHREAD_STACK_MIN** olmalıdır.

stacksize
: Evre için oluşturulan yığıtın büyüklüğünü değiştirir. Değer yığıt için asgari büyüklüğü bayt cinsinden tanımlar.

  Eğer değer sistemin azami yığıt büyüklüğünü aşarsa veya **PTHREAD_STACK_MIN** değerinden küçükse, **pthread_attr_setstacksize** başarısız olur ve **EINVAL** döndürür.

stack
: Yeni evreye, kullanmak için uygulama yönetimli yığıta hem adresi hem de büyüklüğü sağlar. Bellek alanının tabanı, bayt cinsinden *stacksize* büyüklüğündeki *stackaddr*dir.

  Eğer *stacksize* değeri **PTHREAD_STACK_MIN** değerinden az ise veya sistemin azami yığın büyüklüğünden büyükse veya *stackaddr* değeri uygun hizalamadan yoksunsa, **pthread_attr_setstack** başarısız olur ve **EINVAL** döndürür.

guardsize
: Evre yığıtının koruma alanının asgari büyüklüğünü bayt cinsinden değiştirir. Varsayılan büyüklük tek sayfadır. Eğer bu değer belirtilirse, en yakın sayfa büyüklüğüne yuvarlanır. Değer 0'a eşitlenirse, bu evre için koruma alanı oluşturulmaz. Koruma alanı için ayrılan yer yığıt taşmalarını yakalamak için kullanılır. Bu nedenle, yığıt üzerinde büyük yapılar ayrılacağında, yığıt taşmalarını yakalamak için daha büyük koruma alanı gerekebilir.

Çağrıcı kendi yığıtlarını yönetiyorsa (eğer **stackaddr** özelliği belirtildiyse), o zaman **guardsize** özelliği dikkate alınmaz.

Eğer değer **stacksize** değerini aşarsa, **pthread_atrr_setguardsize** başarısız olur ve **EINVAL** döndürür.

10.3. İptaletme

İptaletme bir evrenin başka bir evrenin işletilmesini sonlandırabildiği bir mekanizmadır. Daha kesin bir deyişle, bir evre başka bir evreye bir iptal isteği gönderebilir. Ayarlarına bağlı olarak, hedef evre isteği reddedebilir veya bir iptal noktasına ulaşana kadar erteleyebilir. Evreler **pthread_create** ile ilk oluşturulduklarında, daima iptal isteklerini ertelerler.

Bir evre iptal isteğine cevap verirken aslında, **pthread_exit(PTHREAD_CANCELED)** çağrılmış gibi davranır. Bütün temizlik işleyicileri ters sıraya işletilirler, evrelere özgü veri sonlandırma işlevleri çağrılır ve son olarak evre işletilmeyi durdurur. İptal edilen evre birleşimciyse, **PTHREAD_CANCELED** dönüş değeri bu evre üzerinde *pthread_join* işlevini çağıran evreye gönderilir. Daha fazla bilgi için bkz. **pthread_exit** (sayfa: 695).

İptal noktaları evrelerin bekleyen iptal isteklerini kontrol ettikleri ve uyguladıkları noktalardır. POSIX evre işlevleri **pthread_join**, **pthread_cond_wait**, **pthread_cond_timedwait**, **pthread_testcancel**, **sem_wait** ve **sigwait** iptal noktalarıdır. Ek olarak, bu sistem çağrıları da iptal noktalarıdır:

accept	**open**	**sendmsg**
close	**pause**	**sendto**
connect	**read**	**system**
fcntl	**recv**	**tcdrain**
fsync	**recvfrom**	**wait**
lseek	**recvmsg**	**waitpid**
msync	**send**	**write**
nanosleep		

Bu işlevleri çağıran bütün kütüphane işlevleri de (**printf** gibi) iptal noktalarıdır.

```
int pthread_setcancelstate(int  durum,                                  işlev
                           int *eskidurum)
```

pthread_setcancelstate çağıran evre için iptal durumunu değiştirir – bu iptal isteklerinin dikkate alınıp alınmayacağıdır. *durum* argümanı yeni iptal durumudur: **PTHREAD_CANCEL_ENABLE** iptali etkinleştirir veya **PTHREAD_CANCEL_DISABLE** iptali etkisiz kılar (iptal istekleri dikkate alınmaz).

Eğer *eskidurum* **NULL** değilse, önceki iptal durumu *eskidurum* ile gösterilen yerde saklanır, böylece daha sonra **pthread_setcancelstate** işlevine yapılacak başka bir çağrıyla geri yüklenebilir.

Eğer *durum* argümanı **PTHREAD_CANCEL_ENABLE** veya **PTHREAD_CANCEL_DISABLE** değilse, **pthread_setcancelstate** başarısız olur ve **EINVAL** döndürür. Aksi takdirde 0 döndürür.

```
int pthread_setcanceltype(int  tür,                                     işlev
                          int *eskitür)
```

pthread_setcanceltype çağıran evre için gelen iptal isteklerine verilen cevap türünü değiştirir: zamanuyumsuz (anında) veya ertelenmiş. *tür* argümanı yeni iptal türüdür: Bunlar ya çağıran evre iptal isteğini alır almaz iptal etmek için **PTHREAD_CANCEL_ASYNCHRONOUS** ya da diğer iptal noktasına kadar

iptal isteğini bekletmek için **PTHREAD_CANCEL_DEFERRED**dir. Eğer *eskitür* **NULL** değilse, önceki iptal durumu *eskitür* ile gösterilen yerde saklanır, böylece daha sonra **pthread_setcanceltype** işlevine yapılacak başka bir çağrıyla geri yüklenebilir.

Eğer *tür* argümanı **PTHREAD_CANCEL_DEFERRED** veya **PTHREAD_CANCEL_ASYNCHRONOUS** değilse, **pthread_setcanceltype** başarısız olur ve **EINVAL** döndürür. Aksi takdirde 0 döndürür.

`void pthread_testcancel(void)` işlev

pthread_testcancel bekleyen istekleri sınamak ve onları işletmekten başka birşey yapmaz. Amacı uzun kod yığını içerisinde iptal için açık kontroller ortaya koymaktır. Aksi takdirde bu kodlar iptal noktası işlevlerini çağırmazlar.

10.4. Temizlik İşleyicileri

Temizlik işleyicileri, **pthread_exit** çağrılarak veya iptal nedeniyle bir evre sonlandığında çağrılan işlevlerdir. Temizlik işleyicileri yığıta benzer bir disiplinde kurulurlar ve kaldırılırlar.

Temizlik işleyicilerinin amacı bir evrenin tuttuğu kaynakları evre sonlandığı zaman serbest bırakmaktır. Eğer bir evre kilitli bir karşılıklı red nesnesine (mutex)[13] sahipken çıkarsa veya iptal edilirse, nesne sonsuza kadar kilitli kalır ve diğer evrelerin normal işletilmesini engeller. Bundan kaçınmanın en iyi yolu, karşılıklı red nesnesini kilitlemeden az önce, nesnenin kilidini kaldıracak bir temizlik işleyicisinin kurulmasıdır. Temizlik işleyicileri **malloc** ile ayrılmış blokların serbest bırakılmasında veya evre sonlandırıldığında dosya tanımlayıcılarının kapatılmasında da kullanılabilir.

Burada *mut* kilitliyken evre iptal edilirse, kilidinin kaldırılmasında gerektiği gibi bir *mut* nesnesinin nasıl kilitlenebileceğini görüyoruz.

```
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* biraz çalış */
pthread_mutex_unlock(&mut);
pthread_cleanup_pop(0);
```

Son iki satır

```
pthread_cleanup_pop(1);
```

ile değiştirilebilir.

Unutmayınız ki yukarıdaki kod sadece ertelenmiş iptal kipinde güvenlidir (bkz. **pthread_setcanceltype** (sayfa: 699)). Zamanuyumsuz iptal kipinde, bir iptal **pthread_cleanup_push** ve **pthread_mutex_lock** arasında veya **pthread_mutex_unlock** ve **pthread_cleanup_pop** arasında oluşur, evrenin her iki durumunda muteksin kilidini açmayı deneyen geçerli evre tarafından kilitlenmez. Bu zamanuyumsuz iptalin kullanımının neden zor olduğunun ana sebebidir.

Eğer yukarıdaki kodun zamanuyumsuz iptal kipinde de çalışması gerekiyorsa, muteksi kilitlemek ve kilidi açmak için ertelenmiş iptal kipine çevrilmelidir:

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* Birşeyler yap */
pthread_cleanup_pop(1);
pthread_setcanceltype(oldtype, NULL);
```

Yukarıdaki kod taşınabilir değildir. **pthread_cleanup_push_defer_np** ve **pthread_cleanup_pop_restore_np** işlevlerinin kullanılmasıyla daha kısa ve verimli bir şekilde tekrar yazılabilir:

```
pthread_cleanup_push_defer_np(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* Birşeyler yap */
pthread_cleanup_pop_restore_np(1);
```

```
void pthread_cleanup_push(void (*işlev) (void *),                    işlev
                          void  *arg)
```

pthread_cleanup_push *işlev* işlevini *arg* argümanıyla bir temizlik işleyicisi olarak yükler. Bu noktadan itibaren **pthread_cleanup_pop** ile ilişkili *işlev* işlevi, ne zaman evre **pthread_exit** veya iptal ile sonlansa, *arg* argümanlarıyla çağrılacaktır. Eğer bu noktada çok sayıda temizlik işleyicileri etkinse, LIFO sırasıyla çağrılırlar: en son yüklenen ilk çağrılır.

```
void pthread_cleanup_pop(int çalıştır)                               işlev
```

pthread_cleanup_pop en son yüklenen temizlik işleyicisini kaldırır. Eğer *çalıştır* argümanı 0 değilse, aynı zamanda *işlev* işlevini *arg* argümanlarıyla çağırarak işleyiciyi de işletir. Eğer *çalıştır* argümanı 0 ise, işleyici sadece kaldırılır, işletilmez.

pthread_cleanup_push ve **pthread_cleanup_pop** işlevlerinin eşleşen çiftleri aynı işlev içinde aynı blok yuvalama seviyesinde gerçekleşmelidir. Aslında, **pthread_cleanup_push** ve **pthread_cleanup_pop** birer makrodur, ve **pthread_cleanup_push** bir bloğu başlatan kaşlı ayraca **{**, **pthread_cleanup_pop** ise bunun kapatan kaşlı ayraca **}** karşılık gelir.

```
void pthread_cleanup_push_defer_np(void (*işlev) (void *),           işlev
                                   void  *arg)
```

pthread_cleanup_push_defer_np, **pthread_cleanup_push** ile **pthread_setcanceltype**'ı birleştiren taşınabilir olmayan bir oluşumdur. Sadece **pthread_cleanup_push**'un yaptığı gibi bir temizlik işleyicisi iter, fakat aynı zamanda geçerli iptal türünü kaydeder ve onu ertelenmiş iptale ayarlar. Bu temizleme mekanizmasının ilk başta zamanuyumsuz iptal kipinde olsa bile verimli olduğunu temin eder.

```
void pthread_cleanup_pop_restore_np(int çalıştır)                    işlev
```

pthread_cleanup_pop_restore_np, **pthread_cleanup_push_defer_np** ile tanıştırılan bir temizlik işleyicisini çeker ve iptal türünü **pthread_cleanup_push_defer_np** çağrıldığında olan değerine geri yükler.

pthread_cleanup_push_defer_np ve **pthread_cleanup_pop_restore_np** eşleşen çiftlerde aynı blok yuvalama seviyesinde gerçekleşmelidir.

```
pthread_cleanup_push_defer_np(routine, arg);
...
pthread_cleanup_pop_defer_np(execute);
```

sıralaması

```
{
  int oldtype;
  pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
  pthread_cleanup_push(routine, arg);
```

```
  ...
  pthread_cleanup_pop(execute);
  pthread_setcanceltype(oldtype, NULL);
}
```

ile işlevsel olarak denktir.(fakat daha kısa ve verimlidir)

10.5. Muteksler

Bir mutek bir Karşılıklı Dışlama (MUTual EXclusion) aygıtıdır ve paylaşılan veri yapılarını eşzamanlı değişikliklerden korumak için ve kritik bölümler ve görüntüleme birimlerini uygulamak için kullanışlıdır.

Bir muteksin iki durumu vardır: kilitsiz (herhangi bir evre tarafından sahiplenilmemiş) ve kilitli (bir evre tarafından sahiplenilmiş). Bir muteks hiçbir zaman aynı anda iki farklı evre tarafından sahiplenilemez. Kilitli bir muteksi kilitlemeye teşebbüs eden bir evre sahiplenen evre o muteksin kilidini açana kadar askıya alınır.

Muteks işlevlerinden hiçbiri bir iptal noktası değildir, hatta bir evreyi belirli aralılarla askıya alabilen **pthread_mutex_lock** bile. Evre işletilmeyi durdurmadan önce kilidinin açılması gereken mutekslerı, iptal işleyicilerinin açmalarını sağlayarak, mutekslerin iptal noktalarındaki durumları önceden tahmin edilebilir. Sonuç olarak, ertelenmiş iptal kullanan evreler hiçbir zaman bir muteksi genişletilmiş zaman aralıklarıyla tutmamalıdırlar.

Muteks işlevlerini tek bir sinyal işleyiciden çağırmak güvenli değildir. Özellikle, tek bir sinyal işleyiciden **pthread_mutex_lock** veya **pthread_mutex_unlock** çağırmak, çağıran evreyi kısır döngüye sokabilir.

```
int pthread_mutex_init(pthread_mutex_t            *muteks,                  işlev
                       const pthread_mutexattr_t *muteks_özelliği)
```

pthread_mutex_init *muteks_özelliği* ile mutex özellikleri belirtilen *muteks* ile gösterilen muteks nesnesini hazırlar. Eğer *muteks_özelliği* **NULL** ise, öntanımlı özellikler kullanılır.

LinuxThreads gerçeklemesi sadece bir muteks özelliğini desteklemektedir, bu "hızlı", "özyinelemeli" veya "hata denetimli" olan *muteks türü*dür. Mutex tipi, sahiplenildiği evre tarafından tekrar kilitlenebilirliğini belirler. Varsayılan "hızlı"dır.

pthread_mutex_t türündeki değişkenler de **PTHREAD_MUTEX_INITIALIZER** (zamanlı muteksler için), **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP** (özyinelemeli muteksler için), **PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP** (hızlı muteksler için) ve **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP** (hata denetimli muteksler için) sabitleri kullanılarak ilk değerleri statik olarak alabilirler.

pthread_mutex_init daima 0 ile döner.

```
int pthread_mutex_lock(pthread_mutex_t *muteks)                              işlev
```

pthread_mutex_lock verilen muteksi kilitler. Muteks kilitli değilse, kilitlenir, çağıran evre tarafından sahiplenilir ve **pthread_mutex_lock** hemen döner. Eğer muteks zaten başka bir evre tarafından kilitlendiyse, **pthread_mutex_lock** muteksin kilidi açılana kadar çağıran evreyi askıya alır.

Eğer muteks çağıran evre tarafından önceden kilitlenmişse, **pthread_mutex_lock** davranışı muteks türüne bağlıdır. Eğer muteks "hızlı" türdeyse, çağıran evre askıya alınır. Sonsuza kadar askıda kalır, çünkü başka hiçbir evre muteksin kilidini açamaz. Eğer muteks "hata kontrollü" türdeyse, **pthread_mutex_lock EDEADLK** hata koduyla hemen döner. Mutex "özyinelemeli" türdeyse, **pthread_mutex_lock** başarılı olur ve çağıran evrenin muteksi kaç kez kilitlediğini kaydederek hemen döner. Muteksi kilitsiz hale düşürmek için eşit sayıda **pthread_mutex_unlock** işlemi uygulanmalıdır.

```
int pthread_mutex_trylock(pthread_mutex_t *muteks)                    işlev
```

pthread_mutex_trylock işlevi **pthread_mutex_lock** ile aynı davranır, farklı olarak eğer muteks başka bir evre tarafından (veya "hızlı" muteks türünde çağıran evre tarafından) önceden kilitlenmişse, çağıran evreyi askıya almaz. Bunun yerine, **pthread_mutex_trylock** hemen **EBUSY** hata koduyla döner.

```
int pthread_mutex_timedlock(pthread_mutex_t        *muteks,           işlev
                            const struct timespec *mutlak_zaman)
```

pthread_mutex_timedlock işlevi **pthread_mutex_lock** işlevi ile aynıdır, fakat muteks başka bir evre tarafından kilitlendiğinde, sonsuza kadar askıya almaktansa *mutlak_zaman* ile belirtilen zamana erişildiğinde döner.

Bu işlev sadece standart ("zamanlı") ve "hata denetimli" mutekslerde kullanılabilir. Diğer bütün türler için **pthread_mutex_lock** gibi davranır.

Eğer muteks başarıyla kilitlendiyse, işlev sıfır döndürür. Eğer muteks kilitlenmeden *mutlak_zaman* süresi dolarsa, **ETIMEDOUT** döndürülür.

Bu işlev POSIX standartlarının POSIX.1d uyarlamasıyla tanıtılmıştır.

```
int pthread_mutex_unlock(pthread_mutex_t *muteks)                     işlev
```

pthread_mutex_unlock verilen muteksin kilidini açar. **pthread_mutex_unlock** işlevinin girişinde muteks kilitlenmiş ve çağıran evre tarafından sahiplenilmiş olarak kabul edilir. Eğer muteks "hızlı" türdeyse, **pthread_mutex_unlock** kilitsiz durumuna döndürür. Eğer "özyinelemeli" türdeyse, muteksin kilit sayısını azaltır (çağıran evre tarafından uygulanan **pthread_mutex_lock** işlemi sayısı), ve ancak bu sayı sıfıra ulaştığında muteks kilidi açılmış olur.

"Hata denetimli" mutekslerde, **pthread_mutex_unlock** aslında çalışma–anında muteksin kilitli olup olmadığını girişte kontol eder, ve o an **pthread_mutex_unlock** çağıranın kilitleyen evre olup olmadığına bakar. Eğer bu koşullar karşılanmadıysa, **pthread_mutex_unlock** işlevi **EPERM** döndürür, ve muteks değişmez. "Hızlı" ve "özyinelemeli" muteksler bu tür kontroller uygulamazlar, bu nedenle kilitli bir muteksin sahibinden başka bir evre tarafından kilidinin açılmasına izin verirler. Bu taşınabilir olmayan bir davranıştır ve güvenilemez.

```
int pthread_mutex_destroy(pthread_mutex_t *muteks)                    işlev
```

pthread_mutex_destroy bir muteks nesnesini, sahip olabileceği kaynakları serbest bırakarak yok eder. Girişte muteksin kilitlenmemiş olması gerekir. LinuxThreads gerçeklemesinde, muteks nesneleriyle hiçbir kaynak ilişkilendirilmemiştir, bu nedenle **pthread_mutex_destroy** aslında muteksin kilitli olup olmadığını kontrol etmekten başka bir şey yapmaz.

Eğer muteks bir evre tarafından kilitlendiyse, **pthread_mutex_destroy** işlevi **EBUSY** döndürür. Aksi takdirde 0 döndürür.

Eğer yukarıdaki işlevlerden herhangi biri (**pthread_mutex_init** dışında) hazırlanmamış bir mutekse uygulanırsa, sadece **EINVAL** döndürürler ve başka birşey yapmazlar.

Paylaşılan global bir *x* değişkeni bir muteks tarafından aşağıdaki gibi korunabilir:

```
int x;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

x'e yapılan bütün erişim ve değişiklikler **pthread_mutex_lock** ve **pthread_mutex_unlock** arasına aşağıdaki gibi alınmalıdır.

```
pthread_mutex_lock(&mut);
/* x üzerindeki işlemler*/
pthread_mutex_unlock(&mut);
```

Muteks özellikleri muteksin oluşturulması sırasında **pthread_mutex_init**'e ikinci parametre olarak muteks özellik nesnesinin aktarılması ile belirtilebilirler. **NULL** aktarmak, bütün özellikleri öntanımlı değer verilmiş bir muteks özellik nesnesi aktarmakla eşdeğerdir.

`int` **pthread_mutexattr_init**`(pthread_mutexattr_t *`*öznitelik*`)` işlev

pthread_mutexattr_init işlevi *öznitelik* muteks özellik nesnesini hazırlar ve özelliklerini öntanımlı değerlerle doldurur.

Bu işlev her zaman 0 döndürür.

`int` **pthread_mutexattr_destroy**`(pthread_mutexattr_t *`*öznitelik*`)` işlev

pthread_mutexattr_destroy bir muteks özellik nesnesini yok eder, bu nesne tekrar hazırlanıncaya kadar tekrar kullanılmamalıdır. **pthread_mutexattr_destroy** LinuxThreads gerçeklemesinde birşey yapmaz.

Bu işlev her zaman 0 döndürür.

LinuxThreads sadece bir muteks özelliğini destekler: muteks türü. "hızlı" muteksler için **PTHREAD_MUTEX_ADAPTIVE_NP**, "özyinelemeli" muteksler için **PTHREAD_MUTEX_RECURSIVE_NP**, "zamanlı" muteksler için **PTHREAD_MUTEX_TIMED_NP** , veya "hata denetimli" muteksler için **PTHREAD_MUTEX_ERRORCHECK_NP** olabilir. **NP** ekinin de gösterdiği gibi, POSIX standardının taşınabilir olmayan bir oluşumudur ve taşınabilir yazılımlarda kullanılmamalıdır.

Muteks türü bir evre zaten sahip olduğu bir muteksi **pthread_mutex_lock** ile kilitlemeye çalıştığında ne olacağını belirler. Eğer muteks "hızlı" türdeyse, **pthread_mutex_lock** sadece çağıran evreyi sonsuza kadar askıya alır. Eğer muteks "hata denetimli" türdeyse, **pthread_mutex_lock** hemen **EDEADLK** hata koduyla döner. Eğer muteks "özyinelemeli" türdeyse, **pthread_mutex_lock**'a yapılan çağrı hemen bir başarı dönüş koduyla döner. Evrenin mutekse kaç kez sahip olduğu muteks içinde kayıtlanır. Muteksin kilitsiz duruma dönmesi için, sahip olan evre **pthread_mutex_unlock** işlevini aynı sayıda çağırmalıdır.

Varsayılan muteks türü **PTHREAD_MUTEX_TIMED_NP** olan "zamanlı"dır.

```
int pthread_mutexattr_settype(pthread_mutexattr_t *öznitelik,          işlev
                              int                  tür)
```

pthread_mutexattr_settype işlevi *öznitelik* içindeki muteks türü özelliğini *tür* ile belirtilen değere ayarlar.

Eğer *tür* **PTHREAD_MUTEX_ADAPTIVE_NP**, **PTHREAD_MUTEX_RECURSIVE_NP**, **PTHREAD_MUTEX_TIMED_NP**, veya **PTHREAD_MUTEX_ERRORCHECK_NP** değilse, bu işlev **EINVAL** döndürür ve *öznitelik* değerini değiştirmez.

Standart Unix98 tanıtıcıları olan **PTHREAD_MUTEX_DEFAULT**, **PTHREAD_MUTEX_NORMAL**, **PTHREAD_MUTEX_RECURSIVE** ve **PTHREAD_MUTEX_ERRORCHECK** değerlerine de izin verilmiştir.

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *öznitelik,    işlev
                              int                       *tür)
```

pthread_mutexattr_gettype işlevi *öznitelik* içindeki geçerli muteks türü özelliğini alır ve *tür* ile gösterilen yerde saklar.

Bu işlev her zaman 0 döndürür.

10.6. Koşul Değişkenleri

Bir koşul ("koşul değişkeni" yerine), evrelerin işletilmeyi paylaşılan veride bir karara varıncaya kadar askıya almasına olanak veren bir eşzamanlama aracıdır. Koşullar üzerindeki temel işlemler: koşula sinyal göndermek (Karar doğru çıkınca) ve başka bir evre koşula sinyal gönderene kadar evrenin işletilmesini askıya alarak koşul için beklemek.

Yarış koşulundan (race condition) kaçınmak için bir koşul değişkeni herzaman bir muteks ile ilişkilendirilmelidir. Öyle ki yarış koşulunda bir evre bir koşul değişkeni için beklemeye hazırlanırken başka bir evre ilk evre tam beklemeye başlayacakken koşula sinyal gönderir.

```
int pthread_cond_init(pthread_cond_t           *koşul,                    işlev
                      pthread_condattr_t *koşul_özelliği)
```

pthread_cond_init işlevi *koşul* koşul değişkenini, *koşul_özelliği* içinde belirtilen koşul özelliklerini kullanarak ya da eğer *koşul_özelliği* **NULL** ise öntanımlı özellikleri kullanarak hazırlar. LinuxThreads gerçeklemesi koşullar için özellikleri desteklemez, bu yüzden *koşul_özelliği* parametresi dikkate alınmaz.

pthread_cond_t türündeki değişkenler de **PTHREAD_COND_INITIALIZER** sabitini kullanarak durağan olarak hazırlanabilirler.

Bu işlev her zaman 0 döndürür.

```
int pthread_cond_signal(pthread_cond_t *koşul)                              işlev
```

pthread_cond_signal işlevi *koşul* koşul değişkeni için bekleyen evrelerden birini yeniden başlatır. Eğer hiçbir evre *koşul* için beklemiyorsa, bir şey olmaz. Eğer *koşul* için birçok evre bekliyorsa, araların biri yeniden başlatılır.

Bu işlev her zaman 0 döndürür.

```
int pthread_cond_broadcast(pthread_cond_t *koşul)                           işlev
```

pthread_cond_broadcast işlevi *koşul* koşul değişkeni için bekleyen bütün evreleri yeniden başlatır. *koşul* için bekleyen hiç evre yoksa birşey olmaz.

Bu işlev her zaman 0 döndürür.

```
int pthread_cond_wait(pthread_cond_t  *koşul,                               işlev
                      pthread_mutex_t *muteks)
```

pthread_cond_wait işlevi *muteks*'in kilidini atomik olarak açar (**pthread_unlock_mutex** gibi) ve *koşul* koşul değişkeninin sinyal alması için bekler. Evre işletimi askıya alınır ve koşul değişkeni sinyal alana kadar işlemci zamanı harcamaz. **pthread_cond_wait** girişinde, *muteks* çağıran evre tarafından kilitlenmelidir. Çağıran evreye geri dönmeden önce, **pthread_cond_wait** *muteks*'i tekrar elde eder (**pthread_lock_mutex** gibi).

Muteks kilidini açmak ve koşul değişkeni üzerine askıya almak atomik olarak gerçekleşir. Bu nedenle, eğer bütün evreler koşula sinyal göndermeden önce hep muteksi elde etseler, bu evrenin muteksi kilitlemesiyle koşul değişkeni için bekleyeceği zaman aralığında koşula sinyal gönderilemeyeceğini temin eder.

Bu işlev her zaman 0 döndürür.

```
int pthread_cond_timedwait(pthread_cond_t         *koşul,                   işlev
                           pthread_mutex_t        *muteks,
                           const struct timespec *mutlak_zaman)
```

pthread_cond_timedwait işlevi *muteks* kilidini atomik olarak açar ve **pthread_cond_wait**'in yaptığı gibi *koşul* için bekler, fakat aynı zamanda bekleme süresini sınırlar. Eğer *koşul mutlak_zaman* süresinden önce sinyal almadıysa, *muteks* muteksi tekrar elde edilir ve **pthread_cond_timedwait** **ETIMEDOUT** hata kodunu döndürür. Bekleme de bir sinyal ile kesilebilir; bu durumda **pthread_cond_timedwait** işlevi **EINTR** döndürür.

mutlak_zaman parametresi, **time** ve **gettimeofday** ile aynı kökende kesin bir süre belirler: *mutlak_zaman* için 0 değeri 00:00:00 GMT, January 1, 1970 anlamına gelir.

`int` **pthread_cond_destroy**`(pthread_cond_t *`*koşul*`)` işlev

pthread_cond_destroy işlevi *koşul* koşul değişkenini sahip olabileceği kaynakları serbest bırakarak yok eder. Eğer herhangi bir evre koşul değişkeni için bekliyorsa, **pthread_cond_destroy** *koşul* koşulunu değiştirmeden bırakır ve **EBUSY** döndürür. Aksi takdirde 0 döndürür ve *koşul* tekrar hazırlanana kadar kullanılmamalıdır.

LinuxThreads gerçeklemesinde, koşul değişkenleriyle hiçbir kaynak ilişkilendirilmemiştir, bu yüzden **pthread_cond_destroy** aslında hiçbir şey yapmaz.

pthread_cond_wait ve **pthread_cond_timedwait** iptal noktalarıdır. Eğer bir evre bu işlevlerden biri tarafından askıya alındığında iptal edildiyse, evre çalışmaya geri döner, *muteks* ile belirtilen muteksi tekrar kilitler ve son olarak iptali işletir. Sonuç olarak, temizlik işleyicileri *muteks*'in çağrıldığında kilitli olduğuna emin olurlar.

Bir sinyal işleyicisinden koşul değişken işlevlerinin çağrılması güvenli değildir. Özellikle, bir sinyal işleyicisinden **pthread_cond_signal** veya **pthread_cond_broadcast** çağırmak çağıran evreyi kısır döngüye sokabilir.

mut muteksi tarafından korunan, paylaşılan iki *x* ve *y* değişkeni ve ne zaman *x* *y*'den büyük olsa sinyal alan bir *cond* koşul değişkeni düşünün.

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

x *y*'den büyük olana kadar beklemek aşağıdaki gibi yapılır:

```
pthread_mutex_lock(&mut);
while (x <= y) {
        pthread_cond_wait(&cond, &mut);
}
/* x ve y üzerinde işle*/
pthread_mutex_unlock(&mut);
```

x'in *y*'den büyük olmasına neden olabilecek *x* ve *y* üzerindeki değişiklikler gerektiğinde koşula sinyal göndermeli:

```
pthread_mutex_lock(&mut);
/*  x ve y'yi değiştir*/
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

Eğer en çok bir evrenin uyanması gerektiği kanıtlanabilirse (örneğin, *x* ve *y* üzerinden haberleşen sadece iki evre varsa), **pthread_cond_signal** işlevi **pthread_cond_broadcast**'in daha verimli bir alternatifi olarak kullanılabilir. Şüpheliyseniz **pthread_cond_broadcast** kullanın.

x'in *y*'den büyük oluncaya kadar 5 saniye beklenmesi için:

```
struct timeval now;
```

```
struct timespec timeout;
int retcode;

pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
while (x <= y && retcode != ETIMEDOUT) {
        retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
}
if (retcode == ETIMEDOUT) {
        /* zaman aşımı oluştu */
} else {
        /* x ve y üzerinde işlem yap*/
}
pthread_mutex_unlock(&mut);
```

Koşul özellikleri,**pthread_cond_init** işlevinin ikinci argümanı olarak koşul özellik nesnesi geçirilerek, koşul oluşturulması esnasında belirtilebilir. **NULL** geçirmek, bütün özellikleri öntanımlı değer verilmiş bir koşul özellik nesnesi geçirmekle eşdeğerdir.

LinuxThreads gerçeklemesi koşullar için hiçbir özellik desteklememektedir. Koşul özellikleriyle ilgili işlevler sadece POSIX standardıyla uyum için içerilmektedir.

```
int pthread_condattr_init(pthread_condattr_t *öznitelik)                    işlev
int pthread_condattr_destroy(pthread_condattr_t *öznitelik)                 işlev
```

pthread_condattr_init işlevi *öznitelik* koşul özellik nesnesini hazırlar ve özelliklerini öntanımlı değerlerle doldurur. **pthread_condattr_destroy** işlevi ise *öznitelik* koşul özellik nesnesini yok eder.

Her iki işlev de LinuxThreads gerçeklemesinde bir şey yapmaz.

pthread_condattr_init ve **pthread_condattr_destroy** herzaman 0 döndürr.

10.7. POSIX Semaforları

Semaforlar evreler arasında paylaşılan kaynaklar için sayaçtırlar. Semaforlar üzerindeki temel işlemler: sayacı atomik olarak artırmak ve sayaç sıfırdan farklı oluncaya kadar beklemek ve atomik olarak azaltmak.

Semaforların azami bir değeri vardır ve bunu geçemezler. **SEM_VALUE_MAX** makrosu bu azami değer olmak için tanımlanmıştır. GNU C kütüphanesinde, **SEM_VALUE_MAX** ile **INT_MAX** eşittir. (Bkz. *Bir Tamsayı Türün Aralığı* (sayfa: 821)), fakat diğer sistemlerde oldukça küçük olabilir.

POSIX evreleri kütüphanesi POSIX 1003.1b semaforlarını gerçeklemektedir. Bunlar System V semaforlarıyla (**ipc**, **semctl** ve **semop**) karıştırılmamalıdır.

Bütün semafor işlevleri ve makroları `semaphore.h` içinde tanımlıdır.

```
int sem_init(sem_t        *sem,                                             işlev
             int           paylaşımlı,
             unsigned int  değer)
```

sem_init *sem* ile gösterilen semafor nesnesini hazırlar. Semafor ile ilişkili sayı ilk değer olarak *değer* alır. *paylaşımlı* argümanı semaforun geçerli sürece yerel (sıfır ise) veya süreçler arasında paylaşımlı (sıfırdan farklı ise) olduğunu belirtir.

Başarı halinde **sem_init** 0 döndürür. Başarısızlık halinde –1 döndürür ve *errno* değerini aşağıdakilerden biri yapar:

EINVAL
: *değer* azami sayaç değeri **SEM_VALUE_MAX** değerini aşmaktadır.

ENOSYS
: *paylaşımlı* sıfır değildir. LinuxThreads henüz süreç–paylaşımlı semaforları desteklememektedir. (Bu aslında değişecek.)

`int sem_destroy(sem_t * sem)` işlev

sem_destroy semafor nesnesini sahip olabileceği kaynakları serbest bırakarak yok eder. Eğer herhangi bir evre **sem_destroy** çağrıldığında semaforda bekliyorsa, başarısız olur ve *errno* değerini **EBUSY** yapar.

LinuxThreads gerçeklemesinde, semafor nesneleriyle hiçbir kaynak ilişkilendirilmemiştir, bu nedenle **sem_destroy** aslında semaforda hiçbir evrenin beklemekte olmadığını sınamaktan başka birşey yapmaz. Bu süreç–paylaşımlı semaforlar gerçeklendiğinde değişecektir.

`int sem_wait(sem_t * sem)` işlev

sem_wait *sem* ile gösterilen semafor sıfır olmayan sayıya sahip oluncaya kadar çağıran evreyi askıya alır. Ardından atomik olarak semafor sayısını bir azaltır.

sem_wait bir iptal noktasıdır. Herzaman 0 döndürür.

`int sem_trywait(sem_t * sem)` işlev

sem_trywait işlevi **sem_wait**'in durdurmayan bir türevidir. Eğer *sem* ile gösterilen semafor sıfırdan farklı bir sayıya sahipse, sayı atomik olarak azaltılır ve **sem_trywait** hemen 0 döndürür. Eğer semafor sayısı sıfırsa, **sem_trywait** hemen –1 döndürür ve hata kodunu **EAGAIN** yapar.

`int sem_post(sem_t * sem)` işlev

sem_post işlevi *sem* ile gösterilen semafor sayısını atomik olarak artırır. Bu işlev hiçbir zaman durdurmaz.

Atomik karşılaştırma–ve–değiştirme destekleyen işlemcilerde (Intel 486, Pentium ve sonrası, Alpha, PowerPC, MIPS II, Motorola 68k, Ultrasparc), **sem_post** işlevi sinyal işleyicilerden güvenle çağrılabilir. Bu POSIX evreleri tarafından desteklenen zamanuyumsuz–sinyal güvenli tek evre eşzamanlama işlevidir. Intel 386 ve eski Sparc kıymıklarında, **sem_post**'un geçerli LinuxThreads gerçeklemesi zamanuyumsuz–sinyal güvenli değildir, çünkü donanım gerekli olan atomik işlemleri desteklememektedir.

sem_post semafor sayısı artırıldıktan sonra **SEM_VALUE_MAX** aşılmadığı sürece hep başarılı olur ve 0 döndürür. Bu durumda **sem_post** –1 döndürür ve *errno* hata kodunu **EINVAL** yapar. Semafor sayısı değiştirilmeden bırakılır.

```
int sem_getvalue(sem_t * sem,                                    işlev
                 int   * semdeg)
```

sem_getvalue işlevi *sem* semaforunun geçerli sayısını *semdeg* ile gösterilen yerde saklar. Hep 0 döndürür.

10.8. Evreye Özgü Veri

Yazılımlarda farklı evrelerde farklı değerlere sahip olan global veya statik değişkenlere sıklıkla ihtiyaç duyulur. Evreler bir bellek alanını paylaştıkları için, bu sıradan değişkenlerle başarılamamaktadır. Evreye özgü veri bu ihtiyaca POSIX evrelerinin cevabıdır.

Her evre bir özel bellek bloğunu, evreye özgü veri alanını veya kısaca EÖV alanını zapteder. Bu alan EÖV anahtarlarıyla indekslenir. EÖV alanı **void *** türündeki değerleri EÖV anahtarlarıyla ilişkilendirir. EÖV anahtarları bütün evreler için ortaktır, fakat verilen bir EÖV anahtarına ilişkin değer her evre içinde farklı olabilir.

Somut olarak belirtmek gerekirse, EÖV alanları **void *** göstericilerden oluşan diziler olarak gösterilebilirler, EÖV anahtarları bu diziler için birer tamsayı indisi ve EÖV anahtarının değeri de çağıran evre içindeki ilişkili dizi elemanının değeridir.

Bir evre oluşturulduğunda, onun EÖV alanı ilk başta bütün anahtarlarıyla **NULL** değerini ilişkilendirir.

```
int pthread_key_create((pthread_key_t *anahtar,                                    işlev
                       void          (*yıkıcı_işlev) (void *))
```

pthread_key_create yeni bir EÖV anahtarı ayırır. Anahtar *anahtar* ile gösterilen yerde saklanır. Belirli bir süre içinde ayrılabilecek anahtar sayısı **PTHREAD_KEYS_MAX** ile sınırlandırılmıştır. Döndürülen anahtarla ilişkilendirilmiş değer ilk başta **NULL**dur.

yıkıcı_işlev argümanı, eğer **NULL** değilse anahtarla ilişkili bir yıkıcı işlev belirtir. Bir evre **pthread_exit** ile veya iptal edilerek sonlandığında, evre içindeki anahtarla ilişkili değer üzerine *yıkıcı_işlev* işlevi çağrılır. Eğer anahtar **pthread_key_delete** ile silindiyse veya bir değer **pthread_setspecific** ile değiştirildiyse *yıkıcı_işlev* çağrılmaz. Yıkıcı işlevin evre sonlandırılma esnasındaki çağrılma sırası belirtilmemiştir.

Yıkıcı işlev çağrılmadan önce, **NULL** değeri geçerli evre içindeki anahtar ile ilişkilendirilir. Bir yıkıcı işlev **NULL** olmayan değerleri bu veya başka bir anahtarla tekrar ilişkilendirebilir. Bununla ilgilenmek için, eğer bütün **NULL** olmayan değerler için bütün yıkıcılar çağrıldıktan sonra, hala **NULL** olmayan yıkıcılarla ilişkilendirilmiş bazı değerler varsa, süreç tekrarlanır. LinuxThreads gerçeklemesi **PTHREAD_DESTRUCTOR_ITERATIONS** tekrarlamadan sonra, **NULL** olmayan tanımlayıcılarla ilişkilendirilmiş değer kalsa bile, süreci durdurur. Diğer uygulamalar sonsuza kadar tekrarlayabilir.

pthread_key_create 0 döndürür, ancak **PTHREAD_KEYS_MAX** anahtar zaten ayrılmışsa başarısız olur ve **EAGAIN** döndürür.

```
int pthread_key_delete(pthread_key_t anahtar)                                      işlev
```

pthread_key_delete bir EÖV anahtarını serbest bırakır. Geçerli süreçteki anahtarla **NULL** olmayan değerler ilişkilendirilmiş mi kontrol etmediği gibi, anahtarla ilişkili yıkıcı işlevi de çağırmaz.

Eğer *anahtar* diye bir anahtar yoksa, **EINVAL** döndürür. Aksi takdirde 0 döndürür.

```
int pthread_setspecific(pthread_key_t anahtar,                                     işlev
                        const void    *gösterici)
```

pthread_setspecific çağıran süreçteki *anahtar* ile ilişkili değeri verilen *gösterici* değeri ile değiştirir.

Eğer *anahtar* diye bir anahtar yoksa, **EINVAL** döndürür. Aksi takdirde 0 döndürür.

```
void *pthread_getspecific(pthread_key_t anahtar)                                   işlev
```

pthread_getspecific çağıran süreçteki *anahtar* ile ilişkili geçerli değeri döndürür.

Eğer *anahtar* diye bir anahtar yoksa, **EINVAL** döndürür. Aksi takdirde 0 döndürür.

Aşağıdaki kod parçası 100 karakterlik bir evreye özgü dizi ayırır. Evre çıkışında da otomatik geri alır:

```
/* Evreye özgü tampon için anahtar */
static pthread_key_t buffer_key;

/* Anahtar bir kerelik hazırlanıyor */
static pthread_once_t buffer_key_once = PTHREAD_ONCE_INIT;

/* Evreye özgü tampon ayrılıyor */
void buffer_alloc(void)
{
  pthread_once(&buffer_key_once, buffer_key_alloc);
  pthread_setspecific(buffer_key, malloc(100));
}

/* Evreye özgü tampon döndürülüyor */
char * get_buffer(void)
{
  return (char *) pthread_getspecific(buffer_key);
}

/* Anahtarı ayır */
static void buffer_key_alloc()
{
  pthread_key_create(&buffer_key, buffer_destroy);
}

/* Evreye özgü tamponu serbest bırak */
static void buffer_destroy(void * buf)
{
  free(buf);
}
```

10.9. Evreler ve Sinyal İşleme

```
int pthread_sigmask(int              nasıl,                                  işlev
                    const sigset_t  *yenimaske,
                    sigset_t        *eskimaske)
```

pthread_sigmask çağıran süreç için sinyal maskesini *nasıl* ve *yenimaske* argümanlarıyla belirtildiği şekilde değiştirir. Eğer *eskimaske* **NULL** değilse, önceki sinyal maskesi *eskimaske* ile gösterilen yerde saklanır.

nasıl ve *yenimaske* argümanlarının anlamı **sigprocmask** ile aynıdır. Eğer *nasıl* **SIG_SETMASK** ise, sinyal maskesi *yenimaske* yapılır. Eğer *nasıl* **SIG_BLOCK** ise, *yenimaske* için belirtilen sinyaller geçerli sinyal maskesine eklenir. Eğer *nasıl* **SIG_UNBLOCK** ise, *yenimaske* için belirtilen sinyaller geçerli sinyal maskesinden kaldırılır.

Sinyal maskeleri her evre başına ayarlanır, fakat sinyal hareketleri ve sinyal işleyicileri **sigaction** ile ayarlanır ve bütün evrelerce paylaşılır.

pthread_sigmask işlevi başarı halinde 0 döndürür, hata halinde de aşağıdaki hata kodlarından birini döndürür:

EINVAL

nasıl **SIG_SETMASK**, **SIG_BLOCK** veya **SIG_UNBLOCK** değerlerinden biri değildir.

EFAULT
yenimaske veya *eskimaske* geçersiz bir adresi göstermektedir.

```
int pthread_kill(pthread_t evre,                                    işlev
                 int       sinyalnum)
```

pthread_kill *sinyalnum* sinyal numarasını *evre* evresine gönderir. Sinyal *Sinyal İşleme* (sayfa: 601) içinde anlatıldığı gibi teslim edilir ve işlenir.

pthread_kill başarı halinde 0 , hata halinde aşağıdaki hata kodlarından birini döndürür:

EINVAL
sinyalnum geçerli bir sinyal numarası değildir.

ESRCH
evre evresi mevcut değildir (örn. sonlandırılmış olabilir)

```
int sigwait(const sigset_t *küme,                                   işlev
            int            *sinyal)
```

sigwait *küme* içindeki sinyallerden biri çağıran evreye ulaştırılıncaya kadar çağıran evreyi askıya alır. Ardından, alınan sinyal numarasını *sinyal* ile gösterilen yerde saklar ve döner. *küme* içindeki sinyaller durdurulmalıdır ve **sigwait** girişinde dikkate alınmalıdırlar. Eğer ulaştırılan sinyal yanında sinyal işleyici işlevine sahipse, bu işlev çağrılmaz.

sigwait bir iptal noktasıdır. Her zaman 0 döndürür.

sigwait'in güvenilir çalışması için, beklenen sinyaller sadece çağıran evrede değil bütün evrelerde durdurulmalıdır, aksi takdirde sinyal ulaştırmanın POSIX mantığı, sinyali alacak olan **sigwait**'in o evre olduğunu garanti etmez. Bunu başarmanın en iyi yolu, herhangi bir evre oluşturulmadan önce bu sinyalleri durdurmak ve onları yazılım içinde **sigwait** çağırmak haricinde serbest bırakmamaktır.

LinuxThreads'deki sinyal işleme POSIX standardındakiyle oldukça farklıdır. Standarda göre, "zamanuyumsuz" (dış) sinyaller bütün sürece (evrelerin toplamına) adreslenirler, ardından süreç bunları belirli bir evreye teslim eder. Sinyali alan evre sinyali o anda durdurmayan evrelerden herhangi biridir.

LinuxThreads'de, her evre aslında kendi süreç kimliği ile bir çekirdek sürecidir, bu nedenle dış sinyaller her zaman belirli bir evreye yönlendirilirler. Eğer, örneğin, başka bir evre o sinyalde **sigwait** içinde durdurulduysa, yeniden başlatılmaz.

sigwait'in LinuxThreads gerçeklemesi *küme* içindeki sinyaller için bekleme süresince kukla sinyal işleyicileri kurar. Sinyal işleyicileri bütün evrelerce paylaşıldığı için, diğer evreler bu sinyallere kendi sinyal işleyicilerini eklememeli veya bir seçenek olarak bütün hepsi sinyalleri durdurmalıdırlar (bu daima önerilmektedir).

10.10. Evreler ve Çatallaşmak

Bir çok evreli POSIX süreci **fork** işlevinin çağırdığında ne olması gerektiği kesin değildir. Çatallaşma sırasında doğru çalışan ancak **fork** işlevini kullanmayan kodlar yazmanız gerekebilir. **fork** ve **pthread_once** gibi bazı kütüphane oluşumları ile standart G/Ç akımları arasındaki etkileşime dikkat etmelisiniz.

fork süreçteki bir evre tarafından çağrıldığında, çağıran sürecin bir kopyası olan yeni bir süreç oluşturur. Belirli sistem nesnelerinin kopyalanmasına ek olarak, üst sürecin bellek alanlarının anlık görünümünü alır ve alt süreçte özdeş alanlar oluşturur. Olayı biraz daha karmaşıklaştırmak için, iki veya daha fazla evre kendi içlerinde de çatallaşarak iki veya daha fazla alt süreç oluşturabilirler.

Alt süreç üst sürecinin adres alanının bir kopyasına sahiptir, fakat evrelerinden hiçbirini miras almaz. Alt sürecin işletilmesi **fork** işlevinden 0 değeriyle dönen bir evre ile sağlanır; öyle ki bu alt süreçteki tek evredir. Çatallaşma sırasında evreler miras alınmadığı için bazı sorunlar ortaya çıkar. **fork**'un çağrılması sırasında, üst süreçteki **fork**'u çağıran evre dışındaki evreler kodun kritik bölgelerini işletiyor olabilirler. Sonuç olarak, alt süreç iyi tanımlanmış durumda olmayan nesnelerin kopyasını alabilir. Bu potansiyel sorun yazılımın bütün bileşenlerini etkiler.

Alt süreçte kullanılmaya devam edecek herhangi bir yazılım bileşeni **fork** süresince durumunu doğru bir şekilde ele almalıdır. Bu amaçla, POSIX arayüzü özel bir işlev olan **pthread_atfork**'u, **fork** içinden çağrılan işleyici işlevlere göstericiler kurmak için sağlar.

```
int pthread_atfork(void (*hazırla)(void),                                    işlev
                   void (*üstsüreç)(void),
                   void (*altsüreç)(void))
```

pthread_atfork işlevi **fork** ile yeni bir süreç oluşturmadan hemen önce ve oluşturulduktan hemen sonra çağrılan işleyici işlevleri kaydeder. *hazırla* işleyicisi yeni bir süreç oluşturulmadan önce üst süreçten çağrılacaktır. *üstsüreç* işleyicisi **fork** dönmeden az önce üst süreçten çağrılacaktır. *altsüreç* işleyicisi **fork** dönmeden az önce alt süreçten çağrılacaktır.

pthread_atfork başarı halinde 0, hata halinde sıfırdan farklı bir hata kodu döndürür.

hazırla, *üstsüreç* ve *altsüreç* işleyicilerinden bir veya daha fazlası **NULL** olarak verilebilirler, bu o noktada bir işleyiciye ihtiyaç olmadığı anlamına gelir.

pthread_atfork birçok işleyici kümesi kurmak için birçok kez çağrılabilir. **fork** anında, *üstsüreç* ve *altsüreç* işleyicileri FIFO sırasıyla çağrılırken (ilk eklenen, ilk çağrılır), *hazırla* işleyicileri LIFO sırasında çağrılırlar (**pthread_atfork** ile son eklenen, **fork**'tan önce ilk çağrılır).

Eğer bu işleyicileri kaydetmek için yeterli bellek mevcut değilse, **pthread_atfork** başarısız olur ve **ENOMEM** döndürür. Aksi takdirde 0 döndürür.

fork ve **pthread_atfork** işlevlerine işleyiciler bağlamında çok katılışlı işlev gözüyle bakılmamalıdır. Eğer **fork** içinden çağrılan **pthread_atfork** işleyicisi **pthread_atfork** veya **fork** çağırırsa, davranışı bilinemez.

Çatallaşmada üçlü işleyicilerin kaydı atomik bir işlemdir. Eğer çatallaşma ile aynı zamanda yeni işleyiciler kaydedildiyse ya her üç işleyici de çağrılacaktır ya da hiçbiri.

İşleyiciler alt süreçler tarafından miras alınır ve **exec** ile yeni bir süreç görüntüsü yüklemekten başka onları kaldırmanın hiçbir yolu yoktur.

pthread_atfork'un amacını anlamak için, **fork**'un geçerli kilit durumlarıyla muteksler dahil bütün bellek alanının kopyasını çıkardığını hatırlayınız, fakat sadece çağıran evreyi: diğer evreler alt süreçte çalışmıyordu. **fork**'tan sonra muteksler kullanılamaz ve alt süreçte **pthread_mutex_init** ile tekrar hazırlanmaları gerekir. Bu geçerli gerçeklemenin bir kısıtıdır ve gelecek sürümlerde olabilir de olmayabilir de.

Bundan kaçınmak için, **pthread_atfork** ile şu şekilde işleyiciler kurun: *hazırla* işleyicisi muteksleri kilitler (sırayla) ve *üstsüreç* işleyicisi muteks kilitlerini açar. *altsüreç* işleyicisi muteksleri ve koşul değişkenleri gibi diğer eşzamanlama nesnelerini **pthread_mutex_init** kullanarak sıfırlamalıdır.

Çatallaşmadan önce global mutekslerin kilitlenmesi, bu mutekslerce korunan diğer bütün evrelerin kodun kritik bölgesindeki kilitlerinden kurtulduğundan emin olunmasını sağlar. Bu nedenle ne zaman **fork** üst sürecin adres alanının anlık görüntüsünü alsa, o görüntü geçerli, tutarlı veri içerir. Alt süreçteki eşzamanlama nesnelerinin sıfırlanması, üst sürecin evrelenme alt sistemindeki yapıların temizlendiğini temin eder. Örneğin, bir muteks, kilit

için bekleyen bir evre bekleme sırası miras almıştır; bu bekleme sırası alt süreçte bir şey ifade etmez. Muteksin ilklendirilmesi buna dikkat ister.

10.11. Akımlar ve Çatallaşma

GNU standart G/Ç kütüphanesi, bütün standart C FILE nesnelerinin içteki bağlı listesini koruyan bir iç mutekse sahiptir. **fork** sırasında bu mutekse dikkat edilerek, alt süreç listenin sağlam bir kopyasını alabilir. Bu **fopen** işlevi ve ilgili akım oluşturma işlevlerinin alt süreçte doğru çalışmasına imkan verir, çünkü bu işlevler listeye ekleme ihtiyacı duyarlar.

Tek tek akım kilitleri tam olarak dikkate alınmazlar. Bu yüzden çok evreli uygulamalarda **fork** kullanımında özel önlemler alınmadığı sürece, alt süreç üst süreçten miras aldığı akımları güvenle kullanma imkanı bulamayabilirler. Genellikle, üst süreçte verilen ve alt süreçte kullanılacak herhangi bir akım için, uygulamada **fork** çağrıldığında akımın başka bir evre tarafından kullanılmamasının temin edilmesi gerekir. Aksi takdirde akım nesnesinin tutarsız bir kopyası üretilmiş olur. Bunu garantilemenin kolay yolu **fork** çağırmadan önce **flockfile**'ı akımı kilitlemek için kullanmak ve üst süreçte **funlockfile** ile kilidini açmaktır. Alt süreçte başka özel birşey yapılmasına gerek yoktur, çünkü kütüphane içte bütün akım kilitlerini sıfırlamaktadır.

Unutmayınız ki akım kilitleri üst ve alt süreçler arasında paylaşılmazlar. Örneğin, **stdout** akımının uygun bir şekilde ele alındığını ve alt süreçte güvenle kullanılabileceğini temin etseniz de akım kilitleri üst ve alt süreçler arasında bir dışlama yöntemi sağlamazlar. Eğer her iki süreç **stdout**'a yazıyorsa, **flockfile** veya örtük kilitler uygulanmazsa karmakarışık bir çıktı ortaya çıkabilir.

Ayrıca bu hazırlıklar bir GNU uzantısıdır; diğer sistemler bir çok–evreli sürecin alt sürecinde kullanılacak akımlar için bunları sağlayamayabilirler. POSIX sadece standart G/Ç dahil kütüphanenin büyük kısmını dışlayan zamanuyumsuz güvenli işlevleri kullanarak kendini sınırlayan böyle bir alt sürece ihtiyaç duyar.

10.12. Çeşitli Evre İşlevleri

```
pthread_t pthread_self(void)                                          işlev
```

pthread_self çağıran süreç için evre tanıtıcısını döndürür.

```
int pthread_equal(pthread_t evre1,                                    işlev
                  pthread_t evre2)
```

pthread_equal iki evre tanıtıcısının aynı evreye belirtip belirtmediğini saptar.

Eğer *evre1* ve *evre2* aynı evreyi belirtiyorsa sıfırdan farklı bir değer döndürülür. Aksi takdirde 0 döndürülür.

```
int pthread_detach(pthread_t evre)                                    işlev
```

pthread_detach işlevi *evre* evresini ayrık duruma koyar. Bu, *evre* tarafından harcanan bellek kaynaklarının *evre* sonlandığında hemen serbest bırakılacağını garantiler. Ancak, bu diğer evrelerin *evre*'nin sonlanmasında **pthread_join** kullanarak eşzamanlanmalarını engeller.

Bir evre **detachstate** özelliği **pthread_create** işlevine verilerek ilk başta ayrık oluşturulabilir. Buna karşın **pthread_detach** evrelerin birleşimci durumda oluşturulmalarını sağlar ve daha sonra ayrık duruma konulması gerekir.

pthread_detach tamamlandıktan sonra *evre* üzerinde **pthread_join** uygulama teşebbüsleri başarısız olur. Eğer **pthread_detach** çağrıldığında diğer bir evre *evre* evresiyle birleşiyorsa, **pthread_detach** birşey yapmaz ve *evre*'yi birleşimci durumda bırakır.

Başarı halinde 0 döndürülür. Hata halinde, aşağıdaki hata kodlarından biri döndürülür:

ESRCH
: Belirtilen *evre* evresi bulunamadı

EINVAL
: *evre* evresi zaten ayrık durumda

```
void pthread_kill_other_threads_np(void)                                    işlev
```

pthread_kill_other_threads_np taşınabilir olmayan bir LinuxThreads oluşumudur. Yazılım içindeki çağıran evre hariç bütün evrelerin hemen sonlanmasına neden olur. Bir evre **exec** işlevlerinden birini, örneğin **execve** çağırmadan az önce çağrılmak üzere tasarlanmıştır.

Diğer evrelerin sonlandırılması **pthread_cancel** ile yapılmaz ve iptal mekanizması tamamen atlanır. Bu yüzden geçerli iptal durumu ve iptal türü ayarları dikkate alınmaz ve temizlik işleyicileri sonlandırılan evrelerde işletilmez.

POSIX 1003.1c'ye göre, evrelerden birindeki başarılı bir **exec*** yazılımdaki diğer bütün evreleri otomatik olarak sonlandırmalıdır. Bu davranış henüz LinuxThreads içinde uygulanmamıştır. **pthread_kill_other_threads_np**'in **exec***'den önce çağrılması neredeyse aynı davranışı gösterir, tabii ki eğer **exec*** sonunda başarısız olmadıysa, o zaman zaten diğer bütün evreler sonlandırılmıştır.

```
int pthread_once(pthread_once_t *birkerelik,                                işlev
                 void           (*ilklendirme_yordamı) (void))
```

pthread_once'ın amacı ilklendirme kodunun en çok bir kere işletilmesini temin etmektir. *birkerelik* argümanı **PTHREAD_ONCE_INIT** ile durağan olarak ilklendirilmiş bir statik ya da extern değişkeni gösterir.

birkerelik argümanıyla **pthread_once** ilk çağrıldığında, *ilklendirme_yordamı* yordamını argümansız çağırır ve *once_control* değişkeninin değerini ilklendirmenin yapıldığını belirtmek için değiştirir. **pthread_once**'a aynı **birkerelik** argümanıyla yapılacak tekrarlanan çağrılar birşey yapmazlar.

Eğer bir evre *ilklendirme_yordamı* işletilirken iptal edilirse *birkerelik* değişkeninin durumu sıfırlanır, böylece **pthread_once**'a yapılacak sonraki çağrılar yordamı tekrar çağıracaktır.

Eğer bir veya daha fazla evre, süreç tarafından **pthread_once** ilklendirme yordamlarını işletirken çatallaşırsa, kendi *birkerelik* değişkenlerinin durumları alt süreçte sıfırlanmış olarak görünürler, böylece eğer alt süreç **pthread_once** çağırırsa, yordamlar işletilir.

pthread_once hep 0 döndürür.

```
int pthread_setschedparam(pthread_t                hedef_evre,              işlev
                          int                      ilke,
                          const struct sched_param *param)
```

pthread_setschedparam işlevi *hedef_evre* için *ilke* ve *param* ile belirtildiği gibi zamanlama parametrelerini ayarlar. *ilke*, **SCHED_OTHER** (düzenli, gerçek zamanlı olmayan zamanlama), **SCHED_RR** (gerçek zamanlı, döner turnuva) veya **SCHED_FIFO** (gerçek zamanlı, ilk giren ilk çıkar) olabilir. *param* gerçek zamanlı ilkeler için zamanlama önceliğini belirtir. Zamanlama ilkeleri hakkında daha fazla bilgi için bkz. *Sürecin İşlemci Önceliği ve Zamanlama* (sayfa: 578)

Gerçek zamanlı zamanlama ilkeleri **SCHED_RR** ve **SCHED_FIFO** sadece süper kullanıcı haklarına sahip süreçler için kullanılabilir.

Başarı halinde, **pthread_setschedparam** 0 döndürür. Hata halinde aşağıdaki hata kodlarından birini döndürür:

EINVAL
: *ilke* **SCHED_OTHER**, **SCHED_RR**, **SCHED_FIFO**'dan biri değil veya *param* ile belirtilen öncelik değeri belirtilen ilkeye göre geçerli değil

EPERM
: Gerçek zamanlı zamanlama istendi ancak çağıran süreç yeterli izinlere sahip değil

ESRCH
: *hedef_evre* geçersiz veya sonlandırılmış

EFAULT
: *param* süreç belleği dışında bir yeri gösteriyor

```
int pthread_getschedparam(pthread_t           hedef_evre,                    işlev
                          int                 *ilke,
                          struct sched_param  *param)
```

pthread_getschedparam *hedef_evre* evresi için zamanlama ilkesini ve parametrelerini elde eder, *ilke* ve *param* ile gösterilen yerlerde saklar.

pthread_getschedparam başarı halinde 0 döndürür, hata halinde aşağıdaki hata kodlarından birini döndürür:

ESRCH
: *hedef_evre* geçersiz veya sonlandırılmış

EFAULT
: *ilke* veya *param* süreç bellek alanı dışında bir yeri gösteriyor

```
int pthread_setconcurrency(int seviye)                                        işlev
```

pthread_setconcurrency kullanıcı evrelerinin çekirdek evrelerine eşleştirme konusundaki eksiklerden dolayı LinuxThreads'de kullanılmaz. Kaynak uyumluluğu için bulunmaktadır. *seviye* değerini saklar, böylece sonraki **pthread_getconcurrency** çağrılarında döndürülebilir. Başka bir hareket yapmaz.

```
int pthread_getconcurrency()                                                  işlev
```

pthread_getconcurrency kullanıcı evrelerinin çekirdek evrelerine eşleştirme konusundaki eksiklerden dolayı LinuxThreads'de kullanılmaz. Kaynak uyumluluğu için bulunmaktadır. Ancak, **pthread_setconcurrency**'e yapılan son çağrıda belirlenen değeri döndürür.

XXVII. İş Denetimi

İçindekiler

İş denetimi bir kullanıcıya tek bir ***sisteme giriş oturumu*** içinde çok sayıda ***süreç grubu*** (ya da ***iş***) arasında hareket imkanı veren bir protokoldür. İş denetimi oluşumları çoğu süreci işlerini özdevinmli olarak yapmaları için ayarlar ve bunun olması için bu yazılımların iş denetimi ile ilgili hiçbir şey yapmaları gerekmez. Dolayısıyla, eğer bir kabuk ya da bir oturum açma uygulaması yazmıyacaksanız bu oylumda bahsedilenleri yoksayabilirsiniz.

Bu oylumda bahsedilenleri daha iyi anlayabilmek için *süreç oluşturma* (sayfa: 686) ve *sinyal işleme* (sayfa: 601) ile ilgili kavramlar hakkında bilgi sahibi olmanız gerekir.

1. İş Denetimi Kavramları

Bir etkileşimli kabuğun temel amacı kullanıcının uçbiriminden komutları okumak ve bu komutlarla belirtilen yazılımları çalıştırarak süreçler oluşturmaktır. Kabuk bunu **`fork`** (*Bir Sürecin Oluşturulması* (sayfa: 687)) ve **`exec`** (*Bir Dosyanın Çalıştırılması* (sayfa: 688)) işlevlerini kullanarak yapabilir.

Tek bir komut sadece bir süreci çalıştırmalı gibi düşünülebilir ama çoğunlukla bir komut çeşitli süreçleri kullanır. Bir kabuk komutunda **`|`** işlecini kullanarak, doğrudan kendi süreçleri içinde çeşitli uygulamaları çalıştırabilirsiniz. Ama sadece tek bir uygulamayı çalıştırmak istemiş olsanız bile bu uygulama dahili olarak çok sayıda süreci kullanabilir. Örneğin, **`cc -c foo.c`** gibi tek bir komut (normalde iki olmasına rağmen) genellikle dört süreç gerektirir. Hele bir **`make`** komutunu vardır ki, onun işi ayrı süreçler halinde başka uygulamaları çalıştırmaktır.

Tek bir komuta bağlı olarak çalışan süreçlere ***süreç grubu*** ya da ***iş*** denir. Bu hepsinin aynı anda çalışmasından dolayıdır. Örneğin, **`C-c`** tuşlayıp **`SIGINT`** sinyali göndererek önalan süreç grubundaki tüm süreçleri sonlandırabilirsiniz.

Bir ***oturum*** geniş bir süreçler grubudur. Normalde tüm süreçler tek bir oturum açılışında aynı oturuma ait olarak oluşturulurlar.

Her süreç bir süreç grubuna aittir. Bir süreç oluşturulduğu zaman, onun ata süreci ile aynı süreç grubunun ve oturumun bir üyesi haline gelir. Onu aynı oturuma ait bir süreç grubu olan başka bir süreç grubuna **`setpgid`** işlevini kullanarak koyabilirsiniz.

Bir süreci başka bir oturuma yerleştirmenin tek çaresi onu yeni bir oturumun ilk süreci ya da **setsid** işlevini kullanarak bir ***oturum lideri*** yapmaktır. Bu işlem ayrıca oturum liderini yeni bir süreç grubuna yerleştirir ve artık onu bu süreç grubundan tekrar dışarı taşıyamazsınız.

Çoğunlukla, yeni oturumları sistemin login komutu oluşturur ve oturum lideri kullanıcının oturum açma kabuğunu çalıştıran süreç olur.

İş denetimi desteği olan bir kabuk, her an uçbirimi kullanabilecek işi denetleyecek düzenlemeyi yapmalıdır. Aksi takdirde uçbirimden okumayı deneyen çok sayıda iş olabilir ve kullanıcı tarafından yazılan girdileri hangi sürecin alacağı konusunda karışıklık çıkar. Bundan kaçınmak için kabuk, bu oylumda bahsedilen protokolü kullanarak uçbirim sürücüsü ile işbirliği yapmalıdır.

Kabuk, bir defada sadece bir sürece denetim uçbiriminde sınırsız erişim verebilir. Denetim uçbirimine sınırsız erişim yapabilen sürece ***önalan işi*** denir. Kabuk tarafından yönetilen ve uçbirime bu tür bir erişimi olmayan diğer süreç gruplarına ise ***artalan işleri*** denir.

Eğer artalan işlerinden biri denetim uçbiriminden okuma yapmak ihtiyacı duyarsa, uçbirim sürücüsü tarafından *durdurulur*; eğer **TOSTOP** kipi etkinse, yazmak istediğinde de durdurulur. Bir önalan işini bir kullanıcı *SUSP karakterini* (sayfa: 454) kullanarak durdurabileceği gibi bir süreç de bir **SIGSTOP** sinyali göndererek durdurabilir. İşler durduğunda uyarmak, bunlar hakkında kullanıcıyı uyarmak ve durdurulan işlerin kullanıcı ile etkileşimli olarak sürdürülmesini ve artalan işleri ile önalan işi arasında geçişi mümkün kılmak için mekanizmalar sağlamak kabuğun sorumluluğudur.

Denetim uçbiriminde G/Ç işlemleri hakkında daha fazla bilgi edinmek için *Denetim Uçbirimine Erişim* (sayfa: 717) bölümüne bakınız.

2. İş Denetimi İsteğe Bağlıdır

İşletim sistemlerinin bazıları iş denetimini desteklemez. GNU sistemi iş denetimini desteklese de, kütüphaneyi kullandığınız diğer sistemlerde sistemin kendisi iş denetimini desteklemeyebilir.

Sistemin iş denetimini destekleyip desteklemediğini **_POSIX_JOB_CONTROL** makrosunu kullanarak derleme sırasında sınayabilirsiniz. Bkz. *Sistem Seçenekleri* (sayfa: 785).

İş denetimi desteklenmiyorsa, her oturumda daima önalanda çalışan sadece bir süreç grubu olabilir. Ek süreç grupları oluşturan işlevler **ENOSYS** hata durumu ile başarısız olurlar.

Çeşitli *iş denetim sinyallerini isimlendiren makrolar* (sayfa: 608) iş denetimi desteği olmasa bile tanımlanır. Ancak sistem bu sinyalleri hiçbir zaman üretmez ve bir iş denetim sinyalinin gönderilmesi, incelenmesi ya da eylemlerinin belirtilmesi ya hatalarla raporlanır ya da bir şeye sebep olmaz.

3. Bir Sürecin Denetim Uçbirimi

Bir sürecin özniteliklerinden biri de kendi denetim uçbirimidir. **fork** ile oluşturulan alt süreçler denetim uçbirimlerini kendilerini oluşturan süreçten miras alırlar. Bu yolla, bir oturumdaki tüm süreçler denetim uçbirimini oturum liderinden miras alırlar. Bir oturum liderinin bir uçbirimin denetimine sahip olması, onu, bu uçbirimin ***denetçi süreci*** haline getirir.

Genelde, siz sisteme oturum açarken işlem sizin yerinize sistem tarafından yapıldığından bir denetim uçbiriminin ayrılmasında kullanılan mekanizma hakkında endişelenmeniz gerekmez.

Bir süreç, **setsid** çağrısı ile yeni bir oturumun lideri haline gelerek kendi denetim uçbiriminden kopar. Bkz. *Süreç Grubu İşlevleri* (sayfa: 729).

4. Denetim Uçbirimine Erişim

Bir denetim uçbiriminin önalan işindeki süreçlerin uçbirime erişimi sınırlandırılmamıştır. Bu kısımda, artalandaki bir sürecin kendi denetim uçbirimine erişmeye çalıştığı zaman neler olduğu ayrıntılı olarak açıklanacaktır.

Artalandaki bir süreç kendi denetim uçbiriminden okumaya çalıştığı zaman, süreç grubuna çoğu kez bir **SIGTTIN** sinyali gönderilir. Bu normalde süreç grubundaki tüm süreçlerin durmasına sebep olur (sinyali yakalamıyorlarsa ve kendilerini durdurmamışlarsa). Ancak, eğer okuyan süreç bu sinyali yoksayıyor ya da engelliyorsa, **read** işlevi bir **EIO** hatasıyla başarısız olacaktır.

Benzer olarak, artalandaki bir süreç kendi denetim uçbirimine yazmaya çalıştığı zaman, öntanımlı davranış süreç grubuna bir **SIGTTOU** sinyali göndermektir. Ancak, bu davranış yerel kip seçeneklerinden **TOSTOP** biti tarafından değiştirilir (bkz. *Yerel Kipler* (sayfa: 451)). Bu bit etkin değilse (öntanımlı böyledir), denetim uçbirimine yazma işlemine bir sinyal gönderilmeksizin daima izin verilir. Yazmaya ayrıca, yazan süreç tarafından **SIGTTOU** sinyali yoksayılıyor ya da engelleniyorsa da izin verilir.

Diğer uçbirim işlemlerinden çoğu birer okuma ya da yazma işlemi olarak ele alınır. (Her işlemin açıklamasında hangisi olduğu belirtilmiştir).

İlkel **read** ve **write** işlevleri hakkında daha fazla bilgi için *Girdi ve Çıktı İlkelleri* (sayfa: 308) bölümüne bakınız.

5. Öksüz Süreç Grubu

Bir denetim süreci sonlandığında onun uçbirimi serbest kalır ve yeni bir oturum başlatmaya hazır hale gelir. (Bu işlem gerçekte başka bir kullanıcının sistemde oturum açabilmesi demektir.) Eğer eski oturumdan kalan bir süreç hala uçbirimi kullanmaya çalışırsa bu sorunlara sebep olur.

Bu sorunlardan kaçınmak için, oturumun lideri olan süreç sonlansa bile süreç, ***öksüz süreç grubu*** adı verilen bir süreç grubuna alınarak çalışmasına devam etmesi sağlanır.

Bir süreç grubu öksüz kaldığında, süreçlerine **SIGHUP** sinyali gönderilir. Diğer yandan bir uygulama bu sinyali yoksayar ya da onun için bir sinyal eylemci kurarsa (*Sinyal İşleme* (sayfa: 601)), bu süreci denetleyen süreç sonlandıktan sonra bile süreç, öksüz süreç grubuna alınarak çalışmasına devam edebilir; ancak ne olursa olsun bundan sonra uçbirime erişemez.

6. Bir İş Denetim kabuğunun Gerçeklenmesi

Bu kısımda, iş denetimini gerçekleştirmek için bir kabuğun neler yapması gerektiği, bahsi geçen kavramları örnekleyen kapsamlı bir yazılım örneği de verilerek açıklanacaktır.

6.1. Kabuk için Veri Yapıları

Bu kısımdaki yazılım örneklerinin hepsi basit bir kabuk yazılımının parçalarıdır. Bu bölümdeki veri yapıları ve işlevler örnekte kullanmak içindir.

Örnek kabukta başlıca iki veri yapısı bulunur. **job** türü, birbirine borularla bağlanan altsüreçlerden oluşan bir iş ile ilgili bilgileri içerir. **process** türü ise, tek bir altsüreç hakkında bilgi içerir. Bu veri yapılarının bildirimleri:

```
/* process yapısı tek bir süreçle ilgilidir.  */
typedef struct process
{
  struct process *next;       /* boruhattındaki sonraki süreç */
  char **argv;                /* exec için */
  pid_t pid;                  /* süreç kimliği */
  char completed;             /* süreç tamamlanmışsa doğru */
  char stopped;               /* süreç durmuşsa doğru */
  int status;                 /* raporlanan durum değeri */
} process;
```

```
/* job yapısı boruhattıyla bağlı süreçlerden oluşur.  */
typedef struct job
{
  struct job *next;           /* sonraki etkin iş */
  char *command;              /* komut satırı, iletiler için */
  process *first_process;     /* bu işteki süreçlerin listesi */
  pid_t pgid;                 /* süreç grubu kimliği */
  char notified;              /* duran iş için kullanıcıya uyarı varsa doğru */
  struct termios tmodes;      /* kayıtlı uçbirim kipleri */
  int stdin, stdout, stderr;  /* standart g/ç kanalları */
} job;

/* Etkin iş bir listeyle ilintilenir.  Bu onun başıdır.   */
job *first_job = NULL;
```

job nesneleri üzerinde işlem yapan bazı işlevler:

```
/* pgid ile belirtilen etkin işi bul.  */
job *
find_job (pid_t pgid)
{
  job *j;

  for (j = first_job; j; j = j->next)
    if (j->pgid == pgid)
      return j;
  return NULL;
}

/* İşteki tüm süreçler durmuş ya da tamamlanmışsa "doğru" ile dön.  */
int
job_is_stopped (job *j)
{
  process *p;

  for (p = j->first_process; p; p = p->next)
    if (!p->completed && !p->stopped)
      return 0;
  return 1;
}

/* İşteki tüm süreçler tamamlanmışsa "doğru" ile dön.  */
int
job_is_completed (job *j)
{
  process *p;

  for (p = j->first_process; p; p = p->next)
    if (!p->completed)
      return 0;
  return 1;
}
```

6.2. Kabuğun İlklendirilmesi

Normalde iş denetimi yapan bir kabuk başlatıldığında, kendi iş denetimini yapan başka bir kabuk tarafından

çalıştırılmışsa dikkatli olmak zorundadır.

Etkileşimli olarak çalışan bir altkabuk kendi iş denetimini etkinleştirmeden önce kendini çalıştıran kabuk tarafından önalana mı yerleştirildiğine bakmak zorundadır. **getpgrp** işlevi ile kendi süreç grup kimliğini öğrenip, bunu kendi denetim uçbiriminin önalan işinin süreç grup kimliği (**tcgetpgrp** işlevi ile öğrenebilir) ile karşılaştırarak yapar.

Eğer altkabuk bir önalan işi olarak çalışmıyorsa, kendi süreç grubuna bir **SIGTTIN** sinyali göndererek kendini durdurmalıdır. Kendini keyfi olarak önalana yerleştiremez; bunu kendini çalıştıran kabuğa kullanıcının söylemesini beklemek zorundadır. Eğer alt kabuk yine de devam ederse bu sınamayı tekrarlayıp hala önalanda değilse, kendini tekrar durdurmalıdır.

Bir alt kabuk kendini çalıştıran kabuk tarafından önalana yerliştirildikten sonra kendi iş denetimini etkinleştirebilir. Bunu **setpgid** işlevini çağırıp kendini, kendi süreç grubuna koyduktan sonra **tcsetpgrp** çağrısıyla bu süreç grubunu önalana yerleştirerek yapar.

Bir kabuk iş denetimini etkinleştirdiğinde, kendini tüm iş denetimi durdurma sinyallerini yoksaymaya ayarlayıp kazaen kendi kendini durdurma şansını ortadan kaldırmalıdır. Bunu tüm durdurma sinyalllerine eylem olarak **SIG_IGN** atayarak yapabilirsiniz.

Etklileşimsiz olarak çalışan bir kabuk iş denetimini destekleyemez ve desteklememelidir. Kendi süreç grubunda oluşturulan tüm süreçleri bırakmalıdır; bu onu etkileşimsiz kabuk yapar. Onun alt süreçleri kendini çalıştıran kabuğun tek tek işleri olarak ele alınır. Bunu yapmak kolaydır, iş denetim ilkellerini kullanmamak yeterlidir, ama bunu kabuğa yaptırmayı unutmamalısınız.

Burada bunun örnek kabuğumuz tarafından ilklendirme kodunda nasıl yapıldığı gösterilmiştir.

```
/* Kabuğun özniteliklerini hatırlayalım.  */

#include <sys/types.h>
#include <termios.h>
#include <unistd.h>

pid_t shell_pgid;
struct termios shell_tmodes;
int shell_terminal;
int shell_is_interactive;


/* Başlamadan önce kabuğun önalan işi olarak etkileşimli
   çalışacağından emin olalım. */

void
init_shell ()
{

  /* Etkileşimli mi çalışıyor acaba.  */
  shell_terminal = STDIN_FILENO;
  shell_is_interactive = isatty (shell_terminal);

  if (shell_is_interactive)
    {
      /* Önalana geçinceye kadar döngüde kalsın.  */
      while (tcgetpgrp (shell_terminal) != (shell_pgid = getpgrp ()))
        kill (- shell_pgid, SIGTTIN);

      /* Etkileşimli ve iş denetimi sinyallarini yoksayalım.  */
```

```
      signal (SIGINT, SIG_IGN);
      signal (SIGQUIT, SIG_IGN);
      signal (SIGTSTP, SIG_IGN);
      signal (SIGTTIN, SIG_IGN);
      signal (SIGTTOU, SIG_IGN);
      signal (SIGCHLD, SIG_IGN);

      /* Kendimizi kendi süreç grubumuza koyalım.  */
      shell_pgid = getpid ();
      if (setpgid (shell_pgid, shell_pgid) < 0)
        {
          perror ("Kabuk kendini kendi süreç grubuna yerleştiremedi");
          exit (1);
        }

      /* Uçbirim denetimini ele geçirelim.  */
      tcsetpgrp (shell_terminal, shell_pgid);

      /* Kabuğun öntanımlı uçbirim öznitelikerini saklayalım.  */
      tcgetattr (shell_terminal, &shell_tmodes);
    }
}
```

6.3. İşlerin Başlatılması

Kabuk kendi denetim uçbiriminde iş denetimi uygulamak için sorumluluğu aldıktan sonra, kullanıcı tarafından yazılan komutlara yanıt olarak işleri başlatabilir.

Bir süreç grubunda süreçleri oluşturmak için *Süreç Oluşturma Kavramları* (sayfa: 686) bölümünde açıklandığı gibi **fork** ve **exec** işlevleri kullanılabilir. Çok sayıda alt süreç karışık olduğundan, ister istemez, bazı şeyler biraz karmaşıklaşır, bu bakımdan bunları doğru sırada yapmak önem kazanır. Aksi takdirde istenmeyen yarış koşulları ortaya çıkabilir.

Süreçlerle ilgili ata–çocuk birliktelik ağacının yapılanmasında yerine göre iki seçimden birini kullanabilirsiniz. Ya, süreç grubundaki tüm süreçleri kabuk sürecinin çocukları yaparsınız ya da, süreç grubundaki bir süreci gruptaki diğer süreçlerin atası yaparsınız. Tıpkı, bu oylumun tamamındaki örnekleri basit bir kabuk yazılımının parçaları olarak vermemiz gibi, çünkü bu, bazı şeylerin muhasebesini yapmayı koloylaştırıyor.

Her süreç çatallandığında, **setpgid** çağrısı ile kendini yeni süreç grubuna koymalıdır; bkz. *Süreç Grubu İşlevleri* (sayfa: 729). Süreç grubundaki ilk süreç ***süreç grubunun lideri*** haline gelir ve onun süreç kimliği de ***süreç grup kimliği*** haline gelir.

Kabuk buna ek olarak kendi alt süreçlerinini yeni süreç grubuna koymak için her süreç için ayrı ayrı **setpgid** çağrıları yapmalıdır. Olası bir zamanlama sorunu nedeniyle bunun böyle olması gerekir: her alt süreç kendi alt sürecini çalıştırmaya başlamadan önce süreç grubuna konulmak ve dolayısıyla kabuk çalışamaya devam etmeden önce grubundaki alt süreçlerini bağlamak zorundadır. Eğer hem kabuk hem de alt süreci **setpgid** çağrısı yaparsa, hangi sürecin bunu önce yaptığına bakılmaksızın işler olması gerektiği gibi yürür.

Eğer iş bir önalan işi olarak başlatılmışsa, yeni süreç grubunun ayrıca **tcsetpgrp** kullanılarak denetim uçbiriminde önalana konulması gerekir. Tekrar, yarış koşullarından kaçınmak için bu işlem, hem kabuk tarafından hem de kabuğun alt süreçlerinin her biri tarafından yapılmalıdır.

Bundan sonra, her alt süreç kendi sinyal eylemlerini sıfırlamalıdır.

İlklendirme sırasında, kabuk süreci kendini iş denetim sinyallarini yoksaymaya ayarlamalıdır; bkz. *Kabuğun İlklendirilmesi* (sayfa: 719). Sonuç olarak, bir miras olarak her alt süreç oluşturulduğunda bu sinyalleri

yoksayacaktır. Bu kesinlikle istenmeyen bir durumdur, bu bakımdan kabuğun her alt süreci çatallandığında bu sinyaller **SIG_DFL**'a yani öntanımlı haline getirmelidir.

Kabuklar bu uzlaşımlara uyduğundan, uygulamalar kendini çalıştıran süreçten bu sinyalleri doğru eylemlerle aldıklarını kabul edebilirler. Fakat her uygulamanın durdurma sinyallerinin işlenmesini bozmama sorumluluğu vardır. SUSP karakterinin normal yorumunu iptal eden uygulamalar, kullanıcıya işi durdurmak için başka mekanizmalar sunmalıdır. Kullanıcı bu mekanizmayı çalıştırdığında yazılım sadece kendi sürecinde değil, kendi süreç grubuna da bir **SIGTSTP** sinyali göndermelidir. Bkz. *Başka Bir Sürece Sinyal Gönderme* (sayfa: 628).

Son olarak, her süreç normal yolla **exec** çağrısı yapmalıdır. Bu ayrıca standart girdi ve çıktı kanallarının yönlendirilmesinde önemlidir. Bunun nasıl yapılacağı *Tanıtıcıların Çoğullanması* (sayfa: 339) bölümünde açıklanmıştır.

Örnek kabuk yazılımından, bir uygulamayı çalıştırmakla yükümlü bir işlev aşağıda gösterilmiştir. Kabuk tarafından çatatallandıktan hemen sonra her alt süreç tarafından bu işlev çalıştırılır ve işlev asla dönmez.

```
void
launch_process (process *p, pid_t pgid,
                int infile, int outfile, int errfile,
                int foreground)
{
  pid_t pid;

  if (shell_is_interactive)
    {
      /* Süreci süreç grubuna koy ve uygunsa uçbirimi
         süreç grubuna ver. Olası yarış koşullarının oluşmasını
         engellemek için hem kabuk hem de her alt süreç
         tarafından bu yapılmalıdır.  */
      pid = getpid ();
      if (pgid == 0) pgid = pid;
      setpgid (pid, pgid);
      if (foreground)
        tcsetpgrp (shell_terminal, pgid);

      /* İş denetim sinyallerini öntanımlı eylemlerine ayarlayalım..  */
      signal (SIGINT, SIG_DFL);
      signal (SIGQUIT, SIG_DFL);
      signal (SIGTSTP, SIG_DFL);
      signal (SIGTTIN, SIG_DFL);
      signal (SIGTTOU, SIG_DFL);
      signal (SIGCHLD, SIG_DFL);
    }

  /* Standart girdi/çıktı kanallarını yeni sürece ayarlayalım.  */
  if (infile != STDIN_FILENO)
    {
      dup2 (infile, STDIN_FILENO);
      close (infile);
    }
  if (outfile != STDOUT_FILENO)
    {
      dup2 (outfile, STDOUT_FILENO);
      close (outfile);
    }
  if (errfile != STDERR_FILENO)
    {
```

```
      dup2 (errfile, STDERR_FILENO);
      close (errfile);
    }

  /* Yeni süreci çalıştıralım ve çıkalım.  */
  execvp (p->argv[0], p->argv);
  perror ("execvp");
  exit (1);
}
```

Kabuk etkileşimli olarak çalışmıyorsa, bu işlev süreç grubu ve sinyallerle ilgili olarak hiçbir şey yapmaz. İş denetimi yapmayan bir kabuğun kendi alt süreçlerini kendi süreç grubunda tuttuğunu hatırlayın.

Aşağıda, bir işi tam anlamıyla çalıştıran bir işleve yer verilmiştir. Alt süreç oluşturulduktan sonra, bu işlev yeni oluşturulan işi önalana ya da artalana koyacak bazı işlevleri çağırır; bunlar *Önalan ve Artalan* (sayfa: 724) bölümünde açıklanmıştır.

```
void
launch_job (job *j, int foreground)
{
  process *p;
  pid_t pid;
  int mypipe[2], infile, outfile;

  infile = j->stdin;
  for (p = j->first_process; p; p = p->next)
    {
      /* Set up pipes, if necessary.  */
      if (p->next)
        {
          if (pipe (mypipe) < 0)
            {
              perror ("pipe");
              exit (1);
            }
          outfile = mypipe[1];
        }
      else
        outfile = j->stdout;

      /* Alt süreci çatallayalım.  */
      pid = fork ();
      if (pid == 0)
        /* This is the child process.  */
        launch_process (p, j->pgid, infile,
                        outfile, j->stderr, foreground);
      else if (pid < 0)
        {
          /* Çatallama başarısız.  */
          perror ("fork");
          exit (1);
        }
      else
        {
          /* Bu, ata süreç.  */
          p->pid = pid;
          if (shell_is_interactive)
```

```
            {
              if (!j->pgid)
                j->pgid = pid;
              setpgid (pid, j->pgid);
            }
        }

      /* Boruları temizleyelim.  */
      if (infile != j->stdin)
        close (infile);
      if (outfile != j->stdout)
        close (outfile);
      infile = mypipe[0];
    }

  format_job_info (j, "işe başladı");

  if (!shell_is_interactive)
    wait_for_job (j);
  else if (foreground)
    put_job_in_foreground (j, 0);
  else
    put_job_in_background (j, 0);
}
```

6.4. Önalan ve Artalan

Kabuk tarafından bir iş önalanda başlatılırken kabuk tarafından hangi eylemlerin ele alınması gerektiğine ve bunun bir artalan işi başlatmaktan ne gibi farkları olduğuna bakalım.

Bir önalan işi başlatılırken, ilk olarak kabuk bir **tcsetpgrp** çağrısı yaparak ona denetim uçbiriminde erişim vermelidir. Bundan sonra, kabuk bu süreç grubundaki sürecin sonlanmasını ya da durmasını beklemelidir. Bu ayrıntılı olarak *İşlerin Durdurulması ve Sonlandırılması* (sayfa: 725) bölümünde anlatılmıştır.

Gruptaki tüm süreçler tamamlandığında ya da durduğunda, kabuk tekrar bir **tcsetpgrp** çağrısı yaparak kendi süreç grubu için uçbirim denetimini geri kazanmalıdır. Süreç grubuna, bir artalan işindeki G/Ç işleminden ya da kullanıcı tarafından tuşlanan bir SUSP karakterinden dolayı oluşan durdurma sinyalleri gönderildiğinde normal olarak işteki tüm süreçler durdurulur.

Önalan işi tuhaf bir durumda uçbirimde kalabilir, bu durumda kabuk devam etmeden önce kendi uçbirim kiplerini eski haline getirmelidir. İşin sadece durmuş olması durumunda, iş daha sonra devam edebileceğinden, kabuk önce o anki uçbirim kiplerini kaydetmelidir. Uçbirim kipleri ile ilgili işlemler için kullanılan **tcgetattr** ve **tcsetattr** işlevleri *Uçbirim Kipleri* (sayfa: 444) bölümünde anlatılmıştır.

Bunların hepsini yapan örnek kabuk işlevi:

```
/* j işini önalana koyalım. cont sıfırdan farklıysa,
   kayıtlı uçbirim kiplerini yerine koyalım ve
   biz engellemeden önce sürecin devam edebilmesi için
   süreç grubuna bir SIGCONT sinyali gönderelim.  */

void
put_job_in_foreground (job *j, int cont)
{
  /* İşi önalana koyalım.  */
  tcsetpgrp (shell_terminal, j->pgid);
```

```
  /* Gerekliyse, işe bir devamet sinyali gönderelim.  */
  if (cont)
    {
      tcsetattr (shell_terminal, TCSADRAIN, &j->tmodes);
      if (kill (- j->pgid, SIGCONT) < 0)
        perror ("kill (SIGCONT)");
    }

  /* Rapor vermesini bekleyelim.  */
  wait_for_job (j);

  /* Kabuğu tekrar önalana koyalım.  */
  tcsetpgrp (shell_terminal, shell_pgid);

  /* Kabuğa uçbirim kiplerini geri verelim.  */
  tcgetattr (shell_terminal, &j->tmodes);
  tcsetattr (shell_terminal, TCSADRAIN, &shell_tmodes);
}
```

Eğer süreç grubu bir artalan işi olarak başlatılıyorsa, kabuğun kendisi önalanda kalmalı ve uçbirimden komutları okumaya devam etmelidir.

Örnek kabukta, bir işi artalana koymak için çok fazla bir şey yapmak gerekmemiştir. Bu işlem için kullanılan işlev:

```
/* Bir işi artalana koyacağız. cont doğru ise,
   süreç grubuna devam etmesi için bir
   SIGCONT sinyali gönderelim.  */

void
put_job_in_background (job *j, int cont)
{
  /* Gerekliyse artalan işine bir devamet sinyali gönderelim.  */
  if (cont)
    if (kill (-j->pgid, SIGCONT) < 0)
      perror ("kill (SIGCONT)");
}
```

6.5. İşlerin Durdurulması ve Sonlandırılması

Bir önalan işi başlatıldığında, kabuk, işteki tüm süreçler durana ya da sonlanana kadar beklemelidir. Bunu **waitpid** çağrısı ile yapabilir; bkz. *Süreç Tamamlama* (sayfa: 690). Süreçlerin durması ya da sonlanması halinde durumu raporlamaları için **WUNTRACED** seçeneği kullanılır.

Kabuk bir artalan işinin kullanıcı tarafından durdurulması ya da sonlandırılmasının da raporlanmasını ayrıca beklemelidir; bu **waitpid** işlevinin **WNOHANG** seçeneği ile çağrılması ile yapılabilir. Bu sınamanın yapılacağı en iyi yer yeni bir komut isteminin hemen öncesidir.

Kabuk ayrıca, **SIGCHLD** sinyallerine bir *sinyal eylemci* (sayfa: 601) kurmuş bir alt süreç için durum bilgisi içeren bir eşzamansız uyarı alabilir.

Örnek kabuk yazılımında, **SIGCHLD** sinyalleri normal olarak yoksayılmaktadır. Bu, kabuğun zaman zaman değiştirdiği genel veri yapılarından kaynaklanan evresellik (reentrancy) sorunlarında kaçınmak içindir. Fakat belirli zamanlarda, kabuğun bu veri yapılarını kullanmadığı zamanlarda (örneğin, uçbirimden girdi beklerken), **SIGCHLD** sinyali için bir eylemciyi etkinleştirmeye ihtiyaç duyar. Eşzamanlı durum sınamaları yapmak için kullanılan işlev (bu durumda **do_job_notification** işlevi), bu eylemci tarafından ayrıca çağrılabilir.

Burada, örnek kabuk yazılımından işlerin durumunu sınayıp durumu kullanıcıya bildiren parçası görülmektedir:

```
/* waitpid tarafından döndürülen  süreç kimliğinin durumunu
   saklayalım. İstenen yapıldıysa 0 yoksa sıfırdan farklı
   bir değerle dönelim..  */

int
mark_process_status (pid_t pid, int status)
{
  job *j;
  process *p;

  if (pid > 0)
    {
      /* Süreç için kaydı güncelleyelim.  */
      for (j = first_job; j; j = j->next)
        for (p = j->first_process; p; p = p->next)
          if (p->pid == pid)
            {
              p->status = status;
              if (WIFSTOPPED (status))
                p->stopped = 1;
              else
                {
                  p->completed = 1;
                  if (WIFSIGNALED (status))
                    fprintf (stderr, "%d: %d sinyali ile sonlandırıldı.\n",
                             (int) pid, WTERMSIG (p->status));
                }
              return 0;
              }
      fprintf (stderr, "%d kimlikli bir süreç yok.\n", pid);
      return -1;
    }
  else if (pid == 0 || errno == ECHILD)
    /* Rapor verecek süreç yok.  */
    return -1;
  else {
    /* Diğer tuhaf hatalar.  */
    perror ("waitpid");
    return -1;
  }
}

/* Süreçleri beklemeden durum bilgilerinin varlığını sınayalım.  */

void
update_status (void)
{
  int status;
  pid_t pid;

  do
    pid = waitpid (WAIT_ANY, &status, WUNTRACED|WNOHANG);
  while (!mark_process_status (pid, status));
}

/* Belirtilen işteki tüm süreçleri beklerken
```

```
   durum bilgilerinin varlığını sınayalım.  */

void
wait_for_job (job *j)
{
  int status;
  pid_t pid;

  do
    pid = waitpid (WAIT_ANY, &status, WUNTRACED);
  while (!mark_process_status (pid, status)
         && !job_is_stopped (j)
         && !job_is_completed (j));
}

/* İş durumu ile ilgili bilgileri kullanıcıya sunmak için biçimleyelim.  */

void
format_job_info (job *j, const char *status)
{
  fprintf (stderr, "%ld (%s): %s\n", (long)j->pgid, status, j->command);
}

/* Durmuş ya da sonlandılılmış işler hakkında kullanıcıyı uyaralım.
   Sonlanmış işleri etkin iş listesinden kaldıralım.  */

void
do_job_notification (void)
{
  job *j, *jlast, *jnext;
  process *p;

  /* Alt sürecin durum bilgisini güncelleyelim.  */
  update_status ();

  jlast = NULL;
  for (j = first_job; j; j = jnext)
    {
      jnext = j->next;

      /* Tüm süreçler tamamlanmışsa, kullanıcıya işin tamamlandığını
         bildirelim ve onu etkin işler listesinden silelim.  */
      if (job_is_completed (j)) {
        format_job_info (j, "iş tamamdır");
        if (jlast)
          jlast->next = jnext;
        else
          first_job = jnext;
        free_job (j);
      }

      /* Duran işleri kullanıcıya bildirelim ve
         bunu bir daha yapmamak için onları imleyelim.  */
      else if (job_is_stopped (j) && !j->notified) {
        format_job_info (j, "iş durdu");
        j->notified = 1;
        jlast = j;
```

```
      }

      /* Hala sürmekte olan işler için bir şey söylemek gerekmez.  */
      else
        jlast = j;
    }
}
```

6.6. Duran İşlerin Sürdürülmesi

Kabuk durmuş bir işi onun süreç grubuna bir **SIGCONT** sinyali göndererek çalışmaya devam etmesini sağlayabilir. İşin önalanda sürdürülmesi durumunda, kabuk önce uçbirimde erişim vermek için bir **tcsetpgrp** çağrısı yapmalıdır. İş önalanda kaldığı yerden çalışmaya başladıktan sonra sanki iş önalanda ilk defa başlatılmış gibi kabuk işin tamamlanması ya da durmasını beklemek zorundadır.

Örnek kabuk yazılımında hem yeni oluşturulan hem de devam ettrilen işler aynı işlev çiftiyle, **put_job_in_foreground** ve **put_job_in_background** işleviyle izlenirler. Bu işlevlerin tanımları *Önalan ve Artalan* (sayfa: 724) bölümündeki örnek kod parçasında bulunabilir. Durmuş bir işin devam ettrilmesi durumunda *cont* argümanına sıfırdan farklı bir değer atanarak **SIGCONT** sinyalinin gönderilmesi ve uçbirim kiplerinin eski değerlerine getirilmesi sağlanır.

Burada durmuş işin tekrar çalışmaya devam etmesini sağlayan kod parçası gösterilmiştir:

```
/* Durmuş j işini tekrar çalışmaya başlasın diye imleyelim.  */

void
mark_job_as_running (job *j)
{
  Process *p;

  for (p = j->first_process; p; p = p->next)
    p->stopped = 0;
  j->notified = 0;
}

/* j işi devam etsin.  */

void
continue_job (job *j, int foreground)
{
  mark_job_as_running (j);
  if (foreground)
    put_job_in_foreground (j, 1);
  else
    put_job_in_background (j, 1);
}
```

6.7. Eksik Parçalar

Bu oylumda çeşitli örnekler halinde verilen kabuk yazılımı, yazılımın tamamı değildir. Yazılımın çok küçük bir kısmının örneklenmesi dışında **job** ve **program** veri yapılarının nasıl ayrıldığı ve ilklendirildiği de dahil olmak üzere, hemen hiçbir şey söylenmedi.

Gerçek kabukların çoğu, bir komut dili, değişkenler, kısaltmalar, ikameler, dosya isimleri üzerinde kalıp eşleme gibi pek çok desteği içeren oldukça karmaşık bir kullanıcı arayüzü içerir. Tüm bunların verilmesi için doğaldır

ki, burası yeri değildir. Bunun yerine süreç oluşturulması ile ilgili gerçekleme ve iş denetimi işlevlerinin böyle bir kabuk içinden çağrılmasını gösteren bir özet verdik.

Şimdiye kadar sunduğumuz ana konuları şöyle özetleyebiliriz:

`void` **`init_shell`** `(void)`
: Kabuğun dahili durumunun ilklendirir. Bkz.*Kabuğun İlklendirilmesi* (sayfa: 719).

`void` **`launch_job`** `(job *`*`j`*`, int` *`foreground`*`)`
: *j* işinin hem önalanda hem de artalanda başlatılması için. Bkz. *İşlerin Başlatılması* (sayfa: 721).

`void` **`do_job_notification`** `(void)`
: Sonlanmış ya da durmuş bir işin varlığını sınar ve raporlar. Eşzamanlı çağrılabileceği gibi **`SIGCHLD`** sinyallerinin eylemcisinden de çağrılabilir. Bkz. *İşlerin Durdurulması ve Sonlandırılması* (sayfa: 725).

`void` **`continue_job`** `(job *`*`j`*`, int` *`foreground`*`)`
: *j* işinin sürdürülmesini sağlar. Bkz. *Duran İşlerin Sürdürülmesi* (sayfa: 728).

Süphesiz, gerçek bir kabuk işlerin yönetilmesi için daha fazlasını gerektirir. Örneğin, tüm etkin işleri listeleyen ya da bir işe bir sinyal gönderen (**`SIGKILL`** gibi) komutlar içermesi faydalı olurdu.

7. İş Denetimi İşlevleri

Bu kısımda iş denetimi ile ilgili işlevlerin ayrıntılı açıklamalarına yer verilmiştir.

7.1. Denetim Uçbiriminin İsimlendirilmesi

Denetim uçbirimini açmakta kullanılabilecek dosya ismini almak için **`ctermid`** işlevini kullanabilirsiniz. GNU kütüphanesinde her zaman aynı dizgeyi döndürür: `"/dev/tty"`. Bu o an çalışmakta olan sürecin denetim uçbirimini ifade eden "sihirli" bir özel isimdir. Belli bir uçbirim aygıtının ismini bulmak için ise **`ttyname`** işlevini kullanabilirsiniz; bkz. *Uçbirimlerin Tanımlanması* (sayfa: 442).

`ctermid` işlevi `stdio.h` başlık dosyasında bildirilmiştir.

`char *`**`ctermid`**`(char *`*`dizge`*`)` işlev

> **`ctermid`** işlevi, sürecin denetim uçbiriminin dosya ismini içeren bir dizge ile döner. *dizge* bir boş gösterici değilse, en azından **`L_ctermid`** karakteri tutabilecek bir dizi olmalıdır; istenen dizge bu dizi içinde dönecektir. Aksi takdirde, işlevin sonraki çağrıları ile üzerine yazılabilen, durağan alanda ayrılmış dizgeye bir gösterici ile döner.
>
> Herhangi bir sebeple dosya ismi saptanamazsa bir boş dizge döner. Bir dosya ismi dönmüş olsa bile, bu dosyaya erişim garanti edilmez.

`int` **`L_ctermid`** makro

> Bu makronun değeri, **`ctermid`** işlevi ile döndürülen dosya isminin tutulacağı genişlikte bir dizgenin uzunluğunu ifade eder.

Ayrıca *Uçbirimlerin Tanımlanması* (sayfa: 442) bölümündeki **`isatty`** ve **`ttyname`** işlevlerine de bakınız.

7.2. Süreç Grubu İşlevleri

Bu bölümde süreç grupları ile etkileşen işlevlerin açıklamaları bulunmaktadır. Bu işlevleri kullanabilmek için yazılımınıza `sys/types.h` ve `unistd.h` başlık dosyalarını dahil etmelisiniz.

`pid_t` **`setsid`**`(void)` işlev

setsid işlevi yeni bir oturum oluşturur. Bu işlevi çağıran süreç oturum lideri haline gelir ve süreç, süreç grubunun kimliği kendi süreç kimliği olan süreç grubuna konur. Başlangıçta yeni süreç grubunda başka süreç ve süreç grubu yoktur.

Bu işlev ayrıca çağrıldığı süreci denetim uçbirimsiz süreç durumuna getirir.

Normalde, **setsid** işlevi kendini çağıran sürecin yeni süreç grubu kimliği ile döner. Dönüş değeri **-1** ise bu bir hata oluştuğunu gösterir. Aşağıdaki **errno** hata durumu bu işlev için tanımlanmıştır:

EPERM
: Çağıran süreç zaten bir süreç grubunun lideri ya da aynı süreç grup kimliğine sahip başka bir süreç grubu var

`pid_t getsid(pid_t pid)` işlev

getsid işlevi belirtilen sürecin oturum liderinin süreç grup kimliğini döndürür. *pid*'in değeri **0** ise, işlevi çağıran sürecin oturum liderinin süreç grup kimliği döner.

Bir hata oluşmuşsa işlev **-1** ile döner ve **errno** değişkenine şu hata durumlarından biri atanır:

ESRCH
: *pid* kimlikli bir süreç yok

EPERM
: Çağıran süreç ve *pid* ile belirtilen süreç farklı oturumlara ait ve gerçekleme *pid* ile belirtilen sürecin oturum liderinin süreç grup kimliğine erişime izin vermiyor

getpgrp işlevinin iki tanımı vardır: biri BSD Unix'den diğeri POSIX.1 standardından türetilmiştir. *Özellik sınama makrolarıyla* (sayfa: 25) hangi tanımı kullanmak istediğinizi belirleyebilirsiniz. Özellikle, BSD sürümünü kullanmak istiyorsanız **_BSD_SOURCE**; POSIX sürümünü kullanmak istiyorsanız **_POSIX_SOURCE** veya **_GNU_SOURCE** makrosunu belirtmelisiniz. Özellikle **_BSD_SOURCE** altında tanımlı **getpgrp** işlevini kullanan Eski BSD sistemleri için yazılmış yazılımlar `unistd.h` başlık dosyasını içermeyecektir. Böyle yazılımların BSD tanımlarını elde etmek için **-lbsd-compat** ile ilintilemelisiniz.

`pid_t getpgrp(void)` işlev

Çağrıldığı sürecin süreç grup kimliğini döndüren **getpgrp** işlevinin POSIX.1 tanımı.

`pid_t getpgrp(pid_t pid)` işlev

pid kimlikli sürecin süreç grup kimliğini döndüren **getpgrp** işlevinin BSD tanımı. *pid* değeri olarak **0** verirseniz çağrıldığı sürecin süreç grup kimliğini döndürür.

`int getpgid(pid_t pid)` işlev

getpgid işlevi bir BSD işlevi olarak **getpgrp** işlevi ile aynıdır. *pid* kimlikli sürecin süreç grup kimliğini döndürür. *pid* değeri olarak **0** verirseniz çağrıldığı sürecin süreç grup kimliğini döndürür.

Bir hata oluşmuşsa işlev **-1** ile döner ve **errno** değişkenine şu hata durumlarından biri atanır:

ESRCH
: Çağıran süreç ve *pid* ile belirtilen süreç farklı oturumlara ait ve gerçekleme *pid* ile belirtilen sürecin oturum liderinin süreç grup kimliğine erişime izin vermiyor

```
int setpgid(pid_t pid,                                                    işlev
            pid_t pgid)
```

setpgid işlevi *pid* sürecini *pgid* süreç grubuna koyar. Özel bir durum olarak, çağıran süreci belirtmek üzere *pid* ya da *pgid* sıfır olabilir.

İş denetimini desteklemeyen bir sistemde işlev başarısız olur. Daha fazla bilgi için *İş Denetimi İsteğe Bağlıdır* (sayfa: 717) bölümüne bakınız.

İşlem başarılıysa işlev sıfırla döner. Aksi takdirde **-1** döner ve **errno** değişkenine şu hata durumlarından biri atanır:

EACCES
: *pid* kimlikli alt süreç çatallandığında bir **exec** çağrısı yaptı

EINVAL
: *pgid* değeri geçersiz

ENOSYS
: Sistem iş denetimini desteklemiyor

EPERM
: *pid* ile belirtilen süreç ya bir oturum lideri ya da işlevin çağrıldığı süreçle aynı oturumda değil veya *pgid* argümanı işlevin çağrıldığı süreç ile aynı oturumdaki bir süreç grup kimliğiyle eşleşmiyor

ESRCH
: *pid* kimlikli süreç işlevi çağıran süreç değil ya da işlevi çağıran sürecin bir alt süreci değil.

```
int setpgrp(pid_t pid,                                                    işlev
            pid_t pgid)
```

Bu işlev, **setpgid** işlevinin BSD Unix sürümüdür. Her iki işlev de tamamen aynı işlemi yapar.

7.3. Denetim Uçbirimine Erişim İşlevleri

Bunlar bir uçbirimin önalan süreç grubunu belirlemek ya da okumak için kullanılan işlevlerdir. Bu işlevleri yazılımınızda kullanacaksanız `sys/types.h` ve `unistd.h` başlık dosyalarını yazılımınıza dahil etmelisiniz.

Bu işlevler uçbirim aygıtını belirten bir dosya tanıtıcı almasına rağmen önalan işi, açık bir dosya tanıtıcı ile değil, uçbirim dosyasının kendisi ile ilgilidir.

```
pid_t tcgetpgrp(int dosyatanıtıcı)                                        işlev
```

Bu işlev, *dosyatanıtıcı* ile açılmış uçbirim ile ilişkili önalan süreç grubunun süreç grup kimliğini döndürür.

Bir önalan süreç grubu yoksa, dönüş değeri, mevcut bir süreç grubunun süreç grup kimliği ile eşleşmeyen ve değeri **1**'den büyük bir sayıdır. Eğer, evvelce önalan işi olan ve tüm süreçleri sonlanmış bir iş varsa ve henüz önalana taşınmış bir iş yoksa bu durum ortaya çıkar.

Bir hata durumunda işlev **-1** değeri ile döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
: *dosyatanıtıcı* argümanı geçerli bir dosya tanıtıcı değil

ENOSYS

Sistem iş denetimini desteklemiyor

ENOTTY
dosyatanıtıcı argümanı ile ilişkili uçbirim dosyası işlevin çağrıldığı sürecin denetim uçbirimi değil

```
int tcsetpgrp(int    dosyatanıtıcı,                                    işlev
              pid_t pgid)
```

Bu işlev bir uçbirimin önalan süreç grup kimliğini belirlemede kullanılır. *dosyatanıtıcı* argümanı uçbirimi belirten bir dosya tanıtıcıdır; *pgid* ise süreç grubunu belirtir. İşlevi çağıran süreç ile *pgid* aynı oturumun üyesi olmalı ve aynı denetim uçbirimini kullanıyor olmalıdır.

Uçbirime erişim amaçlarına uygun olarak, bu işlev çıktı olarak kabul edilir. İşlev, denetim uçbiriminin bir artalan sürecinden çağrılmışsa, normalde süreç grubundaki tüm süreçlere bir **SIGTTOU** sinyali gönderilir. İşlevi çağıran sürecin **SIGTTOU** sinyalini engellemesi ya da yoksayması durumunda, işlem yine uygulanır ama sinyal gönderilmez.

İşlev başarılı olursa dönüş değeri **0**'dır. Dönüş değeri **-1** ise bu bir hata oluştuğunu gösterir. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
dosyatanıtıcı argümanı geçerli bir dosya tanıtıcı değil

EINVAL
pgid argümanı geçersiz

ENOSYS
Sistem iş denetimini desteklemiyor

ENOTTY
dosyatanıtıcı argümanı ile ilişkili uçbirim dosyası işlevin çağrıldığı sürecin denetim uçbirimi değil

EPERM
pgid işlevi çağıran süreçle aynı oturumda değil

```
pid_t tcgetsid(int dosyatanıtıcı)                                      işlev
```

Bu işlev, denetim uçbirimi *dosyatanıtıcı* ile belirtilen oturumun süreç grup kimliği ile döner. İşlev başarısız olursa dönüş değeri **(pid_t) -1** olur ve *errno* değişkenine şu değerlerden biri atanır:

EBADF
dosyatanıtıcı argümanı geçerli bir dosya tanıtıcı değil

ENOTTY
İşlevi çağıran sürecin bir denetim uçbirimi yok ya da dosya bir denetim uçbirimine ait değil

XXVIII. Sistem Veritabanları ve İsim Hizmetleri Seçimi

İçindekiler

C kütüphanesindeki çeşitli işlevlerin yerel ortamda düzgün çalışması için yapılandırılmaları gerekir. Geleneksel olarak, bu işlem dosyalar kullanılarak yapılır (örn, `/etc/passwd`), ama diğer isim hizmetleri de (örn, Ağ Bilgi Hizmetleri – NIS, Alan Adı Hizmetleri – DNS) popüler olmuş ve C kütüphanesi içine zaman içinde dahil edilmişlerdir.

GNU C kütüphanesi bu soruna daha temiz bir çözüm içerir. Bu çözüm, Sun Microsystems tarafından Solaris 2'nin C kütüphanesinde kullanılan bir yöntemden sonra tasarlanmıştır. GNU C kütüphanesi bunların isimlerini ve çağrılarını ***İsim Hizmetleri Seçimi*** – *Name Service Switch* (NSS) şeması ile izler.

Arayüzün Sun'ın sürümüne benzer olması beklenirdi ama bir ortak kod bile yoktur. Biz Sun'ın gerçeklemesinden herhangi bir kaynak kod asla almadık, bu yüzden dahili arayüz uyumsuzdur. Daha sonra göreceğimiz gibi bunu dosya isimlerinde de açıkça ortaya koyduk.

1. NSS Temelleri

Anafikir, veritabanlarına erişmeye çalışan farklı hizmetlerin gerçeklemelerini ayrı modüllere koymaktır. Bunun bazı faydaları vardır:

1. Destekçiler yeni hizmetleri GNU C kütüphanesine eklemeksizin NSS gerçeklemesine ekleyebilirler.
2. Modüller birbirinden bağımsız olarak güncellenebilir.
3. C kütüphanesi fazla şişmemiş olur.

Yukarıdaki ilk görevi yerine getirmek için modül arayüzünü[14] aşağıda açıklayacağız. Yeni bir hizmeti doğru gerçeklemek için modüller içinde işlevlerin nasıl çağrıldığını anlamak önemlidir. Yazılımcının doğrudan doğruya kullanabileceği bir yol yoktur. Yazılımcı veritabanlarına sadece belgelenmiş ve standartlaştırılmış işlevleri kullararak erişebilir.

NSS şemasında mevcut veritabanları şunlardır:

`aliases`
: Posta takma adları.

`ethers`
: Ethernet numaraları.

`group`
: Kullanıcı grupları, bkz. *Grup Veritabanı* (sayfa: 762).

`hosts`
: Konak isimleri ve numaraları, bkz. *Konak İsimleri* (sayfa: 412).

`netgroup`
: Ağ çapında kullanıcılar, konaklar ve altağlar, bkz.*Ağ Grubu Veritabanı* (sayfa: 766).

`networks`
: Ağ isimleri ve numaraları, bkz. *Ağ İsimleri Veritabanı* (sayfa: 440).

`protocols`
: Ağ protokolleri, *Protokol Veritabanı* (sayfa: 417).

`passwd`
: Kullanıcı parolaları, bkz. *Kullanıcı Veritabanı* (sayfa: 760).

`rpc`
: Uzak yordam çağrı isimleri ve numaraları.

`services`
: Ağ hizmetleri, bkz. *Servis Veritabanı* (sayfa: 415).

`shadow`
: Gölge kullanıcı parolaları.

Sonradan eklenen **`automount`**, **`bootparams`**, **`netmasks`** ve **`publickey`** gibi veritabanları da vardır.

2. NSS Yapılandırma Dosyası

Ne olursa olsun, NSS kodu kullanıcının isteklerini yerine getirmelidir. **`/etc/nsswitch.conf`** dosyası bu sebeple vardır. Bu dosyada her veritabanı için arama sürecinin nasıl çalışacağı ile ilgili bir belirtim vardır. Dosyanın içeriği şöyle birşeydir:

```
# /etc/nsswitch.conf
#
# Name Service Switch configuration file.
#

passwd:     db files nis
shadow:     files
group:      db files nis

hosts:      files nisplus nis dns
networks:   nisplus [NOTFOUND=return] files

ethers:     nisplus [NOTFOUND=return] db files
protocols:  nisplus [NOTFOUND=return] db files
rpc:        nisplus [NOTFOUND=return] db files
services:   nisplus [NOTFOUND=return] db files
```

İlk sütunda veritabanının ismi bulunur. Satırın kalanında arama sürecinin nasıl çalışacağı belirtilir. Belirttiğiniz yolun her veritabanı için ayrı olduğunu unutmayın. Bu, eski yöntemle, bir tekparça gerçekleme ile yapılamaz.

Her veritabanı için yapılandırma belirtimi iki farklı öğe içerebilir:

- **`files`**, **`db`** veya **`nis`** gibi bir hizmet belirtimi.
- **`[NOTFOUND=return]`** gibi arama sonucuna verilen tepki.

2.1. NSS Yapılandırma Dosyasındaki Hizmetler

Önceki bölümdeki örnek dosyada dört farklı hizmet vardı: **files**, **db**, **nis** ve **nisplus**. Bu, her yerde sadece bu hizmetlerin olduğu anlamına gelmediği gibi bunların her yerde bulunabilen hizmetler olduğu anlamına da gelmez.

Aslında, bu isimler NSS kodu tarafından dolaylı olarak adresli işlevleri bulmak için kullanılan basit dizgelerdir. Dahili arayüz daha sonra açıklanacaktır. Kullanıcıya görünür olan, her biri bir hizmeti gerçekleştiren modüllerdir.

isim dizgesinin arama için kullanılacak hizmet olduğunu varsayalım. Bu hizmeti gerçekleştiren modülün dosyası **libnss_***isim* ismini alır. Paylaşımlı kütüphaneleri destekleyen sistemlerde ise bu isim (örneğin) **libnss_***isim***.so.2**. olacaktır. Dosya isminin sonundaki sayı arayüzün o an geçerli ve pek sık değişmeyen sürümünü ifade eder. Normalde kullanıcılar bu dosyalarla ilgilenmezler, çünkü bunlar özdevinimli olarak bulunabilecekleri bir dizine konulur. Mevcut hizmetlerin sadece isimleri önemlidir.

2.2. NSS Yapılandırmasındaki Eylemler

Belirtimdeki ikinci öğe, kullanıcıya arama sürecini daha iyi denetleme imkanı verir. Eylem öğeleri iki hizmet ismi arasına yerleştirilir ve köşeli parantezlerin arasına yazılır. Genel biçimi şöyledir:

```
[ (!? durum=eylem )+ ]
```

Burada:

```
durum => success | notfound | unavail | tryagain
eylem => return | continue
```

olabilir. Sözcüklerin harf büyüklükleri önemsizdir. *durum* değerleri belirli bir hizmetin bir arama işlevinin çağrısının sonucudur. Bunların anlamları:

`success`
: Bir hata oluşmadan istenen girdi döndü. Öntanımlı eylem **return**, dön'dür.

`notfound`
: Arama süreci tamamlandı ama gerekli değer bulunamadı. Öntanımlı eylem **continue**, aramaya devam et'tir.

`unavail`
: Hizmet geçici olarak kullanımdışıdır. Bu ya gerekli dosyanın olmadığı ya da DNS için, hizmetin kullanımdışı olduğunu veya sorguya izin vermediğini belirtir. Öntanımlı eylem **continue**, aramaya devam et'tir.

`tryagain`
: Hizmet geçici olarak kullanımdışıdır. Bu bir dosyanın kilitli olduğu ya da bir sunucunun o an artık bağlantı kabul edemediğini belirtir. Öntanımlı eylem **continue**, aramaya devam et'tir.

Şöyle bir satırımız varsa:

```
ethers: nisplus [NOTFOUND=return] db files
```

bu satır şuna eşdeğerdir (hepsinin bir satıra yazılması zorunluluğu dışında):

```
ethers: nisplus [SUCCESS=return NOTFOUND=return UNAVAIL=continue
                 TRYAGAIN=continue]
        db      [SUCCESS=return NOTFOUND=continue UNAVAIL=continue
                 TRYAGAIN=continue]
        files
```

Eylemlerin öntanımlı değeri normalde sizin ne istediğinizi gösterir ve sadece olağandışı durumlarda değiştirmek gerekir.

Eğer *durum* dizgesinden önce bir **!** varsa (isteğe bağlıdır), belirtilen eylemin dışındaki tüm eylemlerin geçerli olduğu anlamına gelir. Yani, **!** işleci C dilindeki gibi "bundan farklı her şey" anlamına gelir.

Bu eylem öğesini gerekli yapan olağandışılığı açıklamadan önce bir kaç yorum: **files** hizmetinden sonra bir eylem öğesi eklemenin bir faydası olmadığı açıktır. Eylem *daima* **return** olacağından bundan sonra başka bir hizmet belirtilmez.

Şimdi, bu **[NOTFOUND=return]** eylemi neden kullanışlıdır? Sebebini anlayabilmek için **nisplus** hizmetinin çoğunlukla bir bütünlük arzettiğini bilmemiz gerekir. Yani, bir girdi NIS+ tablolarında mevcut değilse başka hiçbir yerde bulunamaz. Bu durum bu eylem öğesinin neden böyle belirtildiğini açıklar: diğer hizmetler zaten bir sonuç vermeyeceğinden onlara bakmak gereksizdir.

NIS+ hizmetinin makinanın yeniden başlatılması nedeniyle yokluğu farklı bir durum olurdu. Bu durumda arama işlevinin dönüş değeri **notfound** değil, **unavail** olur. Yukarıdaki satırın açılımda da görebileceğiniz gibi bu durumda **db** ve **files** hizmetleri kullanılır. Güzel değil mi? Sistem çalışmaya tamamen hazır olmadığı bir anda (sistem açılışı, kapanışı ya da bir ağ sorunu nedeniyle) sistem yöneticisinin özel bir çaba harcaması gerekmeyecektir.

2.3. NSS Yapılandırma Dosyası için İpuçları

Son olarak bir kaç ipucu. **/etc/nsswitch.conf** dosyasının olmayışı durumunda NSS gerçeklemesi sizi tamamen çaresiz bırakmaz. Desteklenen tüm veritabanları için bir öntanımlı değer vardır, böylece dosya bozuk da olsa hiç olmasa da sistemin normal çalışması mümkün olacaktır.

hosts ve **networks** veritabanları için öntanımlı değer **dns [!UNAVAIL=return] files** şeklindedir. Yani sistem DNS hizmeti olmaksızın hazırsa ama yanıt varsa dönüş kesindir.

passwd, **group** ve **shadow** veritabanları geleneksel olarak özel bir yolla elde edilir. `/etc` dizinindeki ilgili dosyalar okunur, bunlar içinde **+** ile başlayan bir isim varsa NIS kullanılır. Bu çeşit bir arama **compat** adı verilen özel bir arama hizmeti kullanılarak mümkün olur. Bu hizmet için öntanımlı değer **compat [NOTFOUND=return] files** şeklindedir.

Kalan tüm veritabanları için öntanımlı değer **nis [NOTFOUND=return] files** şeklindedir. Bu çözüm NIS ve dosya temelli arama kullanıldığından en iyi şansı verir.

İkinci bir nokta da kullanıcının arama sürecini eniyilemeye çalışmasıdır. Her hizmetin kendine özgü bir yanıt süresi vardır. Basit dosya araması bir yerel dosya üzerinde hızlı olabilirdi ama dosya uzunsa ve gerekli girdi dosyanın sonlarına doğruysa bu biraz vakit alır. Bu durumda büyük veri kümelerine daha hızlı erişim sağlayan **db** hizmetinin kullanılması daha iyi olabilir.

Sıkça rastlanan bir durum NIS gibi kapsamlı bilgi hizmetlerinin kullanıldığı durumdur. Bu durumda **nis** vs. gibi hizmet girdilerinin kullanılması kaçınılmazdır. Fakat bunun gibi yavaş hizmetlerden mümkün olduğunca kaçınılmalıdır.

3. NSS Modül Yapısı

Artık modüllerin nasıl çalıştığını açıklamaya başlayabiliriz. Bir modülde bulunan işlevler isimleriyle anılırlar. Yani, bir sıçrama tablosu ya da benzeri bir şey yoktur. Bunun nasıl yapıldığıyla burada ilgilenmeyeceğiz; bunlarla ilgileniyorsanız Özdevimli İlintileme hakkında birşeyler okumalısınız.

3.1. NSS Modüllerinin İsimlendirme Şeması

Her işlevin ismi çeşitli parçaların bir araya getirilmesiyle oluşur:

_nss_*hizmet*_*işlev*

hizmet şüphesiz bu işlevi içeren modülün isminden gelecektir.[15] *işlev* parçası C kütüphanesindeki arayüz işlevinden türetilir. Eğer kullanılan hizmet **files** ve bu hizmetin modülünde kullanılan işlev **gethostbyname** ise:

libnss_files.so.2

modülündeki

_nss_files_gethostbyname_r

işlevinden bahsediyor, oluruz.

Gördüğünüz gibi, yukarıda açıkladıklarımız gerçeğin tümü değildir. Aslında NSS modülleri arama işlevlerinin sadece evresel sürümlerini içerir. Yani yazılımcı aslında **gethostbyname_r** işlevini kullanacağından bunu işlevin isminde **gethostbyname_r** olarak ('gethostbayname_r'eentrant) belirtmelidir. Tüm kullanıcı arayüzü işlevleri için C kütüphanesi bu çağrıları evresel işlev çağrılarına eşler. Evresel işlevler için arayüz hemen hemen aynı olduğundan bu sıradan bir işlemdir. Evresel eşdeğeri olmayan işlevler için ise kütüphane, kullanılan dahili tamponları kullanıcı tanımlı tamponlarla değiştirerek tutar.

Yani evresel işlevlerin benzer karşılıkları olabilir. Tüm veritabanları için işlevler içermesi için ya da tüm veritabanlarının erişebileceği şekilde tasarlanmış bir hizmet modülü yoktur. Bir işlevin yokluğu halinde işlevin **unavail** döndüreceği varsayılmıştır (Bkz. *NSS Yapılandırmasındaki Eylemler* (sayfa: 735)).

libnss_files.so.2 dosya ismi bir Solaris 2 sistemde **nss_files.so.2** olarak görünür. Bu farktan daha önce sözedilmişti. Sun'ın NSS modülleri sadece dolaylı olarak yüklenen modüller olarak kullanılabilir.

GNU C Kitaplığındaki NSS modülleri, normal kitaplık olarak kullanılmak üzere hazırlanmıştır. Fakat, bu, şu anda geçerli *değildir*. Fakat modüllerdeki isim uzayının organizasyonu, bunu, Solaris'in aksine imkansız kılmamaktadır. Modüllerin kitaplık olması bundan kaynaklanmaktadır. [16]

3.2. NSS Modüllerinde İşlev Arayüzü

Artık, modüllerdeki işlevler hakkında bilgimiz var. Şimdi türleri açıklayalım. Bir önceki bölümde işlevlerin evresel sürümlerinden bahsetmiştik. Bu, işlevin evresel olmayan sürümüne göre ek argümanlar gerektiği anlamına gelir. **gethostbyname** işlevinin evresel olmayan ve evresel sürümlerinin prototiplerine bakalım:

```
struct hostent *gethostbyname (const char *isim)

int gethostbyname_r (const char     *isim,
                     struct hostent  *sonuc_tamponu,
                     char            *tampon,
                     size_t           tampon_uzunlugu,
                     struct hostent **sonuc,
                     int             *hata_durumu)
```

İşlevin NSS modülündeki prototipi ise şöyle olur:

```
enum nss_status _nss_files_gethostbyname_r (const char     *isim,
                                            struct hostent  *sonuc_tamponu,
                                            char            *tampon,
                                            size_t           tampon_uzunlugu,
                                            int             *hata_num,
                                            int             *hata_durumu)
```

Yani, arayüz işlevi aslında *sonuc* argümanı olmayan ve dönüş değeri değişmiş evresel işlevdir. İşlevin evresel olmayan sürümü sonuca bir gösterici ile dönerken evresel sürümü bir **enum nss_status** değeri ile döner:

`NSS_STATUS_TRYAGAIN`
: sayısal değeri: **-2**

`NSS_STATUS_UNAVAIL`
: sayısal değeri: **-1**

`NSS_STATUS_NOTFOUND`
: sayısal değeri: **0**

`NSS_STATUS_SUCCESS`
: sayısal değeri: **1**

Şimdi, **/etc/nsswitch.conf** dosyasında kullanılar eylem öğelerinin yerini görelim.

Kaynak kodunu incelerseniz, beşinci bir değerin varlığını görürsünüz: **NSS_STATUS_RETURN**. Bu sadece dahili olarak kullanılan bir değerdir, bir kaç işlev tarafından yukarıdaki değerin kullanılamadığı yerlerde kullanılır. Eğer gerekliyse, daha fazla ayrıntıya kaynak kodunu inceleyerek ulaşabilirsiniz.

Arayüz işlevinin bir hata döndürmesi durumunda, doğru hata numarasının **hata_num* içinde saklanması önemlidir. Bazı dönüş durum değerleri sadece bir hata kodu ile ilgiliyken diğerleri daha fazlası ile ilgilidir.

NSS_STATUS_TRYAGAIN	**EAGAIN**	Kullanılan işlevlerden biri ya geçici olarak özkaynaksız çalıştı ya da hizmet şu an kullanışsız.
	ERANGE	Belirtilen tampon yeterince geniş değil. İşlev tekrar daha geniş bir tamponla çağrılmalı.
NSS_STATUS_UNAVAIL	**ENOENT**	Gerekli girdi dosyalarından biri bulunamadı.
NSS_STATUS_NOTFOUND	**ENOENT**	İstenen girdi elverişli değil.

Bunlar önerilen değerlerdir. Başka hata kodları olabileceği gibi açıklanan hata kodları farklı anlamlara da gelebilir. *Biri dışında:* **NSS_STATUS_TRYAGAIN** döndüğünde, hata kodu, belirtilen tamponun yetersiz olduğun anlatan **ERANGE** *olmalıdır*. Bunun dışında kritik önemde bir şey yoktur.

Yukarıdaki işlev hemen hemen diğer tüm modül işlevlerinde olmayan bazı özelliklere sahiptir. *hata_durumu* diye bir argümanı var. Bu argümanın gösterdiği değişkene, işlev bir şekilde başarısız olduğunda hata durumu konulacaktır. Evresel işlevler *h_errno* genel değişkenini kullanamazlar; **gethostbyname_r** ile yapılan **gethostbyname** çağrılarında son argüman *&hata_durumu* olarak belirtilir.

get*XXX***by***YYY* işlevleri NSS modüllerinde en önemli işlevlerdir. Ancak başka yöntemlerle erişilen veritabanları da vardır (**setpwent**, **getpwent** ve **endpwent** işlevleri ile erişilen parola veritabanını buna örnek verebiliriz). Bunlar daha sonra ayrıntılı olarak açıklanacaktır. Modül işlevinin imzasını saptayacak genel bir yöntem:

- dönüş değeri **int**'dir;
- ismi *NSS Modüllerinin İsimlendirme Şeması* (sayfa: 736) bölümünde açıklandığı gibidir;
- işlevin ilk argümanları evresel olmayan eşdeğerinin argümanları ile aynıdır;
- sonraki argümanları şunlardır:

  `STRUCT_TYPE` **sonuc_tamponu*
  : sonucun saklandığı tampona gösterici. **STRUCT_TYPE**, normalde veritabanının karşılığı olan bir yapıdır.

  `char` **tampon*
  : sonuçla ilgili ek verilerin saklanabileceği tampona gösterici.

`size_t` *tampon_uzunlugu*
: *tampon* ile gösterilen tamponun uzunluğu.

- konak ismi ve ağ ismi arama işlevlerinde *hata_durumu* hep son argüman olarak görünür.

set…ent ve **end…ent** işlevleri dışında tüm işlevler için, bu liste geçerlidir.

4. NSS'nin Genişletilmesi

Evvelce bahsedildiği gibi NSS'nin getirilerinden biri kolayca genişletilebilmesidir. Genişletme iki yolla yapılabilir: İlki normalde sadece C kütüphanesi geliştiricileri tarafından yapılır. Burada önemli olan, başka bir veritabanının bağımsız olarak eklenmesi gerektiğini unutmamaktır. Çünkü bir hizmet tüm veritabanlarını ve arama işlevlerini desteklemek zorunda değildir.

Bir yeni hizmetin tasarımcısı/gerçekleştiricisi ilgilendiği veritabanlarını seçmekte özgür olduğu gibi kalanı daha sonraya da bırakabilir (veya tamamen terkedebilir).

4.1. NSS'ye Başka Hizmetlerin Eklenmesi

Yeni hizmetin kaynakları GNU C kütüphanesinin parçası olmak zorunda değildir (hatta olmamalıdır). Geliştiricinin kaynaklar ve tasarım üzerinde tam hakimiyeti olmalıdır. C kütüphanesi ile yeni hizmet modülü arasındaki bağları sadece arayüz işlevleri oluşturur.

Her modül burada açıklanan özel bir arayüz belirtimine göre tasarlanır. Şimdilik sürümü 2'dir (arayüzün 1. sürümü yetersizdi) ve bu NSS modülünün paylaşımlı kütüphane nesnesinin sürüm numarası olarak belirtilir: bu nesnelerin isimleri **.2** uzantısını içerir. Eğer arayüz şimdikiyle uyumsuz olarak değitirilirse bu numara artacak ama eski arayüzü kullanan modüller hala kullanılabilir olacaktır.

Yeni hizmetin geliştiricileri modülün doğru arayüz numarası kullanılarak oluşturuduğundan emin olmak zorundadır. Yani, dosyanın ismi doğru tanımlanacak ve ELF sistemlerde paylaşımlı nesne ismi (so uzantısı) ayrıca bu numarayı içerecektir. Bir modül, bir ELF sisteminde GNU CC ile nesne dosyalarından şöyle derlenir:

```
gcc -shared -o libnss_NAME.so.2 -Wl,-soname,libnss_NAME.so.2 nesneler
```

Bu komut satırı hakkında daha fazla bilgi için: `info gcc 'Options for Linking'`

Yeni modülü, onu kullanacak kütüphane bulabilmelidir. Bu özdevimli ilintileyici seçenekleri kullanılarak yapılabilir, böylece ikilik nesne dosyasının yerleştirildiği dizini bulabilir. ELF sistemlerde bu, modülün bulunduğu dizini **LD_LIBRARY_PATH** ortam değişkenine ekleyerek yapılır.

Fakat bu, bazı uygulamalar bu değişkeni yoksaydığından (bunlar kullanıcının kimliğini kullanmazlar) daima mümkün olmaz. Bu nedenle, modülün kararlı sürümü özdevimli ilintileyicinin araştırdığı dizinlere konulması önem kazanır. Normalde bu dizin *önek*`/lib` dizini olmalıdır, burada *önek* derleme öncesi yapılandırma sırasında **--prefix** seçeneğinde belirtilen dizindir. Ama dikkatli olmalısınız: bu sadece, modül herhangi bir bozukluğa yol açmıyorsa yapılabilir. Sistem yöneticisi bu bakımdan dikkatli olmalıdır.

4.2. NSS Modül İşlevlerinin Özellikleri

Şimdiye kadar NSS modülündeki işlevlerin sözdizimsel arayüzünden bahsettik. Her işlevin gerçeklenmesi ister istemez farklı olacağında aslında söylenebilecek fazla bir şey yoktur. Fakat tüm işlevlerin uyması gereken bir kaç genel kuraldan bahsedilebilir.

Aslında arayüzde görülebilecek dört farklı işlev çeşidi vardır. Hepsi sistem veritabanları için kullanılan geleneksel işlevlerden türetilir. Aşağıdaki gösterimde *vt*, veritabanı sözcüğünün kısaltmasıdır.

`enum nss_status _nss_`*veritabanı*`_set`*vt*`ent (void)`

Bu işlev hizmeti belirttiği işlem için hazırlar. Basit bir dosya temelli arama için dosyaları açan, diğer hizmetlerde basitçe hiçbir işlem yapmayan bir işlev olabilir.

Bu işlev için özel bir durum, **sethostent** işlevindeki (*Konak İsimleri* (sayfa: 412)) gibi bazı *veritabanı* veritabanları için ek argümanlar alabilmesidir. (**int setvtent (int)** gösterimi ile karşılaştırılırsa "hosts" veritabanı için "host" kısaltmasının kullanıldığı görülür.)

Normal dönüş değeri *NSS_STATUS_SUCCESS* olmalı; bir hata durumunda *NSS Modüllerinde İşlev Arayüzü* (sayfa: 737) bölümündeki tabloda belirtilen değerlerden biri olmalıdır.

`enum nss_status _nss_`*veritabanı*`_endvtent (void)`

Bu işlev hala açık olan tüm dosyaları kapatır ya da bellekten tamponları kaldırır. Kaldırılacak bir tampon ya da kapatılacak bir dosya yoksa, işlev yine basitçe hiçbir işlem yapmayacaktır.

Normalde *NSS_STATUS_SUCCESS*'dan farklı bir dönüş değeri olmaz.

`enum nss_status _nss_`*veritabanı*`_getvtent_r (`*yapı* `*sonuc, char *`*tampon*`, size_t` *tampon_boyu*`, int *`*hata_num*`)`

Bu işlev peşpeşe girdi almak için bir satırda defalarca çağrılacağından bir durum bilgisi tutmak zorundadır. Ama bu zorunluluk ayrıca işlevin gerçekte evresel olmayacağı anlamına da gelir. Sadece, bu işlev aynı anda yapılan başka çağrılarıyla verinin alındığı yere veri yazmayı denemeyecekse, *sonuc* ile belirtilen tampona yazacaksa evresel olabilir. Fakat, bir ortak durumu paylaşan çağrıların varlığında ve bir dosya erişimi durumunda bu, dosyadaki komşu girdilerin dönmesi anlamına gelir.

tampon tamponunun uzunluğunu belirtmede kullanılan *tampon_boyu* tamponu sonuç ile ilgili bazı ek verilerin saklanması için kullanılabilir ama bu durumda işlevin sonraki çağrılarının aynı tamponla yapılabilmesi mümkün olmaz. Ancak, bu tamponun bazı durum bilgilerini döndürmesinden hareketle bu tampon, bir çağrıdan diğerine durum bilgisini aktarmak için kullanılmamalıdır.

İşlev dönmeden önce, gerçekleme *hata_num* ile gösterilen değeri *errno* genel değişkeninde saklamalıdır. Modülün durağan ilintili yazılımlarla da çalışabilmesini sağlamak için bunun böyle olması önemlidir.

Evvelce açıklandığı gibi bu işlev ayrıca bir ek argüman alabilir. Bu kullanılan veritabanına bağlıdır; sadece **hosts** ve **networks** veritabanlarında görülür.

İşlev **NSS_STATUS_SUCCESS** ile döneceği gibi başka değerlerle de dönebilir. Son girdi okunduktan sonra **NSS_STATUS_NOTFOUND** dönebilir. Belirtilen tampon gereğinden küçükse **NSS_STATUS_TRYAGAIN** ile dönebilir. Bir **_nss_***veritabanı*_**setvtent** çağrısı ile hizmet başta ilklendirilemediğinde bu işlev için izin verilen tüm dönüş değerleri burada ayrıca döndürülebilir.

`enum nss_status _nss_`*veritabanı*`_getvtbyxx_r (`*parametreler*`, yapı *sonuc, char *`*tampon*`, size_t` *tampon_boyu*`, int *`*hata_num*`)`

Bu işlev veritabanından *parametreler* ile adreslenen girdiyi döndürür. Bu argümanların sayısı ve türü değişebilir. Bunlar tek tek kullanıcı seviyesi arayüz işlevlerine bakarak saptanır. İşlevin evresel olmayan sürümünde belirtilen tüm argümanlar burada *parametreler* alanında belirtilmelidir.

Sonuç *sonuc* ile gösterilen yapıda saklanmalıdır. Eğer döndürülecek başka veriler varsa (örn, dizgeler *sonuc* yapısında sadece göstericilerle içerilebilir) işlev *tampon* ya da *tampon_boyu* ile gösterilen tamponları kullanmalıdır. Sabitler biçiminde olmayan hiçbir genel değişkenli veri olmamalıdır.

Bu işlevi gerçeklemesi **setvtent** işlevi tarafından atanan *açıkkal* seçeneği ile de bu bir gereklilikse ilgilenmelidir.

İşlev dönmeden önce, gerçekleme *hata_num* ile gösterilen değeri *errno* genel değişkeninde saklamalıdır. Modülün durağan ilintili yazılımlarla da çalışabilmesini sağlamak için bunun böyle olması önemlidir.

Evvelce açıklandığı gibi bu işlev **hosts** ve **networks** veritabanları için ayrıca bir ek argüman alabilir.

İşlevin dönüş değerleri daima *NSS Modüllerinde İşlev Arayüzü* (sayfa: 737) bölümünde açıklanan kurallara uygun olmalıdır.

XXIX. Kullanıcılar ve Gruplar

İçindekiler

Sistemde bir oturum açan her kullanıcı ***kullanıcı kimliği*** adı verilen özel bir numara ile kimliklendirilir. Her sürecin, süreci oluşturan kullanıcının erişim yetkileriyle belirlenen bir etkin kullanıcı kimliği vardır.

Kullanıcılar erişim denetimi amacıyla ***gruplar*** halinde sınıflandırılır. Her sürecin, dosyalara erişimde kullanabildiği gruplar anlamında çok sayıda ***grup kimliği*** değeri olabilir.

Bir sürecin etkin kullanıcı ve grup kimliği onun ***aidiyeti***ni şekillendirir. Bu, sürecin hangi dosyalara erişebileceğini belirler. Normalde bir süreç aidiyetini ata sürecinden miras alır, ancak bazı özel durumlarda sürecin aidiyeti erişim yetkilerini yeniden düzenlemek amacıyla değiştirebilir.

Ayrıca, sistemdeki her dosya bir kullanıcı ve bir grup kimliğine sahiptir. Dosyalara erişim, çalışan sürecin aidiyeti dosyanın kullanıcı ve grup kimlikleri ile karşılaştırılarak denetim altına alınır.

Sistem kayıtlı kullanıcılar için bir veritabanı, tüm tanımlı gruplar için de ayrı bir veritabanı tutar. Kütüphane, bu veritabanları ile çalışabileceğiniz işlevler içerir.

1. Kullanıcı ve Grup Kimlikleri

Bir bilgisayar sistemindeki her kullanıcı hesabı bir ***kullanıcı ismi*** (veya ***oturum açma ismi***) ile ***kullanıcı kimliği*** içerir. Normalde her kullanıcının eşsiz bir kullanıcı kimliği vardır, ama bu kullanıcı kimliği ile ilişkili çok sayıda kullanıcı ismi olabilir. Kullanıcı kimlikleri ve onunla ilişkili kullanıcı isimleri *Kullanıcı Veritabanı* (sayfa: 760) bölümünde nasıl erişildiğinin açıklandığı bir veritabanında tutulur.

Kullanıcılar ***gruplar*** halinde sınıflandırılır. Her kullanıcı kimliği için bir ***öntanımlı grup*** vardır ve her kullanıcı çok sayıda ***ek grup***un üyesi olabilir. Aynı grubun üyeleri olan kullanıcılar, bu grubun üyesi olmayan kullanıcıların erişemediği özkaynakları paylaşırlar. Her grubun bir ***grup ismi*** ve bir ***grup kimliği*** vardır. Grup ismi ve grup kimliği ile ilgili ayrıntılı bilgiyi *Grup Veritabanı* (sayfa: 762) bölümünde bulabilirsiniz.

2. Bir Sürecin Aidiyeti

Herhangi bir anda, her sürecin bir ***etkin kullanıcı kimliği***, bir ***etkin grup kimliği*** ile çok sayıda ***ek grup kimliği*** olabilir. Bu kimlikler sürecin ayrıcalıklarını belirler. Bunlara bir bütün olarak ***sürecin aidiyeti*** denir, çünkü bunlar sürecin özkaynaklara kimin adına erişebileceğini belirler.

Oturum kabuğunuz sizin kullanıcı kimliğiniz, sizin grup kimliğiniz ve eğer çok sayıda ek grubun üyesi iseniz sizin ek grup kimliklerinizle başlatılır. Normalde, kabuğun çalıştırdığı tüm süreçler bu değerleri miras alırlar.

Bir sürecin ayrıca, süreci oluşturan kullanıcıyı ifade eden bir ***gerçek kullanıcı kimliği*** ve kullanıcının öntanımlı grubunu ifade eden bir ***gerçek grup kimliği*** vardır. Bu değerler, erişim denetimi ile ilgili bir rol oynamazlar, ama yine de sürecin aidiyeti kapsamında kabul edilirler. Ayrıca önemlidirler.

Gerçek kullanıcı ve grup kimliği sürecin yaşam süresi içinde değiştirilebilir; bkz. *Bir Sürecin Aidiyeti Niçin Değiştirilir?* (sayfa: 743).

Süreçlerin dosyalara erişim yetkilerini belirleyen etkin kullanıcı ve grup kimlikleri ile daha ayrıntılı bilgiyi *Erişim İzinleri* (sayfa: 380) bölümünde bulabilirsiniz.

Bir sürecin etkin kullanıcı kimliği ayrıca **`kill`** işlevi ile sinyal gönderme yetkilerini denetlemekte de kullanılır. Bkz. *Başka Bir Sürece Sinyal Gönderme* (sayfa: 628).

Son olarak, bir sürecin etkin kullanıcı kimliğinin sıfır olduğu durumda uygulanabilen bir çok işlem vardır. Bu kullanıcı kimliğine sahip bir sürece ***ayrıcalıklı süreç*** denir. Sıfır kullanıcı kimliği ile ilişkili kullanıcının ismi teamülen **`root`**'dur. Ancak bu kullanıcı kimliği ile ilişkilendirilmiş başka isimler de olabilir.

3. Bir Sürecin Aidiyeti Niçin Değiştirilir?

Bir sürecin kullanıcı ve/veya grup kimliklerinin değişmesinin gerektiği en belirgin durum **`login`** uygulamasıdır. **`login`** çalışmaya başladığında kullanıcı kimliği **`root`**'tur. Görevi, oturum açan kullanıcının kullanıcı ve grup kimlikleriyle bir kabuk başlatmaktır. (Bunu tamamen yerine getirmek için, **`login`** hem kabuğun hem de kendi aidiyetinin gerçek kullanıcı ve grup kimliklerini ayarlamalıdır. Ama bu özel bir durumdur.)

Aidiyeti değiştirmek içi daha genel bir durum sıradan bir kullanıcının, bir yazılımı çalıştırmadan bazı özkaynaklara erişemeyeceği durumdur.

Örneğin, bir uygulama ile oluşturulmuş ama başka kullanıcıların okumasını ve değiştirmesini istemediğiniz bir dosyanız olsun. Bir takım kilitleme protokolleri ile bunu sağlamanın yanında, mahremiyet ve bütünlüğünün bozulmaması gibi gerekçelerle dosyanıza erişilememesini de isteyebilirsiniz. Bu çeşit erişim kısıtlamalarını gerçekleştirmek için yazılımın etkin kullanıcı ve grup kimliklerinin dosyanınkilere uygun olması gerekir.

Farklı bir örnek oyunlardır. Oyunlarda oyuncuların elde ettiği derecelere hiçbir oyuncunun erişememesi, bu dosyaya sadece oyun yazılımının erişebilmesi gerekir. Bu gibi durumlarda oyunlar için bir kullanıcı kimliği ve oturum açma ismi (**`oyunlar`** diyelim) oluşturularak bu sağlanabilir. Oyun yazılımı bu dosyayı güncellemek gerektiğinde etkin kullanıcı kimliğini **`oyunlar`** olarak değiştirilebilir. Uygulamada, yazılımın aidiyeti **`oyunlar`** kullanıcı kimliğine uyarlanmalı, böylece derecelerin bulunduğu dosyaya erişmesi sağlanmalıdır.

4. Bir Sürecin Aidiyeti Nasıl Değiştirilir?

Bir sürece aidiyetini değiştirme yeteneği mahremiyetin korunması amacıyla ister istemez verilebileceği gibi kasıtlı olarak da verilebilir. Bazı olası sorunlar sebebiyle aidiyetin değiştirilmesi özel koşullarla sınırlıdır.

Kendi kullanıcı ve grup kimliklerinizi keyfi olarak belirleyemezsiniz; bunu sadece yetkili süreçler yapabilir. Bir süreç için ise, aidiyeti değiştirmenin normal yolu bunu belli bir kullanıcı ya da gruba önceden ayarlamaktır. Bu bir dosyanın erişim kipinin setuid ve setgid bitleri ile yapılır. Bkz. *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

Bir çalıştırılabilir dosyanın setuid biti etkinse, bu dosyanın çalıştırılması sürece üçüncü bir kullanıcı kimliği sağlar: ***dosya kullanıcı kimliği***. Bu kimlik dosyanın sahibinin kimliğine ayarlanır. Bundan hareketle, sistem etkin kullanıcı kimliği dosyanın kullanıcı kimliğine değiştirir. Gerçek kullanıcı kimliği ise değişmeden kalır. Benzer şekilde, setgid biti etkinse, dosyanın grup kimliği dosyanın grup kimliğine ayarlanır ve sistem etkin grup kimliğini dosya grup kimliğine değiştirir.

Bir süreç bir dosya kimliğine (kullanıcı ya da grup) sahipse, herhangi bir anda kendi etkin kimliğini kendi gerçek kimliğine ve geriye kendi dosya kimliğine değiştirebilir. Yazılımlarda bu özellik, gerçekten ihtiyaç duyulması dışında özel ayrıcalıklarını terketmesi için kullanılır. Böylece bir yazılımın ayrıcalıklarının uygunsuz bazı şeyler için kullanılması bir bakıma önlenir.

Taşınabilirlik Bilgisi

Eski sisitemlerden bazıları dosya kimliklerini içermez. Bir sistem bu özelliğe sahipse derleyicide **_POSIX_SAVED_IDS** tanımlıdır. (POSIX standardında dosya kimliklerinden "kayıtlı kimlikler" (saved IDs) diye bahsedilir.)

Dosya Öznitelikleri (sayfa: 371) bölümünde dosya kipleri ve erişilebilirlik hakkında daha ayrıntılı bilgi bulabilirsiniz.

5. Bir Sürecin Aidiyetinin Okunması

Bu kısımda bir sürecin gerçek ve etkin,kullanıcı ve grup kimliklerinin öğrenilmesinde kullanılan işlevlerin ayrıntılı açıklamalarını bulacaksınız. Bu oluşumları kullanabilmek için `sys/types.h` ve `unistd.h` başlık dosyalarını yazılımınıza dahil etmeniz gerekir.

uid_t veri türü

Kullanıcı kimliğini ifade etmek için kullaılan bir tamsayı veri türüdür. GNU kütüphanesinde bu veri türü **unsigned int**'e eşdeğerdir.

gid_t veri türü

Grup kimliğini ifade etmek için kullaılan bir tamsayı veri türüdür. GNU kütüphanesinde bu veri türü **unsigned int**'e eşdeğerdir.

`uid_t` **getuid**`(void)` işlev

getuid işlevi sürecin gerçek kullanıcı kimliği ile döner.

`gid_t` **getgid**`(void)` işlev

getgid işlevi sürecin gerçek grup kimliği ile döner.

`uid_t` **geteuid**`(void)` işlev

geteuid işlevi sürecin etkin kullanıcı kimliği ile döner.

`gid_t` **getegid**`(void)` işlev

getegid işlevi sürecin etkin grup kimliği ile döner.

```
int getgroups(int     miktar,                                    işlev
              gid_t *gruplar)
```

getgroups işlevi sürecin ek grupları hakkında bilgi almak için kullanılır. Bu grup kimliklerinin en çok *miktar* kadarı *gruplar* dizisine kaydedilir; işlevin dönüş değeri işlev tarafından elde edilebilen grup kimliklerinin sayısı olur. Eğer *miktar* ek grup kimliklerinin sayısından küçükse, işlev **-1** değeri ile döner ve **errno** değişkenine **EINVAL** atanır.

Eğer *miktar* sıfırsa, **getgroups** işlevi ek grup kimliklerin toplam sayısı ile döner. Ek grupları desteklemeyen sistemlerde bu daima sıfır olacaktır.

getgroups işlevinin ek grup kimliklerini öğrenmek için kullanılışına bir örnek:

```
gid_t *
read_all_groups (void)
{
  int ngroups = getgroups (0, NULL);
  gid_t *groups
    = (gid_t *) xmalloc (ngroups * sizeof (gid_t));
  int val = getgroups (ngroups, groups);
  if (val < 0)
    {
      free (groups);
      return NULL;
    }
  return groups;
}
```

6. Kullanıcı Kimliğinin Belirtilmesi

Bu kısımda bir sürecin gerçek ve/veya etkin kullanıcı kimliklerini değiştiren işlevlere yer verilmiştir. Bu oluşumları kullanabilmek için `sys/types.h` ve `unistd.h` başlık dosyalarını yazılımınıza dahil etmeniz gerekir.

```
int seteuid(uid_t etkinkullkim)                                    işlev
```

Bu işlev bir sürecin etkin kullanıcı kimliğini, sürecin etkin kullanıcı kimliğini *etkinkullkim* olarak değiştirmesi mümkünse, *etkinkullkim* ile belirtilen kimliğe ayarlar. Bir ayrıcalıklı süreç (etkin kullanıcı kimliği sıfır olan süreç) kendi etkin kullanıcı kimliğini herhangi bir kullanıcı kimliği ile değiştirebilir. Bir ayrıcalıksız süreç ise kendi etkin kullanıcı kimliğini sadece kendi gerçek kullanıcı kimliğine ya da dosya kullanıcı kimliğine değiştirebilir. Aksi takdirde, sürecin etkin kullanıcı kimliği değişmeyecektir.

seteuid işlevi, kimliği değiştirebilmişse **0** ile döner. **-1** dönüş değeri bir hata oluştuğunu gösterir. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EINVAL`
: *etkinkullkim* argümanının değeri geçersiz

`EPERM`
: Süreç aidiyetini belirtilen kimlikle değiştiremez

_POSIX_SAVED_IDS özelliği olmayan eski sistemlerde bu işlev yoktur.

```
int setuid(uid_t kullkim)                                          işlev
```

Bu işlevi çağıran sürecin yetkisi varsa, sürecin gerçek ve etkin kullanıcı kimliği *kullkim* yapılır. İşlev ayrıca sürecin dosya kullanıcı kimliğini varsa siler. *kullkim* sistemde geçerli herhangi bir kullanıcı kimliği olabilir. (İşlev eski etkin kullanıcı kimliği döndürmez, dolayısıyla bir kere değiştikten sonra tekrar eski etkin kullanıcı kimliğe dönmenin bir yolu yoktur.)

Süreç istenen işlemi yapmaya yetkili değilse ve sistemde **_POSIX_SAVED_IDS** özelliği yoksa bu işlev **seteuid** gibi davranır.

İşlevin dönüş değerleri ve hata durumları **seteuid** ile aynıdır.

```
int setreuid(uid_t gerçekkullkim,                                    işlev
             uid_t etkinkullkim)
```

Bu işlev sürecin gerçek kullanıcı kimliğini *gerçekkullkim* ve etkin kullanıcı kimliğini *etkinkullkim* yapar. Eğer *gerçekkullkim* değeri **-1** ise bu, gerçek kullanıcı kimliğin değiştirilmeyeceği anlamına gelir. Benzer şekilde *etkinkullkim* değeri **-1** ise etkin kullanıcı kimliği değiştirilmez.

setreuid işlevi 4.3 BSD Unix ile uyumluluk adına vardır. Bu işlevi sürecin etkin ve gerçek kullanıcı kimliklerini takaslamak için kullanabilirsiniz. (Ayrıcalıklı süreçler için böyle bir sınırlama yoktur.) Dosya kimlikleri destekleniyorsa, bu işlev bunun için kullanılmaz. Bkz. *Setuid Erişiminin Etkinleştirilmesi ve İptali* (sayfa: 748).

İşlev başarılı ise **0** değilse **-1** döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EPERM
: Sürecin yetkileri işlem için yetersiz; belirtilen kimliğe geçiş için yetkili değilsiniz.

7. Grup Kimliğinin Belirtilmesi

Bu kısımda bir sürecin gerçek ve/veya etkin grup kimliklerini değiştiren işlevlere yer verilmiştir. Bu oluşumları kullanabilmek için `sys/types.h` ve `unistd.h` başlık dosyalarını yazılımınıza dahil etmeniz gerekir.

```
int setegid(gid_t grupkim)                                           işlev
```

Bu işlev bir sürecin etkin grup kimliğini, sürecin etkin grup kimliğini *grupkim* olarak değiştirmesi mümkünse, *grupkim* ile belirtilen kimliğe ayarlar. **seteuid** işlevindeki gibi, bir ayrıcalıklı süreç (etkin kullanıcı kimliği sıfır olan süreç) kendi etkin grup kimliğini herhangi bir kullanıcı kimliği ile değiştirebilir. Bir ayrıcalıksız süreç ise dosya grup kimliğine sahipse, etkin grup kimliğini kendi gerçek grup kimliğine veya kendi dosya grup kimliğine değiştirebilir; aksi takdirde etkin grup kimliği değişmez.

Bir sürecin ayrıcalıklı süreç olması için etkin grup kimliğinin değil, etkin kullanıcı kimliğinin sıfır olması gerektiğini unutmayın. Etkin grup kimliği sadece erişim yetkilerini etkiler.

setegid işlevinin dönüş değerleri ve hata durumları **seteuid** ile aynıdır.

_POSIX_SAVED_IDS özelliği olmayan eski sistemlerde bu işlev yoktur.

```
int setgid(gid_t grupkim)                                            işlev
```

Bu işlevi çağıran sürecin yetkisi varsa, sürecin gerçek ve etkin grup kimliği *grupkim* yapılır. İşlev ayrıca sürecin dosya grup kimliğini varsa siler.

Süreç istenen işlemi yapmaya yetkili değilse, bu işlev **setegid** gibi davranır.

İşlevin dönüş değerleri ve hata durumları **setgid** ile aynıdır.

```
int setregid(gid_t gerçekgrupkim,                                    işlev
             gid_t etkingrupkim)
```

Bu işlev sürecin gerçek grup kimliğini *gerçekgrupkim* ve etkin grup kimliğini *etkingrupkim* yapar. Eğer *gerçekgrupkim* değeri **-1** ise bu, gerçek grup kimliğin değiştirilmeyeceği anlamına gelir. Benzer şekilde *etkingrupkim* değeri **-1** ise etkin grup kimliği değiştirilmez.

setregidişlevi, dosya kimliklerini desteklemeyen 4.3 BSD Unix ile uyumluluk adına vardır. Bu işlevi sürecin etkin ve gerçek kullanıcı kimliklerini takaslamak için kullanabilirsiniz. (Ayrıcalıklı süreçler için böyle bir sınırlama yoktur.) Dosya kimlikleri destekleniyorsa, bu işlev bunun için kullanılmaz. Bkz. *Setuid Erişiminin Etkinleştirilmesi ve İptali* (sayfa: 748).

İşlevin dönüş değerleri ve hata durumları **setreuid** ile aynıdır.

setuid ve **setgid** işlevleri sürecin etkin kullanıcı kimliğinin sıfır olup olmamasına göre farklı davranırlar. Sıfırdan farklıysa, **seteuid** ve **setegid** gibi davranırlar. Bir karışıklıktan kaçınmak için, etkin kullanıcı kimliğinin sıfır olduğunu bilmedikçe ve sürecin aidiyetini kalıcı olarak değiştirmek istemedikçe daima **seteuid** ve **setegid** işlevlerini kullanmanızı öneririz. Bu durum yaygındır ve **login** ve **su** gibi çoğu uygulama buna ihtiyaç duyar.

Eğer yazılımınız **root** dışında bir kullanıcı için setuid ise yetkileri kalıcı olarak değiştirmenin bir yolu yoktur.

Sistem ayrıca yetkin süreçlerin kendi ek grup kimliklerini değiştirmesini mümkün kılmıştır. **setgroups** veya **initgroups** işlevlerini yazılımınızda kullanmak isterseniz, `grp.h` başlık dosyasını yazılımınıza dahil etmelisiniz.

```
int setgroups(size_t miktar,                                         işlev
              gid_t *gruplar)
```

Bu işlev sürecin ek grup kimliklerini ayarlar. Sadece yetkin süreçten çağrılabilir. *miktar* argümanı ile *gruplar* dizisindeki grup kimliklerinin sayısı belirtilir.

İşlev başarılı ise **0** değilse **-1** döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

`EPERM`
Çağıran süreç yetkisiz.

```
int initgroups(const char *kullanıcı,                                işlev
               gid_t       grup)
```

initgroups işlevi sürecin ek gruplarını *kullanıcı* kullanıcısının ek grupları olarak ayarlar. *grup* grubu bunlara eklenir.

Bu işlev grup veritabanını *kullanıcı* ile ilgili grupları bulmak için tarar. Oluşturulan liste için **setgroups** çağrısı yapar.

İşlevin dönüş değerleri ve hata durumları **setgroups** ile aynıdır.

Bir kullanıcının hangi grupların üyesi olduğunu bilmek ama sürecin ek grup kimliklerini değiştirmek istemiyoranız **getgrouplist** işlevini kullanabilirsiniz. **getgrouplist** işlevini kullanabilmek için yazılımınıza `grp.h` başlık dosyasını dahil etmelisiniz.

```
int getgrouplist(const char *kullanıcı,                              işlev
                 gid_t       grup,
                 gid_t      *gruplar,
                 int        *grupsayısı)
```

getgrouplist işlevi *kullanıcı*'nın üyesi olduğu ek grupları saptamak için grup veritabanını tarar. Bu grup kimliklerinin en çok *miktar* kadarı *gruplar* dizisine kaydedilir; işlevin dönüş değeri işlev tarafından elde edilebilen grup kimliklerinin sayısı olur. Eğer *miktar* ek grup kimliklerinin sayısından küçükse, işlev **-1** değeri ile döner ve grupların gerçek sayısını **grupsayısı*'na atar. *grup* grubu bu listeye eklenir.

getgrouplist işlevinin ek grup kimliklerini öğrenmek için kullanılışına bir örnek:

```
gid_t *
supplementary_groups (char *user)
{
  int ngroups = 16;
  gid_t *groups
    = (gid_t *) xmalloc (ngroups * sizeof (gid_t));
  struct passwd *pw = getpwnam (user);

  if (pw == NULL)
    return NULL;

  if (getgrouplist (pw->pw_name, pw->pw_gid, groups, &ngroups) < 0)
    {
      groups = xrealloc (ngroups * sizeof (gid_t));
      getgrouplist (pw->pw_name, pw->pw_gid, groups, &ngroups);
    }
  return groups;
}
```

8. Setuid Erişiminin Etkinleştirilmesi ve İptali

Genellikle setuid yazılımlarda bu özel erişime her zaman gerek duyulmaz. Gerekmedikçe bu erişimin kapalı tutulması daha iyidir, böylece tasarlanmamış bir erişime imkan verilmemiş olur.

Sistem **_POSIX_SAVED_IDS** özelliğini destekliyorsa, bunu **seteuid** ile yapabilirsiniz. Bir oyunun başlatıldığındaki gerçek kullanıcı kimliği **jdoe**, etkin kullanıcı kimliği **games** ise kayıtlı kullanıcı kimliği de **games** olur. Bu yazılımın her iki kullanıcı kimliğinin çalıştırıldığında kaydedilmesi şöyle yapılır:

```
user_user_id = getuid ();game_user_id = geteuid ();
```

Bunun ardından oyun dosyasını erişime şöyle kapatabilirsiniz:

```
seteuid (user_user_id);
```

Oyun dosyasını tekrar erişime açmak isterseniz:

```
seteuid (game_user_id);
```

Bu işlemler sırasında sürecin gerçek kullanıcı kimliği **jdoe** ve dosya kullanıcı kimliği **games** olarak kalır, böylece yazılım etkin kullanıcı kimliğini bunlardan biri yapabilir.

Dosya kullanıcı kimliklerini desteklemeyen sistemlerde, setuid erişimini **setreuid** kullanarak etkin ve gerçek kullanıcı kimlikler arasında takas edebilirsiniz:

```
setreuid (geteuid (), getuid ());
```

Bu daima geçerli bir özel durumdur—hiç başarısız olmaz.

Setuid erişim, gerçek ve etkin kimliklerin takaslanmasından neden etkilenir? Bir oyunun başlatıldığını ve onun gerçek kullanıcı kimliğinin **jdoe** iken etkin kullanıcı kimliğinin **games** olduğunu varsayalım. Bu durumda sürecin

puanların tutulduğu dosyaya yazabildiğini varsayalım. Eğer bu iki kullanıcı kimlik aralarında yer değiştirirse gerçek kullanıcı kimlik **games**, etkin kullanıcı kimlik **jdoe** olur ve süreç artık sadece **jdoe** erişimine sahip olur ve puanların tutulduğu dosyaya yazamaz. Tekrar yapılan bir takasla puanların tutulduğu dosyaya tekrar erişim sağlanır.

Kayıtlı kullanıcı kimlik özelliğinin desteklendiği ve desteklenmediği her iki sistemde de bu özelliği kullanabilmek için önişlemci yordamlarını kullanabilirsiniz:

```
#ifdef _POSIX_SAVED_IDS
  seteuid (user_user_id);
#else
  setreuid (geteuid (), getuid ());
#endif
```

9. Setuid Yazılım Örneği

Buradaki örnekte, kendi etkin kullanıcı kimliğini değiştiren bir yazılım gösterilmiştir.

Örnek, **caber-toss** diye bilinen bir oyundan alınmıştır. Örnek, sadece oyun sürecinin yazabildiği `scores` dosyasının değiştirilmesi için yapılan işlemleri içerir. Oyunun çalıştırılabilir dosyasının setuid bitinin `scores` dosyasının sahibi olan kullanıcı için etkin olarak kaydedildiğini varsayıyoruz. Genellikle sistem yöneticisinin yaptığı bir işlemin sonucu olarak bu amaçla **games** kullanıcısının kullanıldığını varsayalım.

Çalıştırılabilir dosyanın kipinin **4755** olduğunu varsayarsak, **ls -l** komutu şöyle bir çıktı üretir:

```
-rwsr-xr-x   1 games    184422 Jul 30 15:17 caber-toss
```

Setuid biti dosya kiplerinde **s** olarak gösterilir.

`scores` dosyasının kipinin ise **644** olduğunu varsayarak aynı komut şu çıktıyı üretir:

```
-rw-r--r--  1 games           0 Jul 31 15:33 scores
```

Buradaki yazılım parçası kullanıcı kimliklerin nasıl değiştirildiğini gösterir. Yazılım, dosya kimliği desteği varsa bu özelliği yoksa etkin ve gerçek kullanıcıları takaslamak için **setreuid** işlevini kullanmak üzere koşullandırılmıştır.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>


/* Etkin ve gerçek kullanıcı kimlikleri hatırlayalım. */

static uid_t euid, ruid;


/* Etkin kullanıcı kimliği özgün değerine ayarlayalım. */

void
do_setuid (void)
{
  int status;

#ifdef _POSIX_SAVED_IDS
```

```
  status = seteuid (euid);
#else
  status = setreuid (ruid, euid);
#endif
  if (status < 0) {
    fprintf (stderr, "Kullanıcı kimliği etkinleştirilemedi.\n");
    exit (status);
    }
}


/* Etkin kullanıcı kimliği gerçek kullanıcı kimliğe ayarlayalım. */

void
undo_setuid (void)
{
  int status;

#ifdef _POSIX_SAVED_IDS
  status = seteuid (ruid);
#else
  status = setreuid (euid, ruid);
#endif
  if (status < 0) {
    fprintf (stderr, "Kullanıcı kimliği etkinleştirilemedi.\n");
    exit (status);
    }
}

/* Asıl kod. */

int
main (void)
{
  /* Etkin ve gerçek kullanıcı kimlikleri hatırlayalım.  */
  ruid = getuid ();
  euid = geteuid ();
  undo_setuid ();

  /* Oynayalım ve puanları kaydedelim.  */
  ...
}
```

main işlevinin yaptığı ilk işlem etkin kullanıcı kimliğe gerisin geriye gerçek kullanıcı kimliğe ayarlamaktır. Kullanıcı oyunu oynarken bir dosya erişimi yapmak isterse erişim yetkileri gerçek kullanıcı kimliğe göre saptansın diye böyle yapılır. Oyun yazılımı sadece `scores` dosyasına puanı yazacağı zaman etkin kullanıcı kimliğini dosyanın kullanıcı kimliği yapar:

```
/* Puanı kaydedelim. */

int
record_score (int score)
{
  FILE *stream;
  char *myname;

  /* scores dosyasını açalım. */
```

```
  do_setuid ();
  stream = fopen (SCORES_FILE, "a");
  undo_setuid ();

  /* Puanı dosyaya yazalım. */
  if (stream)
    {
      myname = cuserid (NULL);
      if (score < 0)
        fprintf (stream, "%10s: Couldn't lift the caber.\n", myname);
      else
        fprintf (stream, "%10s: %d feet.\n", myname, score);
      fclose (stream);
      return 0;
    }
  else
    return -1;
}
```

10. Setuid Yazılımları Geliştirmek için İpuçları

Setuid yazılımlarla tasarlanmamış kullanıcı erişimi vermek kolaydır, aslında; bundan kaçınmak için çok dikkatli olmanız gerekir. Burada tasarlanmamış erişimden korunma ve ortaya çıktığında zararlarını en aza indirme ile ilgili bazı ipuçlarına yer verilmiştir:

- Çok gerekli olmadıkça özellikle **root** gibi ayrıcalıklı kullanıcı kimliklerle **setuid** yazılımlar kullanmayın. Özkaynaklara erişim için sadece bu özkaynaklara erişim yetkisi olan yeni bir ayrıcalıksız bir kullanıcı oluşturup bu kullanıcıyla o özkaynaklara erişmek daha iyidir. Yazılımınızı özel bir kullanıcı ve grubun kullanabilmesi için yazmanız daha da iyi olur.

- **exec** işlevlerini etkin kullanıcı kimliği değiştirerek kullanırken dikkatli olun. Yazılımınızı kullanarak kullanıcıların değiştirilmiş bir kullanıcı kimlikle keyfi yazılım çalıştırmasına izin vermeyin. Bir kabuğun çalıştırılabilmesi özellikle kötüdür. Daha az belirgin olarak, **execlp** ve **execvp** bir potansiyel risk içerir (yazılımlar kullanıcının **PATH** ortam değişkenine bağlı olarak çalıştırıldığından dolayı).

  Bir yazılımı değiştirilmiş bir kullanıcı kimlikle **exec** işleviyle çalıştırıyorsanız, çalıştırılabilir olarak *dosyanın tam ismini* (sayfa: 233) belirtin ve çalıştırılan bu dosyanın bulunduğu ve eriştiği dizinlerde sıradan kullanıcıların bir değişiklik yapamayacağından emin olun (Bu dosya ismine örneğin bir kabuk yerleştiremesinler ya da yapılandırma dosyasında bir değişiklik yapamasınlar).

  Ayrıca yazılıma aktarılan argümanları umulmadık yan etkilere karşı sınamalısınız. Benzer şekilde, ortam değişkenlerini de incelemelisiniz. Hangi argümanların ve değişkenlerin güvenilir olduğuna karar verdikten sonra tüm diğerlerini reddedin.

  Ayrıcalıklı yazılımlarda **system** işlevini asla kullanmayın, çünkü bu işlev bir kabuk açar.

- Kullanıcı kimliğini sadece yazılımın kullandığı özkaynaklar erişebilen bir kullanıcı için sadece bu özkaynaklara erişirken kullanın, işi bittiği anda etkin kullanıcı kimliği kullanıcının kendi kimliği ile değiştirin. Bkz. *Setuid Erişiminin Etkinleştirilmesi ve İptali* (sayfa: 748).

- Yazılımınızın **setuid** parçasının denetimindeki özkaynaklar dışında başka dosyalara da erişmesi gerekiyorsa, bu dosyalara yazılımı çalıştıran kullanıcının erişim izni olup olmadığına bakmalısınız. Bunu sınamak için **access** işlevini kullanabilirsiniz (bkz. *Erişim İzinleri* (sayfa: 380)); bu işlev etkin kullanıcı kimliği değil gerçek kullanıcı ve grup kimliklerini kullanır.

11. Oturumu Açan Kim?

Bu bölümde açıklanan işlevleri bir süreci çalıştıran kullanıcının ismini saptamakta kullanabilirsiniz. Ayrıca, **getuid** *ailesi işlevlere* (sayfa: 744) bakmayı da unutmayın. Bu bilginin sistem tarafından nasıl toplandığı ve artalanda saklanan bu bilgilerin nasıl denetlendiği/eklendiği/silindiği gibi bilgiler ise *Kullanıcı Hesapları Veritabanı* (sayfa: 752) bölümünde açıklanmıştır.

getlogin işlevi `unistd.h` başlık dosyasında, **cuserid** işlevi ile **L_cuserid** makrosu ise `stdio.h` başlık dosyasında bildirilmiştir.

`char *`**getlogin**`(void)` işlev

> **getlogin** işlevi sürecin denetim uçbiriminde oturum açmış olan kullanıcının ismini içeren bir gösterici ile döner. Bu bilgi saptanamamışsa bir boş gösterici ile döner. Dizge durağan olarak ayrıldığından bu işlevin sonraki çağrılarında veya **cuserid** işlevinin çağrılmasıyla içeriği değişebilir.

`char *`**cuserid**`(char *`*dizge*`)` işlev

> **cuserid** işlevi sürecin etkin kullanıcı kimliği ile ilişkili kullanıcı isminini içeren bir dizgeye gösterici ile döner. *dizge* bir boş gösterici değilse en az **L_cuserid** karakteri tutabilecek bir dizi olmalıdır. Aksi takdirde durağan alanda ayrılmış bir dizgeye gösterici döner. Bu dizge durağan olarak ayrıldığından bu işlevin sonraki çağrılarında veya **getlogin** işlevinin çağrılmasıyla içeriği değişebilir.
>
> Bu işlevin kullanılması artık tavsiye edilmemektedir (XPG4.2'de geri çekilmiş olarak imlidir ve POSIX.1'in daha yeni sürümlerinde kaldırılması planlanmıştır.)

`int` **L_cuserid** makro

> Kullanıcı isminin saklanacağı dizinin ne kadar uzun olabileceğini belirten bir tamsayı sabittir.

Bu işlevler yazılımınızı çalıştıran kullanıcının kim olduğunu ya da o oturumu açan kullanıcının kim olduğunu saptamanıza yarar. (Bunlar setuid yazılımlarla ilgili olarak bahsedilenlerden farklı olabilir.) Bu işlevleri kandırmak için kullanıcı hiçbir şey yapamaz.

Çoğu amaç için, oturum açan kullanıcıyı saptamak için **LOGNAME** ortam değişkenine bakmak faydalıdır. Ancak, kullanıcı **LOGNAME** ortam değişkenini keyfi olarak değiştirebileceğinden bu iyi bir yöntem değildir. Bkz. *Standart Ortam Değişkenleri* (sayfa: 678).

12. Kullanıcı Hesapları Veritabanı

Unix benzeri çoğu işletim sistemi oturum açan kullanıcıların neler yaptıklarını izlemek amacıyla bir kullanıcı hesapları veritabanı tutar. Bu kullanıcı hesapları veritabanında kullanıcının hangi uçbirimden ne zaman oturum açtığı, kullanıcının oturum açtığı uçbirimin süreç kimliği gibi bilgiler yanında, ayrıca sistemin çalışma seviyesi, sistemin yeniden başlatıldığı son tarih ve daha fazlası da saklanabilir.

Kullanıcı hesapları veritabanı genellikle **/etc/utmp**, **/var/adm/utmp** veya **/var/run/utmp** dosyalarında tutulur. Sıradan kullanıcının erişemediği dizinlerde tutuluyor olsalar da, bu dosyalara doğrudan erişilmesi *asla* mümkün olmamalıdır. Kullanıcı hesapları veritabanı ile ilgili okuma va yazma işlemleri bu kısımda açıklanan işlevler ile yapılmalıdır.

12.1. Kullanıcı Hesapları Veritabanına Erişim

Bu amaçla kullanılan işlevler ve veri yapıları `utmp.h` başlık dosyasında bildirilmiştir.

`struct` **exit_status** veri türü

exit_status veri yapısı, kullanıcı hesapları veritabanında **DEAD_PROCESS** olarak imlenmiş süreçlerin çıkış durumları ile ilgili bilgileri saklamakta kullanılır.

`short int` **e_termination**
: Sürecin sonlanma durumu.

`short int` **e_exit**
: Sürecin çıkış durumu.

`struct` **utmp** veri türü

utmp veri yapısı kullanıcı veritabanındaki girdiler hakkında bilgileri tutar. GNU sisteminde şu üyelere sahiptir:

`short int` **ut_type**
: Oturum açma türünü belirtir. **EMPTY**, **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, **NEW_TIME**, **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, **DEAD_PROCESS** ve **ACCOUNTING** sabitlerinden biri olabilir.

`pid_t` **ut_pid**
: Oturumu açan sürecin süreç kimliği numarası.

`char` **ut_line**`[]`
: Uçbirimin aygıt ismi (**/dev/** olmaksızın).

`char` **ut_id**`[]`
: Sürecin inittab kimliği.

`char` **ut_user**`[]`
: Kullanıcının oturum açma ismi.

`char` **ut_host**`[]`
: Kullanıcının kullanarak bağlantı kurduğu konağın ismi.

`struct exit_status` **ut_exit**
: **DEAD_PROCESS** olarak imli sürecin çıkış durumu.

`long` **ut_session**
: Pencereleme için kullanılan oturum kimliği.

`struct timeval` **ut_tv**
: Girdinin yapıldığı zaman. **OLD_TIME** türündeki girdiler için sistem zamanı değişmeden önceki zaman, **NEW_TIME** türündeki girdiler için ise sistem zamanı değiştikten sonraki zamandır.

`int32_t` **ut_addr_v6**`[4]`
: Uzak konağın internet adresi.

ut_type, **ut_pid**, **ut_id**, **ut_tv** ve **ut_host** alanları tüm sistemlerde yoktur. Taşınabilir uygulamalar bu duruma hazırlıklı olmalıdır. Bunu yapmaya yardımcı olmak için, `utmp.h` başlık dosyasında ilgili alanın varlığını belirleyen **_HAVE_UT_TYPE**, **_HAVE_UT_PID**, **_HAVE_UT_ID**, **_HAVE_UT_TV** ve **_HAVE_UT_HOST** makroları tanımlanmıştır. Yazılımcı bu durumları yazılımda **#ifdef**'ler ile sınayabilir.

Aşağıdaki makrolar **utmp** yapısının **ut_type** üyesinde kullanılabilecek değerler olarak tanımlanmıştır. Değerler tamsayı sabitlerdir.

`EMPTY`

Bu makro girdinin hiçbir geçerli kullanıcı hesabı bilgisi içermediğini belirtir.

`RUN_LVL`
Bu makro sistem çalışma seviyesi ile ilgilidir.

`BOOT_TIME`
Bu makro sistemin açıldığı zaman ile ilgilidir.

`OLD_TIME`
Bu makro sistem saatinin değiştiği zaman ile ilgilidir.

`NEW_TIME`
Bu makro sistem değiştikten sonraki zaman ile ilgilidir.

`INIT_PROCESS`
Bu makro init süreci ile çatallanan bir süreç ile ilgilidir.

`LOGIN_PROCESS`
Bu makro kullanıcının oturum açarken kullandığı ilk süreç ile ilgilidir.

`USER_PROCESS`
Bir kullanıcı süreci ile ilgilidir.

`DEAD_PROCESS`
Sonlandırılmış bir süreç ile ilgilidir.

`ACCOUNTING`
???

`ut_line`, **`ut_id`**, **`ut_user`** ve **`ut_host`** dizilerinin boyutları **`sizeof`** işleci kullanılarak bulunabilir.

`time_t` türünde bir üye zamanla ilgili bilgi tutabildiğinden çoğu eski sistemde **`ut_tv`** üyesi yerine **`ut_time`** üyesi bulunur. Bununa birlikte, sadece geriye uyumluluk adına, `utmp.h` başlık dosyasında **`ut_time`**, **`ut_tv.tv_sec`** için bir takma ad olarak tanımlanır.

`void` **`setutent`**`(void)` işlev

Bu işlev kullanıcı hesapları veritabanını taramaya başlamak için açar. Girdileri okumak için **`getutent`**, **`getutid`** veya **`getutline`** işlevlerini; yazmak için ise **`pututline`** işlevini kullanabilirsiniz.

Veritabanı zaten açıksa, girişi veritabanının başlangıcına ayarlar.

`struct utmp *`**`getutent`**`(void)` işlev

`getutent` işlevi kullanıcı hesapları veritabanından sonraki girdiği okur. İşlevin sonraki çağrıları ile üzerine yazılabilecek durağan ayrılmış olarak girdiye bir gösterici ile döner. Yapının bir kopyasını saklamak için dönen verinin içeriğini başka bir değişkende saklamalı ya da veriyi kullanıcı tanımlı bir tamponda saklayan **`getutent_r`** işlevini kullanmalısınız.

Sonraki bir girdinin olmaması durumunda işlev bir boş gösterici döndürür.

`void` **`endutent`**`(void)` işlev

Kullanıcı hesapları veritabanını kapatır.

`struct utmp *`**`getutid`**`(const struct utmp *`*`id`*`)` işlev

Bu işlev veritabanında bulunulan noktadan ileri doğru *id* ile eşleşen girdiyi arar. Eğer *id* yapısının **ut_type** üyesindeki değer **RUN_LVL**, **BOOT_TIME**, **OLD_TIME** veya **NEW_TIME** ise ve veritabanındaki girdinin **ut_type** üyesi bunlardan biri ise eşleşme sağlanır. Eğer, *id* yapısının **ut_type** üyesindeki değer **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS** veya **DEAD_PROCESS** ise ve girdinin **ut_type** üyesindeki değer bu dördünden biri ise ve **ut_id** üyeleri aynıysa eşleşme sağlanır. Bununla birlikte eğer hem *id*'nin hem de okunan girdinin **ut_id** üyesi boşsa, bunun yerine **ut_line** üyesi ile eşleşme aranır. Eğer bir eşleşme sağlanırsa, **getutid** girdiye bir gösterici ile döner. Dönen gösterici durağan ayrılmış olduğundan sonuç sonraki bir **getutent**, **getutid** veya **getutline** çağrısı ile değişebilir. Elde ettiğiniz bilgiyi saklamak isterseniz yapı içeriğini bir değişkene kopyalamalısınız.

Bir eşleşme sağlanamadan veritabanının sonuna gelinmişse işlev bir boş gösterici ile döner.

getutid işlevi okunan son girdiyi arabellekleyebilir. Dolayısıyla, işlevi yapıyla eşleşen birden fazla girdiyi bulmak için kullanıyorsanız, her çağrıdan sonra durağan veriyi sıfırlamanız gerekir. Aksi takdirde, işlev her çağrıda aynı girdiye döner.

```
struct utmp *getutline(const struct utmp *satır)                    işlev
```

Bu işlev veritabanında bulunulan noktadan ileri doğru **ut_type** değeri **LOGIN_PROCESS** veya **USER_PROCESS** olan ve hem *satır* hem de veritabanında **ut_line** üyesi eşleşen girdiyi arar. Eğer bir eşleşme sağlanırsa, **getutline** girdiye bir gösterici ile döner. Dönen gösterici durağan ayrılmış olduğundan sonuç sonraki bir **getutent**, **getutid** veya **getutline** çağrısı ile değişebilir. Elde ettiğiniz bilgiyi saklamak isterseniz yapı içeriğini bir değişkene kopyalamalısınız.

Bir eşleşme sağlanamadan veritabanının sonuna gelinmişse işlev bir boş gösterici ile döner.

getutline işlevi okunan son girdiyi arabellekleyebilir. Dolayısıyla, işlevi yapıyla eşleşen birden fazla girdiyi bulmak için kullanıyorsanız, her çağrıdan sonra durağan veriyi sıfırlamanız gerekir. Aksi takdirde, işlev her çağrıda aynı girdiye döner.

```
struct utmp *pututline(const struct utmp *utmp)                      işlev
```

pututline işlevi kullanıcı hesapları veritabanında uygun yere **utmp* girdisini yerleştirir. Girdiyi doğru yerde bulamazsa, girdiyi yerleştireceği doğru yeri bulmak için **getutid** işlevini kullanır. Ancak, bu **getutent**, **getutid** ve **getutline** tarafından döndürülen durağan yapıyı değiştirmeyecektir. Eğer bu arama başarısız olursa girdi veritabanına eklenir.

pututline işlevi kullanıcı hesapları veritabanına yerleştirilen girdinin bir kopyası ile döner. Girdi veritabanına eklenememişse, bir boş gösterici döner. Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EPERM
: Süreç yeterli yetkiye sahip değil; kullanıcı hesapları veritabanını değiştiremezsiniz

Bahsi geçen tüm **get*** işlevleri bilgiyi saklanmadan önce bir durağan tampon içinde döndürür. Döndürülen veri başka bir evre tarafından değiştirilebileceğinden, bu çok evreli yazılımlarda sorun olabilir. Bu gibi durumlarda kullanılmak üzere GNU C kütüphanesi veriyi kullanıcı tanımlı tamponda döndüren üç ek işlev içerir.

```
int getutent_r(struct utmp  *tampon,                                 işlev
               struct utmp **sonuç)
```

getutent_r işlevi **getutent** işlevi ile aynıdır. Veritabanındaki sonraki girdi ile döner. Fakat veriyi bir durağan ayrılmış tamponda değil, *tampon* parametresi ile gösterilen tamponda saklar.

Çağrı başarılı olursa işlev **0** ile döner ve *sonuç* parametresi ile gösterilen gösterici değişkeni sonucu içeren tampona bir gösterici içerir (çok büyük olasılıkla bu *tampon* değeri olacaktır). Eğer bazı şeyler yanlış giderse işlev **-1** ile döner.

Bu işlev bir GNU oluşumudur.

```
int getutid_r(const struct utmp *id,                                    işlev
              struct utmp       *tampon,
              struct utmp      **sonuç)
```

Bu işlev **getutid** gibi *id* içinde saklanan bilgi ile eşleşen sonraki girdi ile döner. Fakat sonuç, *tampon* parametresi ile gösterilen tamponda saklanır.

Çağrı başarılı olursa işlev **0** ile döner ve *sonuç* parametresi ile gösterilen gösterici değişkeni sonucu içeren tampona bir gösterici içerir (çok büyük olasılıkla bu *tampon* değeri olacaktır). Eğer bazı şeyler yanlış giderse işlev **-1** ile döner.

Bu işlev bir GNU oluşumudur.

```
int getutline_r(const struct utmp *satır,                               işlev
                struct utmp       *tampon,
                struct utmp      **sonuç)
```

Bu işlev **getutline** gibi *satır* içinde saklanan bilgi ile eşleşen sonraki girdi ile döner. Fakat sonuç, *tampon* parametresi ile gösterilen tamponda saklanır.

Çağrı başarılı olursa işlev **0** ile döner ve *sonuç* parametresi ile gösterilen gösterici değişkeni sonucu içeren tampona bir gösterici içerir (çok büyük olasılıkla bu *tampon* değeri olacaktır). Eğer bazı şeyler yanlış giderse işlev **-1** ile döner.

Bu işlev bir GNU oluşumudur.

Kullanıcı hesapları veritabanına ek olarak çoğu sistem buna benzer başka veritabanları da içerir. Örneğin çoğu sistem evvelce açılmış oturumları bir günlük dosyasında tutar (genellikle **/etc/wtmp** veya **/var/log/wtmp** dosyasında).

Hangi veritabanını ile çalışılacağını belirtmek için şu işlev kullanılabilir:

```
int utmpname(const char *dosya)                                         işlev
```

utmpname işlevi çalışılacak veritabanını, ismi *dosya* ile belirtilen veritabanına değiştirir. Öntanımlı olarak, **getutent**, **getutid**, **getutline** ve **pututline** işlevleri kullanıcı hesapları veritabanı ile çalışır.

dosya argümanında kullanılmak üzere tanımlanmış makrolar:

```
char *_PATH_UTMP                                                        makro
```

Bu makro kullanıcı hesapları veritabanını belirtmek için kullanılır.

```
char *_PATH_WTMP                                                        makro
```

Bu makro kullanıcı hesapları günlük dosyasını belirtmek için kullanılır.

utmpname işlevi yeni isim başarıyla saklanmışsa **0** değeri ile, bir hata oluşmuşsa **-1** değeri ile döner. **utmpname** işlevi veritabanını açmayı denemeyeceğinden dönüş değerinin veritabanının başarıyla açıldığına ilişkin bir bilgi vermeyeceğini unutmayın.

Özellikle günlükleme benzeri veritabanları ile çalışmak için GNU C kütüphanesi şu işlevi içerir:

```
void updwtmp(const char          *wtmp_dosyası,                    işlev
             const struct utmp *utmp)
```

updwtmp işlevi **utmp* girdisini *wtmp_dosyası* ile belirtilen veritabanına ekler. *wtmp_dosyası* argümanında kullanılabilecek değerler için **utmpname** işlevinin açıklamasına bakınız.

Taşınabilirlik Bilgisi

Bir çok işletim sistemi bu işlevlerin bir alt kümesini içermekle birlikte, bunlar standartlaşmamıştır. Çoğunlukla dönüş değerleri arasında ince farklar bulunurken, **struct utmp** tanımları arasında da hatırı sayılır farklar vardır. GNU sistemi için yazılım geliştirirken, şüphesiz en iyisi bu bölümde açıklanan işlevlere sadık kalmaktır. Buna rağmen, yine de, yazılımınızın taşınabilir olmasını istiyorsanız *XPG Kullanıcı Hesapları Veritabanı İşlevleri* (sayfa: 757) bölümünde bahsedilen XPG işlevlerini kullanmayı ya da *Oturum Açma ve Kapatma* (sayfa: 759) bölümündeki BSD uyumlu işlevleri kullanmayı tercih edebilirsiniz..

12.2. XPG Kullanıcı Hesapları Veritabanı İşlevleri

Bu işlevler X/Open Taşınabilirlik Rehberinde açıklanmış ve `utmpx.h` başlık dosyasında bildirilmiştir.

```
struct utmpx                                                          veri türü
```

utmpx veri yapısı en azından şu üyeleri içerir:

`short int` **ut_type**
: Oturum açma türünü belirtir; **EMPTY**, **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, **NEW_TIME**, **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS** veya **DEAD_PROCESS** sabitlerinden biri olabilir.

`pid_t` **ut_pid**
: Oturum açan sürecin süreç kimliği numarası.

`char` **ut_line**`[]`
: Uçbirimin aygıt ismi (**/dev/** olmaksızın).

`char` **ut_id**`[]`
: Sürecin inittab kimliği.

`char` **ut_user**`[]`
: Kullanıcının oturum açma ismi.

`struct timeval` **ut_tv**
: Girdinin yapıldığı zaman. **OLD_TIME** türündeki girdiler için sistem zamanı değişmeden önceki zaman, **NEW_TIME** türündeki girdiler için ise sistem zamanı değiştikten sonraki zamandır.

GNU sisteminde, **struct utmpx** yapısı **struct utmp** yapısına bir durum dışında eşdeğerdir: `utmpx.h` başlık dosyasının içerilmesi **struct exit_status** bildirimini görünür yapmaz.

Aşağıdaki makrolar **utmpx** yapısının **ut_type** üyesinde kullanılabilecek değerler olarak tanımlanmıştır. Değerler tamsayı sabitlerdir ve GNU sisteminin `utmp.h` başlık dosyasındaki tanımlarla aynıdır.

`EMPTY`
: Bu makro girdinin hiçbir geçerli kullanıcı hesabı bilgisi içermediğini belirtir.

`RUN_LVL`

Bu makro sistem çalışma seviyesi ile ilgilidir.

`BOOT_TIME`
Bu makro sistemin açıldığı zaman ile ilgilidir.

`OLD_TIME`
Bu makro sistem saatinin değiştiği zaman ile ilgilidir.

`NEW_TIME`
Bu makro sistem değiştikten sonraki zaman ile ilgilidir.

`INIT_PROCESS`
Bu makro init süreci ile çatallanan bir süreç ile ilgilidir.

`LOGIN_PROCESS`
Bu makro kullanıcının oturum açarken kullandığı ilk süreç ile ilgilidir.

`USER_PROCESS`
Bir kullanıcı süreci ile ilgilidir.

`DEAD_PROCESS`
Sonlandırılmış bir süreç ile ilgilidir.

ut_line, **ut_id** ve **ut_user** dizilerinin boyutları **sizeof** işleci kullanılarak bulunabilir.

`void` **setutxent**`(void)` işlev

Bu işlev **setutent** işlevinin benzeridir. GNU sisteminde **setutent** için bir takma addır.

`struct utmpx *`**getutxent**`(void)` işlev

Bu işlev **getutent** işlevinin bir benzeridir, ancak, **struct utmp** yerine **struct utmpx** türünde bir gösterici ile döner. GNU sisteminde **getutent** için bir takma addır.

`void` **endutent**`(void)` işlev

Bu işlev**endutent** işlevinin bir benzeridir. GNU sisteminde **endutent** için bir takma addır.

`struct utmpx *`**getutxid**`(const struct utmpx *`*id*`)` işlev

Bu işlev **getutid** işlevinin bir benzeridir, ancak, **struct utmp** yerine **struct utmpx** türünde bir gösterici ile döner. GNU sisteminde **getutid** için bir takma addır.

`struct utmpx *`**getutxline**`(const struct utmpx *`*satır*`)` işlev

Bu işlev **getutid** işlevinin bir benzeridir, ancak, **struct utmp** yerine **struct utmpx** türünde bir gösterici ile döner. GNU sisteminde **getutid** için bir takma addır.

`struct utmpx *`**pututxline**`(const struct utmpx *`*utmp*`)` işlev

Bu işlev **pututline** işlevinin bir benzeridir, ancak, **struct utmp** yerine **struct utmpx** türünde bir gösterici ile döner. GNU sisteminde **pututline** için bir takma addır.

`int` **utmpxname**`(const char *`*dosya*`)` işlev

Bu işlev **utmpname** işlevinin bir benzeridir. GNU sisteminde **utmpname** için bir takma addır.

Bir geleneksel **struct utmp** yapısı ile bir XPG **struct utmpx** yapısı arasında aşağıdaki işlevlere dönüşüm yapabilirsiniz. GNU sisteminde bu iki yapı aynı olduğundan bu işlevler sadece birer kopyadır.

```
int getutmp(const struct utmpx *utmpx,                                    işlev
            struct utmp        *utmp)
```

getutmp işlevi yapılar mümkün olduğunca uyumlu olacak şekilde *utmpx* yapısından *utmp* yapısına kopyalama yapar.

```
int getutmpx(const struct utmp *utmp,                                     işlev
             struct utmpx      *utmpx)
```

getutmpx işlevi yapılar mümkün olduğunca uyumlu olacak şekilde *utmp* yapısından *utmpx* yapısına kopyalama yapar.

12.3. Oturum Açma ve Kapatma

Bu işlevler BSD'den türetilmiştir, ayrı bir kütüphane olarak **libutil** kütüphanesinde bulunur ve `utmp.h` başlık dosyasında bildirilmişlerdir.

BSD'de **struct utmp** yapısının **ut_user** üyesi **ut_name** ismiyle yeralır. Dolayısıyla, `utmp.h` başlık dosyasında **ut_name**, **ut_user** için bir takma addır.

```
int login_tty(int dosyatanıtıcı)                                          işlev
```

Bu işlev tanıtıcısı *dosyatanıtıcı* olan uçbirimi sürecin denetim uçbirimi yapar. Standart girdi, standart çıktı ve standart hatanın çıktısı bu uçbirime yapılır ve *dosyatanıtıcı* kapatılır.

İşlem başarıyla yerine geririlmişse **0** ile bir hata varsa **-1** ile döner.

```
void login(const struct utmp *girdi)                                      işlev
```

login işlevi kullanıcı hesapları veritabanına bir girdi yerleştirir. **ut_line** üyesine standart girdi üzerindeki uçbirimin ismi atanır. Standart girdi bir uçbirim değilse, uçbirimin ismini saptamak için standart çıktı ya da standart hata kullanılır. **struct utmp** bir **ut_type** üyesine sahipse ona **USER_PROCESS** atanır, **ut_pid** üyesi varsa değeri sürecin süreç kimliği yapılır. Kalan girdiler *girdi*'den kopyalanır.

Girdinin bir kopyası da kullanıcı hesapları günlük dosyasına yazılır.

```
int logout(const char *ut_line)                                           işlev
```

Bu işlev kullanıcı hesapları veritabanında *ut_line* satırındaki kullanıcının oturumu kapattığını belirten bir değişiklik yapar.

logout işlevi girdi veritabanına başaryla yazlımışsa **1** ile bir hata oluşmuşsa **0** ile döner.

```
void logwtmp(const char *ut_line,                                         işlev
             const char *ut_name,
             const char *ut_host)
```

logwtmp işlevi kullanıcı hesapları veritabanına o an için ve *ut_line*, *ut_name* ve *ut_host* argümanları ile belirtilen girdiyi ekler.

Taşınabilirlik Bilgisi

BSD **struct utmp** yapısı sadece **ut_line**, **ut_name**, **ut_host** ve **ut_time** üyelerine sahiptir. Daha eski sistemlerde ise **ut_host** üyesi yoktur.

13. Kullanıcı Veritabanı

Bu bölümde kayıtlı kullanıcılar veritabanında nasıl arama tarama yapılacağı açıklanmıştır. Veritabanı çoğu istemde **/etc/passwd** dosyasında tutulurken bazılarında da özel bir ağ sunucusu üzerinde erişim sağlanır.

13.1. Bir Kullanıcıyı Tanımlayan Veri Yapısı

Sistem kullanıcıları vertabanına erişim için kullanılan veri yapıları ve işlevler `pwd.h` başlık dosyasında bildirilmiştir.

`struct` **passwd** — veri türü

passwd veri yapısı sistem kullanıcıları veritabanındaki girdiler hakkındaki bilgileri tutar. En azından şu üyelere sahiptir:

`char *`**pw_name**
: Kullanıcının oturum açma ismi.

`char *`**pw_passwd**
: Şifrelenmiş parola dizgesi.

`uid_t` **pw_uid**
: Kullanıcı kimliği numarası.

`gid_t` **pw_gid**
: Kullanıcının öntanımlı grup kimliği numarası.

`char *`**pw_gecos**
: Dizge genelde kullanıcının isim ve soyadını içermekle birlikte telefon numarası gibi bilgiler için de kullanılabilir.

`char *`**pw_dir**
: Kullanıcının ev dizini ya da ilk çalışma dizini. Sistem bağımlı yorumlama durumunda bu bir boş gösterici olabilir.

`char *`**pw_shell**
: Kullanıcının öntanımlı kabuğu ya da kullanıcı oturum açtığında çalıştırılacak ilk dosyanın ismi. Sistem öntanımlısının kullanılacağını belirtmek üzere bu üye bir boş gösterici olabilir.

13.2. Bir Kullanıcı Hakkında Bilgi Alınması

Belirli bir kullanıcı için sistem kullanıcıları veritabanında **getpwuid** veya **getpwnam** işlevini kullanarak arama yapabilirsiniz. Bu işlevler `pwd.h` başlık dosyasında bildirilmiştir.

`struct passwd *`**getpwuid**`(uid_t` *kullkim*`)` — işlev

Bu işlev kullanıcı kimliği *kullkim* olan kullanıcı hakkında bilgi içeren durağan olarak ayrılmış bir gösterici ile döner. Bu yapıya sonraki **getpwuid** çağrıları yazabilir.

Dönen bir boş gösterici kullanıcı kimliği *kullkim* olan bir kullanıcı olmadığını belirtir.

```
int getpwuid_r(uid_t           kullkim,                    işlev
               struct passwd  *sonuç_tamponu,
               char           *tampon,
               size_t          tampon_uzunluğu,
               struct passwd **sonuç)
```

Bu işlev kullanıcı kimliği *kullkim* olan kullanıcı hakkında bilgi döndüren **getpwuid** işlevine benzemekle birlikte, bilgi durağan ayrılmış bir tamponda dönmez, kullanıcı tanımlı *sonuç_tamponu* ile gösterilen tamponda saklanır. *tampon* ile gösterilen ek tamponun ilk *tampon_uzunluğu* baytı normalde sonuç yapının elemanları tarafından gösterilen dizgelerden oluşan ek bilgi içerir.

Eğer kullanıcı kimliği *kullkim* olan bir kullanıcı varsa, *sonuç* ile dönen gösterici istenen veriyi içeren kaydı gösterir (yani, *sonuç*, *sonuç_tamponu* değerini içerir). Böyle bir kullanıcı yoksa ya da bir hata oluşmuşsa *sonuç* ile boş gösterici döner. İşlev ya sıfır ya da bir hata kodu ile döner. Eğer *tampon* tamponu gereken tüm bilgiyi saklamak için küçükse **ERANGE** hata kodu döner ve *errno* değişkenine **ERANGE** atanır.

```
struct passwd *getpwnam(const char *isim)                                    işlev
```

Bu işlev kullanıcı ismi *isim* olan kullanıcı hakkında bilgi içeren durağan olarak ayrılmış bir gösterici ile döner. Bu yapıya sonraki **getpwnam** çağrıları yazabilir.

Dönen bir boş gösterici kullanıcı ismi *isim* olan bir kullanıcı olmadığını belirtir.

```
int getpwnam_r(const char      *isim,                                        işlev
               struct passwd  *sonuç_tamponu,
               char           *tampon,
               size_t          tampon_uzunluğu,
               struct passwd **sonuç)
```

Bu işlev kullanıcı ismi *isim* olan kullanıcı hakkında bilgi döndüren **getpwnam** işlevine benzemekle birlikte, bilgi durağan ayrılmış bir tamponda dönmez, kullanıcı tanımlı *sonuç_tamponu* ve *tampon* ile gösterilen tamponlarda saklanır.

Dönüş değerleri **getpwuid_r** işlevi ile aynıdır.

13.3. Kullanıcı Listesinin Taranması

Bu bölümde bir yazılımın sistemdeki tüm kullanıcılar hakkındaki bilgileri bir kerede bir kullanıcı olarak nasıl okuyabileceği anlatılmıştır. Burada bahsedilen işlevler `pwd.h` başlık dosyasında bildirilmiştir.

Belli bir dosyadaki kullanıcı girdilerini okumak için **fgetpwent** işlevini kullanabilirsiniz.

```
struct passwd *fgetpwent(FILE *akım)                                         işlev
```

Bu işlev *akım*'dan sonraki kullanıcı girdisini okur ve girdiye bir gösterici ile döner. Yapı durağan olarak ayrıldığından sonraki **fgetpwent** çağrıları üzerine yazabilir. Aldığınız bilgiyi saklamak istiyorsanız yapıyı bir değişkene kopyalamalısınız.

Akım, standart parola veritabanı dosyası ile aynı biçimdeki bir dosyaya karşılık olmalıdır.

```
int fgetpwent_r(FILE           *akım,                                        işlev
                struct passwd  *sonuç_tamponu,
                char           *tampon,
                size_t          tampon_uzunluğu,
                struct passwd **sonuç)
```

Bu işlev *akım*'dan sonraki kullanıcı girdisini okuyan **fgetpwent** işlevine benzer. Fakat sonuç *sonuç_tamponu* ile gösterilen yapı içinde döner. *tampon* ile gösterilen ek tamponun ilk *tampon_uzunluğu* baytı, normalde sonuç yapının elemanları tarafından gösterilen dizgelerden oluşan ek bilgi içerir.

Akım, standart parola veritabanı dosyası ile aynı biçimdeki bir dosyaya karşılık olmalıdır.

İşlev sıfırla dönmüşse *sonuç* istenen veriyi içeren yapıya göstericidir. Bir hata oluşmuşsa *sonuç* bir boş gösterici içerir ve işlev sıfırdan faklı bir değerle döner.

Kullanıcı veritabanındaki tüm girdiler **setpwent**, **getpwent** ve **endpwent** işlevleri ile taranabilir.

```
void setpwent(void)                                                    işlev
```

Bu işlev, kullanıcı veritabanını okumakta kullanılan **getpwent** ve **getpwent_r** işlevleri için bir akım ilklendirir.

```
struct passwd *getpwent(void)                                          işlev
```

getpwent işlevi **setpwent** işlevi ile ilklendirilen akımdan sonraki girdiyi okur. Yapı durağan olarak ayrıldığından sonraki **getpwent** çağrıları üzerine yazabilir. Aldığınız bilgiyi saklamak istiyorsanız yapıyı bir değişkene kopyalamalısınız.

Okunacak başka girdi kalmadığında işlev bir boş gösterici ile döner.

```
int getpwent_r(struct passwd  *sonuç_tamponu,                          işlev
               char           *tampon,
               int             tampon_uzunluğu,
               struct passwd **sonuç)
```

Bu işlev, **setpwent** işlevi ile ilklendirilen akımdan sonraki girdiyi döndüren **getpwent** işlevine benzer. İstenen bilgiyi **fgetpwent_r** işlevi gibi kullanıcı tanımlı *sonuç_tamponu* ve *tampon* tamponları ile döndürür.

Dönüş değerleri **fgetpwent_r** ile aynıdır.

```
void endpwent(void)                                                    işlev
```

getpwent veya **getpwent_r** tarafından kullanılan dahili akımı kapatır.

13.4. Bir Kullanıcı Girdisinin Yazılması

```
int putpwent(const struct passwd *p,                                   işlev
             FILE *akım)
```

Bu işlev **p* kullanıcı girdisini *akım* akımına standart kullanıcı veritabanı biçimiyle yazar. İşlev başarılı ise sıfırla aksi takdirde sıfırdan farklı bir değerle döner.

Bu işlev SVID ile uyumluluk adına vardır. Bu işlevi kullanmaktan kaçınmanızı öneririz, çünkü **struct passwd** yapısının standart tek üyesi dışında üye içermediği kabulüne duyarlıdır; geleneksel Unix veritabanı ile kullanıcılar hakkındaki diğer genişletilmiş bilgilerin karışık olduğu bir sistem üzerinde bu işlevi kullanarak bir girdinin eklenmesi kaçınılmaz olarak önemli bilginin çoğunu dışarda bırakırdı.

Grup veya kullanıcı ismi bir – veya + işareti ile başlıyorsa grup ve kullanıcı kimliği alanları boş kalır.

putpwent işlevi `pwd.h` başlık dosyasında bildirilmiştir.

14. Grup Veritabanı

Bu bölümde kayıtlı gruplar veritabanında nasıl arama tarama yapılacağı açıklanmıştır. Veritabanı çoğu istemde **/etc/group** dosyasında tutulurken bazılarında da özel bir ağ sunucusu üzerinde erişim sağlanır.

14.1. Grup Veri Yapısı

Sistem grup veritabanına erişim için kullanılan işlevler ve veri yapıları `grp.h` başlık dosyasında bildirilmiştir.

```
struct group                                                           veri türü
```

group yapısı sistem grup veritabanındaki bir girdideki bilgileri tutar. En azından şu üyelere sahiptir:

`char *`**gr_name**
Grubun ismi.

`gid_t` **gr_gid**
Grubun grup kimliği

`char **`**gr_mem**
Gruptaki kullanıcıların isimlerini içeren bir gösterici vektörüdür. Her kullanıcı ismi bir boş karakter sonlandırmalı dizgedir ve vektörün kendisi de bir boş gösterici ile sonlandırılır.

14.2. Bir Grup Hakkında Bilgi Alınması

Belirli bir grup için sistem grup veritabanında bir arama yapmak için **getgrgid** veya **getgrnam** işlevini kullanabilirsiniz. Bu işlevler `grp.h` başlık dosyasında bildirilmiştir.

```
struct group *getgrgid(gid_t grupkim)                                    işlev
```

Bu işlev grup kimliği *grupkim* olan grup hakkında bilgi içeren durağan olarak ayrılmış bir gösterici ile döner. Bu yapıya sonraki **getgrgid** çağrıları yazabilir.

Dönen bir boş gösterici grup kimliği *grupkim* olan bir grup olmadığını belirtir.

```
int getgrgid_r(gid_t           grupkim,                                  işlev
               struct group  *sonuç_tamponu,
               char          *tampon,
               size_t         tampon_uzunluğu,
               struct group **sonuç)
```

Bu işlev grup kimliği *grupkim* olan grup hakkında bilgi döndüren **getgrgid** işlevine benzemekle birlikte, bilgi durağan ayrılmış bir tamponda dönmez, kullanıcı tanımlı *sonuç_tamponu* ile gösterilen tamponda saklanır. *tampon* ile gösterilen ek tamponun ilk *tampon_uzunluğu* baytı normalde sonuç yapının elemanları tarafından gösterilen dizgelerden oluşan ek bilgi içerir.

Eğer grup kimliği *grupkim* olan bir grup varsa, *sonuç* ile dönen gösterici istenen veriyi içeren kaydı gösterir (yani, *sonuç*, *sonuç_tamponu* değerini içerir). Böyle bir grup yoksa ya da bir hata oluşmuşsa *sonuç* ile boş gösterici döner. İşlev ya sıfır ya da bir hata kodu ile döner. Eğer *tampon* tamponu gereken tüm bilgiyi saklamak için küçükse **ERANGE** hata kodu döner ve *errno* değişkenine **ERANGE** atanır.

```
struct group *getgrnam(const char *isim)                                 işlev
```

Bu işlev grup ismi *isim* olan grup hakkında bilgi içeren durağan olarak ayrılmış bir gösterici ile döner. Bu yapıya sonraki **getgrnam** çağrıları yazabilir.

Dönen bir boş gösterici grup ismi *isim* olan bir grup olmadığını belirtir.

```
int getgrnam_r(const char    *isim,                                      işlev
               struct group  *sonuç_tamponu,
               char          *tampon,
               size_t         tampon_uzunluğu,
               struct group **sonuç)
```

Bu işlev grup ismi *isim* olan grup hakkında bilgi döndüren **getgrnam** işlevine benzemekle birlikte, bilgi durağan ayrılmış bir tamponda dönmez, kullanıcı tanımlı *sonuç_tamponu* ve *tampon* ile gösterilen tamponlarda saklanır.

Dönüş değerleri **getgrgid_r** işlevi ile aynıdır.

14.3. Grup Listesinin Taranması

Bu bölümde bir yazılımın sistemdeki tüm gruplar hakkındaki bilgileri bir kerede bir grup olarak nasıl okuyabileceği anlatılmıştır. Burada bahsedilen işlevler `grp.h` başlık dosyasında bildirilmiştir.

Belli bir dosyadaki kullanıcı girdilerini okumak için **fgetgrent** işlevini kullanabilirsiniz.

```
struct group *fgetgrent(FILE *akım)                                    işlev
```

Bu işlev *akım*'dan sonraki grup girdisini okur ve girdiye bir gösterici ile döner. Yapı durağan olarak ayrıldığından sonraki **fgetgrent** çağrıları üzerine yazabilir. Aldığınız bilgiyi saklamak istiyorsanız yapıyı bir değişkene kopyalamalısınız.

Akım, standart grup veritabanı dosyası ile aynı biçimdeki bir dosyaya karşılık olmalıdır.

```
int fgetgrent_r(FILE           *akım,                                  işlev
                struct group   *sonuç_tamponu,
                char           *tampon,
                size_t          tampon_uzunluğu,
                struct group **sonuç)
```

Bu işlev *akım*'dan sonraki kullanıcı girdisini okuyan **fgetgrent** işlevine benzer. Fakat sonuç *sonuç_tamponu* ile gösterilen yapı içinde döner. *tampon* ile gösterilen ek tamponun ilk *tampon_uzunluğu* baytı, normalde sonuç yapının elemanları tarafından gösterilen dizgelerden oluşan ek bilgi içerir.

Akım, standart grup veritabanı dosyası ile aynı biçimdeki bir dosyaya karşılık olmalıdır.

İşlev sıfırla dönmüşse *sonuç* istenen veriyi içeren yapıya göstericidir. Bir hata oluşmuşsa *sonuç* bir boş gösterici içerir ve işlev sıfırdan faklı bir değerle döner.

Grup veritabanındaki tüm girdiler **setgrent**, **getgrent** ve **endgrent** işlevleri ile taranabilir.

```
void setgrent(void)                                                    işlev
```

Bu işlev, grup veritabanını okumakta kullanılan **getgrent** ve **getgrent_r** işlevleri için bir akım ilklendirir.

```
struct group *getgrent(void)                                           işlev
```

getgrent işlevi **setgrent** işlevi ile ilklendirilen akımdan sonraki girdiyi okur. Yapı durağan olarak ayrıldığından sonraki **getgrent** çağrıları üzerine yazabilir. Aldığınız bilgiyi saklamak istiyorsanız yapıyı bir değişkene kopyalamalısınız.

```
int getgrent_r(struct group  *sonuç_tamponu,                           işlev
               char          *tampon,
               size_t         tampon_uzunluğu,
               struct group **sonuç)
```

Bu işlev, **setgrent** işlevi ile ilklendirilen akımdan sonraki girdiyi döndüren **getgrent** işlevine benzer. İstenen bilgiyi **fgetgrent_r** işlevi gibi kullanıcı tanımlı *sonuç_tamponu* ve *tampon* tamponları ile döndürür.

İşlev sıfırla dönmüşse *sonuç* istenen veriyi içeren yapıya göstericidir (normalde *sonuç_tamponu* ile aynıdır). Bir hata oluşmuşsa *sonuç* bir boş gösterici içerir ve işlev sıfırdan faklı bir değerle döner.

`void endgrent(void)` işlev

getgrent veya **getgrent_r** tarafından kullanılan dahili akımı kapatır.

15. Kullanıcı ve Grup Veritabanı Örneği

Burada veritabanları ile çalışan işlevlerin kullanımını gösteren örnek bir yazılıma yer verilmiştir. Bu yazılım, kendisini çalıştıran kullanıcı hakkında bazı bilgiler basar.

```
#include <grp.h>
#include <pwd.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int
main (void)
{
  uid_t me;
  struct passwd *my_passwd;
  struct group *my_group;
  char **members;

  /* Kullanıcı kimliği hakkında bilgi alalım. */
  me = getuid ();
  my_passwd = getpwuid (me);
  if (!my_passwd)
    {
      printf ("%d kullanıcı kimlikli bir kullanıcı bulunamadı.\n", (int) me);
      exit (EXIT_FAILURE);
    }

  /* Bilgileri basalım. */
  printf ("Ben %s.\n", my_passwd->pw_gecos);
  printf ("Oturum açma ismim %s.\n", my_passwd->pw_name);
  printf ("Kullanıcı kimliğim %d.\n", (int) (my_passwd->pw_uid));
  printf ("Ev dizinim %s.\n", my_passwd->pw_dir);
  printf ("Öntanımlı kabuğum %s.\n", my_passwd->pw_shell);

  /* Öntanımlı grup kimliği hakkında bilgi alalım */
  my_group = getgrgid (my_passwd->pw_gid);
  if (!my_group)
    {
      printf ("%d grup kimlikli bir grup bulunamadı.\n",
              (int) my_passwd->pw_gid);
      exit (EXIT_FAILURE);
    }

  /* Bilgileri basalım. */
  printf ("Öntanımlı grubum %s (%d).\n",
          my_group->gr_name, (int) (my_passwd->pw_gid));
  printf ("Üyesi olduğum gruplar:\n");
  members = my_group->gr_mem;
  while (*members)
    {
      printf ("  %s\n", *(members));
      members++;
```

```
    }

  return EXIT_SUCCESS;
}
```

Bu yazılımın çıktısı şöyle olurdu:

```
Ben NBB.
Oturum açma ismim nilgun.
Kullanıcı kimliğim 502.
Ev dizinim /home/nilgun.
Öntanımlı kabuğum /bin/bash.
Öntanımlı grubum belgeler (526).
Üyesi olduğum gruplar:
  nilgun
```

16. Ağ Grubu Veritabanı

16.1. Ağgrubu Verisi

Bazan kullanıcıları başka kriterlere göre gruplamak faydalı olur (Bkz. *Grup Veritabanı* (sayfa: 762)). Örneğin, belli bir grubu bir makina ile ilişkilendirmek yararlıdır. Diğer yandan konak isimlerinin gruplanması artık desteklenmemektedir.

Sun Microsystems SunOS üzerinde yeni bir veritabanı çeşidi bulunur, ağgrubu veritabanı. Konakların, kullanıcıların ve isim alanlarının özgürce her birine bir isim vererek gruplanabilmesini mümkün kılar. Özetle, bir ağgrubu bir konak ismi, bir kullanıcı ismi ve bir alan isminden oluşan üçlülerin listesidir. Girdilerin her biri tüm girdilerle eşleşebilen bir kalıp girdisi olabilir. Son olanak ise, ayrıca diğer ağgruplarının isimlerinin bir ağgrubunu belirten listede verilebilmesidir. Böylece döngü oluşturmadan keyfi hiyerarşiler oluşturulabilmektedir.

Sun'ın gerçeklemesi ağgruplarına sadece **nis** veya **nisplus** hizmeti için izin verir, bkz. *NSS Yapılandırma Dosyasındaki Hizmetler* (sayfa: 735). GNU C kütüphanesindeki gerçekleme ise böyle bir sınırlama içermez. Girdi hizmetlerinin her birinin girdisi aşağıdaki biçimde olmalıdır:

grupismi (*grupismi* | (*konakismi*,*kullanıcısmi*,*alanismi*))+

Üçlüdeki alanların her biri hiçbir şeyle eşleşmediğini belirtmek üzere boş olabilir. İşlevleri açıklarken göreceğiniz gibi tamamen zıddı bir durumda kullanışlıdır. Örneğin, hiçbir girdi ile eşleşmeyen girdiler olabilir. Bunun gibi girdiler için tek bir karakterden oluşan bir isim, **-** kullanılacaktır.

16.2. Bir Ağgrubu Hakkında Bilgi Alınması

Ağgrubu erişim işlevleri tüm diğer sistem veritabanı işlevlerinden birazcık farkıdır. Tek bir ağgrubu çok sayıda girdi içerebildiğinden iki adımlık bir işlem gerekir. Önce tek bir ağgrubu seçilir, sonra da bu ağgrubundaki tüm girdiler üzerinde yineleme yapılır. Bu işlevler `netdb.h` başlık dosyasında bildirilmiştir.

`int` **`setnetgrent`**`(const char *`*`netgrup`*`)` işlev

Bu işleve yapılan bir çağrı *netgrup* isimli ağgrubundaki tüm girdiler üzerinde yinelenecek **getnetgrent** çağrılarını mümkün kılmak üzere kütüphanenin dahili durumunu ilklendirir.

Çağrı başarılı olduğunda (örn, bu isimde bir ağ grubu varsa) **1** değeri döner. **0** dönüş değeri ya bu isimde bir ağgrubu olmadığını ya da bir hata oluştuğunu belirtir.

Ağgruplarının yinelenemesi için sadece tek bir durumun var olduğunu hatırlamak önemlidir. Yazılımcı **getnetgrent_r** işlevini kullansa bile daima tek bir seferde sadece bir ağgrubu işlenebildiğinden sonuç gerçekte evresel olmayacaktır. Eğer yazılım aynı anda bir ağgrubundan fazlasını işlemeyi gerektiriyorsa, yazılımcı harici kiltleme kullanarak bunu korumalıdır. Bu sorun SunOS'daki özgün ağgrubu gerçeklemesinde vardır ve uyumlu kalmak gerektiğinden bunu değiştirmek mümkün değildir.

Ağgrupları katmanını başka işlevler de kullanır. Şimdilik bunlar NSS gerçeklemesinin **compat** hizmeti ile ilgili parçaları ile **innetgr** işlevidir.

```
int getnetgrent(char **konak,                                    işlev
                char **kullanıcı,
                char **alanadı)
```

Bu işlev o an seçili ağgrubunun sonraki işlenmemiş ilk girdisini döndürür. Adresleri *konak*, *kullanıcı* ve *alanadı* argümanlarında aktarılan dizge göstericileri başarılı bir çağrı sonrasında ilgili dizgeleri içerecektir. Eğer sonraki girdide dizge boşsa gösterici **NULL** değerine sahip olur. Dönen dizge göstericileri sadece çağrılmış bir ağgrubu işlevinin olmaması halinde geçerlidir.

Sonraki girdi başarıyla okunmuşsa işlev **1** değeri ile döner. **0** değeri böyle bir girdinin olmadığını ya da dahili bir hata oluştuğunu gösterir.

```
int getnetgrent_r(char **konak,                                  işlev
                  char **kullanıcı,
                  char **alanadı,
                  char  *tampon,
                  int    tampon_uzunluğu)
```

Bu işlev bir şey dışında **getnetgrent** işlevinin benzeridir: *konak*, *kullanıcı* ve *alanadı* dizge göstericilerinin gösterdiği dizgeler, *tampon* ile başlayan *tampon_uzunluğu* baytlık tampona yerleştirilir. Bunun anlamı, çağrılmış bir ağgrubu işlevinin olması halinde bile dönen değerlerin geçerli olduğudur.

Sonraki girdi başarıyla okunmuşsa ve tamponda dizgeler için yeterince yer varsa işlevin dönüş değeri **1**'dir. **0** dönüş değeri böyle bir girdinin olmadığı, tamponun yetersiz olduğu ya da dahili bir hata oluştuğunu gösterir.

Bu işlev bir GNU oluşumudur. SunOS'un özgün gerçeklemesi böyle bir işlevi içermemektedir.

```
void endnetgrent(void)                                           işlev
```

Bu işlev, seçilen son ağgrubu ile ilgili olarak ayrılmış tamponları serbest bırakır. Sonuç olarak, bu çağrıdan sonra yapılan **getnetgrent** çağrılarının döndürdüğü tüm dizge göstericileri geçersizdir.

16.3. Ağgrubu Üyeliğinin Sınanması

Çoğunlukla ilgilenilen konu, bir girdinin seçili ağgrubunun parçası olup olmadığı, olduğundan tüm ağgrubunun taranması gerekmez.

```
int innetgr(const char *netgrup,                                 işlev
            const char *konak,
            const char *kullanıcı,
            const char *alanadı)
```

Bu işlev *konak*, *kullanıcı* ve *alanadı* parametreleri ile belirtilen üçlünün *netgrup* ağgrubunun bir parçası olup olmadığına bakar. Bu işlevi kullanmanın bazı getirileri vardır:

1. Dahili kilitleme uygulandığından genel ağgrubu katmanında başka hiç bir ağgrubu işlevi kullanılamaz.

2. İşlev bu işlem için **set**/**get**/**endnetgrent** gibi işlevlerden daha verimli olacak şekilde tasarlanmıştır.

konak, *kullanıcı* ve *alanadı* göstericilerden herhangi biri bu konumda herhangi bir girdinin geçerli olacağını belirtmek üzere **NULL** olarak verilebilir. Bu herhangi bir ismin **-** olarak belirtilmesi durumunda da geçerlidir.

Belirtilen üçlü ağgrubundaki bir girdi ile eşleştirilebilmişse işlevin dönüş değeri **1**'dir. **0** dönüş değeri böyle bir ağgrubunun olmadığı, ağgrubunda böyle bir üçlünün olmadığı ya da dahili bir hatanın oluştuğu anlamına gelir.

XXX. Sistem Yönetimi

İçindekiler

Bu oylumda bir sürecin altında çalıştığı sistemin (işletim sistemi ve donanım) denetimi ve hakkında bilgi alınması ile ilgili oluşumlar açıklanacaktır. Genellikle bilgilendirme oluşumlarını herkes kullanabilir ama sadece ayrıcalıklı süreç değişiklik yapabilir.

Bir dosya isminin uzunluğu gibi sistem parametreleri hakkında bilgi almak için bu oluşumlar sistemin içinde oluşturulur. Bkz. *Sistem Yapılandırma Parametreleri* (sayfa: 784).

1. Konak İsimlendirmesi

Bu kısımda yazılımınızın çalıştığı belli bir sistemin nasıl isimlendirildiği açıklanacaktır. Önce internetin gelişim tarihçesinden dolayı biraz karmaşık olan bilgisayar sistemleri isimlendirilmesininin çeşitli yollarını gözden geçirelim.

Her Unix sistem (bir konak olarak da bilinir) bir ağa bağlı olup olmamasına bağlı olarak bir konak ismine sahiptir. En basit şekli, bilgisayar ağları ortada yokken **`civciv`** gibi tek bir sözcüktü.

Fakat bir sistem İnternete ya da bir ağa bağlandığında Alan Adı Sisteminin (DNS) parçası olarak daha titiz bir isimlendirme uzlaşımına uyum sağlaması gerekir. DNS'de her konak ismi iki parçadan oluşur:

1. makina adı
2. alan ismi

"makina adı" biraz "konak ismi" gibi görünür, fakat aynı şey değildir ve çoğunlukla konak adlarının tamamına hatalı olarak "alan adları" derler.

DNS'de, tam konak adına FQDN (Fully Qualified Domain Name – Tamamen Nitelenmiş Alan Adı) denir ve makina adı, nokta, alan adı şeklindedir. Alan adının kendisi de aslında noktalarla ayrılmış çok sayıda bileşene sahiptir. Örneğin, bir sistemin makina adı **`truva`** ve alan adı **`ulakbim.gov.tr`** olsun, bu durumda bu sistemin konak ismi (ya da FQDN'si) **`truva.ulakbim.gov.tr`** olur.

Bu karışıklığa ek olarak bir bilgisayarın bilinme gereksinimini karşılamak için DNS tek isim uzayı değildir. Diğer bir isim uzayı da NIS (nam–ı diğer YP) isim uzayıdır. NIS'in amaçlarına uygun olarak NIS alan adı ya da YP alan adı olarak bilinen başka bir alan adı daha vardır ve DNS alan adının gerek duyduğu hiçbir şeye ihtiyaç duymaz.

DNS'de aslında herşey biraz daha karışıktır, aynı sisteme çok sayıda konak ismi vermek mümkündür. Bununla beraber, bunlardan sadece biri gerçek konak ismidir ve "asıl konak ismi" (canonical FQDN) adını alır.

Bazı bağlamlarda, konak ismine "düğüm ismi" dendiği de olur.

DNS konak isimlendirmesi hakkında daha fazla bilgi edinmek için *Konak İsimleri* (sayfa: 412) bölümüne bakınız.

Bu kısımdaki işlevlerin prototipleri `unistd.h` dosyasında bulunur.

hostname, **hostid** ve **domainname** komutları bu işlevleri çağırarak çalışır.

```
int gethostname(char  *isim,                                                    işlev
                size_t boyut)
```

Bu işlev çağrıldığı sistemin konak ismini *isim* dizisinde döndürür. *boyut* argümanı bu dizinin bayt cinsinden boyutudur. Dönen değer DNS makina adı değildir. Sistem DNS'de kayıtlı ise bu konak ismidir (FQDN) (yukarı bakınız).

İşlevin normal dönüş değeri **0**'dır, **−1** dönüşse hata oluşmuş demektir. GNU C kütüphanesinde **gethostname** işlevi *boyut* yeterli değilse başarısız olur; bu durumda daha büyük bir diziyle yeniden denemelisiniz. Aşağıdaki hata durumu bu işlev için tanımlanmıştır:

ENAMETOOLONG
boyut argümanı "konak adının uzunluğu artı bir" değerinden küçük

Bazı sistemlerde, olası en büyük konak ismi uzunluğu için bir sembol vardır: **MAXHOSTNAMELEN**. `sys/param.h` dosyasında tanımlıdır. Fakat bunun mevcudiyetine fazla güvenmeyin, en iyisi bir başarısızlık durumunda yeniden denemektir.

gethostname işlevi, konak ismi *isim* dizisine tam olarak sığmasa bile sığdığı kadarını diziye yerleştirir. Bazı durumlarda kırpılmış bir konak ismi bile yeterlidir. Böyle bir durumda hata kodunu yoksayabilirsiniz.

```
int sethostname(const char *isim,                                               işlev
                size_t      uzunluk)
```

sethostname işlevi çağrıldığı sistemin konak ismini *uzunluk* baytlık bir dizge olarak *isim* dizgesindeki değere ayarlar. Sadece bir ayrıcalıklı süreç bu işlemi yapabilir.

Genellikle **sethostname** işlevi sistem açılışı sırasında sadece bir kere çağrılır. Çoğunlukla, işlevin çağrıldığı yazılım konak ismini **/etc/hostname** dosyasından alır. Konak ismi olarak DNS makina adını değil, tam konak ismini verdiğinizden emin olun.

İşlevin normal dönüş değeri **0**'dır, **−1** dönüşse hata oluşmuş demektir. Aşağıdaki hata durumu bu işlev için tanımlanmıştır:

EPERM
Süreç yetkisiz olduğundan konak ismini veremiyor

```
int getdomainnname(char  *isim,                                                 işlev
                   size_t uzunluk)
```

getdomainname işlevi çağrıldığı sistemin NIS (YP) alan adı ile döner. Bu pek popüler bir DNS alan adı değildir. Bunun yerine **gethostname** işlevi önerilir.

Bu işlevin özellikleri yukarıdaki **gethostname** işlevinin benzeridir.

```
int setdomainnname(const char *isim,                                            işlev
                   size_t      uzunluk)
```

setdomainname işlevi çağrıldığı sistemin NIS (YP) alan adını ayarlar. Bu pek popüler bir DNS alan adı değildir. Bunun yerine **sethostname** işlevi önerilir.

Bu işlevin özellikleri yukarıdaki **sethostname** işlevinin benzeridir.

`long int` **gethostid**`(void)` işlev

Bu işlev çağrıldığı makinanın "konak kimliği" ile döner. Teamülen, bu genelde **long int** bir değere dönüştürülmüş olarak makinanın birincil Internet IP adresidir. Bununla birlikte, bazı sistemlerde bu adres anlamsızdır ama her makina için tam kodlu eşsiz bir numaradır.

Bu geniş çapta kullanılmaz. BSD 4.2'de vardı ama BSD 4.4'de kaldırıldı. POSIX için gerekli değildir.

IP adresini sorgulamak için en uygun yöntem **gethostname** işlevinin sonucu ile **gethostbyname** çağrısı yapmaktır. IP adresleri hakkında daha fazla bilgi için *Konak Adresleri* (sayfa: 408) bölümüne bakınız.

`int` **sethostid**`(long int` *kimlik*`)` işlev

Bu işlev çağrıldığı makinanın "konak kimliği"ni *kimlik* değerine ayarlar. Sadece ayrıcalıklı süreç bu işlemi yapabilir. Sistem açılışı sırasında sadece bir kere çağrılır.

Sisteme birincil IP adresi vermenin uygun yolu **gethostname** tarafından döndürülen sistem konak ismi ile IP adresini ilişklendirecek IP adres çözümleyiciyi yapılandırmaktır. Örneğin, sistem için **/etc/hosts** dosyasına bir kayıt yerleştirin.

Konak kimlikleri için yukarıdaki **gethostid** işlevinde daha fazla bilgi bulabilirsiniz.

İşlevin normal dönüş değeri **0**'dır, **-1** dönmüşse hata oluşmuş demektir. Aşağıdaki hata durumları bu işlev için tanımlanmıştır:

EPERM
Süreç yetkisiz olduğundan konak kimliğini veremiyor

ENOSYS
İşletim sistemi konak kimliği verilmesini desteklemiyor. Bazı sistemlerde konak kimliği anlamsızdır ama her makina için tam kodlu eşsiz bir numaradır.

2. Platform Türü İsimlendirmesi

Yazılımınızın çalıştığı bilgisayarın türü hakkında bilgi edinmek için **uname** işlevini kullanabilirsiniz. Bu işlev ve onunla ilgili veri türü `sys/utsname.h` başlık dosyasında bildirilmiştir.

Ek olarak, **uname** işlevi yazılımınızın çalıştığı sistemi kimliklendiren bazı bilgiler verir. Bu bilgi, *Konak İsimlendirmesi* (sayfa: 769) bölümünde açıklanan işlevlerle alınan bilginin aynısıdır.

`struct` **utsname** veri türü

utsname yapısı **uname** işlevi tarafından döndürülen bilgiyi tutar. Şu üyelere sahiptir:

`char` **sysname**`[]`
Kullanılan işletim sisteminin ismi.

`char` **release**`[]`
İşletim sisteminin o anki dağıtım seviyesi (çekirdeğin kaçıncı defa derlendiğini belirtir).

`char` **version**`[]`
İşletim sisteminin sürüm numarası.

`char` **machine**`[]`
Kullanılan donanım türü.

Bazı sistemler bu bilgi için doğrudan çekirdeği sorguya çeken bir mekanizma sağlar. Bu mekanizma olmayan sistemlerde, GNU C kütüphanesi bu alanları kütüphanenin derlenmesi sırasında belirtilen yapılandırma ismine göre doldurur.

GNU üç parçalı bir sistem yapılandırma ismi kullanır; bunlar tire işaretleri ile ayrılmış *işlemci*, *üretici* ve *sistem–türü* dizgeleridir. Bu üçlünün her olası birleşimi potansiyel olarak anlamlı olmakla birlikte bazı birleşimleri pratik olarak anlamsızdır ve anlamlı olsa bile belli bir GNU yazılımınca desteklenmesi gerekli değildir.

machine değerinin sadece donanımı belirtmesi sebebiyle üçlü yapılandıma isminin ilk iki parçasından oluşur: *işlemci–üretici*. Örnek olarak bunlardan biri olabilir:

"sparc-sun", **"i386-*birşey*"**, **"m68k-hp"**, **"m68k-sony"**, **"m68k-sun"**, **"mips-dec"**

`char` **nodename**`[]`
Bilgisayarın konak ismidir. GNU C kütüphanesinde, bu değer **gethostname** işlevinden dönen değer ile aynıdır; bkz. *Konak İsimlendirmesi* (sayfa: 769).

gethostname() işlevi bir uname() çağrısı ile gerçeklenir.

`char` **domainname**`[]`
NIS ya da YP alan adı. **getdomainname** işlevinden dönen değer ile aynıdır; bkz. *Konak İsimlendirmesi* (sayfa: 769). Bu eleman yapıya görece en son konulandır ve bunun kullanımı yapının kalanının kullanımı açısından taşınabilir değildir.

`int` **uname**`(struct utsname *`*bilgi*`)` işlev

uname işlevi makina ve işletim sistemi ile ilgili bilgileri *bilgi* ile gösterilen yapıya yerleştirerek döner. Negatif olmayan bir dönüş değeri işlevin başarılı olduğunu gösterir.

-1 bir hata oluştuğu anlamına gelir. Olası tek hata **EFAULT** olmasına rağmen bu hep bir olasılık olarak kalacaktır.

3. Dosya Sistemleri ile Çalışma

Dosya sistemindeki tüm dosyalara erişmek için önce o dosya sistemini sisteme bağlamalısınız. Unix'de herşey bir dosya olduğundan bir dosya sisteminin bağlanması hemen hemen herşeyin merkezidir. Bu bölümde bağlı olan dosya sistemlerinin nasıl bulunacağı, hangi dosya sistemlerinin bağlanabileceği ve bağlı olanların nasıl değiştirileceği açıklanacaktır.

Klasik dosya sistemi bir disk sürücününün içeriğidir. Kavram giderek soyutlaşmış ve disk sürücülerden başka şeyler de bağlanabilir olmuştur.

Bazı blok aygıtları disk sürücüleri gibi geleneksel aygıtlara karşılık değildir. Örneğin bir döngü (loop) aygıtı, normal bir dosyayı başka bir dosya sisteminin ortamı gibi kullanan bir blok aygıtıdır. Bu durumda, normal dosya bir dosya sistemi için uygun verileri içeriyorsa, bir döngü aygıtını bağlayarak esasen bir normal dosyayı bağlamış olursunuz.

Bazı dosya sistemlerinin herhangi bir aygıtla ilgisi yoktur. "proc" dosya sistemi böyledir ve içerdiği dosyaları okumak isterseniz dosya sistemi sürücüsü tarafından o an için oluşturulurlar. Ve o dosyaya yazarsanız yazdığınız veri sistemde değişikliğe sebep olur. Hiçbir veri saklanmaz.

3.1. Bağlama Bilgileri

Bazı uygulamalar için, belli bir dosya sisteminin bağlı mı olduğu, bağlıysa nerede bağlı olduğu ya da mevcut dosyasistemlerinin bir listesinin alınması istenebilir hatta gerekli olabilir. GNU C kütüphanesi bu bilginin taşınabilir olarak alınmasını sağlayan işlevler içerir.

Geleneksel olarak Unix sistemlerinde bağlanabilmesi olası tüm dosya sistemlerinin bir listesini içeren `/etc/fstab` isminde bir dosya bulunur. **mount** uygulaması ile sistemin açılışı sırasında, bağlanması gereken tüm dosya sistemlerini bağlamak için bu dosya kullanılır. Bağlanmış tüm dosya sistemlerine ilişkin bilgiler de ayrı bir dosyada tutulur. Bu dosyanın ismi `mtab` dır ve normalde yeri `/var/run` ya da `/etc` dizinidir. Her iki dosyanında sözdizimi aynıdır ve bu sözdizimi artık nihai duruma gelmiştir. Bu bakımdan dosyalara doğrudan asla yazılmaz. Bu işlem bu bölümde açıklanan işlevlerle yapılır ve ayrıca bu işlevler harici metinsel gösterimi dahili gösterime dönüştüren işlevselliği sağlarlar.

fstab ve **mtab** dosyalarını sistemde bulunması bir *uzlaşım* sonucudur. Bu dosyalar sistemde bulunmayabileceği gibi bağlanacak ya da bağlanmış tüm dosya sistemlerini de içermeyebilir. Buna sistem yöneticisi karar verir. Fakat burada açıklanacak işlevler ve dolayısıyla çoğu uygulama genellikle bu dosyaların varlığına ihtiyaç duyar ve bu dosyaları kullanırlar.

Bu dosya isimleri doğrudan kullanılmamalıdır. Bu dosyalarla taşınabilir bir şekilde çalışmak için `fstab.h` içinde tanımlı **_PATH_FSTAB** makrosunu, `mntent.h` içinde tanımlı **_PATH_MNTTAB** ve **_PATH_MOUNTED** makrolarını, `paths.h` içinde bildirilmiş **fstab** ve **mtab** işlevlerini kullanın. Bu makro isimlerine ek olarak başka makrolar da vardır: **FSTAB**, **MNTTAB** ve **MOUNTED**. Bu makroların kullanılması önerilmemektedir ve sadece geriye uyumluluk adına tutulmaktadır. Bunların yerine daima **_PATH_MNTTAB** ve **_PATH_MOUNTED** makroları kullanılmalıdır.

3.1.1. fstab

Dosya içeriğinin kütüphane içindeki gösterimi olan **struct fstab** yapısı `fstab.h` dosyasında tanımlıdır.

`struct` **fstab** veri türü

Bu yapı **getfsent**, **getfsspec** ve **getfsfile** işlevlerinde kullanılır.

`char *fs_spec`
: Dosya sistemi olarak bağlanacak aygıtı belirtir. Normalde bu, bir sabit disk bölümü gibi bir özel aygıt ismidir, fakat az ya da çok soysal bir dizge de olabilir. *NFS* için bir makina adı ile bir dizin isminin birleşimi olabilir.

  Eleman **const** olarak bildirilmemiş olsa bile içeriği değiştirilmemelidir. Bunun eski bir ISO C işlevi olmasıyla **const**'un yokluğu tarihsel sebeplere dayanır. Aynı durum yapının diğer dizge elemanları için de geçerlidir.

`char *`**fs_file**
: Yerel sistem üzerindeki bağlama noktasını belirtir. Yani bu dosya sistemindeki bir dosyaya erişilmek istendiğinde bu dizge dosyaya bir önek olur.

`char *`**fs_vfstype**
: Dosya sisteminin türü. Çekirdek açısından anlamlı bir dizge olmalıdır.

`char *`**fs_mntops**
: Bu dizge **mount** çağrısı ile çekirdeğe aktarılan seçenekleri içerir. Tekrar belirtelim ki, bu çekirdek açısından anlamlı herhangi bir dizge olabilir. Diğerlerinden virgülle ayrılmış birden fazla seçenek belirtilebilir. Her seçenek bir isim ile isteğe bağlı ve **=** karakteri ile tanınan değer alanından oluşur.

  Eğer bu elemanın içeriğinin işlenmesi gerekirse bu ideal olarak **getsubopt** işlevi kullanılarak yapılır; bkz. *Alt Seçeneklerin Çözümlenmesi* (sayfa: 674).

`const char *`**`fs_type`**
: Bu üyenin ismi yanlış seçilmiştir. Bu eleman bağlı dosya sistemi ile ilgili kipleri içeren bir dizgedir (büyük ihtimalle **`fs_mntops`** içindekiler). **`fstab`** olası değerleri açıklayan beş makro tanımlar:

  `FSTAB_RW`
  : Dosya sistemi oku/yaz erişimli bağlanır.

  `FSTAB_RQ`
  : Dosya sistemi oku/yaz erişimli bağlanır ama yazma erişimi kotalarla kısıtlanmıştır.

  `FSTAB_RO`
  : Dosya sistemi salt okunur bağlanır

  `FSTAB_SW`
  : Bu gerçek bir dosya sistemini göstermez, bir takas aygıtıdır.

  `FSTAB_XX`
  : Bu girdinin tamamı **`fstab`** dosyasında yoksayılır.

  Bu değerlerin hepsi dizge olduğundan eşitlik sınaması **`strcmp`** kullanılarak yapılmalıdır. Şüphesiz göstericilerin karşılaştırılması daima başarısız olacaktır.

`int` **`fs_freq`**
: Bu eleman gün cinsinden dökümleme sıklığını belirler.

`int` **`fs_passno`**
: Paralel dökümlemede geçiş sayısıdır. Unix sistemlerinde kullanılan **`dump`** uygulaması ile yakından ilgilidir.

`fstab` dosyasının içeriğinin tamamının okunması amacıyla GNU C kütüphanesi birbiriyle uyumlu çalışan üç işlev içerir.

`int` **`setfsent`**`(void)` işlev

Bu işlev dosya konumlayıcıyı `fstab` dosyasının başlangıcına hizalar. Bu işlem ya dosyayı açarak ya da konumlayıcıyı sıfırlayarak yapılır.

Dosya tanıtıcı kütüphanenin dahili değeri olduğundan bu işlev evresel değildir.

İşlev başarılı olduğunda sıfırdan farklı bir değerle döner. Bu durumda dosyadaki girdileri okumak için **`getfs*`** işlevleri kullanılabilir.

`void` **`endfsent`**`(void)` işlev

Bu işlev evvelce yapılmış **`setfsent`** (doğrudan ya da dolaylı **`getfsent`**) çağrılarıyla edinilen tüm özkaynakları serbest bırakır.

`struct fstab *`**`getfsent`**`(void)` işlev

Bu işlev `fstab` dosyasındaki sonraki girdi ile döner. Bu işleve yapılan çağrı yazılımın başından itibaren `fstab` dosyasıyla çalışmak için yapılan ilk çağrıysa ya da bu çağrıdan önceki son çağrı **`endfsent`** ise dosya açılacaktır.

İşlev **`struct fstab`** türünde bir değişkene gösterici ile döner. Bu değişken tüm evrelerce paylaşılmasına rağmen işlev evresel değildir. Bir hata oluşmuşsa işlev bir boş gösterici döndürür.

`struct fstab *`**`getfsspec`**`(const char *`*isim*`)` işlev

`fstab` dosyasından, **`fs_spec`** elemanındaki dizgenin *isim* ile gösterilen dizgeyle aynı olduğu sonraki girdiyi döndürür. Normalde her özel aygıt için sadece bir girdi olduğundan bu işlevin aynı argümanla yapılan her çağrısı daima aynı girdiyi döndürür. Bu işleve yapılan çağrı yazılımın başından itibaren `fstab` dosyasıyla çalışmak için yapılan ilk çağrıysa ya da bu çağrıdan önceki son çağrı **`endfsent`** ise dosya açılacaktır.

İşlev **`struct fstab`** türünde bir değişkene gösterici ile döner. Bu değişken tüm evrelerce paylaşılmasına rağmen işlev evresel değildir. Bir hata oluşmuşsa işlev bir boş gösterici döndürür.

`struct fstab *`**`getfsfile`**`(const char *`*`isim`*`)` işlev

`fstab` dosyasından, **`fs_file`** elemanındaki dizgenin *isim* ile gösterilen dizgeyle aynı olduğu sonraki girdiyi döndürür. Normalde her bağlama noktası için sadece bir girdi olduğundan bu işlevin aynı argümanla yapılan her çağrısı daima aynı girdiyi döndürür. Bu işleve yapılan çağrı yazılımın başından itibaren `fstab` dosyasıyla çalışmak için yapılan ilk çağrıysa ya da bu çağrıdan önceki son çağrı **`endfsent`** ise dosya açılacaktır.

İşlev **`struct fstab`** türünde bir değişkene gösterici ile döner. Bu değişken tüm evrelerce paylaşılmasına rağmen işlev evresel değildir. Bir hata oluşmuşsa işlev bir boş gösterici döndürür.

3.1.2. `mtab`

Aşağıdaki işlevler ve veri yapısı `mtab` dosyasına erişim için kullanılır.

`struct` **`mntent`** veri türü

Bu yapı **`getmntent`**, **`getmntent_t`**, **`addmntent`** ve **`hasmntopt`** işlevleriyle kullanılır.

`char *`**`mnt_fsname`**
: Bu eleman bağlı dosya sistemini içeren özel aygıtın ismi olan bir dizgeye bir gösterici içerir. Bu eleman **`struct fstab`** yapısının **`fs_spec`** üyesinin karşılığıdır.

`char *`**`mnt_dir`**
: Dosya sisteminin bağlama noktasını belirten dizgeye bir göstericidir. Bu eleman **`struct fstab`** yapısının **`fs_file`** üyesinin karşılığıdır.

`char *`**`mnt_type`**
: **`mnt_type`** dosya sistemini açıklar ve **`struct fstab`** yapısının **`fs_vfstype`** üyesinin karşılığıdır. Bu dizgenin içerebileceği sabitler `mntent.h` dosyasında tanımlanmıştır. Fakat çekirdek dosya sistemlerini keyfi isimlerle destekleyebildiğinden bunları sembolik isimler olarak vermenin bir yararı yoktur. Sembolik ismi bilen biri dosya sistemi ismini de bilir ve bu sembollerin listesini `mntent.h` dosyasında bulabilirsiniz.

  `MNTTYPE_IGNORE`
  : Bu sembol **`"ignore"`** dizgesinin karşılığıdır. Değer bazan `fstab` dosyasındaki bir girdiyi silmeden kullanılamaz duruma getirmek için kullanılır.

  `MNTTYPE_NFS`
  : **`"nfs"`** dizgesinin karşılığıdır. Bu makronun kullanımı, öntanımlı NFS gerçeklemesi ile ilgilidir ve 2. ve 3. sürümleri desteklemektedir.

  `MNTTYPE_SWAP`
  : Bu sembol **`"swap"`** dizgesinin karşılığıdır. `fstab` dosyasında çok sayıda olabilen takas bölümleri ile ilgili girdileri isimlendirir.

`char *`**`mnt_opts`**

Bu eleman dosya sistemi bağlanırken kullanılan seçenekleri içeren bir dizgedir. **struct fstab** yapısının **fs_mntops** üyesinin karşılığı olarak bu dizgenin parçalarına erişmek için **getsubopt** işlevini kullanmak en iyi yöntemdir; bkz. *Alt Seçeneklerin Çözümlenmesi* (sayfa: 674).

mntent.h dosyasında, çekirdek tarafından anlamlandırılabilen bazı seçeneklerin karşılığı olan dizge değerli makrolar tanımlanmıştır. Bu makrolarla kapsanmayan başka seçenekler de olabilir, bunlar mevcut olanların listesidir:

MNTOPT_DEFAULTS
: **"defaults"** dizgesinin karşılığıdır. Bu seçenek tek başına kullanılır, çünkü özelleştirilebilir değerlerin uygun seçilmesi ile oluşturulan bir dizgenin öntanımlı karşılığıdır.

MNTOPT_RO
: **"ro"** dizgesinin karşılığıdır. Dosya sisteminin salt okunur bağlanması anlamına gelen **FSTAB_RO** değerine bakınız.

MNTOPT_RW
: **"rw"** dizgesinin karşılığıdır. Dosya sisteminin oku/yaz erişimiyle bağlanması anlamına gelen **FSTAB_RW** değerine bakınız.

MNTOPT_SUID
: **"suid"** dizgesinin karşılığıdır. Dosya sisteminden bir uygulama başlatıldığında SUID bitine riayet edilecek anlamındadır; bkz. *Bir Sürecin Aidiyeti Nasıl Değiştirilir?* (sayfa: 743).

MNTOPT_NOSUID
: **"nosuid"** dizgesinin karşılığıdır. Dosya sisteminden bir uygulama başlatıldığında SUID biti yoksayılacak anlamında olup **MNTOPT_SUID** makrosunun zıddıdır.

MNTOPT_NOAUTO
: **"noauto"** dizgesinin karşılığıdır. Sistem açılışında **mount** uygulaması fstab dosyasındaki tüm bağlanabilir dosya sistemlerini bağlamasını belirten **-a** seçeneği ile çalıştırıldığında bu girdi yoksayılacaktır.

Önceki bölümlerde **FSTAB_*** girdilerinde bahsedildiği gibi eşitlik sınaması yapmak için **strcmp** kullanılması önemlidir.

mnt_freq
: Bu eleman **fs_freq**'in karşılığıdır ve ayrıca dökümlemenin yapılma sıklığını gün cinsinden belirtir.

mnt_passno
: **dump** gibi uygulamaların bu dosya sistemi ile ilgilenmemesi anlamında olan **fs_passno**'nun karşılığıdır.

mtab dosyasının içeriğinin tamamının okunması amacıyla GNU C kütüphanesi birbiriyle uyumlu çalışan üç işlev içerir. fstab dosyası ile çalışan işlevelerin tersine bu işlevler bir sabit dosyaya erişmez ve ayrıca get işlevlerinin evresel sürümleri de vardır. Bunun yanında GNU C kütüphanesi dosyayı değiştirmek ve bazı seçenekleri sınamak için de işlevler içerir.

```
FILE *setmntent(const char *dosya,                                    işlev
                const char *kip)
```

setmntent işlevi, ailenin diğer işlevlerinin ihtiyaçlarına uygun olarak kullanılmak üzere fstab ve mtab biçiminde *dosya* isimli dosyayı hazırlar. *kip* parameresi **fopen** işlevinin *açıştürü* parametrsinin seçiminde kullanılan yolla seçilebilir (bkz. *Akımların Açılması* (sayfa: 238)). Eğer dosya yazma amacıyla açılıyorsa ayrıca boş olmasına da izin verilir.

İşlev dosyayı başarıyla açmışsa kullanılacak akım ile döner. Aksi takdirde **NULL** döner ve hata durumu **errno** değişkenine atanır.

```
int endmntent(FILE *akım)                                                    işlev
```

Bu işlev önceki bir **setmntent** çağrısından dönen *akım* akımını kapatır ve tüm özkaynaklarını serbest bırakır.

Hata durumunda 0 aksi takdirde 1 ile döner.

```
struct mntent *getmntent(FILE *akım)                                         işlev
```

Bu işlev önceki bir **setmntent** çağrısından dönen *akım* akımını argüman olarak alır ve dosyadan sonraki girdiyi içeren **struct mntent** türünde bir değişkene gösterici ile döner.

Kullanılan dosya biçimi gereğince alanları ayırmak için boşluklar ve sekme karakterleri kullanılır. Bu durum, bu karakterleri içeren isimlerin kullanılmasını güçleştirir. Bu bakımdan bu karakterleri dosyalarda kodlarken ve **getmntent** işlevinde çözümlerken dikkatli olunmalıdır. Bir boşluk karakteri `'\040'` ile, bir sekme karakteri `'\011'` ile, satırsonu karakteri `'\012'` ile ve tersbölü karakteri `'\\'` ile kodlanır.

Bir hata oluşursa ya da dosya sonuna gelinmişse işlev **NULL** ile döner.

Bu işlevin çağrıları aynı durağan değişkene bir gösterici ile döndüğünden işlev evresel değildir. Dosyaya çok evreli erişim için **getmntent_r** işlevi kullanılmalıdır.

```
struct mntent *getmntent_r(FILE           *akım,                             işlev
                           struct mentent *sonuç,
                           char           *tampon,
                           int             tamponboyu)
```

getmntent_r işlevi **getmntent** işlevinin evresel benzeridir. Ayrıca dosyadaki sonraki girdiye bir gösterici ile döner. Döndürdüğü değişken aslında durağan değildir. Döndürdüğü değeri *sonuç* parametresi ile belirtilen göstericinin gösterdiği değişkende saklar. Ek bilgiler (sonucun elemanları ile gösterilen dizgeler) uzunluğu *tamponboyu* kadar olan *tampon* tamponunda saklanır.

Öncelemeli karakterler (boşluk, sekme, tersbölü) **getmentent** işlevindeki gibi geri dönüştürülür.

İşlev bir hata durmunda bir boş gösterici ile döner. Olası hatalar:

- dosya okunurken hata oluştu,
- dosyanın sonuna gelindi,
- *tamponboyu* tam bir girdiyi okumak için yetersiz.

```
int addmntent(FILE                 *akım,                                    işlev
              const struct mntent *girdi)
```

addmntent işlevi önceki bir **setmntent** çağrısı ile açılan dosyaya yeni bir girdi eklemeye imkan verir. Yeni girdi daima dosyanın sonuna eklenir. Yani dosya konumlayıcı dosyanın sonunda olmasa bile bu işlev girdiyi dosya konumlayıcının bulunduğu yere değil daima dosyanın sonuna yazar.

Bunun sonucu olarak bir dosyadan bir girdiyi silmek için bu girdiyi içermeyen yeni bir dosya oluşturmak, bu dosyayı kapatmak ve eski dosyayı silip yeni dosyanın adını eskisi olarak değiştirmek gerekir.

Bu işlev isimlerin içindeki boşluklar ve sekmeler bakımından dikkatlidir. Bunları ve tersbölü karakterlerini evvelce **getmntent** işlevinde açıklandığı gibi dönüştürür.

İşlev başarı durumunda 0, aksi takdirde 1 değeri ile döner ve hata durumu **errno** değişkenine atanır.

```
char *hasmntopt(const struct mntent *girdi,                                    işlev
                const char          *seçenek)
```

Bu işlev, *girdi* ile gösterilen yapının **mnt_opts** elemanının içerdiği dizgede *seçenek* dizgesi var mı diye bakar. *seçenek* bir boş gösterici değilse **mnt_opts** içindeki seçeneğin başlangıcına bir gösterici ile döner. Böyle bir seçenek yoksa işlev boş gösterici döndürür.

Bu işlev belli bir seçeneğin varlığını sınamak için kullanışlıdır ama tüm seçenekler işlenmek zorunda olduğunda dizgedeki tüm seçenekler üzerinde yinelenen **getsubopt** işlevini kullanmak daha iyidir.

3.1.3. Diğer Bağlama Bilgileri

Linux çekirdekli ve **proc** dosya sistemli bir sistemde `proc` dosya sistemindeki `mounts` dosyasından o an bağlı olan dosya sistemleri hakkında bilgi edinebilirsiniz. Bu dosyanın biçimi `mtab` dosyasınınkine benzer. Çekirdek bu dosyayı, gerçekten bağlı dosya sistemleri hakkında bilgiyi dışarda aramamak için kendisi güncel tutar.

3.2. Bağlama, Ayırma, Yeniden Bağlama

Bu bölümde dosya sistemlerinin bağlanması, ayrılması ve yeniden bağlanması ile ilgili işlevlerden bahsedilecektir.

Bir dosya sistemini sadece sistem yöneticisi bağlayabilir, ayırabilir veya yeniden bağlayabilir.

Bu işlevler `fstab` ve `mtab` dosyalarına erişmezler. Bunu ayrıca kendiniz sağlamalısınız. Bkz. *Bağlama Bilgileri* (sayfa: 772).

Bu bölümdeki semboller `sys/mount.h` dosyasında bildirilmiştir.

```
int mount(const char        *özel_dosya,                                       işlev
          const char        *dizin,
          const char        *dstürü,
          unsigned long int  seçenekler,
          const void        *veri)
```

mount işlevi bir dosya sistemini bağlamak ya da bağlı bir dosya sistemini yeniden bağlamak için kullanılır. İki işlem açıkça farklıdır ve doğal olmayan bir şekilde aynı işlevin içinde birbirine katıştırılmıştır. Aşağıda açıklanacağı gibi **MS_REMOUNT** seçeneği ile işlevin sıfırdan bağlama mı yoksa yeniden bağlama mı yapacağı saptanır.

İşlev dosya sistemini bağlamak amacıyla kullanıldığında, dosya sistemini içeren blok aygıtı *özel_dosya* isimli aygıt özel dosyası ile, bağlanacağı nokta ise *dizin* ile belirtilir. Bu durumda *dizin* dizinindeki herşey dosya sistemi sistemden ayrılana kadar erişilebilir olmayacak, onların yerine bağlanan dosya sisteminin kök dizininin içeriği erişilebilir olacaktır.

Bir olağandışılık olarak, eğer dosya sistemi türü bir aygıta ait değilse ("proc" gibi), **mount** dosya sistemini hemen görür ve *özel_dosya*'yı yoksayarak *dizin* dizinine onu bağlar.

İşlev dosya sistemini yeniden bağlamak amacıyla kullanıldığında, *dizin* dosya sisteminin bağlı olduğu yeri belirtir ve *özel_dosya* yoksayılır. Bir dosya sistemini yeniden bağlamanın amacı, evvelce bağlanırken kullanılan seçenekleri değiştirmektir. Bu işlem dosya sistemini önce ayırıp sonra tekrar bağlamak anlamında yapılmaz.

Sıfırdan bağlama amaçlı kullanımda dosya sisteminin türü *dstürü* argümanı ile belirtilir. Bu tür, çekirdeğin dosya sistemine erişirken kullanacağı dosya sistemi sürücüsünün ismi olarak belirtilir. Kabul edilen

değerler sistemden sistem değişir. Linux çekirdekli ve **proc** dosya sistemli bir sistemde olası dosya sistemi türleri `/proc/filesystems` dosyasında listelenir (komut satırına **cat /proc/filesystems** yazarak bu listeyi görebilirsiniz). Linux çekirdeğinde **mount** ile bağlanabilen dosya sistemi türlerinin isimleri, çekirdeğin yapılandırılması sırasında yüklenebilir modül olarak belirtilen dosya sistemi sürücüsü olan modüllerin isimleridir. *dstürü* olarak belitilebilecek çok bilinen bir örnek **ext2**'dir.

Yeniden bağlama amaçlı kullanımda *dstürü* argümanı yoksayılır.

seçenekler ile dosya sistemi ayrılana ya da yeniden bağlanana kadar geçerli olacak çeşitli seçenekler belirtilir. Her seçenek dosya sistemine özgüdür, bazı seçenekleri birçok dosya sistemi kabul etse de bu hepsi için geçerli değildir. Bazı dosya sistemlerinde bu seçeneklerden bazıları (ama asla **MS_RDONLY** değil) **ioctl** ile dosya erişimi sırasında değiştirilebilir.

seçenekler bir bit dizgesidir. Bit alanları aşağıdaki maske ve maske değerli makrolar kullanılarak tanımlanmıştır:

MS_MGC_MASK
: Bu çokbitli alan sihirli bir sayı içerir. Eğer **MS_MGC_VAL** değerini içermiyorsa, **mount** aşağıdaki tüm bitlerin sıfır olduğunu ve *veri* argümanının ise içerdiği değerlere bakılmaksızın bir boş dizge olduğunu varsayar.

MS_REMOUNT
: Bu bit dosya sisteminin yeniden bağlanacağını belirtir. Yokluğu sıfırdan bağlama yapılacak anlamındadır.

MS_RDONLY
: Bu bit bağlanan dosya sisteminde yazma işlemlerine izin verilmediğini belirtir ve **ioctl** tarafından bitin durumu değiştirilemez. Bu seçenek hemen tüm dosya sistemlerinde vardır.

S_IMMUTABLE
: Bu bit bağlanan dosya sisteminde yazma işlemlerine izin verilmediğini belirtir. Fakat uygun olarak yetkilendirilmiş bir **ioctl** çağrısı bu bitin durumunu değiştirebilir. Bu seçenek görece diğerlerinden daha yeni olduğundan bir çok dosya sisteminde bulunmayabilir.

S_APPEND
: Bu bit bağlanan dosya sisteminde yazma işlemlerine sadece dosyaların sonuna ekleme yapılacaksa izin verildiğini belirtir. Bazı dosya sistemlerinde uygun olarak yetkilendirilmiş bir **ioctl** çağrısı bu bitin durumunu değiştirebilir. Bu seçenek görece diğerlerinden daha yeni olduğundan bir çok dosya sisteminde bulunmayabilir.

MS_NOSUID
: Bu bit bağlanan dosya sisteminde dosyaların Setuid and Setgid yetkilerinin yoksayılacağını belirtir.

MS_NOEXEC
: Bu bit bağlanan dosya sisteminde dosyaların çalıştırılamayacağını belirtir.

MS_NODEV
: Bu bit bağlanan dosya sisteminde aygıtlara özel dosyaların erişilebilir olmayacağını belirtir.

MS_SYNCHRONOUS
: Bu bit bağlanan dosya sisteminde dosyalara yapılan tüm yazma işlemlerinin eşzamanlanacağını belirtir. Yani veri her yazma işleminde sonradan dosyaya yazılmak üzere bir tamponda saklanmaz doğrudan dosyaya yazılır.

MS_MANDLOCK

Bu bit bağlanan dosya sisteminde dosyalarda zorlayıcı kilitlere izin verildiğini belirtir.

MS_NOATIME
Bu bit bağlanan dosya sisteminde dosyaların erişim zamanlarının dosyalara erişildiğinde güncellenmeyeceğini belirtir.

MS_NODIRATIME
Bu bit bağlanan dosya sisteminde dizinlerin erişim zamanlarının dizinlere erişildiğinde güncellenmeyeceğini belirtir.

Yukarıdaki bit maskelerinin kapsamında olmayan bütün bitler sıfır olmalıdır; aksi takdirde sonuçları tanımsızdır.

veri'nin anlamlandırılması dosya sistemi türüne bağlıdır ve tamamen çekirdekteki dosya sistemi sürücüsü tarafından değerlendirilir.

Örnek:

```
#include <sys/mount.h>

mount("/dev/hdb", "/cdrom", MS_MGC_VAL | MS_RDONLY | MS_NOSUID, "");

mount("/dev/hda2", "/mnt", MS_MGC_VAL | MS_REMOUNT, "");
```

mount işlevinin ilgili argümanları teamülen **fstab** tablosuna kaydedilir. Bkz. *Bağlama Bilgileri* (sayfa: 772).

İşlev başarılı olduğunda sıfırla döner. Aksi takdirde **-1** döner ve **errno** değişkenine hata durumu atanır. **errno** değerleri dosya sistemine bağımlıdır, fakat burada ortak bir listeye yer verilmiştir:

EPERM
Süreç sistem yöneticisne ait değil

ENODEV
dstürü dosya sistemi türünü çekirdek bilmiyor

ENOTBLK
dev dosyası bir blok aygıtına özel dosya değil

EBUSY
- Aygıt zaten bağlı
- Bağlama noktası meşgul (Yani ya bu dizin bazı süreçlerin çalışma dizini ya da onun üzerinde de bir dosya sistemi hala bağlı)
- Yeniden salt–okunur bağlama istendi ama yazma amacıyla açılmış dosyalar var

EINVAL
- Yeniden bağlama istendi ama belirtilen bağlama noktasında bağlı bir dosya sistemi yok
- Belirtilen dosya sistemi geçersiz süperblok içeriyor.

EACCES
- Dosya sistemi kalıtsal olarak salt–okunur (aygıt üzerindeki donanımsal bir anahtar nedeniyle olabilir) ve süreç onu oku/yaz olarak bağlamaya çalıştı (**MS_RDONLY** biti sıfır olarak belirtilerek)
- *özel_dosya* ya da *dizin* dosya erişim izinleri nedeniyle erişilebilir değil
- *özel_dosya* erişilebilir değil, çünkü üzerinde bulunduğu dosya sistemi **MS_NODEV** seçeneği ile bağlı

EM_FILE
İsimsiz (dummy) aygıtlar tablosu dolu. **mount**, bir aygıtla bağlanamayacak bir dosya sistemini bağlamaya çalışırsa bir dummy aygıt oluşturmak ister.

```
int umount2(const char *dosya,                                    işlev
            int         seçenekler)
```

umount2 bağlı bir dosya sistemini sistemden ayırır.

Ayrılacak dosya istemini *dosya* argümanında bir aygıta özel dosya olarak belirtebileceğiniz gibi bağlama noktası olarak da belirtebilirsiniz. Etkisi aynıdır.

seçenekler aşağıdaki maske makrosu ile tek bitlik bir alan olarak belirtilir:

MNT_FORCE
Bu bit bağlı dosya sisteminin meşgul olması durumnda bile ayrılacağını belirtir (önce meşgul değil durumuna getirilir). Bu bitin yokluğunda meşgul bir dosya sistemi ayrılmaz ve işlev **errno**= **EBUSY** ile başarısız olur. Bu bitin sağladığı zorlamayı bazı sistemler desteklemeyebilir, bazılarında ise meşgul olmama durumu olmayabilir. Yani bu bitin davranışı dosya sistemine bağlıdır.

seçenekler ile belirtilen diğer tüm bitler sıfır olmalıdır; aksi takdirde sonuçları tanımsızdır.

Örnek:

```
#include sys/mount.h

umount2("/mnt", MNT_FORCE);

umount2("/dev/hdd1", 0);
```

Dosya sistemi ayrıldıktan sonra bağlama noktası olan dizinin içeriğindeki dosyalara erişilebilir.

Dosya sisteminin ayrılması bağlamında **umount2** dosya sistemin eşzamanlar. Yani açık dosyalara tamponlar boşaltılır.

Ayırma işlemi başarılı olmuşsa işlev sıfırla döner. Aksi takdirde **-1** ile döner ve **errno** değişkenine hata durumu atanır:

EPERM
Süreç sistem yöneticisine ait değil

EBUSY
Dosya sistemi meggul olduğundan ayrılamıyor. Yani dosya sistemindeki bir dizin ya bir sürecin çalışma dizini ya da süreçlerden biri bir dosya açmış. Bazı dosya sistemlerinde bu hata durumu **MNT_FORCE** seçeneği ile aşılabilir.

EINVAL
dosya aslında bir dosya ama ayrılmak istenen dosya sistemine ait ne bir bağlama noktası ne de aygıta özel bir dosya

Bu işlev dosya sistemlerinin tamamında kullanılabilir değildir.

```
int umount(const char *dosya)                                     işlev
```

Bu işlev **umount2** işlevinde *seçenekler* argümanının sıfır olması durumunda aynı işlevselliğe sahiptir. **umount2**'den daha geniş kullanım alanına sahip olmakla birlikte bir dosya sisteminin ne olursa olsun ayrılmasını sağlayan **umount2**'nin dosya sisteminde desteklenmesi durumunda kullanılması önerilmemektedir.

4. Sistem Parametreleri

Bu bölümde çeşitli sistem parametrelerinin okunması ve belirtilmesi için kullanılan **sysctl** işlevinden bahsedilecektir.

Bu kısımdaki semboller `sysctl.h` dosyasında bildirilmiştir.

işlev
```
int sysctl(int    *isimler,
           int     isim_uzunluğu,
           void   *eski_değer,
           size_t *eski_değer_uzn,
           void   *yeni_değer,
           size_t  yeni_değer_uzn)
```

sysctl işlevi belirtilen sistem parametresini ya okur ya da belirler. Bu parametreler o kadar çoktur ki, burada hepsinin listelenmesi pratik olmayacaktır. Fakat bazı örnekler verilebilir:

- ağ alan adları
- sayfalama parametreleri
- Ağ adres çözümleme protokolü (ARP) zaman aşımı
- açılabilecek dosyalarını azami sayısı
- kök dosya sistemi aygıtı
- çekirdeğin derleme zamanı

Kullanılabilir parametrelerin tamamı çekirdek yapılandırmasına bağlıdır ve sistem çalışırken yüklenebilir modüllerin yüklenmesi ve kaldırılması ile kısmi olarak değiştirilebilir.

sysctl ile ilgili sistem parametreleri bir hiyerarşik dosya sistemindeki gibi bir hiyerarşik yapıda düzenlenmiştir. Belli bir parametreyi belirtmek için, bir dosyanın dosya yolu ile belirtilmesine benzer şekilde parametreyi yapı üzerinde bir yol ile belirtebilirsiniz. Yolu oluşuran her eleman bir tamsayı ile belirtilir ve bu tamsayıların her biri için `sysctl.h` dosyasında bir makro tanımlanmıştır. *isimler* bir tamsayılar dizisi şeklinde bir yol belirtir. Yolu oluşturan her bir eleman sırasıyla dizideki bir elemandır. *isim_uzunluğu* ile dizideki bu elemanların sayısı belirtilir.

Örneğin, tüm sayfalama parametreleri için yolun ilk elemanı **CTL_VM** değeridir. Serbest sayfa eşikleri için yolun ikinci elemanı **VM_FREEPG** değeridir. Bu durumda serbest sayfa eşiklerini almak için *isimler* dizisini **CTL_VM** ve **VM_FREEPG** elemanlarını içeren bir dizi olarak belirtip, *isim_uzunluğu*= 2 yapılır.

Bir parametre değerinin biçimi parametreye bağlıdır. Bazan bir tamsayı, bazan bir ASCII dizge, bazan özenle oluşturulmuş bir yapıdır. Yukarıdaki örnekte görüldüğü gibi serbest sayfa eşiklerinin kullanıldığı durumda parametre değeri çeşitli tamsayılar içeren bir yapıdır.

Her durumda, dönen parametrenin değerinin adresi için *eski_değer* argümanını kullanabilir ve bu tamponun boyutunu **eski_değer_uzn* ile belirtebilirsiniz. **eski_değer_uzn* çift görevlidir, çünkü ayrıca dönen değerin gerçek uzunluğu da burada döndürülür.

Eğer parametre değerinin dönmesini istemiyorsanız *eski_değer* için boş gösterici belirtebilirsiniz.

Bir parametreye değer atamak isterseniz yeni değerin adresini ve uzunluğu *yeni_değer* ve *yeni_değer_uzn* ile belirtin. Bunu istemiyorsanız *yeni_değer* için boş göserici belirtin.

Bir parametrenin o anki değerini okuyup yeni bir değer atamak için aynı **sysctl** çağrısını kullanabilirsiniz.

Her sistem parametresi için bir dosyanın izinlerine benzeyen erişim izinleri vardır ve bunlar bir paramatrenin okunması ve değiştirilmesi ile ilgili izinlerdir. Amaca uygun olarak her parametrenin sahibinin süper kullanıcı ve grubunun 0 olduğu kabul edilir. Böylece bu etkin kullanıcı ya da grup kimliğine sahip süreçler sistem parametrelerinde daha yüksek erişim iznine sahip olur. Dosyalardaki durumun tersine süper kullanıcı bütün sistem parametrelerinde tam yetkili değildir, çünkü bazı parametre değerleri asla değiştirilemeyecek biçimce tasarlanmıştır.

sysctl başarılı olursa sıfır ile döner. Aksi takdirde **-1** ile döner ve **errno** değişkenine hata durumu atanır. Tüm sistem çağrılarına uygulanan hata durumlarına ek olarak aşağıdaki **errno** kodları olası diğer hataları belirtir:

EPERM
: Sürecin, sistem parametresinin yolu üzerindeki elemanlardan birine erişim yetkisi yok ya da sistem parametresinin kendisine istenen yolla (okuma ya da yazma) erişim izni yok

ENOTDIR
: *isim*'in karşılığı olan bir sistem parametresi yok

EFAULT
: *eski_değer* bir boş gösterici değil ama *eski_değer_uzn* sıfır belirtilmiş, dolayısıyla değer döndürecek yer yok

EINVAL
: - Sürecin belirttiği değer parametre için uygun değil
  - Sistem parametresinin dönüş değeri için ayrılan yer yetersiz

ENOMEM
: Bu değer sistem parametresini dönüş değeri için ayrılan yerin çok küçük olması durumunda **EINVAL** hata durumundan daha doğru bir hata durumu belirtir.

Linux çekirdekli ve `proc` dosya sistemli bir sisteminiz varsa çoğu sistem parametresine `proc` dosya sisteminin `sys` dizinindeki dosyaları okuyarak ya da onlara yazarak erişebilirsiniz. `sys` dizini, parametre yapısıyla aynı hiyerarşiye sahiptir. Örneğin serbest sayfa eşiklerini göstermek için komut satırında şu komutu verin:

```
# cat /proc/sys/vm/freepages
```

Bazı sistem parametrelerini okumak veya belirtmek için kullanılan biraz daha geleneksel ve daha geniş kullanım alanı bulan ama pek genel amaçlı olmayan bazı GNU C kütüphanesi işlevleri:

- **getdomainname**, **setdomainname**
- **gethostname**, **sethostname** (bkz. *Konak İsimlendirmesi* (sayfa: 769).)
- **uname** (bksz *Platform Türü İsimlendirmesi* (sayfa: 771).)
- **bdflush**

XXXI. Sistem Yapılandırma Parametreleri

İçindekiler

Bu oylumda listelenen işlevler ve makrolar işletim sisteminin yapılandırma parametreleri hakkında bilgi verirler. Örneğin, kapasite sınırları, isteğe bağlı POSIX özellikleri ve çalıştırılabilir dosyaların arama yolları (bkz. *Dizge Değerli Parametreler* (sayfa: 801)).

1. Genel Sınırlar

POSIX.1 ve POSIX.2 standartları, sistemin kapasite sınırlamasını açıklayan bir miktar parametre belirlemiştir. Bu sınırlar belli bir işletim sistemi için değişmez sabitler olabileceği gibi makinadan makinaya değişen sabitler de olabilir. Örneğin, bazı sınır değerler sistem yöneticisi tarafından sistemin çalışması sırasında ya da çekirdeği derlerken yapılandırılabilir ve bu uygulama yazılımlarının yeniden derlenmesini gerektirmez.

Aşağıdaki sınır parametrelerinin herbiri `limits.h` başlık dosyasında tanımlanmış birer makro olup, sadece parametre için sistem bir sabit ve tektip bir sınıra sahipse anlamlıdırlar. Eğer sistem farklı dosya sistemleri ya da dosyalara farklı sınırlar atanmasına izin veriyorsa, makro tanımsızdır; böyle bir makina üzerinde belli bir anda uygulanan sınırı öğrenmek için **`sysconf`** kullanılır. Bkz. *sysconf Kullanımı* (sayfa: 787).

Bu parametrelerin her biri için ismi **`_POSIX`** ile başlayan başka makrolar da vardır. Bunlar herhangi bir POSIX sisteminde izin verilen sınırın en düşük değerini verirler. Bkz. *Asgari Değerler* (sayfa: 794).

`int` **`ARG_MAX`** makro

Tanımlıysa, **`exec`** işlevlerine aktarılabilen *argv* ve *environ* argümanlarının uzunluklarının toplamı için değişmez bir azami değerdir.

`int` **`CHILD_MAX`** makro

Tanımlıysa, aynı anda çalışabilecek aynı gerçek kullanıcı kimlikli süreçlerin azami sayısını belirten değişmez bir değerdir. BSD ve GNU'da bu **`RLIMIT_NPROC`** özkaynak sınırı ile denetlenir; bkz. *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

`int` **`OPEN_MAX`** makro

Tanımlıysa, tek bir sürecin aynı anda açabileceği dosya sayısının azami sayısını belirten değişmez bir değerdir. BSD ve GNU'da bu **RLIMIT_NOFILE** özkaynak sınırı ile denetlenir; bkz. *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

`int` **STREAM_MAX** makro

Tanımlıysa, tek bir sürecin aynı anda açabileceği akım sayısının azami sayısını belirten değişmez bir değerdir. Bkz. *Akımların Açılması* (sayfa: 238).

`int` **TZNAME_MAX** makro

Tanımlıysa, zaman dilimi isminin azami uzunluğunu belirten değişmez bir değerdir. Bkz. *Zaman Dilimi Değişkenleri ve İşlevleri* (sayfa: 566).

Bu sınır makroları daima `limits.h` dosyasında tanımlıdır.

`int` **NGROUPS_MAX** makro

Bir sürecin sahip olabileceği ek grup kimliklerinin azami sayısı.

Bu makronun değeri aslında azami değerin alt sınırıdır. Yani, siz bu sayıda ek grubun üyesi olabilirsiniz ama bir makina size daha fazlası için izin verebilir. Bir makinanın size daha fazlası için izin verip vermediğini öğrenmek için **sysconf** kullanabilirsiniz (bkz. *sysconf Kullanımı* (sayfa: 787)).

`int` **SSIZE_MAX** makro

ssize_t türünde bir nesneye kapasitebilen en büyük değer. Etkin olarak, bu, tek bir işlem sırasında okunabilecek ya da yazılabilecek baytların sayısı ile ilgili bir sınırdır.

Bu makro tüm POSIX sistemlerinde tanımlıdır, çünkü bu sınır asla yapılandırılmaz.

`int` **RE_DUP_MAX** makro

Bir düzenli ifadede `\{`*enaz*`,`*ençok*`\}` yapısında garanti edilmiş en büyük yineleme sayısıdır.

Bu makronun değeri aslında azami değerin alt sınırıdır. Yani, siz bu sayıda yineleme sayabilirsiniz ama bir makina size daha fazlası için izin verebilir. Bir makinanın size daha fazlası için izin verip vermediğini öğrenmek için **sysconf** kullanabilirsiniz (bkz. *sysconf Kullanımı* (sayfa: 787)). Hatta **sysconf**'un söylediği değer de sadece bir alt sınırdır—daha büyük değerler çalışabilir.

Bu makro tüm POSIX.2 sistemlerinde tanımlıdır, çünkü POSIX.2 zorlanmış özel bir sınırın olmamasının istendiği durumda bile bunun daima tanımlanmış olmasını gerektirir.

2. Sistem Seçenekleri

POSIX standardı tüm POSIX sistemlerinde desteklenmesi gerekmeyen bazı sisteme özel seçenekler tanımlamıştır. Bu seçenekler kütüphane de değil çekirdek içinde sağlandığından, GNU C kütüphanesi kullanılarak bu özelliklerin desteklenmesi garanti edilmemiştir; bu, kullandığınız sisteme bağlıdır.

Bu bölümdeki makroları **sysconf** işlevi ile birlikte kullanarak belirtilen bir seçeneğin kullanılabilirliğini sınayabilirsiniz. Bu makrolar `unistd.h` dosyasında tanımlıdır.

Aşağıdaki makrolar için, eğer makro `unistd.h` dosyasında tanımlıysa seçenek desteklenir. Aksi takdirde seçenek desteklenebilir de desteklenmeyebilir de; bunun için **sysconf** işlevini kullanın. Bkz. *sysconf Kullanımı* (sayfa: 787).

`int` **_POSIX_JOB_CONTROL** makro

Bu sembol tanımlıysa, sistemin iş denetimini desteklediğini belirtir. Aksi takdirde, gerçekleme bir oturumdaki tüm süreçler tek bir süreç grubuna aitmiş gibi davranır. Bkz. *İş Denetimi* (sayfa: 716).

`int` **_POSIX_SAVED_IDS** makro

Bu sembol tanımlıysa, sistem, bir sürecin çalıştırılabilir dosyasını set–user–ID or set–group–ID bitleriyle çalıştırmadan önce sürecin etkin kullanıcı ve grup kimliklerini hatırlar ve etkin kullanıcı ve grup kimliklerinin tekrar ve doğrudan bu değerlerle değiştirilmesine izin verir. Bu seçenek tanımlı değilse, bir ayrıcalıksız süreç kendi etkin kullanıcı ve grup kimliklerini, kendi gerçek kullanıcı ve grup kimlikleri ile değiştirdikten sonra tekrar eski duruma dönemez. Bkz. *Setuid Erişiminin Etkinleştirilmesi ve İptali* (sayfa: 748).

Aşağıdaki makrolar için, eğer makro `unistd.h` dosyasında tanımlıysa, değeri seçeneğin desteklenip desteklenmediğini belirtir. **-1** değeri desteklenmediğini, bunun dışında herhangi bir değer ise desteklendiğini belirtir. Eğer makro tanımlı değilse, seçenek desteklenebilir de desteklenmeyebilir de; bunun için **sysconf** işlevini kullanın. Bkz. *sysconf Kullanımı* (sayfa: 787).

`int` **_POSIX2_C_DEV** makro

Bu sembol tanımlıysa, sistem, POSIX.2 C derleyici komutu, **c89**'a sahiptir. Eğer C derleyiciye sahip değilseniz, GNU C kütüphanesi de gerekmeyeceğinden, kütüphanede bu makro daima **1** ile tanımlanmıştır.

`int` **_POSIX2_FORT_DEV** makro

Bu sembol tanımlıysa, sistem, POSIX.2 Fortran derleyici komutu, **fort77**'ye sahiptir. Sistemde olup olmadığı baştan bilinemeyeceğinden bu makro kütüphanede tanımsızdır.

`int` **_POSIX2_FORT_RUN** makro

Bu sembol tanımlıysa, sistem, Fortran taşıma denetimini yorumlayan POSIX.2 **asa** komutuna sahiptir. Sistemde olup olmadığı baştan bilinemeyeceğinden bu makro kütüphanede tanımsızdır.

`int` **_POSIX2_LOCALEDEF** makro

Bu sembol tanımlıysa, sistem, POSIX.2 **localedef** komutuna sahiptir. Sistemde olup olmadığı baştan bilinemeyeceğinden bu makro kütüphanede tanımsızdır.

`int` **_POSIX2_SW_DEV** makro

Bu sembol tanımlıysa, sistem, POSIX.2 **ar**, **make** ve **strip** komutuna sahiptir. Kütüphaneyi kurmak için **ar** ve **make** gerekli olduğundan GNU C kütüphanesinde bu makronun değeri daima **1**'dir. Tersine, bunların varlığı durumunda **strip** komutunun yokluğu söz konusudur.

3. POSIX'in Hangi Sürümü Var?

`long int` **_POSIX_VERSION** makro

Bu sabit gerçeklemenin uyumlu olduğu POSIX.1 standardının sürümünü ifade eder. 1995 POSIX.1 standardına uyumlu bir gerçekleme için bu değer **199506L** tamsayısıdır.

Herhangi bir POSIX sisteminde **_POSIX_VERSION** `unistd.h` dosyasında tanımlıdır.

Kullanım Bilgisi

`unistd.h` başlık dosyasını yazılımınıza dahil ederek **_POSIX_VERSION** tanımlı mı diye bir sınama yapmaya çalışmayın. POSIX olmayan bir sistemde `unistd.h` dosyası olmayacağından bu sınama başarısız olacaktır. Hedef sisteminizin POSIX desteği olup olmadığını ya da `unistd.h` dosyasının mevcut olup olmadığını gerektiği gibi sınayacak bir yöntem bilinmemektedir.

GNU C derleyicisi hedef sistem bir POSIX sistemi ise **__POSIX__** sembolünü tanımlar. POSIX sistemlerde **defined (__POSIX__)** sınamasını diğer derleyicileri kullanıyorsanız yapmayın.

`long int` **_POSIX2_C_VERSION** makro

Bu sabit kütüphanede ve sistem çekirdeğinde desteklenen POSIX.2 standardının sürümünü ifade eder. Bu değer, standardın resmen yayınlandığı yılı ve ayı belirten bir numara olduğundan bunun POSIX.2 standardının ilk sürümü olup olmadığı bilinemez.

Bu sembolün değeri sistemde kurulu araçlar hakkında hiçbir bilgi vermez.

Kullanım Bilgisi

Bu makroyu POSIX.1 sistem kütüphanesinde POSIX.2 desteği de olup olmadığını sınamak için kullanabilirsiniz. Herhangi bir POSIX sistemi `unistd.h` dosyasını içerir, dolayısıyla bu dosyayı içeren bir sistemde **defined (_POSIX2_C_VERSION)** sınamasını yapabilirsiniz.

4. sysconf Kullanımı

Sisteminiz yapılandırılabilen sistem sınırlarına sahipse, herhangi bir makinada bu değerleri **sysconf** işlevi ile öğrenebilirsiniz. Bu işlev ve bu işlevle ilgili *parametre* sabitleri `unistd.h` başlık dosyasında bildirilmiştir.

4.1. Sysconf Tanımı

`long int` **sysconf**`(int` *parametre*`)` işlev

Bu işlem çalışma anı sistem parametrelerini sorgulamak için kullanılır. *parametre* argümanı olarak aşağıda listelenen **_SC_** sembollerinden biri kullanılır.

İşlevin normal dönüş değeri işlevden istediğiniz değerdir. **-1** değeri dönmüşse bu gerçeklemenin böyle bir sınır içermediğini belirtebileceği gibi bir hata durumunu da gösterebilir.

Aşağıdaki **errno** hata durumu bu işlev için tanımlanmıştır:

EINVAL
: *parametre* değeri geçersiz

4.2. sysconf Parametreleri

Buradaki sembolik sabitler **sysconf** işlevinde *parametre* argümanı olarak kullanmak içindir. Değerlerin tümü tamsayı sabitlerdir (daha doğrusu, sıralı tamsayı sabitlerdir).

_SC_ARG_MAX
: **ARG_MAX**'ın karşılığı olan bilgi.

_SC_CHILD_MAX
: **CHILD_MAX**'ın karşılığı olan bilgi.

_SC_OPEN_MAX
: **OPEN_MAX**'ın karşılığı olan bilgi.

_SC_STREAM_MAX
: **STREAM_MAX**'ın karşılığı olan bilgi.

_SC_TZNAME_MAX

TZNAME_MAX'ın karşılığı olan bilgi.

_SC_NGROUPS_MAX
NGROUPS_MAX'ın karşılığı olan bilgi.

_SC_JOB_CONTROL
_POSIX_JOB_CONTROL'un karşılığı olan bilgi.

_SC_SAVED_IDS
_POSIX_SAVED_IDS'nin karşılığı olan bilgi.

_SC_VERSION
_POSIX_VERSION'un karşılığı olan bilgi.

_SC_CLK_TCK
CLOCKS_PER_SEC'in karşılığı olan bilgi. Bkz. *İşlemci Zamanının Sorgulanması* (sayfa: 540).

_SC_CHARCLASS_NAME_MAX
Genişletilmiş bir yerel belirtiminde bir karakter sınıf ismi için izin verilen olası en büyük uzunluğun karşılığı olan bilgi.

_SC_REALTIME_SIGNALS
_POSIX_REALTIME_SIGNALS'in karşılığı olan bilgi.

_SC_PRIORITY_SCHEDULING
_POSIX_PRIORITY_SCHEDULING'in karşılığı olan bilgi.

_SC_TIMERS
_POSIX_TIMERS'ın karşılığı olan bilgi.

_SC_ASYNCHRONOUS_IO
_POSIX_ASYNCHRONOUS_IO'ın karşılığı olan bilgi.

_SC_PRIORITIZED_IO
_POSIX_PRIORITIZED_IO'nun karşılığı olan bilgi.

_SC_SYNCHRONIZED_IO
_POSIX_SYNCHRONIZED_IO'nun karşılığı olan bilgi.

_SC_FSYNC
_POSIX_FSYNC'in karşılığı olan bilgi.

_SC_MAPPED_FILES
_POSIX_MAPPED_FILES'ın karşılığı olan bilgi.

_SC_MEMLOCK
_POSIX_MEMLOCK'un karşılığı olan bilgi.

_SC_MEMLOCK_RANGE
_POSIX_MEMLOCK_RANGE'in karşılığı olan bilgi.

_SC_MEMORY_PROTECTION
_POSIX_MEMORY_PROTECTION'ın karşılığı olan bilgi.

_SC_MESSAGE_PASSING
_POSIX_MESSAGE_PASSING'in karşılığı olan bilgi.

_SC_SEMAPHORES

_POSIX_SEMAPHORES'un karşılığı olan bilgi.

_SC_SHARED_MEMORY_OBJECTS
_POSIX_SHARED_MEMORY_OBJECTS'in karşılığı olan bilgi.

_SC_AIO_LISTIO_MAX
_POSIX_AIO_LISTIO_MAX'ın karşılığı olan bilgi.

_SC_AIO_MAX
_POSIX_AIO_MAX'ın karşılığı olan bilgi.

_SC_AIO_PRIO_DELTA_MAX
Bir sürecin zamanlama önceliğinden itibaren eşzamansız G/Ç öncelik seviyesini arttırabileceği değer. Bu bir derleme anı değeri olan **AIO_PRIO_DELTA_MAX**'ın karşılığıdır.

_SC_DELAYTIMER_MAX
_POSIX_DELAYTIMER_MAX'ın karşılığı olan bilgi.

_SC_MQ_OPEN_MAX
_POSIX_MQ_OPEN_MAX'ın karşılığı olan bilgi.

_SC_MQ_PRIO_MAX
_POSIX_MQ_PRIO_MAX'ın karşılığı olan bilgi.

_SC_RTSIG_MAX
_POSIX_RTSIG_MAX'ın karşılığı olan bilgi.

_SC_SEM_NSEMS_MAX
_POSIX_SEM_NSEMS_MAX'ın karşılığı olan bilgi.

_SC_SEM_VALUE_MAX
_POSIX_SEM_VALUE_MAX'ın karşılığı olan bilgi.

_SC_SIGQUEUE_MAX
_POSIX_SIGQUEUE_MAX'ın karşılığı olan bilgi.

_SC_TIMER_MAX
_POSIX_TIMER_MAX'ın karşılığı olan bilgi.

_SC_PII
_POSIX_PII'nin karşılığı olan bilgi.

_SC_PII_XTI
_POSIX_PII_XTI'nin karşılığı olan bilgi.

_SC_PII_SOCKET
_POSIX_PII_SOCKET'in karşılığı olan bilgi.

_SC_PII_INTERNET
_POSIX_PII_INTERNET'in karşılığı olan bilgi.

_SC_PII_OSI
_POSIX_PII_OSI'nin karşılığı olan bilgi.

_SC_SELECT
_POSIX_SELECT'in karşılığı olan bilgi.

_SC_UIO_MAXIOV

_POSIX_UIO_MAXIOV'un karşılığı olan bilgi.

_SC_PII_INTERNET_STREAM
_POSIX_PII_INTERNET_STREAM'in karşılığı olan bilgi.

_SC_PII_INTERNET_DGRAM
_POSIX_PII_INTERNET_DGRAM'ın karşılığı olan bilgi.

_SC_PII_OSI_COTS
_POSIX_PII_OSI_COTS'un karşılığı olan bilgi.

_SC_PII_OSI_CLTS
_POSIX_PII_OSI_CLTS'nin karşılığı olan bilgi.

_SC_PII_OSI_M
_POSIX_PII_OSI_M'nin karşılığı olan bilgi.

_SC_T_IOV_MAX
T_IOV_MAX değişkeni ile ilgili değerin değeri hakkında bilgi.

_SC_THREADS
_POSIX_THREADS'in karşılığı olan bilgi.

_SC_THREAD_SAFE_FUNCTIONS
_POSIX_THREAD_SAFE_FUNCTIONS'ın karşılığı olan bilgi.

_SC_GETGR_R_SIZE_MAX
_POSIX_GETGR_R_SIZE_MAX'ın karşılığı olan bilgi.

_SC_GETPW_R_SIZE_MAX
_POSIX_GETPW_R_SIZE_MAX'ın karşılığı olan bilgi.

_SC_LOGIN_NAME_MAX
_POSIX_LOGIN_NAME_MAX'ın karşılığı olan bilgi.

_SC_TTY_NAME_MAX
_POSIX_TTY_NAME_MAX'ın karşılığı olan bilgi.

_SC_THREAD_DESTRUCTOR_ITERATIONS
_POSIX_THREAD_DESTRUCTOR_ITERATIONS'ın karşılığı olan bilgi.

_SC_THREAD_KEYS_MAX
_POSIX_THREAD_KEYS_MAX'ın karşılığı olan bilgi.

_SC_THREAD_STACK_MIN
_POSIX_THREAD_STACK_MIN'in karşılığı olan bilgi.

_SC_THREAD_THREADS_MAX
_POSIX_THREAD_THREADS_MAX'ın karşılığı olan bilgi.

_SC_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKADDR'nin karşılığı olan bilgi.

_SC_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_ATTR_STACKSIZE'ın karşılığı olan bilgi.

_SC_THREAD_PRIORITY_SCHEDULING

_POSIX_THREAD_PRIORITY_SCHEDULING'in karşılığı olan bilgi.

_SC_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_INHERIT'in karşılığı olan bilgi.

_SC_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIO_PROTECT'in karşılığı olan bilgi.

_SC_THREAD_PROCESS_SHARED
_POSIX_THREAD_PROCESS_SHARED'ın karşılığı olan bilgi.

_SC_2_C_DEV
Sistemin POSIX.2 C derleyici komutu **c89**'u içerip içermediği hakkında bilgi.

_SC_2_FORT_DEV
Sistemin POSIX.2 Fortran derleyici komutu **fort77**'yi içerip içermediği hakkında bilgi.

_SC_2_FORT_RUN
Sistemin, Fortran taşıma denetimini yorumlayacak POSIX.2 **asa**komutunu içerip içermediği hakkında bilgi.

_SC_2_LOCALEDEF
Sistemin POSIX.2 **localedef** komutunu içerip içermediği hakkında bilgi.

_SC_2_SW_DEV
Sistemin **ar**, **make** ve **strip** POSIX.2 komutlarını içerip içermediği hakkında bilgi.

_SC_BC_BASE_MAX
bc aracındaki azami **obase** değeri hakkında bilgi.

_SC_BC_DIM_MAX
bc aracındaki bir dizinin azami boyutu hakkında bilgi.

_SC_BC_SCALE_MAX
bc aracındaki azami **scale** değeri hakkında bilgi.

_SC_BC_STRING_MAX
bc aracındaki bir dizge sabitin azami uzunluğu hakkında bilgi.

_SC_COLL_WEIGHTS_MAX
Bir yerelin karşılaştırma dizilimlerini tanımlamakta kullanılması gerekebilen azami önem sayısı hakkında bilgi.

_SC_EXPR_NEST_MAX
expr aracı kullanıldığında parantezler içine alınarak iç içe kullanılabilecek ifadelerin azami sayısı hakkında bilgi.

_SC_LINE_MAX
POSIX.2 metin araçları ile işlenebilecek bir metin satırının azami uzunluğu hakkında bilgi.

_SC_EQUIV_CLASS_MAX
Yerel tanımında **LC_COLLATE** kategorisinin **order** anahtar sözcüğünün bir girdisine atanabilecek azami önem sayısı hakkında bilgi. GNU C kütüphanesi halen yerel tanımlarını desteklememektedir.

_SC_VERSION
Kütüphanenin ve çekideğin desteklediği POSIX.1 sürüm numarası hakkında bilgi.

_SC_2_VERSION
Sistem araçlarının desteklediği POSIX.2 sürüm numarası hakkında bilgi

_SC_PAGESIZE
Makinanın sanal bellek sayfası genişliği hakkında bilgi. **getpagesize** işlevi de bu bilgi ile döner; bkz.*Bellek Parametrelerinin Sorgulanması* (sayfa: 590).

_SC_NPROCESSORS_CONF
Yapılandırılmış işlemci sayısı hakkında bilgi.

_SC_NPROCESSORS_ONLN
Kullanılabilir işlemci sayısı hakkında bilgi.

_SC_PHYS_PAGES
Sistemdeki fiziksel bellek sayfalarının sayısı hakkında bilgi.

_SC_AVPHYS_PAGES
Sistemdeki kullanılabilir fiziksel bellek sayfalarının sayısı hakkında bilgi.

_SC_ATEXIT_MAX
atexit ile sonlanma sırasında çalışmak üzere kaydedilecek işlevlerin sayısı hakkında bilgi; bkz. *Çıkışta Temizlik* (sayfa: 682).

_SC_XOPEN_VERSION
_XOPEN_VERSION'un karşılığı olan bilgi.

_SC_XOPEN_XCU_VERSION
_XOPEN_XCU_VERSION'un karşılığı olan bilgi.

_SC_XOPEN_UNIX
_XOPEN_UNIX'in karşılığı olan bilgi.

_SC_XOPEN_REALTIME
_XOPEN_REALTIME'ın karşılığı olan bilgi.

_SC_XOPEN_REALTIME_THREADS
_XOPEN_REALTIME_THREADS'in karşılığı olan bilgi.

_SC_XOPEN_LEGACY
_XOPEN_LEGACY'nin karşılığı olan bilgi.

_SC_XOPEN_CRYPT
_XOPEN_CRYPT'in karşılığı olan bilgi.

_SC_XOPEN_ENH_I18N
_XOPEN_ENH_I18N'nin karşılığı olan bilgi.

_SC_XOPEN_SHM
_XOPEN_SHM'nin karşılığı olan bilgi.

_SC_XOPEN_XPG2
_XOPEN_XPG2'nin karşılığı olan bilgi.

_SC_XOPEN_XPG3
_XOPEN_XPG3'ün karşılığı olan bilgi.

_SC_XOPEN_XPG4

_XOPEN_XPG4'ün karşılığı olan bilgi.

_SC_CHAR_BIT
char türünde bir değişkendeki bit sayısı hakkında bilgi.

_SC_CHAR_MAX
char türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_CHAR_MIN
char türünde bir değişkende saklanabilecek asgari değer hakkında bilgi.

_SC_INT_MAX
int türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_INT_MIN
int türünde bir değişkende saklanabilecek asgari değer hakkında bilgi.

_SC_LONG_BIT
long int türünde bir değişkendeki bit sayısı hakkında bilgi.

_SC_WORD_BIT
bir yazmaç sözcüğü değişkenindeki bit sayısı hakkında bilgi.

_SC_MB_LEN_MAX
Bir geniş karakter değerinin çok baytlı gösteriminin azami uzunluğu.

_SC_NZERO
Sıfır öncelikli süreçlerin öncelik seviyesini dahili olarak ifade etmekte kullanılan değer hakkında bilgi.

SC_SSIZE_MAX
ssize_t türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_SCHAR_MAX
signed char türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_SCHAR_MIN
signed char türünde bir değişkende saklanabilecek asgari değer hakkında bilgi.

_SC_SHRT_MAX
short int türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_SHRT_MIN
short int türünde bir değişkende saklanabilecek asgari değer hakkında bilgi.

_SC_UCHAR_MAX
unsigned char türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_UINT_MAX
unsigned int türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_ULONG_MAX
unsigned long int türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_USHRT_MAX
unsigned short int türünde bir değişkende saklanabilecek azami değer hakkında bilgi.

_SC_NL_ARGMAX

NL_ARGMAX'ın karşılığı olan bilgi.

_SC_NL_LANGMAX
NL_LANGMAX'ın karşılığı olan bilgi.

_SC_NL_MSGMAX
NL_MSGMAX'ın karşılığı olan bilgi.

_SC_NL_NMAX
NL_NMAX'ın karşılığı olan bilgi.

_SC_NL_SETMAX
NL_SETMAX'ın karşılığı olan bilgi.

_SC_NL_TEXTMAX
NL_TEXTMAX'ın karşılığı olan bilgi.

4.3. `sysconf` Örnekleri

Önce ilgilendiğiniz parametre için bir makro tanımını sınadıktan sonra, sadece makro tanımlı değilse **sysconf** çağrısı yapın. Örneğin, burada sistemde iş denetimi desteği olup olmadığı sınanmaktadır:

```
int
have_job_control (void)
{
#ifdef _POSIX_JOB_CONTROL
  return 1;
#else
  int value = sysconf (_SC_JOB_CONTROL);
  if (value < 0)
    /* Sistem hata verdiğine göre denenecek
       başka birşey kalmadı.  */
    fatal (strerror (errno));
  return value;
#endif
}
```

Burada ise bir sayısal sınırın değeri alınmaktadır:

```
int
get_child_max ()
{
#ifdef CHILD_MAX
  return CHILD_MAX;
#else
  int value = sysconf (_SC_CHILD_MAX);
  if (value < 0)
    fatal (strerror (errno));
  return value;
#endif
}
```

5. Asgari Değerler

Bu kısımda sistem sınır parametreleri için POSIX asgari üst sınırlarının isimlerine yer verilmiştir. Bu değerlerin önemi, belli bir sistem için bu sınırların uzun uzadıya sınanmadan rahatça kullanılabilmesidir.

_POSIX_AIO_LISTIO_MAX
Bir G/Ç listesi çağrısında belirtilebilecek G/Ç işlemlerinin azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu sabitin değeri **2**'dir; yani, yapılacak işlemler listesine en fazla iki yeni girdi ekleyebilirsiniz.

_POSIX_AIO_MAX
Yapılacak eşzamansız G/Ç işlemlerinin azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu sabitin değeri **1**'dir. Yani, eşzamansız uyarılar alarak normal çalışmanın devamı sırasında bir anda bir işlemden fazlasının yapılmasını bekleyemezsiniz.

_POSIX_ARG_MAX
exec işlevlerine aktarılabilen *argv* ve *environ* argümanlarının uzunluklarının toplamının azami değeri için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **4096**'dır.

_POSIX_CHILD_MAX
Her gerçek kullanıcı kimlik için aynı anda çalışabilecek süreçlerin azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **6**'dır.

_POSIX_NGROUPS_MAX
Bir sürecin sahip olabileceği ek grup kimliklerinin azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **0**'dır.

_POSIX_OPEN_MAX
Tek bir sürecin aynı anda açabileceği dosya sayısının azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **16**'dır.

_POSIX_SSIZE_MAX
ssize_t türünde bir nesneye kapasitebilen en büyük değer için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **32767**'dir.

_POSIX_STREAM_MAX
Tek bir sürecin aynı anda açabileceği akım sayısının azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **8**'dir.

_POSIX_TZNAME_MAX
Zaman dilimi isminin azami uzunluğu için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **3**'tür.

_POSIX2_RE_DUP_MAX
Bir düzenli ifadede `\{enaz,ençok\}` yapısında garanti edilmiş en büyük yineleme sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **255**'tir.

6. Dosya Sistemi Kapasite Sınırları

POSIX.1 standardı, dosya sistemini sınırlarını açıklayan bir dizi parametre belirtir. Sistem için sabit, bir parametre için tektip sınır olabilirler fakat genel bir durumu yansıtmazlar. Çoğu sistemde, farklı dosya sistemlerinde (hatta bazı parametreler, hatta farklı dosyalarda) farklı azami sınırlar olabilmektedir. Örneğin, bazı dosya sistemlerini farklı makinalara NFS üzerinden bağlıyorsanız bu durumlarla karşılaşırsınız.

Aşağıdaki makroların herbiri sadece sistem o parametre için bir sabit ve tektip sınıra sahipse `limits.h` içinde tanımlıdır. Eğer sistem farklı dosya sistemleri için farklı sınırlara izin veriyorsa, makro tanımsızdır; böyle bir durumda belli bir dosyaya uygulanan sınırı öğrenmek için **pathconf** ya da **fpathconf** kullanın. Bkz. *pathconf Kullanımı* (sayfa: 798).

Bu parametrelerin her biri için ismi **_POSIX** ile başlayan başka makrolar da vardır. Bunlar herhangi bir POSIX sisteminde izin verilen sınırın en düşük değerini verirler. Bkz. *Dosyalarla İlgili Asgari Değerler* (sayfa: 797).

`int` **LINK_MAX** makro

Bir dosyaya verilebilecek isim sayısı için (varsa) tektip sistem sınırı. Bkz. *Sabit Bağlar* (sayfa: 364).

`int` **MAX_CANON** makro

Girdi düzenlemesi etkin olduğunda bir girdi satırındaki metnin uzunluğu için (varsa) tektip sistem sınırı. Bkz. *İki Girdi Tarzı: Kurallı veya Kuralsız* (sayfa: 443).

`int` **MAX_INPUT** makro

Bir girdiye baştan sona yazılan karakterlerin toplam sayısı için (varsa) tektip sistem sınırı. Bkz. *G/Ç Kuyrukları* (sayfa: 443).

`int` **NAME_MAX** makro

Bir dosya ismi bileşeninin uzunluğu için (varsa) tektip sistem sınırı.

`int` **PATH_MAX** makro

Bir tam dosya ismi uzunluğu için (varsa) tektip sistem sınırı. (**open** çağrısındaki gibi bir sistem çağrısında belirtilen bir argüman gibi tam dosya ismi).

`int` **PIPE_BUF** makro

Bir boruya atomik olarak yazılabilen baytların sayısı için (varsa) tektip sistem sınırı. Eğer aynı boruya aynı anda çok sayıda süreç yazıyorsa, farklı süreçlerdeki çıktı bu boyutun parçaları halinde saçılmış olabilir. Bkz. *Borular ve FIFOlar* (sayfa: 393).

Bunlar aynı bilgi için isimleri farklı diğer makrolardır:

`int` **MAXNAMLEN** makro

NAME_MAX makrosunun BSD karşılığıdır. `dirent.h` dosyasında tanımlıdır.

`int` **FILENAME_MAX** makro

Bu makronun değeri bir dosya ismi dizgesinin azami uzunlığunu ifade eden bir tamsayı sabittir ve `stdio.h` dosyasıda tanımlıdır.

PATH_MAX'ın tersine, bu makro aslında bir sınır bulunmasa bile tanımlanır. Böyle bir durumda genellikle oldukça büyük bir sayıdır. *GNU sisteminde de bu daima böyledir.*

Kullanım Bilgisi

Bir dosya ismini saklayacak bir dizinin boyutu olarak **FILENAME_MAX** kullanmayın! Bu kadar büyük bir dizi yapmanız mümkün değildir! Bunu yapmaktansa *özdevimli ayırma* (sayfa: 49) yapın.

7. Dosya Desteği Seçenekleri

POSIX, dosyalarla çalışan sistem çağrıları için bazı sisteme özel seçenekler tanımlar. Bazı sistemler bu seçenekleri desteklerken bazıları da desteklemeyebilir. Bu seçenekler kütüphane ile değil, çekirdek tarafından sağlandığından bu seçenekler için GNU C kütüphanesinin kullanımı bunların desteklendiğini garanti etmez; bu tamamen kullandığınız sisteme bağlıdır. Bunlar ayrıca tek bir makina üzerindeki farklı dosya sistemleri arasında bile değişiklik gösterir.

Bu kısımda açıklanan makroları ilgili seçeneğin makinanızda desteklenip desteklenmediğini saptamak için sınayabilirsiniz. Eğer ilgili makro `unistd.h` dosyasında tanımlıysa, onun değeri bu özelliğin sistemde desteklenip desteklenmediği bilgisini içerir. (**-1** değeri desteklenmediğini; bundan farklı bir değer ise desteklendiğini belirtir). Eğer makro tanımsızsa, belli dosyalar bu özelliği destekleyebilir de desteklemeyebilir de.

GNU C kütüphanesini destekleyen tüm makinalar ayrıca NFS desteğine de sahip olduklarından, tüm dosya sistemlerinde **_POSIX_CHOWN_RESTRICTED** ve **_POSIX_NO_TRUNC** destekleri var mı yok mu belirleyen bir genel deyim asla yapılamaz. Bu makro isimleri bu bakımdan GNU C kütüphanesinde makro olarak asla tanımlanmaz.

`int` **_POSIX_CHOWN_RESTRICTED** makro

Bu seçenek etkinse, **chown** kısıtlanır; bir ayrıcalıksız sürecin bir dosyanın grubunu sadece ya sürecin etkin grup kimliğine ya da sürecin ek grup kimliklerinden birine ayarlamasına izin verilir. Bkz. *Dosya İyeliği* (sayfa: 377).

`int` **_POSIX_NO_TRUNC** makro

Bu seçenek etkinse, **NAME_MAX**'dan daha uzun dosya ismi bileşenleri bir **ENAMETOOLONG** hatası üretir. Aksi takdirde gereğinden uzun dosya isimleri sadece kırpılır.

`unsigned char` **_POSIX_VDISABLE** makro

Bu seçenek sadece uçbirim aygıtlarının dosyaları için anlamlıdır. Seçenek etkinse, özel denetim karakterlerinin işlenmesi tek tek iptal edilebilir. Bkz. *Özel Karakterler* (sayfa: 454).

Bu makrolardan biri tanımsızsa, bu seçeneğin bazı dosyalarda etkili bazılarında etkisiz olduğu anlamına gelir. Bir seçeneğin belli bir dosyada etkin olup olmadığını **pathconf** veya **fpathconf** işlevi ile öğrenebilirsiniz. Bkz. *pathconf Kullanımı* (sayfa: 798).

8. Dosyalarla İlgili Asgari Değerler

Bu kısımda önceki bölümde bahsedilen parametreler için POSIX asgari üst sınırlarının isimlerine yer verilmiştir. Bu değerlerin önemi, belli bir sistem için bu sınırların uzun uzadıya sınanmadan rahatça kullanılabilmesidir. Asıl sınır gerekirse istenebilir.

_POSIX_LINK_MAX
Bir dosyanın bağ sayısının azami değeri için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **8**'dir; yani bir sistem sınırlaması ile karşılaşmadan yapabileceğiniz dosya bağlarının sayısı sekizdir.

_POSIX_MAX_CANON
Bir uçbirim aygıtında bir kurallı girdi satırındaki baytların azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **255**'tir.

_POSIX_MAX_INPUT
Bir uçbirim aygıtı *girdi kuyruğundaki* (sayfa: 447) (ya da sürekli yazma tamponunda) baytların azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **255**'tir.

_POSIX_NAME_MAX
Bir dosya ismi bileşenindeki baytların azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **14**'tür.

_POSIX_PATH_MAX
Bir tam dosya ismindeki baytların azami sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **256**'dır.

_POSIX_PIPE_BUF

Bir boruya atomik olarak yazılabilen azami bayt sayısı için POSIX tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **512**'dir.

SYMLINK_MAX
Bir sembolik bağdaki azami bayt sayısı.

POSIX_REC_INCR_XFER_SIZE
Dosya iletim boyu olarak **POSIX_REC_MIN_XFER_SIZE** ve **POSIX_REC_MAX_XFER_SIZE** değerleri arasında önerilen artış.

POSIX_REC_MAX_XFER_SIZE
Önerilen azami dosya iletim boyu.

POSIX_REC_MIN_XFER_SIZE
Önerilen asgari dosya iletim boyu.

POSIX_REC_XFER_ALIGN
Önerilen dosya iletim tamponu hizalaması.

9. `pathconf` Kullanımı

Makinanızda bir dosya sistemi parametresi için farklı dosyalara farklı değerler varsa, bu kısımdaki işlevleri kullanarak belli bir dosya için ilgilendiğiniz değerin ne olduğunu öğrenebilirsiniz.

Bu işlevler ve *parametre* argümanında belirtilebilecek sabitler `unistd.h` başlık dosyasında bildirilmiştir.

```
long int pathconf(const char *dosyaismi,                    işlev
                  int         parametre)
```

Bu işlev ismi *dosyaismi* olan dosyaya uygulanan sınırlar hakkında bilgi almak için kullanılır.

parametre argümanı aşağıda listelenen **_PC_** sabitlerinden biri olmalıdır.

İşlevin normal dönüş değeri işlevden istediğiniz değerdir. **-1** değeri dönmüşse bu gerçeklemenin böyle bir sınır içermediğini belirtebileceği gibi bir hata durumunu da gösterebilir. İlk durumda **errno** değişkenine bir değer atanmazken ikinci durumda değişken hata durumunu içerir. Bu bakımdan dönen değerin bir hata durumu içerip içermediğini saptayabilmek için islevi çağırmadan hemen önce **errno** değişlenine **0** değeri atanmalıdır.

Dosya ismi hatalarına (sayfa: 234) ek olarak, bu işlev için şu hata durumu tanımlanmıştır:

EINVAL
parametre değeri geçersiz ya da gerçeklemede bu dosya için *parametre* desteği yok.

```
long int fpathconf(int dosyatanıtıcı,                       işlev
                   int parametre)
```

pathconf işlevi gibi olmakla birlikte bilgi bir dosya ismi için değil bir açık dosya tanıtıcı için döner.

Aşağıdaki **errno** hata durumları bu işlev için tanımlanmıştır:

EBADF
dosyatanıtıcı argümanı geçerli bir dosya tanıtıcı değil

EINVAL

parametre değeri geçersiz ya da gerçeklemede bu dosya için *parametre* desteği yok.

Buradaki sembolik sabitler **pathconf** ve **fpathconf** işlevlerinin *parametre* argümanında kullanmak içindir. Değerlerin hepsi tamsayı sabitlerdir.

_PC_LINK_MAX
: **LINK_MAX**'in değeri hakkında bilgi.

_PC_MAX_CANON
: **MAX_CANON**'un değeri hakkında bilgi.

_PC_MAX_INPUT
: **MAX_INPUT**'un değeri hakkında bilgi.

_PC_NAME_MAX
: **NAME_MAX**'ın değeri hakkında bilgi.

_PC_PATH_MAX
: **PATH_MAX**'ın değeri hakkında bilgi.

_PC_PIPE_BUF
: l**PIPE_BUF**'ın değeri hakkında bilgi.

_PC_CHOWN_RESTRICTED
: **_POSIX_CHOWN_RESTRICTED**'in değeri hakkında bilgi.

_PC_NO_TRUNC
: **_POSIX_NO_TRUNC**'ın değeri hakkında bilgi.

_PC_VDISABLE
: **_POSIX_VDISABLE**'ın değeri hakkında bilgi.

_PC_SYNC_IO
: **_POSIX_SYNC_IO**'nun değeri hakkında bilgi.

_PC_ASYNC_IO
: **_POSIX_ASYNC_IO**'nun değeri hakkında bilgi.

_PC_PRIO_IO
: **_POSIX_PRIO_IO**'nun değeri hakkında bilgi.

_PC_SOCK_MAXBUF
: **_POSIX_PIPE_BUF**'un değeri hakkında bilgi.

_PC_FILESIZEBITS
: Dosya sisteminde geniş dosyaların kullanılabilirliği hakkında bilgi.

_PC_REC_INCR_XFER_SIZE
: **POSIX_REC_INCR_XFER_SIZE**'ın değeri hakkında bilgi.

_PC_REC_MAX_XFER_SIZE
: **POSIX_REC_MAX_XFER_SIZE**'ın değeri hakkında bilgi.

_PC_REC_MIN_XFER_SIZE
: **POSIX_REC_MIN_XFER_SIZE**'ın değeri hakkında bilgi.

_PC_REC_XFER_ALIGN

l**POSIX_REC_XFER_ALIGN**'ın değeri hakkında bilgi.

10. Bazı Araçların Kapasite Sınırları

POSIX.2 standardı, işletim sistemi ve kütüphane davranışlarından başka bazı uygulamaların davranışlarına uygulanan ve **sysconf** üzerinden erişilen bazı sistem sınırları da belirtir.

GNU C kütüphanesi bu sınırlar için makrolar tanımlar ve eğer bu sınırları öğrenmek isterseniz **sysconf** bu değerleri döndürür. Bu değerler sadece POSIX.2 tarafından izin verilen en küçük değerlerdir.

`int` **BC_BASE_MAX** makro

bc uygulamasının desteklemeyi garanti ettiği en büyük **obase** değeridir.

`int` **BC_DIM_MAX** makro

bc uygulamasının desteklemeyi garanti ettiği bir dizinin eleman sayısının en büyük değeridir.

`int` **BC_SCALE_MAX** makro

bc uygulamasının desteklemeyi garanti ettiği en büyük **scale** değeridir.

`int` **BC_STRING_MAX** makro

bc uygulamasının desteklemeyi garanti ettiği bir dizge sabitinin karakter sayısının en büyük değeridir.

`int` **COLL_WEIGHTS_MAX** makro

Bir yerelin karşılaştırma dizilimlerini tanımlamakta kullanılması gerekebilen azami önem sayısı.

`int` **EXPR_NEST_MAX** makro

expr aracı kullanıldığında parantezler içine alınarak iç içe kullanılabilecek ifadelerin azami sayısı.

`int` **LINE_MAX** makro

POSIX.2 metin araçları ile işlenebilecek bir metin satırının azami uzunluğu. (Bu araçların GNU sürümlerini kullanıyorsanız aslında sanal belleğin büyüklüğü dışında bir sınır söz konusu değildir, fakat kütüphanede bunun belirtilebileceği bir yöntem yoktur.)

`int` **EQUIV_CLASS_MAX** makro

Yerel tanımında **LC_COLLATE** kategorisinin **order** anahtar sözcüğünün bir girdisine atanabilecek azami önem sayısı. GNU C kütüphanesi halen yerel tanımlarını desteklememektedir.

11. Araç Sınırları için Asgari Değerler

_POSIX2_BC_BASE_MAX

bc uygulamasının desteklemeyi garanti ettiği en büyük **obase** değeri için POSIX.2 tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **99**'dur.

_POSIX2_BC_DIM_MAX

bc uygulamasının desteklemeyi garanti ettiği bir dizinin eleman sayısının en büyük değeri için POSIX.2 tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **2048**'dir.

_POSIX2_BC_SCALE_MAX

bc uygulamasının desteklemeyi garanti ettiği en büyük **scale** değeri için POSIX.2 tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **99**'dur.

_POSIX2_BC_STRING_MAX

bc uygulamasının desteklemeyi garanti ettiği bir dizge sabitinin karakter sayısının en büyük değeri için POSIX.2 tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **1000**'dir.

_POSIX2_COLL_WEIGHTS_MAX
Bir yerelin karşılaştırma dizilimlerini tanımlamakta kullanılması gerekebilen azami önem sayısı için POSIX.2 tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **2**'dir.

_POSIX2_EXPR_NEST_MAX
expr aracı kullanıldığında parantezler içine alınarak iç içe kullanılabilecek ifadelerin azami sayısı için POSIX.2 tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **32**'dir.

_POSIX2_LINE_MAX
POSIX.2 metin araçları ile işlenebilecek bir metin satırının azami uzunluğu için POSIX.2 tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **2048**'dir.

_POSIX2_EQUIV_CLASS_MAX
Yerel tanımında **LC_COLLATE** kategorisinin **order** anahtar sözcüğünün bir girdisine atanabilecek azami önem sayısı için POSIX.2 tarafından izin verilen en kısıtlayıcı sınırdır. Bu değer **2**'dir. GNU C kütüphanesi halen yerel tanımlarını desteklememektedir.

12. Dizge Değerli Parametreler

POSIX.2, işletim sisteminden dizge değerli parametrelerin değerlerinin öğrenilmesi için **confstr** işlevini tanımlamıştır:

```
size_t confstr(int    parametre,                                    işlev
               char   *tampon,
               size_t uzunluk)
```

Bu işlev bir dizge değerli sistem parametresinin değerini okur ve *tampon*'da başlayan bellek alanının *uzunluk* baytına bu dizgeyi yerleştirerek döner. *parametre* argümanı aşağıda listelenen **_CS_** sembollerinden biri olmalıdır.

İşlevin normal dönüş değeri istenen dizgenin uzunluğudur. *tampon* olarak bir boş dizge verilmişse işlev dizgeyi buraya yerleştirmeye çalışmaz, sadece dizgenin uzunluğu ile döner. **0** dönüş değeri bir hata oluştuğunu gösterir.

Eğer istenen dizge için tamponda yeterince yer yoksa (yani *uzunluk* **-** **1**'den daha uzunsa), işlev dizgenin ilk *uzunluk* **-** **1** baytını (sonlandırıcı boş karaktere yer bırakarak) yerleştirir. Bu durumun oluştuğunu işlev *uzunluk* bayta eşit ya da daha büyük bir değerle dönerek bildirir.

Aşağıdaki **errno** hata durumu bu işlev için tanımlanmıştır:

EINVAL
parametre değeri geçersiz

confstr işlevinin okuyabileceği parametreler:

_CS_PATH
Çalıştırılabilir dosyaların aranacağı öntanımlı dosya yollarının önerilen değeridir. Kullanıcı sisteme oturum açtığında öntanımlı olarak bu dosya yollarına sahip olur.

_CS_LFS_CFLAGS
Eğer bir kaynak **_LARGEFILE_SOURCE** kullanarak derlenmişse, C derleyiciye hangi ek seçeneklerin verileceğini belirten bir dizge döner. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

_CS_LFS_LDFLAGS
Eğer bir kaynak **_LARGEFILE_SOURCE** kullanarak derlenmişse, ilintileyiciye hangi ek seçeneklerin verileceğini belirten bir dizge döner. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

_CS_LFS_LIBS
Eğer bir kaynak **_LARGEFILE_SOURCE** kullanarak derlenmişse, uygulamanın hangi ek kütüphanelerle ilintileneceğini belirten bir dizge döner. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

_CS_LFS_LINTFLAGS
Eğer bir kaynak **_LARGEFILE_SOURCE** kullanarak derlenmişse, lint aracına hangi ek seçeneklerin verileceğini belirten bir dizge döner. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

_CS_LFS64_CFLAGS
Eğer bir kaynak **_LARGEFILE64_SOURCE** kullanarak derlenmişse, C derleyiciye hangi ek seçeneklerin verileceğini belirten bir dizge döner. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

_CS_LFS64_LDFLAGS
Eğer bir kaynak **_LARGEFILE64_SOURCE** kullanarak derlenmişse, ilintileyiciye hangi ek seçeneklerin verileceğini belirten bir dizge döner. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

_CS_LFS64_LIBS
Eğer bir kaynak **_LARGEFILE64_SOURCE** kullanarak derlenmişse, uygulamanın hangi ek kütüphanelerle ilintileneceğini belirten bir dizge döner. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

_CS_LFS64_LINTFLAGS
Eğer bir kaynak **_LARGEFILE64_SOURCE** kullanarak derlenmişse, lint aracına hangi ek seçeneklerin verileceğini belirten bir dizge döner. Bkz. *Özellik Sınama Makroları* (sayfa: 25).

confstr işlevini dönecek dizgeye keyfi bir sınır belirtmeden kullanmanın tek yolu işlevi iki kere çağırmaktır; İlk çağrıda dizgenin uzunluğu döner, buna göre tamponu ayırıp işlevi tamponu doldurması için tekrar çağırırsınız. Örnek:

```
char *
get_default_path (void)
{
  size_t len = confstr (_CS_PATH, NULL, 0);
  char *buffer = (char *) xmalloc (len);

  if (confstr (_CS_PATH, buf, len + 1) == 0)
    {
      free (buffer);
      return NULL;
    }

  return buffer;
}
```

XXXII. Şifrelemeyle İlgili İşlevler

İçindekiler

Bir çok sistem üzerinde, kullanıcı kimlik denetimine gerek yoktur; örneğin, bir ağa bağlı olmayan bir iş istasyonu sanırım herhangi bir kullanıcı kimlik denetimine ihtiyaç duymaz, çünkü davetsiz bir misafirin makinayı kullanması için fiziksel erişime ihtiyacı vardır.

Bazen, bir kullanıcının bir makinanın sağladığı bir servisi kullanmak için yetkili olduğundan emin olmak gerekir––örneğin, belirli bir kullanıcı kimliği ile oturum açmak (Bkz. *Kullanıcılar ve Gruplar* (sayfa: 742)). Bunu yapmanın geleneksel bir yolu her kullanıcı için gizli bir ***parola*** seçmektir; böylece sistem kullanıcı olduğunu iddia eden birine kullanıcının parolasının ne olduğunu sorabilir ve eğer kişi doğru parolayı verirse sistem uygun yetkileri verebilir.

Eğer bütün parolalar sadece bir yerdeki dosyada saklanıyorsa, o zaman bu dosya çok dikkatli korunmalıdır. Bundan kaçınmak için, şifreler, dosyada saklanmadan önce, çıktısına bakılarak girdisinin ne olduğunun kolayca anlaşılamayacağı, bir ***tekyönlü işlev***den geçirilir.

GNU C kütüphanesi, FreeBSD 2.0 ile tanıdığımız bir işlev olan `crypt` işlevinin davranışı ile uyumlu bir tek yönlü işlev sağlar. Bu işlev, iki tekyönlü algoritma sağlar: biri modern BSD sistemleriyle uyumlu MD5 temelli ileti özümleme (bazıları ileti özeti der) (message–digest) algoritmasıdır, diğeri ise Unix sistemlerle uyumlu Veri Şifreleme Standardını (Data Encryption Standard – DES) temel almaktadır.

Ayrıca güvenli uzak yordam çağrılarını (Secure RPC) ve normal DES şifrelemede kullanmak için bazı kütüphane işlevlerini sağlamaktadır.

1. Yasal Sorunlar

Kanunların sürekli değişmesinden dolayı, kriptografiyi etkileyen kanunlar hakkında kesin bir inceleme yapmak mümkün değildir. Bunun yerine, bu bölüm sizi bazı üzücü noktalar hakkında uyarmaktadır; bu ülkenizin kanunlarında neleri aramanız gerektiği hakkında size yardımcı olabilir.

Bazı ülkelerde kriptografi kullanmak, sahip olmak veya ithal etmek için bir ruhsata sahip olmanız gerekir. Bu ülkeler arasında Belarus, Birmanya, Hindistan, Endonezya, İsrail, Kazakistan, Pakistan, Rusya ve Suudi Arabistan vardır.

Bazı ülkeler şifrelenmiş iletilerin radyo ile iletimini sınırlar; bazı telekomünikasyon taşıyıcıları kendi ağlarında şifrelenmiş iletilerin iletimini sınırlarlar.

Birçok ülkenin şifreleme yazılımı için bir takım ihraç kontrolleri vardır. Wassenaar Anlaşması 33 ülke arasındaki çok taraflı bir anlaşmadır ve bir takım şifreleme ihracını kısıtlarlar (Arjantin, Avustralya, Avusturya, Belçika, Bulgaristan, Kanada, Çek Cumhuriyeti, Danimarka, Finlandiya, Fransa, Almanya, Yunanistan, Macaristan, İrlanda, İtalya, Japonya, Lüksemburg, Hollanda, Yeni Zelanda, Norveç, Polonya, Portekiz, Kore Cumhuriyeti, Romanya, Rusya Federasyonu, Slovak Cumhuriyeti, İspanya, İsveç, İsviçre, Türkiye, Ukrayna, İngiltere ve Amerika Birleşik Devletleri). Farklı ülkeler düzenlemeyi farklı yollarla uygularlar; bazıları belli başlı "kamu alanı" yazılımlar için istisnalara izin vermezler, bazıları yazılımın ihracının somut biçimde yapılışını kısıtlar ve diğerleri önemli ek kısıtlamalar uygularlar.

Bireşik Devletler'in ek kuralları vardır. Bu yazılım genelde, "şifreleme kaynak kodu"nun ihracına izin veren, "kamuya açık" ve "kaynak kodla geliştirilen bir ürünün satışı veya ticari bir üretim anlaşması için telif hakkı veya ruhsat ücreti söz konusu olmayan", sözcükleriyle ifade edilebilen 15 CFR 740.13(e) altında ihraç edilebilir;

Bu alandaki kurallar sürekli değişmektedir. Eğer burada bahsedilen süresi dolmuş bir bilgiye rastlarsanız lütfen bunu hatalar veritabanına bildiriniz. *Yazılım Hatalarının Raporlanması* (sayfa: 956).

2. Parolaların Okunması

Parola okunacağında, onun gizli kalması için ekranda göstermekten kaçınmak gerekir. Aşağıdaki işlev bunu uygun bir şekilde yapmaktadır.

`char *getpass(const char *istem)` işlev

> **getpass** kullanıcının karşısına *istem*'i getirir, ardından uçbirimden girilen dizgeyi ekrana yazmadan okur. Kullanıcıların dosyalara düz metin şifreleri koymamaları için, **/dev/tty** gerçek uçbirimine bağlanmaya çalışır; bağlanamazsa **stdin** ve **stderr** akımlarını kullanır. **getpass** aynı zamanda **ISIG** uçbirim özelliğini kullanarak INTR, QUIT ve SUSP karakterlerini uçbirimde iptal eder. (bkz. *Yerel Kipler* (sayfa: 451)). Uçbirim **getpass** öncesinde ve sonrasında temizlenir, böylece yanlış yazılan şifre karakterleri kazara görünmez.
>
> Diğer C kütüphanelerinde, **getpass** parolanın sadece ilk **PASS_MAX** baytını döndürebilir. GNU C kütüphanesinin bir sınırı yoktur, yani **PASS_MAX** tanımlanmamıştır.
>
> Bu işlevin prtotipi `unistd.h` içindedir. **PASS_MAX** ise `limits.h` içinde tanımlanabilirdi.

Bu işlemler bütün durumlara uymayabilirler. Bu durumda, kullanıcıların kendi **getpass** eşdeğerini yazmaları önerilir. Örneğin, çok basit bir uygulaması şöyle olabilir:

```
#include <termios.h>
#include <stdio.h>

ssize_t
my_getpass (char **lineptr, size_t *n, FILE *stream)
{
  struct termios old, new;
  int nread;

  /* Ekrana yazmayı kapat, eğer yapamazsan başarısız ol. */
  if (tcgetattr (fileno (stream), &old) != 0)
    return -1;
  new = old;
  new.c_lflag &= ~ECHO;
  if (tcsetattr (fileno (stream), TCSAFLUSH, &new) != 0)
    return -1;

  /* Şifreyi oku. */
  nread = getline (lineptr, n, stream);

  /* Uçbirimi geri yükle. */
  (void) tcsetattr (fileno (stream), TCSAFLUSH, &old);

  return nread;
}
```

Eşdeğer uygulamamız **getline** ile aynı parametreleri alır (bkz. *Satır Yönlenimli Girdi* (sayfa: 250)); kullanıcıdan istenilen biçimde bilgi isteminde bulunulabilir.

3. Parolaların Şifrelenmesi

```
char *crypt(const char *anahtar,                                    işlev
            const char *tuz)
```

crypt işlevi parolayı, bir *anahtar* dizgesi ile aşağıda ne olduğu anlatıldığı gibi bir *tuz* karakter dizisi olarak alır ve başka bir tuz ile başlayan yazılabilir bir ASCII dizge döndürür. İşlevin çıktısından, onu üreten *anahtar* değerini bulmanın en iyi yolunun *anahtar*'ın gerçek değerini bulana kadar *anahtar* için tahminde bulunmak olduğuna inanılmaktadır.

tuz parametresi iki şey yapar. Öncelikle, hangi algoritmanın kullanılacağını seçer, MD5–temelli olanı mı yoksa DES–temelli olanı mı. İkinci olarak, parolaları içeren bir dosya üzerinde parola tahmin etmeye çalışan birilerine hayatı dar eder; *tuz* olmadan, bir davetsiz misafir **crypt** çalıştırarak ve sonucu dosyadaki parolalarla karşılaştırarak tahminde bulunabilir. *tuz* ile davetsiz misafir **crypt**'i her farklı tuz ile çalıştırmak zorunda kalır.

MD–5 temelli algoritma için, *tuz* **1** dizgesi ile başlayan, en çok 8 karakterle devam eden ve **$** ile ya da dizge sonu ile sonlandırılan bir dizgeden oluşmalıdır. **crypt**'in sonucu, eğer tuz bir ile bitmiyorsa *tuz*ı takip eden bir **$** ve bunu da izleyen **./0-9A-Za-z** alfabesinden 22 karakterle devam eder, toplamda en çok 34 karakter olabilir. *anahtar*'deki her karakter anlamlıdır.

DES temelli algoritma için, *tuz*, **./0-9A-Za-z** alfabesindeki iki karakterden oluşmalıdır ve **crypt** işlevinin sonucu bu iki karakteri takip eden aynı alfabeden 11 karakterle birlikte toplam 13 karakterden oluşur. *anahtar* değerinin sadece ilk 8 karakteri anlamlıdır.

MD5 temelli algoritmanın kullanışlı olan uzunluğu hakkında bir sınır yoktur ve daha güvenlidir. Bu nedenle DES temelliye nazaran tercih edilir.

Kullanıcı parolasını ilk girdiğinde, *tuz* rastgele yeni bir dizge olarak ayarlanmalıdır. Bir parolayı **crypt**'in önceki çağrısının sonucu ile doğrulamak için, önceki çağrının sonucunu *tuz* olarak geçirin.

Aşağıdaki kısa program **crypt**'in parola ilk kez girildiğinde nasıl kullanılacağına bir örnektir. Buradaki *tuz* üretiminin ancak kabul edilebilir olduğunu unutmayınız; bu makinalar arasında eşsiz değildir ve bir çok uygulamada saldıran kişi kullanıcının parolasını ne zaman ayarladığı bilgisine erişemez.

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <crypt.h>

int
main(void)
{
  unsigned long seed[2];
  char salt[] = "$1$........";
  const char *const seedchars =
    "./0123456789ABCDEFGHIJKLMNOPQRST"
    "UVWXYZabcdefghijklmnopqrstuvwxyz";
  char *password;
  int i;

  /* (Hemen hemen) Rastgele tohum üret.
     Bundan daha iyi yapmalısınız. */
  seed[0] = time(NULL);
  seed[1] = getpid() ^ (seed[0] >> 14 & 0x30000);
```

```
  /* 'tohum karakter'leri yazılabilir karakterlere dönüştür. */
  for (i = 0; i < 8; i++)
    salt[3+i] = seedchars[(seed[i/5] >> (i%5)*6) & 0x3f];

  /* Kullanıcı parolasını oku ve şifrele. */
  password = crypt(getpass("Parola:"), salt);

  /* Sonuçları yaz. */
  puts(password);
  return 0;
}
```

Diğer yazılım bir parolanın nasıl doğrulanacağını göstermektedir. Kullanıcıya parola isteminde bulunur ve ekrana "Erişim onaylandı." basar; eğer kullanıcı **GNU libc manual** yazarsa.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <crypt.h>

int
main(void)
{
  /* "GNU libc manual" değerinin haşlaması. */
  const char *const pass = "$1$/iSaq7rB$EoUw5jJPPvAPECNaaWzMK/";

  char *result;
  int ok;

  /* Kullanıcı parolasını oku ve beklenilen parolayı
     salt olarak geçirerek şifrele. */
  result = crypt(getpass("Parola:"), pass);

  /* Sonucu sına. */
  ok = strcmp (result, pass) == 0;

  puts(ok ? "Erisim onaylandi.": "Erisim reddedildi.");
  return ok ? 0: 1;
}
```

```
char *crypt_r(const char          *anahtar,                    işlev
              const char          *tuz,
              struct crypt_data * veri)
```

crypt_r işlevi **crypt** ile aynı şeyi yapar, fakat ek bir parametre ile işlev sonucu için yer bulundurur, bu yüzden evresel (reentrant) olabilir. **crypt_r** ilk kez çağrılmadan önce *veri* sıfırlarla doldurularak ilklendirilmelidir.

crypt_r işlevi bir GNU oluşumudur.

crypt ve **crypt_r** işlevlerinin prototipleri `crypt.h` başlık dosyası içinde bulunur.

4. DES Şifreleme

Veri Şifreleme Standardı (Data Encryption Standard), Ulusal Standartlar ve Teknoloji Enstitüsü (National Institute of Standards and Technology – NIST) tarafından yayınlanan ABD Hükümeti Federal Bilgi İşleme Standartları (US

Government Federal Information Processing Standards – FIPS) 46–3 içinde anlatılmaktadır. DES geliştirildiği 1970lerden beri tamamen analiz edilmiştir ve yeni önemli bir kusuru bulunamamıştır.

Ancak, DES sadece 56 bitlik bir anahtar kullanmaktadır (artı 8 eşlik biti), ve 1998 yapımı 200.000 Amerikan Doları değerinde bir makina 6 günde olası bütün anahtarları deneyebilmektedir; daha fazla parayla daha hızlı sonuçlar elde edilebilmektedir. Bu basit DES'i bir çok amaç için güvensiz kılmaktadır ve NIST, US hükümet sistemlerinde basit DES'in kullanım iznini kaldırmıştır.

Ciddi şifreleme işlevselliği için, bu yordamlar yerine özgür şifreleme kütüphanelerinden birinin kullanılması tavsiye edilir.

DES 64–bitlik bir blok ve 64–bitlik anahtar alarak 64–bitlik başka bir blok üreten, tersine çevrilebilir bir işlemdir. Genelde bitler numaralandırılır böylece en–anlamlı bit, yani her bloğun ilk biti 1 ile numaralandırılır.

Bu numaralama altında, anahtarın her 8. biti (8., 16., vs.) şifreleme algoritmasında kullanılmaz. Fakat anahtar tekil eşlik bitine sahip olmalıdır; 1'den 8'e, 9'dan 16'ya, şeklinde devam eden bitler '1'in tek sayılı eş bitlerine sahip olmalıdır ve bu tamamen kullanılmayan bitleri belirtir.

```
void setkey(const char *anahtar)                                    işlev
```

setkey işlevi *anahtar*'ın genişletilmiş bir biçimi olan bir iç veri yapısı kurar. *anahtar* her biti **char** içinde saklanan 64 bitlik bir dizi olarak tanımlanmıştır, ilk bit *anahtar[0]* ve 64. bit *anahtar[63]*'dür. *anahtar* doğru eşlik değerine sahip olmalıdır.

```
void encrypt(char *blok,                                            işlev
             int   bayrak)
```

encrypt işlevi eğer *bayrak* 0 ise *blok*'u şifreler, aksi takdirde **setkey** ile belirtilen anahtarı kullanarak *blok*'u deşifre eder. Sonuç *blok* içinde saklanır.

setkey gibi, *blok* da her biti **char** içinde saklanan 64 bitlik dizi olarak tanımlanır, fakat *blok* içinde eşlik biti yoktur.

```
void setkey_r(const char          *anahtar,                         işlev
              struct crypt_data * veri)
void encrypt_r(char               *blok,                            işlev
               int                 bayrak,
               struct crypt_data * veri)
```

Bunlar **setkey** ve **encrypt**'in evresel (reentrant) sürümleridir. Tek fark *anahtar* değerini saklayan ek parametresidir. **setkey_r** ilk kez çağrılmadan önce *veri* sıfırlarla doldurularak ilklendirilmelidir.

setkey_r ve **encrypt_r** işlevleri GNU oluşumlarıdır. **setkey**, **encrypt**, **setkey_r** ve **encrypt_r** işlevleri `crypt.h` içinde tanımlıdır.

```
int ecb_crypt(char     *anahtar,                                    işlev
              char     *bloklar,
              unsigned uzunluk,
              unsigned kip)
```

ecb_crypt işlevi DES kullanarak bir veya daha fazla blok şifreler veya şifresini çözer. Her blok bağımsız olarak şifrelenir.

bloklar ve *anahtar* 8–bitlik bayt paketleri içinde saklanır, böylece anahtarın ilk biti *anahtar[0]*'in en–anlamlı biti olarak, anahtarın 63. biti *anahtar[7]*'nin en az anlamlı biti olarak saklanır. *anahtar* doğru eşlik değerine sahip olmalıdır.

uzunluk, *bloklar* içindeki bayt sayısıdır. Bu değer 8'in katları olmalıdır (böylece bütün bloklar şifrelenebilir). *uzunluk*, **DES_MAXDATA** azami bayt sayısı ile sınırlandırılmıştır.

Şifrelemenin sonucu *bloklar* girdisi ile değiştirilir.

kip parametresi aşağıdaki ikisinin bit bit VEYAlanmasıdır:

DES_ENCRYPT
: Bu sabit, *kip* parametresinde kullanılır ve *bloklar*'ın şifreleneceğini belirtir.

DES_DECRYPT
: Bu sabit, *kip* parametresinde kullanılır ve *bloklar*'ın şifresinin çözüleceğini belirtir.

DES_HW
: Bu sabit, *kip* parametresinde kullanılır ve bir donanım cihazı kullanılacak mı sorar. Eğer donanım yoksa, şifreleme gerçekleşir ancak yazılımla.

DES_SW
: Bu sabit, *kip* parametresinde kullanılır ve donanım kullanılmayacağını belirtir.

İşlev sonucu aşağıdaki değerlerden biri olur:

DESERR_NONE
: Şifreleme başarıldı.

DESERR_NOHWDEVICE
: Şifreleme başarıldı, ancak donanım bulunamadı.

DESERR_HWERROR
: Şifreleme donanım sorunu nedeniyle başarılamadı.

DESERR_BADPARAM
: Şifreleme kötü parametre nedeniyle başarılamadı, örneğin *uzunluk* 8'in katı değil veya *uzunluk* **DES_MAXDATA** değerinden büyük.

```
int DES_FAILED(int hata)                                                    makro
```

Bu makro eğer **ecb_crypt** veya **cbc_crypt** sonucu *hata* bir 'başarı' sonuç koduysa 1, aksi takdirde 0 döndürür.

```
int cbc_crypt(char     *anahtar,                                            işlev
              char     *bloklar,
              unsigned uzunluk,
              unsigned kip,
              char     *yedek)
```

cbc_crypt işlevi Zincirleme Blok Şifreleme (Cipher Block Chaining) kipinde DES kullanarak bir veya daha fazla bloğu şifreler veya şifre çözer.

CBC kipinde şifreleme için, her blok şifrelenmeden önce *yedek* ile XOR'lanır, ardından *yedek* şifreleme sonucuyla yer değiştirilir, sonra diğer blok işlenir. Şifre çözme bu işlemin tersidir.

Bunun avantajı, şifrelenmeden önce aynı olan blokları şifrelendikten sonra yine aynı ama oldukça farklı yaptığı için, veri içindeki şablonları algılamanın daha zor olmasıdır.

Genellikle, şifreleme başlamadan önce *yedek* 8 rastgele bayt ile ayarlanır. Ardından 8 rastgele bayt şifrelenen veriyle birlikte aktarılır (kendisi şifrelenmeden) ve şifre çözmek için *yedek* olarak geri aktarılır. Diğer olasılık *yedek*'i ilk başta 8 sıfır yapmak ve ilk bloğu 8 rastgele bayt ile şifrelemektir.

Aksi takdirde, bütün parametreler **ecb_crypt** için olduğu gibidir.

`void` **des_setparity**`(char *`*anahtar*`)` işlev

des_setparity işlevi her baytın düşük bitlerini değiştirerek tek eşlik bitine sahip olmak için 8–bitlik baytlar içinde paketlenerek saklanan 64–bit *anahtar*'ı değiştirir,

ecb_crypt, **cbc_crypt** ve **des_setparity** işlevleri ve onlara eşlik eden makrolar `rpc/des_crypt.h` başlık dosyası içinde tanımlıdır.

XXXIII. Hata Ayıklama Desteği

İçindekiler

Uygulamaların hataları genellikle amacı hata ayıklamak olan yazılımlar kullanılarak ayıklanır. Fakat bazan bu mümkün olmaz ve sorunlar hakkında deneyim kazandıkça yazılımcıya mümkün olan en fazla bilgiyi sağlamak gerekir. Bu sebeple, yazılımcının sorunun kaynağına daha kolay erişebilmesini mümkün kılacak bir kaç işlev sağlanmıştır.

1. Köken Arama Listeleri

Bir *köken arama listesi* (backtrace), bir evre içindeki o an etkin olan işlev çağrılarının bir listesidir. Bir yazılımın köken arama listesini elde etmenin en uygun yolu **gdb** gibi harici bir hata ayıklayıcı kullanmaktır. Ancak, bazan bir köken arama listesini günlük tutma, tanı koyma gibi amaçlarla yazılım içinde kodlayarak elde etmek de gerekebilir.

Geçerli evrenin köken arama listesini elde eden ve onunla çalışan üç işlev vardır ve bunlar `execinfo.h` başlık dosyasında bildirilmiştir.

```
int backtrace(void **tampon,                                          işlev
              int     boyut)
```

backtrace işlevi o anki evrenin köken arama listesini elde eder ve bir liste göstericisi olarak *tampon* içine yerleştirir. *boyut* argümanı *tampon* içinde bulunacak **void *** türündeki elemanların sayısı olmalıdır. İşlev *tampon* içine konmuş olan listenin eleman sayısı ile döner.

tampon içine yerleştirilen göstericiler aslında araştırılan yığıttan edinilen dönüş adresleridir, yani her yığıt çerçevesi için bir dönüş adresi vardır.

Belli derleyici eniyilemelerinin edinilen bir geçerli köken arama listesi ile etkileşeceğini gözardı etmemelisiniz. Satıriçine alma işlemi satıriçi işlevlerin bir yığıt çerçevesine sahip olmamasına sebep olur; uç çağrı eniyilemesi bir yığıt çerçevesini bir diğeri ile değiştirir; çerçeve göstericisi elemesi yığıt içeriğinin **backtrace** tarafından doğru olarak yorumlanmasının engelleyecektir.

```
char **backtrace_symbols(void *const *tampon,                         işlev
                         int          boyut)
```

backtrace_symbols işlevi **backtrace** işleviyle edinilen listeyi bir dizge dizisine dönüştürür. *tampon* argümanı **backtrace** işleviyle elde edilen adres dizisinin göstericisi, *boyut* ise bu dizinin eleman sayısı (**backtrace** işlevinin dönüş değeri) olmalıdır.

İşlevin dönüş değeri *boyut* dizgelik dizge dizisine bir göstericidir. Her dizge *tampon* içindeki her elemanın basılabilir içeriğini gösterir. Saptanabiliyorsa işlev ismi, işleve bir başlangıç konumu ve geçerli dönüş adresini (onaltılık tabanda) içerir.

Şimdilik işlev ismi ve başlangıç konumu sadece kütüphaneler ve uygulamalar için ELF ikilik biçimi kullanılan sistemlerde elde edilebilmektedir. Diğer sistemlerde ise sadece onaltılık tabandaki dönüş adresi elde edilebilmektedir. Ayrıca, yazılımın işlev isimlerini içermesi için ilintileyiciye ek seçenekler belirtilebilir. (Örneğin, GNU **ld**'ye `--rdynamic` seçeneğini aktarabilirsiniz.)

backtrace_symbols işlevinin dönüş değeri **malloc** işlevi üzerinden edinilen ve **free** ile serbest bırakılması gereken bir göstericidir. Yalnız, sadece dönüş değeri serbest bırakılmalı, içerdiği dizgeler serbest bırakılmamalıdır.

Edinilen dizgeleri saklamak için bellek yetersizse işlev **NULL** ile döner.

```
void backtrace_symbols_fd(void *const *tampon,                    işlev
                          int          boyut,
                          int          dosyatanıtıcı)
```

backtrace_symbols_fd işlevi **backtrace_symbols** işlevinin yaptığı dönüşümün aynısını yapar, ancak farklı olarak, dizgelere bir gösterici döndürmek yerine her dizge bir satır olmak üzere dizgeleri *dosyatanıtıcı* tanıtıcısına yazar. **malloc** işlevini kullanmaz ve dosya tanıtıcılara yazan işlevlerin başarısız olmasına sebep olan durumlarda bu işlev de başarısız olabilir.

Aşağıdaki yazılımda bu işlevlerin kullanımı gösterilmiştir. **backtrace** tarafından döndürülen gösterici dizisinin adresinin yığıt üzerine ayrıldığına dikkat edin. Bu bakımdan, bu kod **malloc** üzerinden bellek ayrılamayan sistemlerde kullanılabilir (bu gibi durumlarda **backtrace_symbols** yerine **backtrace_symbols_fd** işlevi de kullanılabilir). Dönüş adreslerinin sayısı normalde çok fazla olmayacaktır. Nadiren diyelim ki 50'den fazla iç içelik içeren çok karmaşık yazılımlarda bile 200 girdi olabilir.

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>

/* Köken arama listesini edinip stdout'a basalım. */
void
print_trace (void)
{
  void *array[10];
  size_t size;
  char **strings;
  size_t i;

  size = backtrace (array, 10);
  strings = backtrace_symbols (array, size);

  printf ("%zd yığıt çerçevesi elde edildi.\n", size);

  for (i = 0; i < size; i++)
      printf ("%s\n", strings[i]);

  free (strings);
}

/* Köken arama listesini biraz daha ilginç hale getirmek
   için (iç içelik olsun diye) bir işlev. */
void
dummy_function (void)
{
  print_trace ();
}

int
main (void)
{
  dummy_function ();
```

```
  return 0;
}
```

A. Kütüphanedeki C Dili Oluşumları

C kütüphanesi tarafından gerçekleştirilen bazı oluşumların aslında C dilini kendisinde bulunması gerekir. Bu oluşumlar kütüphane kılavuzunun değil C Dili Kılavuzunun bir parçası olmalıydı, ancak henüz bir dil kılavuzumuz olmadığı için bu oluşumların belgelendirilmesi burada yapılmıştır.

A.1. Dahilî Kararlılığın Doğrudan Denetlenmesi

Bir yazılımı geliştirirken "imkansız" hatalara ve temel kabullerdeki çatışmalara karşı bazı stratejik yerlere denetimler yerleştirmek iyi olur. Bu çeşit denetimler örneğin, yazılımın farklı parçaları arasında, arayüzlerle hata ayıklama sorunlarını gidermeye yardımcı olur.

assert makrosu `assert.h` başlık dosyasında tanımlanmıştır ve yazılımda bir hata saptandığında bir ileti basarak yazılımın çıkması için kullanışlı bir yol sağlar.

Yazılımınızın artık hatalardan arındığı düşündüğünüzde, **assert** makrosu tarafından uygulanan hata denetimlerini, **NDEBUG** makrosunu tanımladıktan sonra yazılımınızı yeniden derleyerek iptal edebilirsiniz. Yani, bu denetimleri iptal etmek için yazılımınızın kaynak kodunu değiştirmeniz gerekmez.

Ancak bu kararlılık denetimlerinin iptal edilmesi, bu denetimler yazılımınızın çalışmasını kaydadeğer oranda yavaşlatmadıkça yapılmasa iyi olur. Yazılımın çalışmasına çok etki etmedikçe daha fazla hata denetimi yapılması daha iyidir. Başlangıç seviyesinde bir kullanıcı bir yazılımın çökmesi sırasında hiçbir uyarı görmezse herşeyi yanlış yaptığı sanısına kapılabilir.

`void` **assert**`(int` *ifade*`)` — makro

Yazılımın o noktasında *ifade* sıfırdan farklıysa yazılımcının kanaatini doğrular.

NDEBUG tanımlı değilse, **assert** *ifade*nin değerini sınamaya tabi tutar. Sınama sonucu yanlışsa (sıfır), **assert** aşağıdakine benzer bir hata iletisini *standart hataya* (sayfa: 237) (**stderr**) basarak *yazılımı sonlandırır* (sayfa: 683):

dosya`:`*satırno*`:` *işlev*`: ‘`*ifade*`’ savı başarısız oldu.`

Dosya ismi ve satır numarası **__FILE__** ve **__LINE__** C önişlemci makrolarıyla saptanır ve **assert** çağrısının yapıldığı yeri belirtir. GNU C derleyicisi kullanılırken, işlevin yani **assert** çağrısının işlev ismi **__PRETTY_FUNCTION__** yerleşik değişkeninden alınır; daha eski derleyicilerde işlev ismi ve onu izleyen iki nokta üstüste yoktur.

NDEBUG önişlemci makrosu `assert.h` dosyasının içerildiği satırdan önce tanımlanmışsa, **assert** makrosu mutlak olarak hiçbir şey yapmamak üzere tanımlanmış olur.

Uyarı

NDEBUG etkili ise argüman olarak verilen *ifade* değerlendirilmez. Bu yüzden, yan etkileri olan ifadeleri argüman olarak kullanmayın. Örneğin **assert (++i > 0);** gibi bir deyimi asla kullanmayın, çünkü **NDEBUG** tanımlıysa **i** hiçbir zaman arttırılmayacaktır.

Bazan bir işletim sistemi işlevinden dönen bir hatanın denetlenmesinin istenebileceği "imkansız" durumlar olabilir. Bu durumda sadece yazılımın nerede çöktüğünü değil, neden çöktüğünün de bildirilmesi faydalıdır. **assert_perror** makrosu bunu kolayca yapar.

`void` **assert_perror**`(int` *hatanum*`)` — makro

hatanum değerinin sıfır olduğunu doğruladığı durumda **assert** işlevine benzer.

NDEBUG tanımlı değilse, **assert_perror** *hatanum* değerine bakar. Değer sıfırdan farklıysa, **assert_perror** aşağıdaki iletiye benzer bir hata iletisini standart hataya basarak yazılımı sonlandırır:

dosya:*satırno*: *işlev*: *hata iletisi*

Dosya ismi, satır numarası ve işlev ismi **assert** işlevindeki gibi elde edilir. *hata iletisi* ise, **strerror** (*hatanum*) işlevinin sonucudur. Bkz. *Hata İletileri* (sayfa: 41).

assert işlevinde olduğu gibi, **NDEBUG** önişlemci makrosu `assert.h` dosyasının içerildiği satırdan önce tanımlanmışsa, **assert_perror** makrosu mutlak olarak hiçbir şey yapmamak üzere tanımlanmış olur. Bu durumda argüman olarak verilen *hatanum* değerlendirilmez. Yani, *hatanum* bir yan etki barındırmamalıdır. En iyisi *hatanum* değerinin basit bir değişken başvurusu olmasıdır; doğrudan **errno** vermek daha iyi olacaktır.

Bu makro bir GNU oluşumudur.

Kullanım Bilgisi:

assert oluşumu *dahili kararlılığın saptanması* için tasarlanmıştır. Yazılımın *kullanıcı* tarafından yanlış kullanımının ya da geçersiz girdilerinin bildirilmesi için kullanışlı değildir.

assert ve **assert_perror** tarafından basılan tanı iletilerinden elde edilen bilgiler size yani yazılımcıya yöneliktir. Bir yazılım hatası ile ilgilidir. Kullanıcıya girdilerinin neden geçersiz olduğunu ya da neden bir komutun gerçekleştirilemediğini bildirmek için değildir. Neden yazılımınız, kullanıcının yaptığı bir hatadan dolayı **assert** işlevinin yaptığı şekilde (bir yazılım hatası vererek) sonlansın ki. Bkz. *Çıkış Durumu* (sayfa: 682).

Yazılımınızdaki bir yazılım hatasının sonucu *olmayan* sorunlarda hata iletilerinin basılması ile ilgili bilgileri *Hata İletileri* (sayfa: 41) bölümünde bulabilirsiniz.

A.2. Değişkin İşlevler

ISO C, değişen sayıda argüman alabilecek bir işlevin bildirilebilmesi için bir sözdizimi tanımlamıştır. Buna rağmen, dilin kendisi bu tür işlevlerin gerekli olmayan argümanlarına erişim için bir mekanizma sağlamaz; bu nedenle, `stdarg.h`[B1157] başlık dosyasındaki makrolarla tanımlanmış olan değişken argümanları kullanacaksınız.

Bu tür işlevler, ***değişken argümanlı işlevler*** [varargs functions] ya da ***değişkin işlevler*** [variadic functions] adını alır.

Bu bölümde değişkin işlevlerin nasıl bildirildikleri, nasıl yazıldıkları ve nasıl çağrıldıkları anlatılmıştır.

Uyumluluk Bilgisi:

Birçok eski C gerçeklemesi, `stdarg.h` kullanılarak değişken sayıda argümanla işlevleri tanımlama mekanizmasına benzeyen ama uyumlu olmayan bir mekanizma sağlar.

A.2.1. Değişkin İşlevler Neden Kullanılır

Sıradan C işlevleri sabit sayıda argüman alır. Bir işlevi tanımlarken her argüman için veri türünü de belirtirsiniz. Her işlev çağrısında istenen sayıda argüman verirken argümanlarını da gerekirse tür dönüşümü yaparak istenen türde sağlarsınız. Bu noktada, örneğin, `foo` işlevi `int foo (int, char *);` deyimi ile bildirilmişse, onu iki argümanla; bir sayı ve bir dizge göstericisi ile çağırmalısınız.

Fakat bazı işlevler sınırsız sayıda argümanı anlamlı olarak kabul ederek işlemler uygulayabilir.

Bazı durumlarda bir işlev çok sayıda değeri bir blok olarak işleyerek elde edebilir. Örneğin, varsayalım ki, bir işlev belirli sayıda değeri tutmak için **malloc** ile bir tek boyutlu dizi ayırsın. Dizinin uzunluğuna bağlı sayıda değer üzerinde işlem yapılabileceğinden değişen sayıda argümanlı oluşumlar olmaksızın, her olası dizi uzunluğu için ayrı bir işlev tanımlamak zorunda kalırsınız.

Kütüphane işlevi **printf** (Bkz. *Biçimli Çıktı* (sayfa: 255)) değişken argümanlı başka bir işlev sınıfından bir örnek olarak verilebilir. Bu işlev bir biçim şablonu dizgesi altında (değişken sayıda ve türdeki) argümanlarını basar.

Bunlar, argüman sayısı çağrı sırasında seçilen bir *değişkin* işlev tanımlamak için iyi sebeplerdir.

open gibi bazı işlevler sabit sayıda argüman almasına rağmen arasıra son birkaçını yoksayar. ISO C'ye kesin bağlılık durumunda bu işlevlerin değişkin olarak tanımlanması gerekir; uygulamada ise, GNU C derleyicisi ve diğer bir çok C derleyicisi böyle bir işlevi sabit sayıda argümanla tanımlamayı ve sadece *bildirme* işlemini yaparken işlevi değişkin olarak bildirerek (ya da argümanlarının tümünü bildirmeyerek) yapmayı tercih eder.

A.2.2. Değişkin İşlevler Nasıl Tanımlanır ve Kullanılır

Bir değişkin işlevin tanımlanması ve kullanılması üç adımdan oluşur:

Tanım
: Bir işlevin değişkin olarak tanımlanması, argüman listesinde bir üçlü nokta (**...**) kullanarak ve değişken sayıda argümana erişmek için özel makrolar kullanılarak yapılır. Bkz. *Argüman değerlerinin Alınması* (sayfa: 816).

Bildirim
: Bir işlevin değişkin olarak bildirilmesi, onu çağıran tüm dosyalarda bir üçlü nokta (**...**) içeren bir prototip kullanarak yapılır. Bkz. *Değişen Sayıda Argüman için Sözdizimi* (sayfa: 815).

Çağrı
: İşlev çağrısı sabit agümanlara ek olarak değişken sayıdaki argümanları yazarak yapılır. Bkz. *Değişkin İşlevlerin Çağrılması* (sayfa: 817).

A.2.2.1. Değişen Sayıda Argüman için Sözdizimi

Değişken sayıda argüman kabul eden bir işlev şimdi açıklanacağı gibi bir prototiple bildirilmelidir. Önce sabit argümanları yazacaksınız, ardından da ek argümanlar olduğunu belirtmek üzere bir üçlü nokta (**...**) gelecek. ISO C sözdizimi üçlü noktadan önce en az bir sabit argüman gerektirir. Örnek:

```
int
func (const char *a, int b, …)
{
  …
}
```

Bu örnekte, **int** türünde bir dönüş değeri olan ve biri **const char *** türünde diğeri **int** türünde iki argüman gerektiren **func** işlevi tanımlanmıştır. Ayrıca işlevin gerekli olan iki argümandan başka belirsiz sayıda anonim argümanları da vardır.

Uyumluluk Bilgisi:

Bazı C derleyicileri için son gerekli argümanın işlev tanımında **register** olarak bildirilmemesi gerekir. Bundan başka, bu argümanın türü *kendinden terfili* olmalıdır: Şöyleki, öntanımlı terfiler onun türünü değiştirmemelidir. Bu kurallar dizi ve işlev türleri ile **float**, **char** (signed ya da değil) ve **short int** (signed ya da değil) türleri dışarda tutar. Bu aslında bir ISO C gerekliliğidir.

A.2.2.2. Argüman değerlerinin Alınması

Gerekli olan argümanların isimleri vardır ve bu isimleri kullanarak onları değerlerine erişirsiniz. Ama isteğe bağlı argümanların isimleri yoktur, çünkü onlar bir üçlü nokta ile ifade edilmiştir. O halde bu argümanlara nasıl erişilecek?

Onlara erişmenin tek yolu onlara yazıldıkları sırayla erişmektir. Bunun için `stdarg.h` başlık dosyasındaki özel makroları aşağıdaki üç adımlık işlemlerle kullanmalısınız:

1. **`va_start`** kullanarak **`va_list`** türünde bir argüman gösterici değişkenini ilklendirin. Argüman gösterici, ilklendirildiğinde ilk isteğe bağlı argümanı gösterecektir.

2. İsteğe bağlı argümanlara **`va_arg`** çağırarak erişirsiniz. İlk **`va_arg`** çağrısı ilk isteğe bağlı argümanı, ikinci çağrı ikincisini, böyle gider.

   Bu çağrı işlemini kalan isteğe bağlı argümanları yoksayacağınız yere kadar sürdürebilirsiniz. Bir işlevin argümanlarından daha azına erişmek bir sorun çıkarmaz ama daha fazla sayıda argümana erişmeye çalışırsanız bozuk değerler alırsınız.

3. Argüman gösterici ile işiniz bittiğinde bunu **`va_end`** çağrısıyla belirtin.

   (Uygulamada, birçok C derleyicisi **`va_end`** çağrısında hiçbir şey yapmaz. Bu GNU C derleyicisi için de geçerlidir. Ancak, yazılımınızın bir gün bu çağrıyı gerektiren bir derleyici ile derlenebileceğini gözönüne alarak yine de **`va_end`** çağrısını yaparsanız iyi olur.)

`va_start`,**`va_arg`** ve **`va_end`** tanımları için *Argümana Erişim Makroları* (sayfa: 817) bölümüne bakınız.

1 den 3 e kadar adımlar isteğe bağlı argümanları kabul eden işlevin içinde uygulanmalıdır. Buna rağmen **`va_list`** değişkenini bir argüman olarak başka bir işleve aktararak 2. adımı ya da tamamını burada uygulayabilirsiniz.

Bu üç adımlık işlemi tek bir işlevi defalarca çağırarak da uygulayabilirsiniz. İsteğe bağlı argümanları yoksaymak istediğinizde ise bu adımları sıfır kere uygulayabilirsiniz.

İsterseniz, birden fazla argüman gösterici değişkeniniz olabilir ve bu değişkenlerin her birini istediğiniz zaman **`va_start`** çağrılarıyla ayrı ayrı ilklendirebilirsiniz. Her argüman gösterici ile istediğiniz kadar isteğe bağlı argümanı alabilirsiniz. Her argüman gösterici değişkeni daima aynı argüman kümesine ama kendi alanında sahip olacaktır.

Taşınabilirlik Bilgisi:

Bazı derleyicilerle, bir argüman gösterici değerini bir alt işleve aktardıktan sonra, alt işlev döndüğünde aynı argüman gösterici değerini kullanımda tutmamalısınız. Tam taşınabilirlik için, onu **`va_end`**'e aktarmalısınız. Bu aslında bir ISO C gerekliliği olmakla birlikte birçok ANSI C derleyicisi ile de sorunsuz çalışır.

A.2.2.3. Aktarılan Argümanların Sayısı

Bir işleve aktarılan isteğe bağlı argümanların sayısını ve türünü saptamak için genel bir yol yoktur. Yani işlevi tasarlayan her kim ise, genellikle çağrıcı tarafından kullanılmak üzere argümanların sayısının ve türünün belirtileceği bir yol da tasarlamalıdır. Bu kimse siz olduğunuza göre her değişkin işlev için bir çağrı yöntemi tespit edip, çağrıları yazarken bunları uygulamalısınız.

Çağrı yöntemlerinden biri, isteğe bağlı argümanların sayısını sabit argümanlardan birinde belirtmektir. Bu yöntem sadece tüm isteğe bağlı argümanlar aynı türde ise çalışır.

Bir diğer çağrı yöntemi ise, sabit argümanlardan birinin bir isteğe bağlı argümanın sağlayabileceği her olası amaç için bir bit olmak üzere bir bit maskesi içermesidir. Bu bitleri önceden tanımlanmış bir sırayla sınayarak; eğer bit bir ise sonraki argümanın değerini alırsınız, değilse bir öntanımlı değer kullanırsınız.

Bir sabit argüman hem argüman sayısını hem de türünü belirten bir kalıp olarak kullanılabilir. **printf** işlevinin biçim dizgesi argümanı buna bir örnektir (Bkz. *Biçimli Çıktı İşlevleri* (sayfa: 263)).

Diğer bir olasılık da, son isteğe bağlı argüman olarak bir "son belirten" değer aktarmaktır. Örneğin isteğe bağlı argümanları göstericilerden oluşan bir işleve sonuncu argüman olarak bir boş gösterici verilebilir. Örneğin, **execl** işlevi bu yöntemi kullanır. Bkz. *Bir Dosyanın Çalıştırılması* (sayfa: 688).

A.2.2.4. Değişkin İşlevlerin Çağrılması

Bir değişkin işlev çağrısına özel hiçbir şey yoktur. Parantez içine önce gerekli sonra da isteğe bağlı argümanları virgüllerle ayırarak yazarsınız. Ama önce işlevi bir prototiple bildirmeniz gerekir, böylece argüman değerlerinin nasıl dönüştürüleceğini bilirsiniz.

Prensip olarak, değişkin olarak *tanımlanan işlevler* çağrılmadan önce bir *işlev prototipi* (sayfa: 815) kullanarak değişkin olarak *bildirilmelidir*. Bazı C derleyicileri, işlevin aldığı sabit ve değişken sayıdaki argümanlara bağlı olarak bir işleve aktarılacak aynı argüman değerleri kümesi için farklı çağrı yöntemleri kullandığı için bu böyledir.

Uygulamada, GNU C derleyici argümanların gerekli mi, isteğe bağlı mı olduğuna bakmaksızın bir verilmiş argüman türleri kümesini hep aynı yolla aktarır. Yani argüman türleri kendiden terfili olduğu sürece onların bildirilmesini rahatça ihmal edebilirsiniz. Genellikle, değişkin işlevlerin argüman türlerini ve hatta tüm işlevleri bildirmek iyi bir fikirdir. Ancak çok kullanışlı bir kaç işlev vardır ki değişkin olarak bildirilmez; örneğin, **open** ve **printf**.

İşlev prototipinde isteğe bağlı argümanların türleri belirtilmediğinden, bir değişkin işlev çağrısında isteğe bağlı argüman değerleri üzerinde *öntanımlı argüman terfileri* uygulanır. Yani, nesne türleri **char** veya **short int** (signed ya da değil) ise ya **int** ya da **unsigned int** türüne, nesne türü **float** ise **double** türüne terfi ettirilir. Böylece örneğin, çağrı sırasına **char** türünde belirtilen bir argüman, **int** türüne terfi ettirilerek, işlev argümana **va_arg (***arg_gstr***, int)** ile erişebilir.

Gerekli argümanların dönüşümleri ise işlev prototipi tarafından genel bir yolla denetlenir: argüman işlev prototipinde bildirilen türde değilse, prototipte bildirilen türe dönüştürülür.

A.2.2.5. Argümana Erişim Makroları

Burada isteğe bağlı argümanlara erişmek için kullanılan makrolar açıklanmıştır. Bu makrolar `stdarg.h` başlık dosyasında tanımlıdır.

va_list — veri türü

va_list türü argüman gösterici değişkeni için kullanılır.

`void` **va_start**`(va_list` *arglist–gstr*`,` *gerekli–son–arg*`)` — makro

Bu makro, kullanıldığı işlevin ilk isteğe bağlı argümanını gösterecek olan *arglist_gstr* argüman gösterici değişkenini ilklendirir. *gerekli–son–arg* işlevdeki gerekli son argüman olmalıdır.

Bu makronun bir alternatifi olarak `varargs.h` başlık dosyasında tanımlanmış olan **va_start** makrosu için *Eski Moda Değişkin İşlevler* (sayfa: 819) bölümüne bakınız.

tür **va_arg**`(va_list` *arglist–gstr*`,` *tür*`)` — makro

Bu makro sonraki isteğe bağlı argümanın değeri ile döner ve *arglist_gstr* değişkeninin değerini sonraki argümanı gösterecek şekilde değiştirir. Böylece her **va_arg** kullanımında sırayla bir isteğe bağlı argümanın değeri alınır.

va_arg tarafından döndürülen değerin türü çağrı sırasında *tür* ile belirtilir. *tür* argümanın türü ile eşleşen kendinden terfili bir tür olmalıdır (**char**, **short int** veya **float** değil).

`void` **va_end**`(va_list` *arglist–gstr*`)` makro

Bu makro *arglist_gstr* kullanımını sonlandırır. Bir **va_end** çağrısından sonraki **va_arg** çağrıları bu *arg_gstr* ile çalışamaz. Aynı *arglist_gstr* argümanı için bir **va_start** çağrısı yapmadan önce bir **va_end** çağrısı yapılmalıdır.

GNU C kütüphanesinde, **va_end** hiçbir şey yapmaz ve taşınabilirlik sebebi dışında kulanmanız gerekmez.

Bazan parametre listesini defalarca çözümlemeniz gerekir ya da argümanlardan birinin listedeki konumunu hatırlamak istersiniz. Bu durumda, o anki argüman değerinin bir kopyasını yaparsınız. Ancak **va_list** şeffaf bir tür değildir ve **va_list** türünden bir değişkenin değeri başka bir değişkene atanamaz (bazı türleri sadece makrolar oluşturabilir).

`void` **__va_copy**`(va_list` *hedef*`,` makro
`va_list` *kaynak*`)`

Bu makro, **va_list** türünden nesnelerin bir bütünleyen tür olmasa bile kopyalanmasını mümkün kılar. *hedef* argüman göstericisi, *kaynak* göstericisi ile aynı argümanı göstermek üzere ilklendirilir.

Bu makro bir GNU oluşumudur ve ISO C standardının gelecek güncellemesinde ayrıca kullanılabilir olacağı umulmaktadır.

__va_copy makrosunu kullanmak isterseniz, bu makronun kullanılabilir olmayabileceği durumlara karşı hazırlıklı olmalısınız. Basit atamanın geçersiz olduğu mimarilerde **__va_copy** makrosunun bulunmakta olduğunu umarak yazılımınızda bu makroyu daima şöyle yazmalısınız:

```
{
  va_list ap, save;
  …
#ifdef __va_copy
  __va_copy (save, ap);
#else
  save = ap;
#endif
  …
}
```

A.2.3. Bir Değişkin İşlev Örneği

Burada değişken sayıda argüman kabul eden bir işlevle ilgili tam bir örneğe yer verilmiştir. İşlevin ilk argümanı diğer argümanların sayısını içermektedir. İşlev anlamsız olsa da değişken argüman oluşumunun kullanımı hakkında fikir vermek için yeterlidir.

```
#include <stdarg.h>
#include <stdio.h>

int
topla (int miktar,...)
{
```

```
  va_list ag;
  int i, toplam;

  va_start (ag, miktar);          /* Argüman listesi ilklendiriliyor. */

  toplam = 0;
  for (i = 0; i < miktar; i++)
    toplam += va_arg (ag, int);    /* Sonraki argümanın değeri alınıyor. */

  va_end (ag);                   /* Temizlik. */
  return toplam;
}

int
main (void)
{
  /* Bu çağrı 16 basar. */
  printf ("%d\n", topla (3, 5, 5, 6));

  /* Bu çağrı 55 basar. */
  printf ("%d\n", topla (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

  return 0;
}
```

A.2.3.1. Eski Moda Değişkin İşlevler

ISO C öncesi çağlarda yazılımcılar değişkin işlevleri yazmak için az çok farklı bir oluşum kullandılar. GNU C derleyicisi hala onu desteklemektedir; şu anda, ISO C için destek hala evrensel olmadığından, ISO C oluşumundan daha taşınabilir durumdadır. Bu eski moda değişkin işlevler oluşumu `varargs.h` başlık dosyasında tanımlanmıştır.

`varargs.h` kullanımı `stdarg.h` kullanımı ile hemen hemen aynıdır. (Bkz. *Değişkin İşlevlerin Çağrılması* (sayfa: 817)) Değişkin işlevlerin çağrılması açısından bir fark yoktur. Tek fark onların nasıl tanımlandığı ile ilgilidir. Herşeyden önce, eski moda prototipsiz sözdizimini şöyle kullanmalısınız:

```
tree
build (va_alist)
     va_dcl
{
```

İkinci olarak, **`va_start`**'ı tek bir argüman ile vermelisiniz, bunun gibi:

```
  va_list p;
  va_start (p);
```

Eski moda değişkin işlevleri tanımlamak için kullanılan özel makrolar şunlardır:

`va_alist` makro

Bu makro bir değişkin işlevdeki gerekli argüman isimlerinin listesi için kullanılır.

`va_dcl` makro

Bu makro bir değişkin işlev için dolaylı argüman ya da argümanları bildirir.

`void` **`va_start`**`(va_list` *arglist–gstr*`)` makro

Bu makro `varargs.h` dosyasında tanımlanmış olarak, kullanıldığı işlevin ilk argümanını gösteren *arg_gstr* argüman gösterici değişkenini ilklendirir.

Diğer argüman makroları, **va_arg** ve **va_end** için `varargs.h` ile `stdarg.h` kullanımları arasında fark yoktur; ayrıntılar için *Argümana Erişim Makroları* (sayfa: 817) bölümüne bakınız.

`varargs.h` ile `stdarg.h` dosyaları aynı derleme altında birlikte kullanılamazlar, çünkü **va_start** tanımları isimleri dışında birbiriyle aynı değildir.

A.3. Boş Gösterici Sabiti

Boş gösterici sabiti herhangi bir gerçek nesneyi göstermemeyi garanti eder. **void *** türünde olduğundan onu herhangi bir gösterici değişkenine atayabilirsiniz. Bir boş gösterici sabitini yazmak için önerilen yol onu **NULL** ile belirtmektir.

`void *` **NULL** — makro

Bu bir boş gösterici sabitidir.

Ayrıca, bir boş gösterici sabiti olarak **0** veya **(void *)0** kullanabilirsiniz, ancak **NULL** kullanımı daha temizdir çünkü sabitin amacını daha net ortaya koyar.

Bir boş gösterici sabitini bir işlev argümanı olarak kullanırsanız, tam taşınabilirlik açısından işlevin bir prototip bildirimi olduğundan emin olmalısınız. Aksi takdirde, hedef makina iki farklı gösterici tanımına sahipse derleyici bu argüman için hangi tanımı kullanacağını bilemeyecektir. Bu sorundan kaçınmak için açıkça bir tür dönüşümü ile sabiti doğru gösterici türüne ayarlamalısınız. Fakat biz bunu yapmak yerine çağırdığınız işlev için bir prototip eklemenizi öneririz.

A.4. Önemli Veri Türleri

İki gösterici ile yapılan çıkartma işleminin sonucu C'de daima bir tamsayıdır, ancak doğru veri türü C derleyicisinden C derleyicisine değişir. Benzer şekilde, **sizeof**'un sonucunun veri türü de derleyiciler arasında değişiklik gösterir. ISO bu iki tür için standart isimler tanımlar, böylece bunlar kullanılarak taşınabilirlik sorunları çözülür. Bunlar `stddef.h` başlık dosyasında tanımlanmıştır.

ptrdiff_t — veri türü

Bu göstericiler arasındaki çıkartma işleminin sonucunun işaretli tamsayı türüdür. Örneğin, **char *p1, *p2;** bildirimiyle **p2 - p1** ifadesi **ptrdiff_t** türündedir. Bu şüphesiz standart işaretli tamsayı türlerinden biri (**short int**, **int** veya **long int**) olacaktır, ancak sadece bu amaca yönelik olarak bir standart dışı tür de mevcut olabilir.

size_t — veri türü

Bu bir işaretsiz tamsayı türdür ve nesnelerin boyutları için kullanılır. **sizeof** işlecinin sonucu bu türdendir ve **malloc** (sayfa: 50), **memcpy** (sayfa: 94) gibi işlevler argümanlarında nesne boyutlarını bu türde kabul ederler.

Kullanım Bilgisi:

size_t, bir nesnenin boyutunu tutan herhangi bir değişken ya da argüman bildiriminde önerilen yoldur.

GNU sisteminde **size_t** ya **unsigned int** ya da **unsigned long int** türüne eşdeğerdir. Bu iki tür GNU sisteminde eşanlamlıdır ve bir çok kullanım amacına yönelik olarak biri diğerinin yerine kullanılabilir. Yine de, onlar farklı veri türleridir ve bazı bağlamlarda veri türü olarak bir fark oluşturabilir.

Örneğin, bir işlev prototipinde bir işlev argümanının türünü belirtirseniz, bunlardan diğerinin kullanımı bir fark oluşturabilir. Sistem başlık dosyasında **malloc** işlevi **size_t** türünden bir argümanla bildirilmişse ve siz onu **unsigned int** türünden bir argümanla bildirirseniz, sisteminizde **size_t** türü **unsigned long int** türüne karşılıksa derleme sırasında hata alırsınız.

Uyumluluk Bilgisi:

ISO C'den önceki C gerçeklemeleri genellikle nesne boyutları için **unsigned int** ve gösterici çıkartma ifadeleri için **int** türünü kullandılar, **size_t** ya da **ptrdiff_t** tanımlamak gereği duymadılar. Unix sistemleri **size_t** türünü **sys/types.h** dosyasında tanımladı ama bu tanım bir işaretli türdü.

A.5. Veri Türü Ölçüleri

Çoğu zaman, yazılımınızda her nesne için uygun bir C veri türü seçersiniz ve onun nasıl gösterildiği veya kaç bit kullandığıyla ilgilenmezsiniz. Böyle bir bilgiye ihtiyaç duyarsanız C dilinin kendisi bu bilgiyi sağlamaz. `limits.h` ve `float.h` başlık dosyaları size bu bilgiyi tüm ayrıntıları ile veren makrolar içerir.

A.5.1. Bir Tamsayı Veri Türünün Genişliğinin Hesaplanması

Çok bilinen sebeplerle bir yazılım bir tamsayı türünde kaç bit bulunduğunu bilmeye ihtiyaç duyabilir. Örneğin, **long int** türünde bir diziyi bir bit vektörü olarak kullanıyorsanız, bitlere *n* indisi ile aşağıdaki gibi erişebilirsiniz:

```
vector[n / LONGBITS] & (1 << (n % LONGBITS))
```

Burada **LONGBITS**, bir **long int** içindeki bitlerin sayısı olarak tanımlanmış olmalıdır.

C dilinde bir tamsayı veri türündeki bitlerin sayısını verecek bir işleç yoktur. Fakat onu, `limits.h` başlık dosyasında tanımlanmış olan **CHAR_BIT** makrosuyla hesaplayabilirsiniz.

CHAR_BIT
Bir **char** içindeki bitlerin sayısıdır. Birçok sistemde değeri sekizdir ve bu değer **int** türündendir.

tür ile belirtilecek herhangi bir veri türünün bitlerinin sayısını şöyle hesaplayabilirsiniz:

```
sizeof (tür) * CHAR_BIT
```

A.5.2. Bir Tamsayı Türün Aralığı

Varsayalım ki, sıfırdan bir milyona kadar bir aralıktaki tamsayı değerleri saklamak istiyorsunuz. Kullanabileceğiniz en küçük tür hangisidir? Bunun için genel bir kural yoktur; C derleyicisine ve hedef makinaya bağlıdır. Hangi tür ile çalışacağınızı bulmak için `limits.h` başlık dosyasında tanımlanmış olan **MIN** ve **MAX** makrolarını kullanabilirsiniz.

Her işaretli tamsayı türü için tutabileceği en küçük ve en büyük değerleri veren bir çift makro vardır. Her işaretsiz tamsayı türü içinde böyle makrolar vardır; en büyük değer için, en küçük değer yani sıfır için.

Bu makroların değerlerinin hepsi tamsayı sabit ifadeleridir. **char** ve **short int** için **MAX** ve **MIN** makroları **int** türünden değerlere sahiptir. Diğer türlerin **MAX** ve **MIN** makrolarının değerleri de makro tarafından açıklanmış aynı türde değerlerdir. Örneğin, **ULONG_MAX** makrosunun değeri **unsigned long int** türündendir.

SCHAR_MIN
Bir **signed char** tarafından tutulabilen en küçük değerdir.

SCHAR_MAX

UCHAR_MAX

Sırasıyla **signed char** ve **unsigned char** tarafından tutulabilen en büyük değerlerdir.

CHAR_MIN

Bir **char** tarafından tutulabilen en küçük değerdir. **char** işaretli ise **SCHAR_MIN**'e eşittir, yoksa sıfırdır.

CHAR_MAX

Bir **char** tarafından tutulabilen en büyük değerdir. **char** işaretli ise **SCHAR_MAX**'a eşittir, yoksa **UCHAR_MAX**'a eşittir.

SHRT_MIN

Bir **signed short int** tarafından tutulabilen en küçük değerdir. GNU C kütüphanesinin çalıştığı çoğu makinada **short** tamsayılar 16 bit genişliktedir.

SHRT_MAX

USHRT_MAX

Sırasıyla **signed short int** ve **unsigned short int** tarafından tutulabilen en büyük değerlerdir.

INT_MIN

Bir **signed int** tarafından tutulabilen en küçük değerdir. GNU C kütüphanesinin çalıştığı çoğu makinada **int** tamsayılar 32 bit genişliktedir.

INT_MAX

UINT_MAX

Sırasıyla **signed int** ve **unsigned int** tarafından tutulabilen en büyük değerlerdir.

LONG_MIN

Bir **signed long int** tarafından tutulabilen en küçük değerdir. GNU C kütüphanesinin çalıştığı çoğu makinada **long** tamsayılar **int** ile aynı olarak 32 bit genişliktedir.

LONG_MAX

ULONG_MAX

Sırasıyla **signed long int** ve **unsigned long int** tarafından tutulabilen en büyük değerlerdir.

LONG_LONG_MIN

Bir **signed long long int** tarafından tutulabilen en küçük değerdir. GNU C kütüphanesinin çalıştığı çoğu makinada **long long** tamsayılar 64 bit genişliktedir.

LONG_LONG_MAX

ULONG_LONG_MAX

Sırasıyla **signed long long int** ve **unsigned long long int** tarafından tutulabilen en büyük değerlerdir.

WCHAR_MAX

Bir **wchar_t** tarafından tutulabilen en büyük değerdir. Bkz. *Genişletilmiş Karakterlere Giriş* (sayfa: 126).

`limits.h` başlık dosyası ayrıca işletim sistemi ve dosya sistemi sınırlarını belirleyen bazı sabitler de bulunmaktadır. Bu sabitler *Sistem Yapılandırma Parametreleri* (sayfa: 784) bölümünde açıklanmıştır.

A.5.3. Gerçek Sayı Türü Makroları

Gerçek sayılara özel gösterim makinadan makinaya değişir. Çünkü gerçek sayılar dahili olarak yaklaşık niceliklerle gösterilir. Gerçek sayı verilerle çalışan algoritmalar çoğunlukla makinanın doğru gerçek sayı gösterim ayrıntılarına dikkat etmeyi gerektirir.

C kütüphanesindeki bazı işlevlerin kendileri bu bilgiye gereksinim duyar; örneğin, gerçek sayıları okumak ve basmak (Bkz. *Akımlar Üzerinde Giriş/Çıkış* (sayfa: 236)) için ve trigonometrik ve gerçel işlevlerin (Bkz. *Matematik* (sayfa: 475)) hesaplanmasında kullanılan algoritmalar yuvarlama hatalarından ve hassasiyet kayıplarından kaçınmak için bunu kullanır. Sayısal analiz teknikleri gerçekleştiren bazı kullanıcı yazılımları da çoğunlukla hata sınırlarının küçültmesi ya da hesaplanması sırasında bu bilgiye ihtiyaç duyar.

`float.h` başlık dosyası makinanızda kullanılan biçimi açıklar.

A.5.3.1. Gerçek Sayı Gösterimi ile İlgili Kavramlar

Bu bölümde gerçek sayı gösterimleri ile ilgili terminolojiye değinilmiştir.

Büyük ihtimalle gerçek sayılar için üstel gösterim veya bilimsel terimler olarak bu kavramların çoğuna zaten aşinasınızdır. Örneğin `123456.0` sayısı üstel olarak `1.23456e+05` biçiminde, `1.23456` sayısı ile `10` üssü `5`'in çarpımı olarak gösterilir.

Daha biçimsel olarak, gerçek sayıların bit gösterimi aşağıdaki terimlerle karakterize edilebilir:

- ***İşaret biti*** ya `-1` ya da `1`'dir.
- Üs alma için ***taban***, `1`'den büyük bir tamsayıdır. Belirli bir gösterim için bu bir sabittir.
- ***Üstel kısım*** tabanın kendisiyle kaç defa çarpılacağını gösterir. Bir belirli gösterim için üs değerinin alt ve üst sınırları birer sabittir.

  Bazan gerçek sayının bit gösteriminde üssü ifade eden kısım bir sabit eklenmesiyle daima işaretsiz bir nicelik yapılır. Bu sadece bit alanlarını ayırıp gerçek sayıyı kendiniz oluşturmak isterseniz önemli olur, ancak GNU kütüphanesinde böyle bir işlem için destek yoktur.
- ***Ondalık kısım*** her gerçek sayının bir parçası olan bir işaretsiz tamsayıdır.
- Ondalık kısmın ***hassasiyeti***. Eğer bit gösteriminde taban b ise hassasiyet, ondalık kısmın taban–b sayıda basamağıdır. Bu belirli bir gösterim için bir sabittir.

  Birçok gerçek sayı gösterimi ondalık kısım içinde bir örtük ***gizli bit*** içerir. Bu ondalık kısım içinde olduğu varsayılan bir bittir ancak bellekte saklanmaz, çünkü bir normalleştirilmiş sayı içinde daima 1 dir. Hassasiyet ile ilgili kısım kendi içinde çok sayıda gizli bit içerebilir.

  Tekrar belirtelim, GNU kütüphanesi gerçek sayıların düşük seviye (ikilik tabandaki bit gösterimi) gösterimleri ile ilgili oluşumlara destek sağlamamaktadır.

Bir gerçek sayının bit gösterimindeki ondalık kısım dolaylı olarak paydası üstel taban hassasiyetteki bir kesri ifade eder. Bu ondalık kısım ile gösterilebilecek en büyük sayı bu paydadan bir eksiği olduğundan kesrin değeri daima birden küçüktür. Bu bit gösteriminin matematiksel değeri bu kesir ile işaret ve üstel tabanın çarpımıdır.

Bilgi

Ç.N. – Özgün metinde burada anlatılan biraz karışık olmuş. Kafa karıştırmamak için çevirmedim. Basitçe ifade etmek gerekirse 32 bitlik gerçek sayı gösterimiyle ifade edilebilecek en küçük değer 2^{-149} olsa da bu sayı **float** türü için 2^{-126} olarak normalleştirilmiştir. En azından GNU C kütüphanesinde bu böyle.

A.5.3.2. Gerçek Sayılar ile İlgili Makrolar

Bu makroların tanımlarını `float.h` başlık dosyasında bulabilirsiniz.

FLT_ ile başlayan makro isimleri **float** türü ile, **DBL_** ile başlayanlar **double** türü ile ve **LDBL_** ile başlayanlar da **long double** türü ile ilgilidir. (GCC hedef makinada bir veri türü olarak **long double** türünü desteklemezse, **LDBL_** ile başlayan sabitler **double** türüne karşılık olan sabitlere eşitlenir.)

Bu makrolardan sadece **FLT_RADIX** bir sabit ifadesi olarak garantilidir. Burada listelenmiş diğer sabitler **#if** önişlemci komutu ya da durağan dizilerin boyutları gibi yerlerde sabit ifadesi olarak güvenilir olamaz.

ISO C standardı bu parametrelerin en küçük ve en büyük değerlerini belirtse de, GNU C gerçeklemesi hedef makinada desteklenen gerçek sayı gösterimlerini açıklayan değerleri kullanır. Yani GNU C prensip olarak ISO C gereksinimlerini hedef makinanın yapabildiği kadarıyla karşılar. Uygulamada tüm makinalarda bu destekler zaten vardır.

FLT_ROUNDS
Bu değer gerçek sayı toplamasında yuvarlama kipini belirler. Standart yuvarlama kiplerinin değerleri:

-1
Bu kip belirlenebilir değildir.

0
Sıfıra yuvarlar.

1
En yakın sayıya yuvarlar.

2
Pozitif sonsuza yuvarlar.

3
Negatif sonsuza yuvarlar.

Diğer değerler, eğer varsa, makina bağımlı standart dışı yuvarlama kiplerini belirtir.

Gerçek sayılar için IEEE standardı gereğince çoğu makinada **1** değeri kullanılır.

Aşağıdaki tabloda **FLT_ROUNDS** sabitinin olası değerleri ile yuvarlamanın nasıl yapıldığı gösterilmiştir. Yuvarlama IEEE tek hassasiyetli gerçek sayılar standardına uygun olarak yapılmıştır.

```
                0       1             2             3
 1.00000003    1.0     1.0           1.00000012    1.0
 1.00000007    1.0     1.00000012    1.00000012    1.0
-1.00000003   -1.0    -1.0          -1.0          -1.00000012
-1.00000007   -1.0    -1.00000012   -1.0          -1.00000012
```

FLT_RADIX
Bit gösteriminde üstel kısmın tabanına karşılık gelen değerdir. Bu bölümde açıklanan diğer makroların aksine bir sabit ifadesi olarak garantilidir. IBM 360 ve türevleri dışında bilinen tüm makinalar için değeri 2 dir.

FLT_MANT_DIG

float veri türü için gerçek sayının ondalık kısmındaki taban–**FLT_RADIX** basamağın basamak sayısıdır. Aşağıdaki ifade ondalık kısım basamaklarının sınırlı olmasından dolayı **1.0**'a gider (matematiksel olarak olmasada):

```
float radix = FLT_RADIX;

1.0f + 1.0f / radix / radix / … / radix
```

Burada **radix**, **FLT_MANT_DIG** kere uygulanır.

DBL_MANT_DIG
LDBL_MANT_DIG
Sırasıyla **double** ve **long double** veri türleri için gerçek sayının ondalık kısmındaki taban–**FLT_RADIX** basamağın basamak sayısıdır.

FLT_DIG
float türü için hassasiyeti belirleyen ondalık basamakların sayısıdır. Teknik olarak h ve i sırasıyla ikilik gösterimdeki hassasiyet ve taban ise ve ondalık basamakların sayısı o onluk gösterimdeki hassasiyet ise, örneğin, 10 tabanındaki o basamaklı bir gerçek sayı b tabanındaki h basamağa yuvarlanır ve o ondalık basamak sayısı değiştirilmeksizin tekrar geri alınır.

Bu makronun değerinin ISO C gereksinimlerini karşılamak üzere en azından **6** olacağı varsayılır.

DBL_DIG
LDBL_DIG
FLT_DIG'e benzerler ancak sırasıyla **double** ve **long double** veri türleri içindir. Bu makroların değerlerinin en azından **10** olacağı varsayılır.

FLT_MIN_EXP
float türü için ikilik gösterimdeki mümkün en küçük üs değeridir. Daha ayrıntılı ifade etmek gerekirse, **float** türündeki normalleştirilmiş bir gerçek sayı olarak **FLT_RADIX** değerinin bu değerden bir eksiğinin artan kuvvetlerinden elde edilebilecek en küçük değerini sağlayacak olan en küçük negatif tamsayıdır. (Pratikte **float** türü için en küçük değer 2^{-125-1} dir ve burada **FLT_MIN_EXP** -125 tir.)

DBL_MIN_EXP
LDBL_MIN_EXP
FLT_MIN_EXP'e benzerler ancak sırasıyla **double** ve **long double** veri türleri içindir.

FLT_MIN_10_EXP
float türü için onluk tabanda üssün en küçük negatif değeri olan bir tamsayıdır. Normalleştirilmiş gerçek sayıların mümkün en küçük değeri için **10**'un bu değerden 1 eksiği artan kuvvetindeki değerine karşılıktır. Bu değerin **-37** veya daha az olduğu varsayılır.

DBL_MIN_10_EXP
LDBL_MIN_10_EXP
FLT_MIN_10_EXP'e benzerler ancak sırasıyla **double** ve **long double** veri türleri içindir.

FLT_MAX_EXP
float türü için ikilik gösterimdeki mümkün en büyük üs değeridir. Daha ayrıntılı ifade etmek gerekirse, **float** türündeki normalleştirilmiş bir gerçek sayı olarak **FLT_RADIX** değerinin bu değerden bir eksiğinin artan kuvvetlerinden elde edilebilecek en büyük değerini sağlayacak olan en büyük pozitif tamsayıdır.

DBL_MAX_EXP
LDBL_MAX_EXP

FLT_MAX_EXP'e benzerler ancak sırasıyla **double** ve **long double** veri türleri içindir.

FLT_MAX_10_EXP
: **float** türü için onluk tabanda üssün en büyük değeri olan bir pozitif tamsayıdır. Normalleştirilmiş gerçek sayıların mümkün en küçük değeri için **10**'un bu değerden 1 eksiği artan kuvvetindeki değerine karşılıktır. Bu değerin en azından **37** olduğu varsayılır.

DBL_MAX_10_EXP
LDBL_MAX_10_EXP
: **FLT_MAX_10_EXP**'e benzerler ancak sırasıyla **double** ve **long double** veri türleri içindir.

FLT_MAX
: Bu makronun değeri **float** türünde ifade edilebilecek en büyük gerçek sayının değeridir. Bu değerin en azından **1E+37** olacağı varsayılır ve bu değer **float** türündendir.

İfade edilebilir en küçük sayı ise **- FLT_MAX**'tır.

DBL_MAX
LDBL_MAX
: **FLT_MAX**'e benzerler ancak sırasıyla **double** ve **long double** veri türleri içindir. Makro değerinin veri türü kendi türü ile aynıdır.

FLT_MIN
: Bu makronun değeri **float** türünde ifade edilebilecek en küçük gerçek sayının değeridir. Bu değerin **1E-37**'den daha büyük olmayacağı varsayılır ve bu değer **float** türündendir.

DBL_MIN
LDBL_MIN
: **FLT_MIN**'e benzerler ancak sırasıyla **double** ve **long double** veri türleri içindir. Makro değerinin veri türü kendi türü ile aynıdır.

FLT_EPSILON
: **1.0 + FLT_EPSILON != 1.0** gibi bir ifadeyi doğrulayan **float** türündeki en küçük pozitif gerçek sayıdır. **1E-5**'den büyük olmayacağı varsayılır.

DBL_EPSILON
LDBL_EPSILON
: **FLT_MIN**'e benzerler ancak sırasıyla **double** ve **long double** veri türleri içindir. Makro değerinin veri türü kendi türü ile aynıdır. Bu değerlerin **1E-9**'dan daha büyük olmayacağı varsayılır.

A.5.3.3. IEEE Gerçek Sayı Gösterimleri

Burada, en genel gerçek sayı gösterimi olan [IEEE Standard for Binary Floating Point Arithmetic (ANSI/IEEE Std 754–1985)] tarafından belirtilmiş gerçek sayı metrikleri için bir örnek gösterilmektedir. 1980 lerden sonra tasarlanan tüm bilgisayarlar bu biçimi kullanır.

IEEE tek hassasiyetli gerçek sayı biçimi ikilik tabanı kullanır. Bir işaret biti, 23 artı bir gizli bit olmak üzere (yani toplam hassasiyet 24 taban–2 basamak) 24 bitlik ondalık kısım ile –125 ile 128 aralığındaki üs değerleri için 8 bitlik üstel kısımdan oluşan bir gösterim sunar.

Bu gösterimin **float** veri türünü gerçeklemekte kullanılan ilgili değerleri aşağıda gösterilmiştir.

```
FLT_RADIX                       2
FLT_MANT_DIG                   24
FLT_DIG                         6
```

```
FLT_MIN_EXP                          -125
FLT_MIN_10_EXP                        -37
FLT_MAX_EXP                           128
FLT_MAX_10_EXP                        +38
FLT_MIN                  1.17549435E-38F
FLT_MAX                  3.40282347E+38F
FLT_EPSILON              1.19209290E-07F
```

Bunlar da **double** veri türü içindir:

```
DBL_MANT_DIG                          53
DBL_DIG                               15
DBL_MIN_EXP                        -1021
DBL_MIN_10_EXP                       -307
DBL_MAX_EXP                          1024
DBL_MAX_10_EXP                        308
DBL_MAX          1.7976931348623157E+308
DBL_MIN          2.2250738585072014E-308
DBL_EPSILON      2.2204460492503131E-016
```

A.5.4. Yapı Alanı Konum Ölçüleri

Bir yapı içindeki bir yapı üyesinin konumunu **offsetof** kullanarak bulabilirsiniz.

`size_t` **offsetof**(*tür*, *üye*) makro

İşlev, *tür* türündeki bir yapı içindeki *üye* isimli yapı üyesinin konumu olarak bir tamsayı sabit ifadesi olarak yorumlanır. Örneğin **offsetof (struct s, elem)** ifadesi, **struct s** içindeki **elem** üyesinin bayt cinsinden başlangıç konumunu verir.

Bu makro, *üye* bir bit alanı ise çalışmayacaktır. Bu durumda C derleyicisinden bir hata alırsınız.

B. Kütüphane Oluşumlarının Özeti

Bu ek, GNU C kütüphanesi ile sağlanmış başlık dosyaları içinde bildirilmiş oluşumların eksiksiz bir listesidir. Her girdi, türetildiği standart ya da diğer kaynağın ismi ile onun hakkında daha ayrıntılı bilgiyi bu kılavuzun neresinde bulabileceğinize ilişkin bilgi de içerir.

B.1. A

`long int` **a64l** `(const char *`*string*`)`
`stdlib.h` (XPG): *İkilik Verinin Kodlanması* (sayfa: 120).

`void` **abort** `(void)`
`stdlib.h` (ISO): *Anormal Sonlanma* (sayfa: 683).

`int` **abs** `(int` *number*`)`
`stdlib.h` (ISO): *Mutlak Değer* (sayfa: 519).

`int` **accept** `(int` *socket*`, struct sockaddr *`*addr*`, socklen_t *`*length_ptr*`)`
`sys/socket.h` (BSD): *Bağlantıların Kabul Edilmesi* (sayfa: 424).

`int` **access** `(const char *`*filename*`, int` *how*`)`
`unistd.h` (POSIX.1): *Dosya Erişim İzinlerinin Sınanması* (sayfa: 382).

ACCOUNTING

`error_t` **argz_create** (`char *const` *argv*`[], char **`*argz*`, size_t *`*argz_len*)
    `argz.h` (GNU): *Argz İşlevleri* (sayfa: 122).

`error_t` **argz_create_sep** (`const char *`*string*`, int` *sep*`, char **`*argz*`, size_t *`*argz_len*)
    `argz.h` (GNU): *Argz İşlevleri* (sayfa: 122).

`void` **argz_delete** (`char **`*argz*`, size_t *`*argz_len*`, char *`*entry*)
    `argz.h` (GNU): *Argz İşlevleri* (sayfa: 122).

`void` **argz_extract** (`char *`*argz*`, size_t` *argz_len*`, char **`*argv*)
    `argz.h` (GNU): *Argz İşlevleri* (sayfa: 122).

`error_t` **argz_insert** (`char **`*argz*`, size_t *`*argz_len*`, char *`*before*`, const char *`*entry*)
    `argz.h` (GNU): *Argz İşlevleri* (sayfa: 122).

`char *` **argz_next** (`char *`*argz*`, size_t` *argz_len*`, const char *`*entry*)
    `argz.h` (GNU): *Argz İşlevleri* (sayfa: 122).

`error_t` **argz_replace** (`char **`*argz*`, size_t *`*argz_len*`, const char *`*str*`, const char *`*with*`, unsigned *`*replace_count*)
    `argz.h` (GNU): *Argz İşlevleri* (sayfa: 122).

`void` **argz_stringify** (`char *`*argz*`, size_t` *len*`, int` *sep*)
    `argz.h` (GNU): *Argz İşlevleri* (sayfa: 122).

`char *` **asctime** (`const struct tm *`*brokentime*)
    `time.h` (ISO): *Zaman Değerlerinin Biçimlenmesi* (sayfa: 550).

`char *` **asctime_r** (`const struct tm *`*brokentime*`, char *`*buffer*)
    `time.h` (POSIX.1c): *Zaman Değerlerinin Biçimlenmesi* (sayfa: 550).

`double` **asin** (`double` *x*)
    `math.h` (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`float` **asinf** (`float` *x*)
    `math.h` (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`double` **asinh** (`double` *x*)
    `math.h` (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`float` **asinhf** (`float` *x*)
    `math.h` (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`long double` **asinhl** (`long double` *x*)
    `math.h` (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`long double` **asinl** (`long double` *x*)
    `math.h` (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`int` **asprintf** (`char **`*ptr*`, const char *`*template*`, …`)
    `stdio.h` (GNU): *Biçimli Çıktıyı Özdevimli Ayırma* (sayfa: 266).

`void` **assert** (`int` *expression*)
    `assert.h` (ISO): *Dahilî Kararlılığın Doğrudan Denetlenmesi* (sayfa: 813).

B.2. B

termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B134
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B150
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B1800
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B19200
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B200
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B230400
termios.h (GNU): *Hat Hızı* (sayfa: 453).

B2400
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B300
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B38400
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B460800
termios.h (GNU): *Hat Hızı* (sayfa: 453).

B4800
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B50
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B57600
termios.h (GNU): *Hat Hızı* (sayfa: 453).

B600
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B75
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

B9600
termios.h (POSIX.1): *Hat Hızı* (sayfa: 453).

`int` **`backtrace`** `(void **`*buffer*`, int` *size*`)`
execinfo.h (GNU): *Köken Arama Listeleri* (sayfa: 810).

`char **` **`backtrace_symbols`** `(void *const *`*buffer*`, int` *size*`)`
execinfo.h (GNU): *Köken Arama Listeleri* (sayfa: 810).

`void` **`backtrace_symbols_fd`** `(void *const *`*buffer*`, int` *size*`, int` *fd*`)`

stdlib.h (ISO): *Dizi Arama İşlevleri* (sayfa: 203).

`wint_t` **btowc** (int *c*)
wchar.h (ISO): *Bir Karakterin Dönüştürülmesi* (sayfa: 132).

`int` **BUFSIZ**
stdio.h (ISO): *Tamponlama Çeşidinin Seçimi* (sayfa: 294).

`void` **bzero** (void **block*, size_t *size*)
string.h (BSD): *Kopyalama ve Birleştirme* (sayfa: 94).

B.3. C

`double` **cabs** (complex double *z*)
complex.h (ISO): *Mutlak Değer* (sayfa: 519).

`float` **cabsf** (complex float *z*)
complex.h (ISO): *Mutlak Değer* (sayfa: 519).

`long double` **cabsl** (complex long double *z*)
complex.h (ISO): *Mutlak Değer* (sayfa: 519).

`complex double` **cacos** (complex double *z*)
complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`complex float` **cacosf** (complex float *z*)
complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`complex double` **cacosh** (complex double *z*)
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex float` **cacoshf** (complex float *z*)
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex long double` **cacoshl** (complex long double *z*)
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex long double` **cacosl** (complex long double *z*)
complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`void *` **calloc** (size_t *count*, size_t *eltsize*)
malloc.h, stdlib.h (ISO): *Temizlenmiş Bellek Ayırma* (sayfa: 53).

`char *` **canonicalize_file_name** (const char **name*)
stdlib.h (GNU): *Sembolik Bağlar* (sayfa: 365).

`double` **carg** (complex double *z*)
complex.h (ISO): *Karmaşık Sayıların İzdüşümleri, Eşlenikleri ve Analizi* (sayfa: 528).

`float` **cargf** (complex float *z*)
complex.h (ISO): *Karmaşık Sayıların İzdüşümleri, Eşlenikleri ve Analizi* (sayfa: 528).

`long double` **cargl** (complex long double *z*)
complex.h (ISO): *Karmaşık Sayıların İzdüşümleri, Eşlenikleri ve Analizi* (sayfa: 528).

`complex double` **casin** (complex double *z*)

complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`complex float` **`casinf`** `(complex float` *z*`)`
complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`complex double` **`casinh`** `(complex double` *z*`)`
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex float` **`casinhf`** `(complex float` *z*`)`
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex long double` **`casinhl`** `(complex long double` *z*`)`
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex long double` **`casinl`** `(complex long double` *z*`)`
complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`complex double` **`catan`** `(complex double` *z*`)`
complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`complex float` **`catanf`** `(complex float` *z*`)`
complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`complex double` **`catanh`** `(complex double` *z*`)`
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex float` **`catanhf`** `(complex float` *z*`)`
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex long double` **`catanhl`** `(complex long double` *z*`)`
complex.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

`complex long double` **`catanl`** `(complex long double` *z*`)`
complex.h (ISO): *Ters Trigonometrik İşlevler* (sayfa: 478).

`nl_catd` **`catopen`** `(const char *`*cat_name*`, int` *flag*`)`
nl_types.h (X/Open): *catgets İşlevleri* (sayfa: 182).

`int` **`cbc_crypt`** `(char *`*key*`, char *`*blocks*`, unsigned` *len*`, unsigned` *mode*`, char *`*ivec*`)`
rpc/des_crypt.h (SUNRPC): *DES Şifreleme* (sayfa: 806).

`double` **`cbrt`** `(double` *x*`)`
math.h (BSD): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **`cbrtf`** `(float` *x*`)`
math.h (BSD): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`long double` **`cbrtl`** `(long double` *x*`)`
math.h (BSD): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`complex double` **`ccos`** `(complex double` *z*`)`
complex.h (ISO): *Trigonometrik İşlevler* (sayfa: 476).

`complex float` **`ccosf`** `(complex float` *z*`)`
complex.h (ISO): *Trigonometrik İşlevler* (sayfa: 476).

`complex double` **`ccosh`** `(complex double` *z*`)`

B.4. D

libgen.h (XPG): *Bir Dizgeyi Dizgeciklere Ayırma* (sayfa: 115).

`div_t` **div** (int *numerator*, int *denominator*)
stdlib.h (ISO): *Tamsayı Bölme* (sayfa: 508).

div_t
stdlib.h (ISO): *Tamsayı Bölme* (sayfa: 508).

`char *` **dngettext** (const char **domain*, const char **msgid1*, const char **msgid2*, unsigned long int *n*)
libintl.h (GNU): *Gelişkin gettext işlevleri* (sayfa: 193).

`double` **drand48** (void)
stdlib.h (SVID): *SVID Rasgele Sayı İşlevleri* (sayfa: 500).

`int` **drand48_r** (struct drand48_data **buffer*, double **result*)
stdlib.h (GNU): *SVID Rasgele Sayı İşlevleri* (sayfa: 500).

`double` **drem** (double *numerator*, double *denominator*)
math.h (BSD): *Kalan İşlevleri* (sayfa: 523).

`float` **dremf** (float *numerator*, float *denominator*)
math.h (BSD): *Kalan İşlevleri* (sayfa: 523).

`long double` **dreml** (long double *numerator*, long double *denominator*)
math.h (BSD): *Kalan İşlevleri* (sayfa: 523).

`mode_t` **DTTOIF** (int *dtype*)
dirent.h (BSD): *Dizin Girdileri* (sayfa: 353).

`int` **dup** (int *old*)
unistd.h (POSIX.1): *Tanıtıcıların Çoğullanması* (sayfa: 339).

`int` **dup2** (int *old*, int *new*)
unistd.h (POSIX.1): *Tanıtıcıların Çoğullanması* (sayfa: 339).

B.5. E

`int` **E2BIG**
errno.h (POSIX.1: Argüman listesi çok uzun): *Hata Kodları* (sayfa: 32).

`int` **EACCES**
errno.h (POSIX.1: İzinler yetersiz): *Hata Kodları* (sayfa: 32).

`int` **EADDRINUSE**
errno.h (BSD: İstenen soket adresi kullanımda): *Hata Kodları* (sayfa: 32).

`int` **EADDRNOTAVAIL**
errno.h (BSD: İstenen soket adresi kullanıma uygun değil): *Hata Kodları* (sayfa: 32).

`int` **EADV**
errno.h (Linux???: Dikkat çekme hatası): *Hata Kodları* (sayfa: 32).

`int` **EAFNOSUPPORT**
errno.h (BSD: Soket için belirtilen adres ailesi desteklenmiyor): *Hata Kodları* (sayfa: 32).

`int` **EAGAIN**

errno.h (POSIX.1: Özkaynak geçici olarak kullanımdışı): *Hata Kodları* (sayfa: 32).

int **EALREADY**
errno.h (BSD: Engellenmemesi öngörülmüş nesne üzerindeki işlem hala sürüyor): *Hata Kodları* (sayfa: 32).

int **EAUTH**
errno.h (BSD: Kimlik kanıtlama hatası): *Hata Kodları* (sayfa: 32).

int **EBACKGROUND**
errno.h (GNU: Artalan süreci için ilgisiz işlem): *Hata Kodları* (sayfa: 32).

int **EBADE**
errno.h (Linux???: Geçersiz değişim): *Hata Kodları* (sayfa: 32).

int **EBADF**
errno.h (POSIX.1: Dosya tanımlayıcı hatalı): *Hata Kodları* (sayfa: 32).

int **EBADFD**
errno.h (Linux???: Dosya tanımlayıcı hatalı durumda): *Hata Kodları* (sayfa: 32).

int **EBADMSG**
errno.h (XOPEN: Hatalı ileti): *Hata Kodları* (sayfa: 32).

int **EBADR**
errno.h (Linux???: İstek tanımlayıcı hatalı): *Hata Kodları* (sayfa: 32).

int **EBADRPC**
errno.h (BSD: RPC yapısı hatalı): *Hata Kodları* (sayfa: 32).

int **EBADRQC**
errno.h (Linux???: İstek kodu geçersiz): *Hata Kodları* (sayfa: 32).

int **EBADSLT**
errno.h (Linux???: Yuva geçersiz): *Hata Kodları* (sayfa: 32).

int **EBFONT**
errno.h (Linux???: Yazıtipi dosyasının biçimi hatalı): *Hata Kodları* (sayfa: 32).

int **EBUSY**
errno.h (POSIX.1: Aygıt ya da özkaynak meşgul): *Hata Kodları* (sayfa: 32).

int **ECANCELED**
errno.h (POSIX.1: İşlem iptal edildi): *Hata Kodları* (sayfa: 32).

int **ecb_crypt** (char *key, char *blocks, unsigned len, unsigned mode)
rpc/des_crypt.h (SUNRPC): *DES Şifreleme* (sayfa: 806).

int **ECHILD**
errno.h (POSIX.1: Hiç alt süreç yok): *Hata Kodları* (sayfa: 32).

tcflag_t **ECHO**
termios.h (POSIX.1): *Yerel Kipler* (sayfa: 451).

tcflag_t **ECHOCTL**
termios.h (BSD): *Yerel Kipler* (sayfa: 451).

tcflag_t **ECHOE**

errno.h (BSD: Kullanıcının disk kotası aşıldı): *Hata Kodları* (sayfa: 32).

int **EEXIST**
errno.h (POSIX.1: Dosya var): *Hata Kodları* (sayfa: 32).

int **EFAULT**
errno.h (POSIX.1: Hatalı adres): *Hata Kodları* (sayfa: 32).

int **EFBIG**
errno.h (POSIX.1: Dosya çok büyük): *Hata Kodları* (sayfa: 32).

int **EFTYPE**
errno.h (BSD: İlgisiz dosya türü ya da biçimi): *Hata Kodları* (sayfa: 32).

int **EGRATUITOUS**
errno.h (GNU: Gereksiz hata): *Hata Kodları* (sayfa: 32).

int **EGREGIOUS**
errno.h (GNU: Bu kez onu gerçekten harcadınız): *Hata Kodları* (sayfa: 32).

int **EHOSTDOWN**
errno.h (BSD: İstenen ağ bağlantısındaki uzak konak çökük): *Hata Kodları* (sayfa: 32).

int **EHOSTUNREACH**
errno.h (BSD: İstenen ağ bağlantısındaki uzak konak erişilebilir değil): *Hata Kodları* (sayfa: 32).

int **EIDRM**
errno.h (XOPEN: Belirteç kaldırıldı): *Hata Kodları* (sayfa: 32).

int **EIEIO**
errno.h (GNU: Bilgisayar çiftlik oldu): *Hata Kodları* (sayfa: 32).

int **EILSEQ**
errno.h (ISO: Çok baytlı ya da geniş karakter tamamlanmamış ya da geçersiz): *Hata Kodları* (sayfa: 32).

int **EINPROGRESS**
errno.h (BSD: Engellenmemesi öngörülmüş bir nesne üzerinde başlatılmış bir işlem tamamlanmadı): *Hata Kodları* (sayfa: 32).

int **EINTR**
errno.h (POSIX.1: İşlev çağrısı engellendi): *Hata Kodları* (sayfa: 32).

int **EINVAL**
errno.h (POSIX.1: Geçersiz argüman): *Hata Kodları* (sayfa: 32).

int **EIO**
errno.h (POSIX.1: Giriş/Çıkış hatası): *Hata Kodları* (sayfa: 32).

int **EISCONN**
errno.h (BSD: Zaten bağlı olan bir sokete bağlanmayı denediniz): *Hata Kodları* (sayfa: 32).

int **EISDIR**
errno.h (POSIX.1: Dosya bir dizin): *Hata Kodları* (sayfa: 32).

int **EISNAM**
errno.h (Linux???: Bir isimli dosya mı): *Hata Kodları* (sayfa: 32).

int **EKEYEXPIRED**
errno.h (Linux: Key has expired): *Hata Kodları* (sayfa: 32).

int **EKEYREJECTED**
errno.h (Linux: Key was rejected by service): *Hata Kodları* (sayfa: 32).

int **EKEYREVOKED**
errno.h (Linux: Key has been revoked): *Hata Kodları* (sayfa: 32).

int **EL2HLT**
errno.h (Obsolete: 2. seviye kapandı): *Hata Kodları* (sayfa: 32).

int **EL2NSYNC**
errno.h (Obsolete: 2. seviye eşzamanlı değil): *Hata Kodları* (sayfa: 32).

int **EL3HLT**
errno.h (Obsolete: 3. seviye kapandı): *Hata Kodları* (sayfa: 32).

int **EL3RST**
errno.h (Obsolete: 3. seviye sıfırlandı): *Hata Kodları* (sayfa: 32).

int **ELIBACC**
errno.h (Linux???: Gerekli bir paylaşımlı kütüphaneye erişilemiyor): *Hata Kodları* (sayfa: 32).

int **ELIBBAD**
errno.h (Linux???: Bozulmuş bir paylaşımlı kütüphaneye erişim): *Hata Kodları* (sayfa: 32).

int **ELIBEXEC**
errno.h (Linux???: Bir paylaşımlı kütüphane doğrudan çalıştırılamaz): *Hata Kodları* (sayfa: 32).

int **ELIBMAX**
errno.h (Linux???: Çok fazla paylaşımlı kütüphane ilintilenmeye çalışılıyor): *Hata Kodları* (sayfa: 32).

int **ELIBSCN**
errno.h (Linux???: a.out içindeki .lib bölümü bozulmuş): *Hata Kodları* (sayfa: 32).

int **ELNRNG**
errno.h (Linux???: Bağ numarası aralık dışında): *Hata Kodları* (sayfa: 32).

int **ELOOP**
errno.h (BSD: Bir dosya ismine bakılırken çok seviyeli sembolik bağlar saptandı): *Hata Kodları* (sayfa: 32).

int **EMEDIUMTYPE**
errno.h (Linux???: Ortam türü yanlış): *Hata Kodları* (sayfa: 32).

int **EMFILE**
errno.h (POSIX.1: Mevcut süreç çok fazla dosya açmış ve daha fazlasını açamaz): *Hata Kodları* (sayfa: 32).

int **EMLINK**
errno.h (POSIX.1: Çok fazla bağ var): *Hata Kodları* (sayfa: 32).

EMPTY
utmp.h (SVID): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

EMPTY

utmpx.h (XPG4.2): *XPG Kullanıcı Hesapları Veritabanı İşlevleri* (sayfa: 757).

`int` **EMSGSIZE**
errno.h (BSD: Bir sokete gönderilen iletinin uzunluğu desteklenenden fazla): *Hata Kodları* (sayfa: 32).

`int` **EMULTIHOP**
errno.h (XOPEN: Çoklu sıçrama denendi): *Hata Kodları* (sayfa: 32).

`int` **ENAMETOOLONG**
errno.h (POSIX.1: Dosya ismi çok uzun): *Hata Kodları* (sayfa: 32).

`int` **ENAVAIL**
errno.h (Linux???: Kullanılabilir bir XENIX semaforu yok): *Hata Kodları* (sayfa: 32).

`void` **encrypt** `(char *`*block*`, int` *edflag*`)`
crypt.h (BSD, SVID): *DES Şifreleme* (sayfa: 806).

`void` **encrypt_r** `(char *`*block*`, int` *edflag*`, struct crypt_data *` *data*`)`
crypt.h (GNU): *DES Şifreleme* (sayfa: 806).

`void` **endfsent** `(void)`
fstab.h (BSD): *fstab* (sayfa: 773).

`void` **endgrent** `(void)`
grp.h (SVID, BSD): *Grup Listesinin Taranması* (sayfa: 764).

`void` **endhostent** `(void)`
netdb.h (BSD): *Konak İsimleri* (sayfa: 412).

`int` **endmntent** `(FILE *`*stream*`)`
mntent.h (BSD): *mtab* (sayfa: 775).

`void` **endnetent** `(void)`
netdb.h (BSD): *Ağ İsimleri Veritabanı* (sayfa: 440).

`void` **endnetgrent** `(void)`
netdb.h (BSD): *Bir Ağgrubu Hakkında Bilgi Alınması* (sayfa: 766).

`void` **endprotoent** `(void)`
netdb.h (BSD): *Protokol Veritabanı* (sayfa: 417).

`void` **endpwent** `(void)`
pwd.h (SVID, BSD): *Kullanıcı Listesinin Taranması* (sayfa: 761).

`void` **endservent** `(void)`
netdb.h (BSD): *Servis Veritabanı* (sayfa: 415).

`void` **endutent** `(void)`
utmp.h (SVID): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

`void` **endutxent** `(void)`
utmpx.h (XPG4.2): *XPG Kullanıcı Hesapları Veritabanı İşlevleri* (sayfa: 757).

`int` **ENEEDAUTH**
errno.h (BSD: Kimlik kanıtlayıcı gerekli): *Hata Kodları* (sayfa: 32).

`int` **ENETDOWN**

errno.h (BSD: Ağ çökük olduğundan soket işlemi başarısız oldu): *Hata Kodları* (sayfa: 32).

int **ENETRESET**
errno.h (BSD: Uzak konak çöktüğünden ağ bağlantısı sıfırlandı): *Hata Kodları* (sayfa: 32).

int **ENETUNREACH**
errno.h (BSD: Uzak konağı içeren alt ağ erişilemez olduğundan soket işlemi başarısız oldu): *Hata Kodları* (sayfa: 32).

int **ENFILE**
errno.h (POSIX.1: Sistemin bütününde çok fazla farklı dosya açılışı var): *Hata Kodları* (sayfa: 32).

int **ENOANO**
errno.h (Linux???: Anot yok): *Hata Kodları* (sayfa: 32).

int **ENOBUFS**
errno.h (BSD: Çekirdeğin G/Ç tamponlarının hepsi kullanımda): *Hata Kodları* (sayfa: 32).

int **ENOCSI**
errno.h (Linux???: Kullanılabilir bir CSI yapısı yok): *Hata Kodları* (sayfa: 32).

int **ENODATA**
errno.h (XOPEN: Kullanılabilir veri yok): *Hata Kodları* (sayfa: 32).

int **ENODEV**
errno.h (POSIX.1: Belli bir aygıtın verilmesi umulan bir işleve yanlış aygıt türü verildi): *Hata Kodları* (sayfa: 32).

int **ENOENT**
errno.h (POSIX.1: Böyle bir dosya ya da dizin yok): *Hata Kodları* (sayfa: 32).

int **ENOEXEC**
errno.h (POSIX.1: Çalıştırılabilir dosya biçimi geçersiz): *Hata Kodları* (sayfa: 32).

int **ENOKEY**
errno.h (Linux: Required key not available): *Hata Kodları* (sayfa: 32).

int **ENOLCK**
errno.h (POSIX.1: Kullanılabilir bir kilit yok): *Hata Kodları* (sayfa: 32).

int **ENOLINK**
errno.h (XOPEN: Bağ kopmuştu): *Hata Kodları* (sayfa: 32).

int **ENOMEDIUM**
errno.h (Linux???: Ortam bulunamadı): *Hata Kodları* (sayfa: 32).

int **ENOMEM**
errno.h (POSIX.1: Yeterli bellek yok): *Hata Kodları* (sayfa: 32).

int **ENOMSG**
errno.h (XOPEN: İstenen türde ileti yok): *Hata Kodları* (sayfa: 32).

int **ENONET**
errno.h (Linux???: Makina ağ üzerinde değil): *Hata Kodları* (sayfa: 32).

int **ENOPKG**
errno.h (Linux???: Paket kurulu değil): *Hata Kodları* (sayfa: 32).

`int` **ENOPROTOOPT**
`errno.h` (BSD: Belirttiğiniz soket seçeneği protokol için uygun değil): *Hata Kodları* (sayfa: 32).

`int` **ENOSPC**
`errno.h` (POSIX.1: Aygıt üzerinde yer yok): *Hata Kodları* (sayfa: 32).

`int` **ENOSR**
`errno.h` (XOPEN: Akımdışı özkaynaklar): *Hata Kodları* (sayfa: 32).

`int` **ENOSTR**
`errno.h` (XOPEN: Aygıt bir akım değil): *Hata Kodları* (sayfa: 32).

`int` **ENOSYS**
`errno.h` (POSIX.1: İşlev henüz gerçeklenmedi): *Hata Kodları* (sayfa: 32).

`int` **ENOTBLK**
`errno.h` (BSD: Blok aygıtı gerekli): *Hata Kodları* (sayfa: 32).

`int` **ENOTCONN**
`errno.h` (BSD: Soket hiçbir şeye bağlı değil): *Hata Kodları* (sayfa: 32).

`int` **ENOTDIR**
`errno.h` (POSIX.1: Bir dizin gerekliyken belirtilen dosya bir dizin değil): *Hata Kodları* (sayfa: 32).

`int` **ENOTEMPTY**
`errno.h` (POSIX.1: Dizin boş değil, işlem için dizin boş olmalı): *Hata Kodları* (sayfa: 32).

`int` **ENOTNAM**
`errno.h` (Linux???: İsimli türde bir XENIX dosyası değil): *Hata Kodları* (sayfa: 32).

`int` **ENOTRECOVERABLE**
`errno.h` (Linux: State not recoverable): *Hata Kodları* (sayfa: 32).

`int` **ENOTSOCK**
`errno.h` (BSD: Bir soket gerektiği halde belirtilen dosya bir soket değil): *Hata Kodları* (sayfa: 32).

`int` **ENOTSUP**
`errno.h` (POSIX.1: Desteklenmiyor): *Hata Kodları* (sayfa: 32).

`int` **ENOTTY**
`errno.h` (POSIX.1: İlgisiz G/Ç denetimi işlemi): *Hata Kodları* (sayfa: 32).

`int` **ENOTUNIQ**
`errno.h` (Linux???: İsim ağ üzerinde eşsiz değil): *Hata Kodları* (sayfa: 32).

`char ** environ`
`unistd.h` (POSIX.1): *Ortama Erişim* (sayfa: 677).

`error_t` **envz_add** `(char **`*envz*`, size_t *`*envz_len*`, const char *`*name*`, const char *`*value*`)`
`envz.h` (GNU): *Envz İşlevleri* (sayfa: 124).

`char *` **envz_entry** `(const char *`*envz*`, size_t` *envz_len*`, const char *`*name*`)`
`envz.h` (GNU): *Envz İşlevleri* (sayfa: 124).

`char *` **envz_get** `(const char *`*envz*`, size_t` *envz_len*`, const char *`*name*`)`
`envz.h` (GNU): *Envz İşlevleri* (sayfa: 124).

`error_t` **envz_merge** `(char **`*envz*`, size_t *`*envz_len*`, const char *`*envz2*`, size_t` *envz2_len*`, int` *override*`)`
envz.h (GNU): *Envz İşlevleri* (sayfa: 124).

`void` **envz_strip** `(char **`*envz*`, size_t *`*envz_len*`)`
envz.h (GNU): *Envz İşlevleri* (sayfa: 124).

`int` **ENXIO**
errno.h (POSIX.1: Böyle bir adres ya da aygıt yok.): *Hata Kodları* (sayfa: 32).

`int` **EOF**
stdio.h (ISO): *Dosya Sonu ve Hatalar* (sayfa: 286).

`int` **EOPNOTSUPP**
errno.h (BSD: İstediğiniz işlem desteklenmiyor): *Hata Kodları* (sayfa: 32).

`int` **EOVERFLOW**
errno.h (XOPEN: Tanımlı veri türü için değer çok büyük): *Hata Kodları* (sayfa: 32).

`int` **EOWNERDEAD**
errno.h (Linux: Owner died): *Hata Kodları* (sayfa: 32).

`int` **EPERM**
errno.h (POSIX.1: İşleme izin verilmedi): *Hata Kodları* (sayfa: 32).

`int` **EPFNOSUPPORT**
errno.h (BSD: İstediğiniz soket iletişim protokolü ailesi desteklenmiyor): *Hata Kodları* (sayfa: 32).

`int` **EPIPE**
errno.h (POSIX.1: Kırık boruhattı): *Hata Kodları* (sayfa: 32).

`int` **EPROCLIM**
errno.h (BSD: Süreç üzerindeki kullanıcı başına sınır fork ile aşıldı): *Hata Kodları* (sayfa: 32).

`int` **EPROCUNAVAIL**
errno.h (BSD: RPC, uygulama için hatalı yordam yürüttü): *Hata Kodları* (sayfa: 32).

`int` **EPROGMISMATCH**
errno.h (BSD: RPC uygulaması sürümü yanlış): *Hata Kodları* (sayfa: 32).

`int` **EPROGUNAVAIL**
errno.h (BSD: RPC uygulaması kullanılabilir değil): *Hata Kodları* (sayfa: 32).

`int` **EPROTO**
errno.h (XOPEN: Protokol hatası): *Hata Kodları* (sayfa: 32).

`int` **EPROTONOSUPPORT**
errno.h (BSD: Soket istenen iletişim protokolünü desteklemiyor): *Hata Kodları* (sayfa: 32).

`int` **EPROTOTYPE**
errno.h (BSD: İstenen protokol bu soket türünü desteklemiyor): *Hata Kodları* (sayfa: 32).

`int` **EQUIV_CLASS_MAX**
limits.h (POSIX.2): *Bazı Araçların Kapasite Sınırları* (sayfa: 800).

`double` **erand48** `(unsigned short int` *xsubi*`[3])`
stdlib.h (SVID): *SVID Rasgele Sayı İşlevleri* (sayfa: 500).

`int` **erand48_r** `(unsigned short int` *xsubi*`[3], struct drand48_data *`*buffer*`, double *`*result*`)`
`stdlib.h` (GNU): *SVID Rasgele Sayı İşlevleri* (sayfa: 500).

`int` **ERANGE**
`errno.h` (ISO: Sayısal sonuç aralığın dışında): *Hata Kodları* (sayfa: 32).

`int` **EREMCHG**
`errno.h` (Linux???: Uzak adres değişti): *Hata Kodları* (sayfa: 32).

`int` **EREMOTE**
`errno.h` (BSD: Bir uzak dosya sistemi zaten kullanımda olan NFS bağlama dosyası ismi ile sisteme bağlanmaya çalışıldı): *Hata Kodları* (sayfa: 32).

`int` **EREMOTEIO**
`errno.h` (Linux???: Uzak G/Ç hatası): *Hata Kodları* (sayfa: 32).

`int` **ERESTART**
`errno.h` (Linux???: Engellenen sistem çağrısı yeniden başlatılmalı): *Hata Kodları* (sayfa: 32).

`double` **erf** `(double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`double` **erfc** `(double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`float` **erfcf** `(float` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`long double` **erfcl** `(long double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`float` **erff** `(float` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`long double` **erfl** `(long double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`int` **EROFS**
`errno.h` (POSIX.1: Salt okunur dosya sisteminde birşeyler değiştirilmeye çalışıldı): *Hata Kodları* (sayfa: 32).

`int` **ERPCMISMATCH**
`errno.h` (BSD: RPC sürümü yanlış): *Hata Kodları* (sayfa: 32).

`void` **err** `(int` *status*`, const char *`*format*`, …)`
`err.h` (BSD): *Hata İletileri* (sayfa: 41).

volatile int errno
`errno.h` (ISO): *Hata Denetimi* (sayfa: 31).

`void` **error** `(int` *status*`, int` *errnum*`, const char *`*format*`, …)`
`error.h` (GNU): *Hata İletileri* (sayfa: 41).

`void` **error_at_line** `(int` *status*`, int` *errnum*`, const char *`*fname*`, unsigned int` *lineno*`, const char *`*format*`, …)`

error.h (GNU): *Hata İletileri* (sayfa: 41).

`unsigned int` **error_message_count**
error.h (GNU): *Hata İletileri* (sayfa: 41).

`int` **error_one_per_line**
error.h (GNU): *Hata İletileri* (sayfa: 41).

`void` **(* error_print_progname)** `(void)`
error.h (GNU): *Hata İletileri* (sayfa: 41).

`void` **errx** `(int` *status*`, const char *`*format*`, …)`
err.h (BSD): *Hata İletileri* (sayfa: 41).

`int` **ESHUTDOWN**
errno.h (BSD: Soket zaten kapatılmış): *Hata Kodları* (sayfa: 32).

`int` **ESOCKTNOSUPPORT**
errno.h (BSD: Soket türü desteklenmiyor): *Hata Kodları* (sayfa: 32).

`int` **ESPIPE**
errno.h (POSIX.1: Konumlama işlemi geçersiz): *Hata Kodları* (sayfa: 32).

`int` **ESRCH**
errno.h (POSIX.1: Belirtilen süreç kimliği ile eşleşen bir süreç yok): *Hata Kodları* (sayfa: 32).

`int` **ESRMNT**
errno.h (Linux???: Srmount hatası): *Hata Kodları* (sayfa: 32).

`int` **ESTALE**
errno.h (BSD: Eskimiş NFS dosya kaydı): *Hata Kodları* (sayfa: 32).

`int` **ESTRPIPE**
errno.h (Linux???: Akımlarda boruhattı hatası): *Hata Kodları* (sayfa: 32).

`int` **ETIME**
errno.h (XOPEN: Timer zamanaşımına uğradı): *Hata Kodları* (sayfa: 32).

`int` **ETIMEDOUT**
errno.h (BSD: Soket işlemine zamanaşımı süresinde bir yanıt gelmedi): *Hata Kodları* (sayfa: 32).

`int` **ETOOMANYREFS**
errno.h (BSD: Çok fazla başvuru: uçlar birbirine bağlanamıyor): *Hata Kodları* (sayfa: 32).

`int` **ETXTBSY**
errno.h (BSD: Metin dosyası meşgul): *Hata Kodları* (sayfa: 32).

`int` **EUCLEAN**
errno.h (Linux???: Yapı temizlik gerektiriyor): *Hata Kodları* (sayfa: 32).

`int` **EUNATCH**
errno.h (Linux???: Protokol sürücüsü bağlı değil): *Hata Kodları* (sayfa: 32).

`int` **EUSERS**
errno.h (BSD: Çok fazla kullanıcı olduğundan dosya kotası sistemi bozuldu): *Hata Kodları* (sayfa: 32).

`int` **EWOULDBLOCK**

errno.h (BSD: Özkaynak geçici olarak kullanımdışı): *Hata Kodları* (sayfa: 32).

`int` **EXDEV**
errno.h (POSIX.1: Dosya sistemlerine uygunsuz bir bağ oluşturulmaya çalışılıyor): *Hata Kodları* (sayfa: 32).

`int` **execl** (`const char *`*filename*`, const char *`*arg0*`, …`)
unistd.h (POSIX.1): *Bir Dosyanın Çalıştırılması* (sayfa: 688).

`int` **execle** (`const char *`*filename*`, const char *`*arg0*`, char *const` *env*`[], …`)
unistd.h (POSIX.1): *Bir Dosyanın Çalıştırılması* (sayfa: 688).

`int` **execlp** (`const char *`*filename*`, const char *`*arg0*`, …`)
unistd.h (POSIX.1): *Bir Dosyanın Çalıştırılması* (sayfa: 688).

`int` **execv** (`const char *`*filename*`, char *const` *argv*`[]`)
unistd.h (POSIX.1): *Bir Dosyanın Çalıştırılması* (sayfa: 688).

`int` **execve** (`const char *`*filename*`, char *const` *argv*`[], char *const` *env*`[]`)
unistd.h (POSIX.1): *Bir Dosyanın Çalıştırılması* (sayfa: 688).

`int` **execvp** (`const char *`*filename*`, char *const` *argv*`[]`)
unistd.h (POSIX.1): *Bir Dosyanın Çalıştırılması* (sayfa: 688).

`int` **EXFULL**
errno.h (Linux???: Değişim kotası doldu): *Hata Kodları* (sayfa: 32).

`void` **_Exit** (`int` *status*)
stdlib.h (ISO): *Sonlandırmanın İçyapısı* (sayfa: 684).

`void` **_exit** (`int` *status*)
unistd.h (POSIX.1): *Sonlandırmanın İçyapısı* (sayfa: 684).

`int` **EXIT_FAILURE** (`int` *status*)
stdlib.h (ISO): *Çıkış Durumu* (sayfa: 682).

`int` **EXIT_SUCCESS**
stdlib.h (ISO): *Çıkış Durumu* (sayfa: 682).

`void` **exit** (`int` *status*)
stdlib.h (ISO): *Normal Sonlandırma* (sayfa: 681).

`double` **exp** (`double` *x*)
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`double` **exp10** (`double` *x*)
math.h (GNU): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **exp10f** (`float` *x*)
math.h (GNU): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`long double` **exp10l** (`long double` *x*)
math.h (GNU): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`double` **exp2** (`double` *x*)
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **exp2f** (`float` *x*)

B.6. F

FLT_DIG
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_EPSILON
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_MANT_DIG
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_MAX
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_MAX_10_EXP
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_MAX_EXP
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_MIN
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_MIN_10_EXP
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_MIN_EXP
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_RADIX
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

FLT_ROUNDS
`float.h` (ISO): *Gerçek Sayılar ile İlgili Makrolar* (sayfa: 824).

`void` **_flushlbf** `(void)`
`stdio_ext.h` (GNU): *Tamponların Boşaltılması* (sayfa: 292).

`tcflag_t` **FLUSHO**
`termios.h` (BSD): *Yerel Kipler* (sayfa: 451).

`double` **fma** `(double` *x*`, double` *y*`, double` *z*`)`
`math.h` (ISO): *Çeşitli Gerçek Sayı Aritmetik İşlevleri* (sayfa: 526).

`float` **fmaf** `(float` *x*`, float` *y*`, float` *z*`)`
`math.h` (ISO): *Çeşitli Gerçek Sayı Aritmetik İşlevleri* (sayfa: 526).

`long double` **fmal** `(long double` *x*`, long double` *y*`, long double` *z*`)`
`math.h` (ISO): *Çeşitli Gerçek Sayı Aritmetik İşlevleri* (sayfa: 526).

`double` **fmax** `(double` *x*`, double` *y*`)`
`math.h` (ISO): *Çeşitli Gerçek Sayı Aritmetik İşlevleri* (sayfa: 526).

`float` **fmaxf** `(float` *x*`, float` *y*`)`
`math.h` (ISO): *Çeşitli Gerçek Sayı Aritmetik İşlevleri* (sayfa: 526).

`long double` **fmaxl** `(long double` *x*`, long double` *y*`)`
`math.h` (ISO): *Çeşitli Gerçek Sayı Aritmetik İşlevleri* (sayfa: 526).

`FILE *` **fmemopen** `(void *`*buf*`, size_t` *size*`, const char *`*opentype*`)`

`FILE * **fopencookie** (void *cookie, const char *opentype, cookie_io_functions_t io–functions)`
`stdio.h` (GNU): *Özel Akımlar ve Çerezler* (sayfa: 298).

`int **FOPEN_MAX**`
`stdio.h` (ISO): *Akımların Açılması* (sayfa: 238).

`pid_t **fork** (void)`
`unistd.h` (POSIX.1): *Bir Sürecin Oluşturulması* (sayfa: 687).

`int **forkpty** (int *amaster, char *name, struct termios *termp, struct winsize *winp)`
`pty.h` (BSD): *Bir Uçbirimsi Çiftinin Açılması* (sayfa: 466).

`long int **fpathconf** (int filedes, int parameter)`
`unistd.h` (POSIX.1): *pathconf Kullanımı* (sayfa: 798).

`int **fpclassify** ( float–typex)`
`math.h` (ISO): *Gerçek Sayı Sınıflama İşlevleri* (sayfa: 510).

FPE_DECOVF_TRAP
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

FPE_FLTDIV_FAULT
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

FPE_FLTDIV_TRAP
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

FPE_FLTOVF_FAULT
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

FPE_FLTOVF_TRAP
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

FPE_FLTUND_FAULT
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

FPE_FLTUND_TRAP
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

FPE_INTDIV_TRAP
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

FPE_INTOVF_TRAP
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

`size_t **__fpending** (FILE *stream)`
`stdio_ext.h` (GNU): *Tamponlama Çeşidinin Seçimi* (sayfa: 294).

FPE_SUBRNG_TRAP
`signal.h` (BSD): *Yazılım Hatalarının Sinyalleri* (sayfa: 604).

`int **FP_ILOGB0**`
`math.h` (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`int **FP_ILOGBNAN**`

stdio.h (GNU): *Blok Girişi ve Çıkışı* (sayfa: 254).

`int __fwriting (FILE *stream)`
stdio_ext.h (GNU): *Akımların Açılması* (sayfa: 238).

F_WRLCK
fcntl.h (POSIX.1): *Dosya Kilitleri* (sayfa: 346).

`int fwscanf (FILE *stream, const wchar_t *template, ...)`
wchar.h (ISO): *Biçimli Girdi İşlevleri* (sayfa: 284).

B.7. G

`double gamma (double x)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`float gammaf (float x)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`long double gammal (long double x)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`void (*__gconv_end_fct) (struct gconv_step *)`
gconv.h (GNU): *glibc iconv Gerçeklemesi* (sayfa: 152).

`int (*__gconv_fct) (struct __gconv_step *, struct __gconv_step_data *, const char **, const char *, size_t *, int)`
gconv.h (GNU): *glibc iconv Gerçeklemesi* (sayfa: 152).

`int (*__gconv_init_fct) (struct __gconv_step *)`
gconv.h (GNU): *glibc iconv Gerçeklemesi* (sayfa: 152).

`char * gcvt (double value, int ndigit, char *buf)`
stdlib.h (SVID, Unix98): *Eski Moda System V Sayıdan Dizgeye Dönüşüm İşlevleri* (sayfa: 534).

`long int get_avphys_pages (void)`
sys/sysinfo.h (GNU): *Bellek Parametrelerinin Sorgulanması* (sayfa: 590).

`int getc (FILE *stream)`
stdio.h (ISO): *Karakter Girdilerinin Alınması* (sayfa: 248).

`int getchar (void)`
stdio.h (ISO): *Karakter Girdilerinin Alınması* (sayfa: 248).

`int getchar_unlocked (void)`
stdio.h (POSIX): *Karakter Girdilerinin Alınması* (sayfa: 248).

`int getcontext (ucontext_t *ucp)`
ucontext.h (SVID): *Bütünsel Bağlam Denetimi* (sayfa: 596).

`int getc_unlocked (FILE *stream)`
stdio.h (POSIX): *Karakter Girdilerinin Alınması* (sayfa: 248).

`char * get_current_dir_name (void)`
unistd.h (GNU): *Çalışma dizini* (sayfa: 351).

`int` **`getgrnam_r`** `(const char *`*`name`*`, struct group *`*`result_buf`*`, char *`*`buffer`*`, size_t` *`buflen`*`, struct group **`*`result`*`)`
`grp.h` (POSIX.1c): *Bir Grup Hakkında Bilgi Alınması* (sayfa: 763).

`int` **`getgrouplist`** `(const char *`*`user`*`, gid_t` *`group`*`, gid_t *`*`groups`*`, int *`*`ngroups`*`)`
`grp.h` (BSD): *Grup Kimliğinin Belirtilmesi* (sayfa: 746).

`int` **`getgroups`** `(int` *`count`*`, gid_t *`*`groups`*`)`
`unistd.h` (POSIX.1): *Bir Sürecin Aidiyetinin Okunması* (sayfa: 744).

`struct hostent *` **`gethostbyaddr`** `(const char *`*`addr`*`, size_t` *`length`*`, int` *`format`*`)`
`netdb.h` (BSD): *Konak İsimleri* (sayfa: 412).

`int` **`gethostbyaddr_r`** `(const char *`*`addr`*`, size_t` *`length`*`, int` *`format`*`, struct hostent *restrict` *`result_buf`*`, char *restrict` *`buf`*`, size_t` *`buflen`*`, struct hostent **restrict` *`result`*`, int *restrict` *`h_errnop`*`)`
`netdb.h` (GNU): *Konak İsimleri* (sayfa: 412).

`struct hostent *` **`gethostbyname`** `(const char *`*`name`*`)`
`netdb.h` (BSD): *Konak İsimleri* (sayfa: 412).

`struct hostent *` **`gethostbyname2`** `(const char *`*`name`*`, int` *`af`*`)`
`netdb.h` (IPv6 Basic API): *Konak İsimleri* (sayfa: 412).

`int` **`gethostbyname2_r`** `(const char *`*`name`*`, int` *`af`*`, struct hostent *restrict` *`result_buf`*`, char *restrict` *`buf`*`, size_t` *`buflen`*`, struct hostent **restrict` *`result`*`, int *restrict` *`h_errnop`*`)`
`netdb.h` (GNU): *Konak İsimleri* (sayfa: 412).

`int` **`gethostbyname_r`** `(const char *restrict` *`name`*`, struct hostent *restrict` *`result_buf`*`, char *restrict` *`buf`*`, size_t` *`buflen`*`, struct hostent **restrict` *`result`*`, int *restrict` *`h_errnop`*`)`
`netdb.h` (GNU): *Konak İsimleri* (sayfa: 412).

`struct hostent *` **`gethostent`** `(void)`
`netdb.h` (BSD): *Konak İsimleri* (sayfa: 412).

`long int` **`gethostid`** `(void)`
`unistd.h` (BSD): *Konak İsimlendirmesi* (sayfa: 769).

`int` **`gethostname`** `(char *`*`name`*`, size_t` *`size`*`)`
`unistd.h` (BSD): *Konak İsimlendirmesi* (sayfa: 769).

`int` **`getitimer`** `(int` *`which`*`, struct itimerval *`*`old`*`)`
`sys/time.h` (BSD): *Bir Alarmın Ayarlanması* (sayfa: 568).

`ssize_t` **`getline`** `(char **`*`lineptr`*`, size_t *`*`n`*`, FILE *`*`stream`*`)`
`stdio.h` (GNU): *Satır Yönlenimli Girdi* (sayfa: 250).

`int` **`getloadavg`** `(double` *`loadavg`*`[], int` *`nelem`*`)`
`stdlib.h` (BSD): *İşlemci Özkaynakları* (sayfa: 591).

`char *` **`getlogin`** `(void)`
`unistd.h` (POSIX.1): *Oturumu Açan Kim?* (sayfa: 752).

`struct mntent *` **`getmntent`** `(FILE *`*`stream`*`)`

`struct servent * getservbyname (const char *name, const char *proto)`
netdb.h (BSD): *Servis Veritabanı* (sayfa: 415).

`struct servent * getservbyport (int port, const char *proto)`
netdb.h (BSD): *Servis Veritabanı* (sayfa: 415).

`struct servent * getservent (void)`
netdb.h (BSD): *Servis Veritabanı* (sayfa: 415).

`pid_t getsid (pid_t pid)`
unistd.h (SVID): *Süreç Grubu İşlevleri* (sayfa: 729).

`int getsockname (int socket, struct sockaddr *addr, socklen_t *length–ptr)`
sys/socket.h (BSD): *Adresin Okunması* (sayfa: 403).

`int getsockopt (int socket, int level, int optname, void *optval, socklen_t *optlen–ptr)`
sys/socket.h (BSD): *Soket Seçenek İşlevleri* (sayfa: 438).

`int getsubopt (char **optionp, const char* const *tokens, char **valuep)`
stdlib.h (stdlib.h): *Suboptions Example* (sayfa: 674).

`char * gettext (const char *msgid)`
libintl.h (GNU): *gettext ile Çeviri* (sayfa: 190).

`int gettimeofday (struct timeval *tp, struct timezone *tzp)`
sys/time.h (BSD): *Yüksek Çözünürlüklü Zaman* (sayfa: 543).

`uid_t getuid (void)`
unistd.h (POSIX.1): *Bir Sürecin Aidiyetinin Okunması* (sayfa: 744).

`mode_t getumask (void)`
sys/stat.h (GNU): *Dosya İzinlerinin Atanması* (sayfa: 380).

`struct utmp * getutent (void)`
utmp.h (SVID): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

`int getutent_r (struct utmp *buffer, struct utmp **result)`
utmp.h (GNU): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

`struct utmp * getutid (const struct utmp *id)`
utmp.h (SVID): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

`int getutid_r (const struct utmp *id, struct utmp *buffer, struct utmp **result)`
utmp.h (GNU): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

`struct utmp * getutline (const struct utmp *line)`
utmp.h (SVID): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

`int getutline_r (const struct utmp *line, struct utmp *buffer, struct utmp **result)`
utmp.h (GNU): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

`int getutmp (const struct utmpx *utmpx, struct utmp *utmp)`
utmp.h (GNU): *XPG Kullanıcı Hesapları Veritabanı İşlevleri* (sayfa: 757).

(GNU): *Özellik Sınama Makroları* (sayfa: 25).

`int` **`grantpt`** `(int` *filedes*`)`
`stdlib.h` (SVID, XPG4.2): *Uçbirimsilerin Ayrılması* (sayfa: 464).

`int` **`gsignal`** `(int` *signum*`)`
`signal.h` (SVID): *Kendine Sinyal Gönderme* (sayfa: 627).

`int` **`gtty`** `(int` *filedes*`, struct sgttyb *`*attributes*`)`
`sgtty.h` (BSD): *BSD Uçbirim Kipleri* (sayfa: 460).

B.8. H

`char *` **`hasmntopt`** `(const struct mntent *`*mnt*`, const char *`*opt*`)`
`mntent.h` (BSD): *mtab* (sayfa: 775).

`int` **`hcreate`** `(size_t` *nel*`)`
`search.h` (SVID): *İsim–Değer Çiftleri ile Arama İşlevi* (sayfa: 208).

`int` **`hcreate_r`** `(size_t` *nel*`, struct hsearch_data *`*htab*`)`
`search.h` (GNU): *İsim–Değer Çiftleri ile Arama İşlevi* (sayfa: 208).

`void` **`hdestroy`** `(void)`
`search.h` (SVID): *İsim–Değer Çiftleri ile Arama İşlevi* (sayfa: 208).

`void` **`hdestroy_r`** `(struct hsearch_data *`*htab*`)`
`search.h` (GNU): *İsim–Değer Çiftleri ile Arama İşlevi* (sayfa: 208).

`HOST_NOT_FOUND`
`netdb.h` (BSD): *Konak İsimleri* (sayfa: 412).

`ENTRY *` **`hsearch`** `(ENTRY` *item*`, ACTION` *action*`)`
`search.h` (SVID): *İsim–Değer Çiftleri ile Arama İşlevi* (sayfa: 208).

`int` **`hsearch_r`** `(ENTRY` *item*`, ACTION` *action*`, ENTRY **`*retval*`, struct hsearch_data *`*htab*`)`
`search.h` (GNU): *İsim–Değer Çiftleri ile Arama İşlevi* (sayfa: 208).

`uint32_t` **`htonl`** `(uint32_t` *hostlong*`)`
`netinet/in.h` (BSD): *Bayt Sırası Dönüşümü* (sayfa: 417).

`uint16_t` **`htons`** `(uint16_t` *hostshort*`)`
`netinet/in.h` (BSD): *Bayt Sırası Dönüşümü* (sayfa: 417).

`double` **`HUGE_VAL`**
`math.h` (ISO): *Hataların Matematiksel İşlevlerce Raporlanması* (sayfa: 515).

`float` **`HUGE_VALF`**
`math.h` (ISO): *Hataların Matematiksel İşlevlerce Raporlanması* (sayfa: 515).

`long double` **`HUGE_VALL`**
`math.h` (ISO): *Hataların Matematiksel İşlevlerce Raporlanması* (sayfa: 515).

`tcflag_t` **`HUPCL`**
`termios.h` (POSIX.1): *Denetim Kipleri* (sayfa: 449).

`double` **hypot** (double *x*, double *y*)
    `math.h` (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **hypotf** (float *x*, float *y*)
    `math.h` (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`long double` **hypotl** (long double *x*, long double *y*)
    `math.h` (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

B.9. I

`tcflag_t` **ICANON**
    `termios.h` (POSIX.1): *Yerel Kipler* (sayfa: 451).

`size_t` **iconv** (iconv_t *cd*, char **inbuf*, size_t **inbytesleft*, char **outbuf*, size_t **outbytesleft*)
    `iconv.h` (XPG2): *Soysal Dönüşüm Arayüzü* (sayfa: 146).

`int` **iconv_close** (iconv_t *cd*)
    `iconv.h` (XPG2): *Soysal Dönüşüm Arayüzü* (sayfa: 146).

`iconv_t` **iconv_open** (const char **tocode*, const char **fromcode*)
    `iconv.h` (XPG2): *Soysal Dönüşüm Arayüzü* (sayfa: 146).

iconv_t
    `iconv.h` (XPG2): *Soysal Dönüşüm Arayüzü* (sayfa: 146).

`tcflag_t` **ICRNL**
    `termios.h` (POSIX.1): *Girdi Kipleri* (sayfa: 447).

`tcflag_t` **IEXTEN**
    `termios.h` (POSIX.1): *Yerel Kipler* (sayfa: 451).

`void` **if_freenameindex** (struct if_nameindex *ptr)
    `net/if.h` (IPv6 basic API): *Arayüz İsimlendirmesi* (sayfa: 403).

`char *` **if_indextoname** (unsigned int ifindex, char *ifname)
    `net/if.h` (IPv6 basic API): *Arayüz İsimlendirmesi* (sayfa: 403).

`struct if_nameindex *` **if_nameindex** (void)
    `net/if.h` (IPv6 basic API): *Arayüz İsimlendirmesi* (sayfa: 403).

`unsigned int` **if_nametoindex** (const char *ifname)
    `net/if.h` (IPv6 basic API): *Arayüz İsimlendirmesi* (sayfa: 403).

`size_t` **IFNAMSIZ**
    `net/if.h` (net/if.h): *Arayüz İsimlendirmesi* (sayfa: 403).

`int` **IFTODT** (mode_t *mode*)
    `dirent.h` (BSD): *Dizin Girdileri* (sayfa: 353).

`tcflag_t` **IGNBRK**
    `termios.h` (POSIX.1): *Girdi Kipleri* (sayfa: 447).

`tcflag_t` **IGNCR**
    `termios.h` (POSIX.1): *Girdi Kipleri* (sayfa: 447).

wctype.h (GNU): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswcntrl** (wint_t *wc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswctype** (wint_t *wc*, wctype_t *desc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswdigit** (wint_t *wc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswgraph** (wint_t *wc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswlower** (wint_t *wc*)
ctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswprint** (wint_t *wc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswpunct** (wint_t *wc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswspace** (wint_t *wc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswupper** (wint_t *wc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **iswxdigit** (wint_t *wc*)
wctype.h (ISO): *Geniş Karakterlerin Sınıflandırılması* (sayfa: 84).

`int` **isxdigit** (int *c*)
ctype.h (ISO): *Karakterlerin Sınıflandırılması* (sayfa: 82).

ITIMER_PROF
sys/time.h (BSD): *Bir Alarmın Ayarlanması* (sayfa: 568).

ITIMER_REAL
sys/time.h (BSD): *Bir Alarmın Ayarlanması* (sayfa: 568).

ITIMER_VIRTUAL
sys/time.h (BSD): *Bir Alarmın Ayarlanması* (sayfa: 568).

`tcflag_t` **IXANY**
termios.h (BSD): *Girdi Kipleri* (sayfa: 447).

`tcflag_t` **IXOFF**
termios.h (POSIX.1): *Girdi Kipleri* (sayfa: 447).

`tcflag_t` **IXON**
termios.h (POSIX.1): *Girdi Kipleri* (sayfa: 447).

B.10. J

`double` **j0** (double *x*)

math.h (SVID): *Özel İşlevler* (sayfa: 484).

`float` **`j0f`** `(float` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`long double` **`j0l`** `(long double` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`double` **`j1`** `(double` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`float` **`j1f`** `(float` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`long double` **`j1l`** `(long double` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`jmp_buf`
setjmp.h (ISO): *Yerel Olmayan Çıkışların Ayrıntıları* (sayfa: 594).

`double` **`jn`** `(int n, double` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`float` **`jnf`** `(int n, float` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`long double` **`jnl`** `(int n, long double` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`long int` **`jrand48`** `(unsigned short int` *xsubi*`[3])`
stdlib.h (SVID): *SVID Rasgele Sayı İşlevleri* (sayfa: 500).

`int` **`jrand48_r`** `(unsigned short int` *xsubi*`[3], struct drand48_data *`*buffer*`, long int *`*result*`)`
stdlib.h (GNU): *SVID Rasgele Sayı İşlevleri* (sayfa: 500).

B.11. K

`int` **`kill`** `(pid_t` *pid*`, int` *signum*`)`
signal.h (POSIX.1): *Başka Bir Sürece Sinyal Gönderme* (sayfa: 628).

`int` **`killpg`** `(int` *pgid*`, int` *signum*`)`
signal.h (BSD): *Başka Bir Sürece Sinyal Gönderme* (sayfa: 628).

B.12. L

`char *` **`l64a`** `(long int` *n*`)`
stdlib.h (XPG): *İkilik Verinin Kodlanması* (sayfa: 120).

`long int` **`labs`** `(long int` *number*`)`
stdlib.h (ISO): *Mutlak Değer* (sayfa: 519).

`LANG`
locale.h (ISO): *Yerellerin Etkilediği Eylemlerin Sınıflandırılması* (sayfa: 165).

math.h (ISO): *Yuvarlama İşlevleri* (sayfa: 521).

`long long int` **`llroundl`** `(long double` *x*`)`
math.h (ISO): *Yuvarlama İşlevleri* (sayfa: 521).

`struct lconv *` **`localeconv`** `(void)`
locale.h (ISO): *localeconv: Taşınabilirdir ama ...* (sayfa: 168).

`struct tm *` **`localtime`** `(const time_t *`*time*`)`
time.h (ISO): *Yerel Zaman* (sayfa: 545).

`struct tm *` **`localtime_r`** `(const time_t *`*time*`, struct tm *`*resultp*`)`
time.h (POSIX.1c): *Yerel Zaman* (sayfa: 545).

`double` **`log`** `(double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`double` **`log10`** `(double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **`log10f`** `(float` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`long double` **`log10l`** `(long double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`double` **`log1p`** `(double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **`log1pf`** `(float` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`long double` **`log1pl`** `(long double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`double` **`log2`** `(double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **`log2f`** `(float` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`long double` **`log2l`** `(long double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`double` **`logb`** `(double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **`logbf`** `(float` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`long double` **`logbl`** `(long double` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`float` **`logf`** `(float` *x*`)`
math.h (ISO): *Üstel ve Logaritmik İşlevler* (sayfa: 479).

`void` **`login`** `(const struct utmp *`*entry*`)`

math.h (ISO): *Yuvarlama İşlevleri* (sayfa: 521).

`void * lsearch (const void *key, void *base, size_t *nmemb, size_t size, comparison_fn_t compar)`
search.h (SVID): *Dizi Arama İşlevleri* (sayfa: 203).

`off_t lseek (int filedes, off_t offset, int whence)`
unistd.h (POSIX.1): *Dosya Konumu İlkeli* (sayfa: 313).

`off64_t lseek64 (int filedes, off64_t offset, int whence)`
unistd.h (Unix98): *Dosya Konumu İlkeli* (sayfa: 313).

L_SET
sys/file.h (BSD): *Dosyalarda Konumlama* (sayfa: 288).

`int lstat (const char *filename, struct stat *buf)`
sys/stat.h (BSD): *Bir Dosyanın Özniteliklerinin Okunması* (sayfa: 374).

`int lstat64 (const char *filename, struct stat64 *buf)`
sys/stat.h (Unix98): *Bir Dosyanın Özniteliklerinin Okunması* (sayfa: 374).

`int L_tmpnam`
stdio.h (ISO): *Geçici Dosyalar* (sayfa: 389).

`int lutimes (const char *filename, struct timeval tvp[2])`
sys/time.h (BSD): *Dosya Zamanları* (sayfa: 383).

L_XTND
sys/file.h (BSD): *Dosyalarda Konumlama* (sayfa: 288).

B.13. M

`int madvise (void *addr, size_t length, int advice)`
sys/mman.h (POSIX): *Bellek Eşlemli G/Ç* (sayfa: 319).

`void makecontext (ucontext_t *ucp, void (*func) (void), int argc, ...)`
ucontext.h (SVID): *Bütünsel Bağlam Denetimi* (sayfa: 596).

`struct mallinfo mallinfo (void)`
malloc.h (SVID): *malloc ile Bellek Ayırma İstatistikleri* (sayfa: 59).

`void * malloc (size_t size)`
malloc.h, stdlib.h (ISO): *Özdevimli Olarak Basit Bellek Ayırma* (sayfa: 50).

__malloc_hook
malloc.h (GNU): *Bellek Ayırma Kancaları* (sayfa: 57).

__malloc_initialize_hook
malloc.h (GNU): *Bellek Ayırma Kancaları* (sayfa: 57).

`int MAX_CANON`
limits.h (POSIX.1): *Dosya Sistemi Kapasite Sınırları* (sayfa: 795).

`int MAX_INPUT`
limits.h (POSIX.1): *Dosya Sistemi Kapasite Sınırları* (sayfa: 795).

`int MAXNAMLEN`

`void * memchr (const void *block, int c, size_t size)`
`string.h` (ISO): *Arama İşlevleri* (sayfa: 111).

`int memcmp (const void *a1, const void *a2, size_t size)`
`string.h` (ISO): *Dizi/Dizge Karşılaştırması* (sayfa: 104).

`void * memcpy (void *restrict to, const void *restrict from, size_t size)`
`string.h` (ISO): *Kopyalama ve Birleştirme* (sayfa: 94).

`void * memfrob (void *mem, size_t length)`
`string.h` (GNU): *Bayağı Şifreleme* (sayfa: 119).

`void * memmem (const void *haystack, size_t haystack–len,const void *needle, size_t needle–len)`
`string.h` (GNU): *Arama İşlevleri* (sayfa: 111).

`void * memmove (void *to, const void *from, size_t size)`
`string.h` (ISO): *Kopyalama ve Birleştirme* (sayfa: 94).

`void * mempcpy (void *restrict to, const void *restrict from, size_t size)`
`string.h` (GNU): *Kopyalama ve Birleştirme* (sayfa: 94).

`void * memrchr (const void *block, int c, size_t size)`
`string.h` (GNU): *Arama İşlevleri* (sayfa: 111).

`void * memset (void *block, int c, size_t size)`
`string.h` (ISO): *Kopyalama ve Birleştirme* (sayfa: 94).

`int mkdir (const char *filename, mode_t mode)`
`sys/stat.h` (POSIX.1): *Dizinlerin Oluşturulması* (sayfa: 370).

`char * mkdtemp (char *template)`
`stdlib.h` (BSD): *Geçici Dosyalar* (sayfa: 389).

`int mkfifo (const char *filename, mode_t mode)`
`sys/stat.h` (POSIX.1): *FIFO Özel Dosyaları* (sayfa: 396).

`int mknod (const char *filename, int mode, int dev)`
`sys/stat.h` (BSD): *Özel Dosyaların Oluşturulması* (sayfa: 388).

`int mkstemp (char *template)`
`stdlib.h` (BSD): *Geçici Dosyalar* (sayfa: 389).

`char * mktemp (char *template)`
`stdlib.h` (Unix): *Geçici Dosyalar* (sayfa: 389).

`time_t mktime (struct tm *brokentime)`
`time.h` (ISO): *Yerel Zaman* (sayfa: 545).

`int mlock (const void *addr, size_t len)`
`sys/mman.h` (POSIX.1b): *Sayfaları Kilitleyen ve Kilitlerini Açan İşlevler* (sayfa: 79).

`int mlockall (int flags)`
`sys/mman.h` (POSIX.1b): *Sayfaları Kilitleyen ve Kilitlerini Açan İşlevler* (sayfa: 79).

`void * mmap (void *address, size_t length,int protect, int flags, int filedes, off_t offset)`

B.14. N

ftw.h (XPG4.2): *Dizin Ağaçlarıyla Çalışma* (sayfa: 361).

`int` **nftw64** `(const char *`*filename*`, __nftw64_func_t` *func*`, int` *descriptors*`, int` *flag*`)`
ftw.h (Unix98): *Dizin Ağaçlarıyla Çalışma* (sayfa: 361).

__nftw64_func_t
ftw.h (GNU): *Dizin Ağaçlarıyla Çalışma* (sayfa: 361).

__nftw_func_t
ftw.h (GNU): *Dizin Ağaçlarıyla Çalışma* (sayfa: 361).

`char *` **ngettext** `(const char *`*msgid1*`, const char *`*msgid2*`, unsigned long int` *n*`)`
libintl.h (GNU): *Gelişkin gettext işlevleri* (sayfa: 193).

`int` **NGROUPS_MAX**
limits.h (POSIX.1): *Genel Sınırlar* (sayfa: 784).

`int` **nice** `(int` *increment*`)`
unistd.h (BSD): *Geleneksel Zamanlama İşlevleri* (sayfa: 585).

nlink_t
sys/types.h (POSIX.1): *Dosya Özniteliklerinin Anlamları* (sayfa: 371).

`char *` **nl_langinfo** `(nl_item` *item*`)`
langinfo.h (XOPEN): *Yerel Verisine Noktasal Erişim* (sayfa: 171).

NO_ADDRESS
netdb.h (BSD): *Konak İsimleri* (sayfa: 412).

`tcflag_t` **NOFLSH**
termios.h (POSIX.1): *Yerel Kipler* (sayfa: 451).

`tcflag_t` **NOKERNINFO**
termios.h (BSD): *Yerel Kipler* (sayfa: 451).

NO_RECOVERY
netdb.h (BSD): *Konak İsimleri* (sayfa: 412).

`long int` **nrand48** `(unsigned short int` *xsubi*`[3])`
stdlib.h (SVID): *SVID Rasgele Sayı İşlevleri* (sayfa: 500).

`int` **nrand48_r** `(unsigned short int` *xsubi*`[3], struct drand48_data *`*buffer*`, long int *`*result*`)`
stdlib.h (GNU): *SVID Rasgele Sayı İşlevleri* (sayfa: 500).

`int` **NSIG**
signal.h (BSD): *Standart Sinyaller* (sayfa: 604).

`uint32_t` **ntohl** `(uint32_t` *netlong*`)`
netinet/in.h (BSD): *Bayt Sırası Dönüşümü* (sayfa: 417).

`uint16_t` **ntohs** `(uint16_t` *netshort*`)`
netinet/in.h (BSD): *Bayt Sırası Dönüşümü* (sayfa: 417).

`int` **ntp_adjtime** `(struct timex *`*tptr*`)`

sys/timex.h (GNU): *Yüksek Doğrulukta Saat* (sayfa: 547).

`int` **ntp_gettime** `(struct ntptimeval *`*tptr*`)`
sys/timex.h (GNU): *Yüksek Doğrulukta Saat* (sayfa: 547).

`void *` **NULL**
stddef.h (ISO): *Boş Gösterici Sabiti* (sayfa: 820).

B.15. O

`int` **O_ACCMODE**
fcntl.h (POSIX.1): *Dosya Erişim Kipleri* (sayfa: 342).

`int` **O_APPEND**
fcntl.h (POSIX.1): *G/Ç İşlem Kipleri* (sayfa: 344).

`int` **O_ASYNC**
fcntl.h (BSD): *G/Ç İşlem Kipleri* (sayfa: 344).

`void` **obstack_1grow** `(struct obstack *`*obstack–ptr*`, char` *c*`)`
obstack.h (GNU): *Büyüyen Nesneler* (sayfa: 69).

`void` **obstack_1grow_fast** `(struct obstack *`*obstack–ptr*`, char` *c*`)`
obstack.h (GNU): *Çok Hızlı Büyüyen Nesneler* (sayfa: 70).

`int` **obstack_alignment_mask** `(struct obstack *`*obstack–ptr*`)`
obstack.h (GNU): *Yığınaktaki Verinin Adreslenmesi* (sayfa: 72).

`void *` **obstack_alloc** `(struct obstack *`*obstack–ptr*`, int` *size*`)`
obstack.h (GNU): *Bir Yığınağa Nesne Eklenmesi* (sayfa: 66).

obstack_alloc_failed_handler
obstack.h (GNU): *Yığınakları Kullanıma Hazırlama* (sayfa: 65).

`void *` **obstack_base** `(struct obstack *`*obstack–ptr*`)`
obstack.h (GNU): *Bir Yığınağın Durumu* (sayfa: 71).

`void` **obstack_blank** `(struct obstack *`*obstack–ptr*`, int` *size*`)`
obstack.h (GNU): *Büyüyen Nesneler* (sayfa: 69).

`void` **obstack_blank_fast** `(struct obstack *`*obstack–ptr*`, int` *size*`)`
obstack.h (GNU): *Çok Hızlı Büyüyen Nesneler* (sayfa: 70).

`int` **obstack_chunk_size** `(struct obstack *`*obstack–ptr*`)`
obstack.h (GNU): *Yığınak Tomarları* (sayfa: 72).

`void *` **obstack_copy** `(struct obstack *`*obstack–ptr*`, void *`*address*`, int` *size*`)`
obstack.h (GNU): *Bir Yığınağa Nesne Eklenmesi* (sayfa: 66).

`void *` **obstack_copy0** `(struct obstack *`*obstack–ptr*`, void *`*address*`, int` *size*`)`
obstack.h (GNU): *Bir Yığınağa Nesne Eklenmesi* (sayfa: 66).

`void *` **obstack_finish** `(struct obstack *`*obstack–ptr*`)`
obstack.h (GNU): *Büyüyen Nesneler* (sayfa: 69).

`void` **obstack_free** `(struct obstack *`*obstack–ptr*`, void *`*object*`)`

`fcntl.h` (BSD): *G/Ç İşlem Kipleri* (sayfa: 344).

`int` **O_TRUNC**
`fcntl.h` (POSIX.1): *Açış Anı Seçenekleri* (sayfa: 343).

`int` **O_WRITE**
`fcntl.h` (GNU): *Dosya Erişim Kipleri* (sayfa: 342).

`int` **O_WRONLY**
`fcntl.h` (POSIX.1): *Dosya Erişim Kipleri* (sayfa: 342).

`tcflag_t` **OXTABS**
`termios.h` (BSD): *Çıktı Kipleri* (sayfa: 449).

B.16. P

PA_CHAR
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_DOUBLE
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_FLAG_LONG
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_FLAG_LONG_DOUBLE
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_FLAG_LONG_LONG
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

`int` **PA_FLAG_MASK**
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_FLAG_PTR
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_FLAG_SHORT
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_FLOAT
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_INT
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_LAST
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

PA_POINTER
`printf.h` (GNU): *Bir Şablon Dizgesinin Çözümlenmesi* (sayfa: 269).

`tcflag_t` **PARENB**
`termios.h` (POSIX.1): *Denetim Kipleri* (sayfa: 449).

`tcflag_t` **PARMRK**

stdio.h (SVID): *Karakterlerin ve Satırların Basit Çıktılanması* (sayfa: 246).

`wint_t` **putwc** (`wchar_t` *wc*, `FILE *`*stream*)
wchar.h (ISO): *Karakterlerin ve Satırların Basit Çıktılanması* (sayfa: 246).

`wint_t` **putwchar** (`wchar_t` *wc*)
wchar.h (ISO): *Karakterlerin ve Satırların Basit Çıktılanması* (sayfa: 246).

`wint_t` **putwchar_unlocked** (`wchar_t` *wc*)
wchar.h (GNU): *Karakterlerin ve Satırların Basit Çıktılanması* (sayfa: 246).

`wint_t` **putwc_unlocked** (`wchar_t` *wc*, `FILE *`*stream*)
wchar.h (GNU): *Karakterlerin ve Satırların Basit Çıktılanması* (sayfa: 246).

`ssize_t` **pwrite** (`int` *filedes*, `const void *`*buffer*, `size_t` *size*, `off_t` *offset*)
unistd.h (Unix98): *Girdi ve Çıktı İlkelleri* (sayfa: 308).

`ssize_t` **pwrite64** (`int` *filedes*, `const void *`*buffer*, `size_t` *size*, `off64_t` *offset*)
unistd.h (Unix98): *Girdi ve Çıktı İlkelleri* (sayfa: 308).

B.17. Q

`char *` **qecvt** (`long double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*)
stdlib.h (GNU): *Eski Moda System V Sayıdan Dizgeye Dönüşüm İşlevleri* (sayfa: 534).

`int` **qecvt_r** (`long double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*, `char *`*buf*, `size_t` *len*)
stdlib.h (GNU): *Eski Moda System V Sayıdan Dizgeye Dönüşüm İşlevleri* (sayfa: 534).

`char *` **qfcvt** (`long double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*)
stdlib.h (GNU): *Eski Moda System V Sayıdan Dizgeye Dönüşüm İşlevleri* (sayfa: 534).

`int` **qfcvt_r** (`long double` *value*, `int` *ndigit*, `int *`*decpt*, `int *`*neg*, `char *`*buf*, `size_t` *len*)
stdlib.h (GNU): *Eski Moda System V Sayıdan Dizgeye Dönüşüm İşlevleri* (sayfa: 534).

`char *` **qgcvt** (`long double` *value*, `int` *ndigit*, `char *`*buf*)
stdlib.h (GNU): *Eski Moda System V Sayıdan Dizgeye Dönüşüm İşlevleri* (sayfa: 534).

`void` **qsort** (`void *`*array*, `size_t` *count*, `size_t` *size*, `comparison_fn_t` *compare*)
stdlib.h (ISO): *Dizi Sıralama İşlevi* (sayfa: 204).

B.18. R

`int` **raise** (`int` *signum*)
signal.h (ISO): *Kendine Sinyal Gönderme* (sayfa: 627).

`int` **rand** (`void`)
stdlib.h (ISO): *ISO C Rasgele Sayı İşlevleri* (sayfa: 498).

`int` **RAND_MAX**
stdlib.h (ISO): *ISO C Rasgele Sayı İşlevleri* (sayfa: 498).

`long int` **random** (`void`)

REG_BADBR
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_BADPAT
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_BADRPT
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

`int` **regcomp** `(regex_t *restrict` *compiled*`, const char *restrict` *pattern*`, int` *cflags*`)`
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_EBRACE
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_EBRACK
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_ECOLLATE
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_ECTYPE
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_EESCAPE
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_EPAREN
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_ERANGE
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

`size_t` **regerror** `(int` *errcode*`, const regex_t *restrict` *compiled*`, char *restrict` *buffer*`, size_t` *length*`)`
regex.h (POSIX.2): *POSIX Şablonunun Temizlenmesi* (sayfa: 224).

REG_ESPACE
regex.h (POSIX.2): *Derlenmiş POSIX Düzenli İfadelerinin Eşleştirilmesi* (sayfa: 222).

REG_ESPACE
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_ESUBREG
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

`int` **regexec** `(const regex_t *restrict` *compiled*`, const char *restrict` *string*`, size_t` *nmatch*`, regmatch_t` *matchptr*`[restrict], int` *eflags*`)`
regex.h (POSIX.2): *Derlenmiş POSIX Düzenli İfadelerinin Eşleştirilmesi* (sayfa: 222).

regex_t
regex.h (POSIX.2): *POSIX Düzenli İfadelerinin Derlenmesi* (sayfa: 220).

REG_EXTENDED
regex.h (POSIX.2): *POSIX Düzenli İfade Seçenekleri* (sayfa: 222).

`stdlib.h` (stdlib.h): *Evet/Hayır Yanıtları* (sayfa: 179).

RUN_LVL
`utmp.h` (SVID): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

RUN_LVL
`utmpx.h` (XPG4.2): *XPG Kullanıcı Hesapları Veritabanı İşlevleri* (sayfa: 757).

RUSAGE_CHILDREN
`sys/resource.h` (BSD): *Özkaynak Kullanımı* (sayfa: 572).

RUSAGE_SELF
`sys/resource.h` (BSD): *Özkaynak Kullanımı* (sayfa: 572).

B.19. S

`int` **SA_NOCLDSTOP**
`signal.h` (POSIX.1): *sigaction Seçenekleri* (sayfa: 616).

`int` **SA_ONSTACK**
`signal.h` (BSD): *sigaction Seçenekleri* (sayfa: 616).

`int` **SA_RESTART**
`signal.h` (BSD): *sigaction Seçenekleri* (sayfa: 616).

`void *`**sbrk** `(ptrdiff_t` *delta*`)`
`unistd.h` (BSD): *Veri Bölütünün Boyunun Değiştirilmesi* (sayfa: 77).

_SC_2_C_DEV
`unistd.h` (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_2_FORT_DEV
`unistd.h` (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_2_FORT_RUN
`unistd.h` (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_2_LOCALEDEF
`unistd.h` (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_2_SW_DEV
`unistd.h` (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_2_VERSION
`unistd.h` (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_AIO_LISTIO_MAX
`unistd.h` (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_AIO_MAX
`unistd.h` (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_AIO_PRIO_DELTA_MAX
`unistd.h` (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

`double` **scalb** `(double` *value*`, int` *exponent*`)`

_SC_BC_STRING_MAX
unistd.h (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_CHAR_BIT
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_CHARCLASS_NAME_MAX
unistd.h (GNU): *sysconf Parametreleri* (sayfa: 787).

_SC_CHAR_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_CHAR_MIN
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_CHILD_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_CLK_TCK
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_COLL_WEIGHTS_MAX
unistd.h (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_DELAYTIMER_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_EQUIV_CLASS_MAX
unistd.h (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_EXPR_NEST_MAX
unistd.h (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_FSYNC
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_GETGR_R_SIZE_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_GETPW_R_SIZE_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

SCHAR_MAX
limits.h (ISO): *Bir Tamsayı Türün Aralığı* (sayfa: 821).

SCHAR_MIN
limits.h (ISO): *Bir Tamsayı Türün Aralığı* (sayfa: 821).

`int` **sched_getparam** `(pid_t` *pid*`, const struct sched_param *`*param*`)`
sched.h (POSIX): *Temel Zamanlama İşlevleri* (sayfa: 581).

`int` **sched_get_priority_max** `(int *`*policy*`);`
sched.h (POSIX): *Temel Zamanlama İşlevleri* (sayfa: 581).

`int` **sched_get_priority_min** `(int *`*policy*`);`
sched.h (POSIX): *Temel Zamanlama İşlevleri* (sayfa: 581).

`int` **sched_getscheduler** `(pid_t` *pid*`)`

sched.h (POSIX): *Temel Zamanlama İşlevleri* (sayfa: 581).

`int` **sched_rr_get_interval** (pid_t *pid*, struct timespec **interval*)
sched.h (POSIX): *Temel Zamanlama İşlevleri* (sayfa: 581).

`int` **sched_getaffinity** (pid_t *pid*, size_t *cpusetsize*, cpu_set_t **cpuset*)
sched.h (GNU): *İşlemciler Arasında İcra Sınırlaması* (sayfa: 587).

`int` **sched_setaffinity** (pid_t *pid*, size_t *cpusetsize*, const cpu_set_t **cpuset*)
sched.h (GNU): *İşlemciler Arasında İcra Sınırlaması* (sayfa: 587).

`int` **sched_setparam** (pid_t *pid*, const struct sched_param **param*)
sched.h (POSIX): *Temel Zamanlama İşlevleri* (sayfa: 581).

`int` **sched_setscheduler** (pid_t *pid*, int *policy*, const struct sched_param **param*)
sched.h (POSIX): *Temel Zamanlama İşlevleri* (sayfa: 581).

`int` **sched_yield** (void)
sched.h (POSIX): *Temel Zamanlama İşlevleri* (sayfa: 581).

_SC_INT_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_INT_MIN
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_JOB_CONTROL
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_LINE_MAX
unistd.h (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_LOGIN_NAME_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_LONG_BIT
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_MAPPED_FILES
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_MB_LEN_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_MEMLOCK
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_MEMLOCK_RANGE
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_MEMORY_PROTECTION
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_MESSAGE_PASSING
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_MQ_OPEN_MAX

unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_MQ_PRIO_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_NGROUPS_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_NL_ARGMAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_NL_LANGMAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_NL_MSGMAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_NL_NMAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_NL_SETMAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_NL_TEXTMAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_NPROCESSORS_CONF
unistd.h (GNU): *sysconf Parametreleri* (sayfa: 787).

_SC_NPROCESSORS_ONLN
unistd.h (GNU): *sysconf Parametreleri* (sayfa: 787).

_SC_NZERO
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_OPEN_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_PAGESIZE
unistd.h (GNU): *sysconf Parametreleri* (sayfa: 787).

_SC_PHYS_PAGES
unistd.h (GNU): *sysconf Parametreleri* (sayfa: 787).

_SC_PII
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_INTERNET
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_INTERNET_DGRAM
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_INTERNET_STREAM
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_OSI

unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_OSI_CLTS
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_OSI_COTS
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_OSI_M
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_SOCKET
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PII_XTI
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_PRIORITIZED_IO
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_PRIORITY_SCHEDULING
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_REALTIME_SIGNALS
unistdh.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_RTSIG_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_SAVED_IDS
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_SCHAR_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_SCHAR_MIN
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_SELECT
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_SEMAPHORES
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_SEM_NSEMS_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_SEM_VALUE_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_SHARED_MEMORY_OBJECTS
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_SHRT_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_SHRT_MIN

unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_SIGQUEUE_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

SC_SSIZE_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_STREAM_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_SYNCHRONIZED_IO
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_ATTR_STACKADDR
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_ATTR_STACKSIZE
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_DESTRUCTOR_ITERATIONS
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_KEYS_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_PRIO_INHERIT
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_PRIO_PROTECT
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_PRIORITY_SCHEDULING
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_PROCESS_SHARED
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREADS
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_SAFE_FUNCTIONS
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_STACK_MIN
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_THREAD_THREADS_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_TIMER_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_TIMERS
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_T_IOV_MAX

unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_TTY_NAME_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_TZNAME_MAX
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_UCHAR_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_UINT_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_UIO_MAXIOV
unistd.h (POSIX.1g): *sysconf Parametreleri* (sayfa: 787).

_SC_ULONG_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_USHRT_MAX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_VERSION
unistd.h (POSIX.1): *sysconf Parametreleri* (sayfa: 787).

_SC_VERSION
unistd.h (POSIX.2): *sysconf Parametreleri* (sayfa: 787).

_SC_WORD_BIT
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_CRYPT
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_ENH_I18N
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_LEGACY
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_REALTIME
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_REALTIME_THREADS
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_SHM
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_UNIX
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_VERSION
unistd.h (X/Open): *sysconf Parametreleri* (sayfa: 787).

_SC_XOPEN_XCU_VERSION

unistd.h (POSIX.1): *Süreç Grubu İşlevleri* (sayfa: 729).

`int` **`setpgrp`** `(pid_t` *pid*`, pid_t` *pgid*`)`
unistd.h (BSD): *Süreç Grubu İşlevleri* (sayfa: 729).

`int` **`setpriority`** `(int` *class*`, int` *id*`, int` *niceval*`)`
sys/resource.h (BSD,POSIX): *Geleneksel Zamanlama İşlevleri* (sayfa: 585).

`void` **`setprotoent`** `(int` *stayopen*`)`
netdb.h (BSD): *Protokol Veritabanı* (sayfa: 417).

`void` **`setpwent`** `(void)`
pwd.h (SVID, BSD): *Kullanıcı Listesinin Taranması* (sayfa: 761).

`int` **`setregid`** `(gid_t` *rgid*`, gid_t` *egid*`)`
unistd.h (BSD): *Grup Kimliğinin Belirtilmesi* (sayfa: 746).

`int` **`setreuid`** `(uid_t` *ruid*`, uid_t` *euid*`)`
unistd.h (BSD): *Kullanıcı Kimliğinin Belirtilmesi* (sayfa: 745).

`int` **`setrlimit`** `(int` *resource*`, const struct rlimit *`*rlp*`)`
sys/resource.h (BSD): *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

`int` **`setrlimit64`** `(int` *resource*`, const struct rlimit64 *`*rlp*`)`
sys/resource.h (Unix98): *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

`void` **`setservent`** `(int` *stayopen*`)`
netdb.h (BSD): *Servis Veritabanı* (sayfa: 415).

`pid_t` **`setsid`** `(void)`
unistd.h (POSIX.1): *Süreç Grubu İşlevleri* (sayfa: 729).

`int` **`setsockopt`** `(int` *socket*`, int` *level*`, int` *optname*`, void *`*optval*`, socklen_t` *optlen*`)`
sys/socket.h (BSD): *Soket Seçenek İşlevleri* (sayfa: 438).

`void *` **`setstate`** `(void *`*state*`)`
stdlib.h (BSD): *BSD Rasgele Sayı İşlevleri* (sayfa: 499).

`int` **`setstate_r`** `(char *restrict` *statebuf*`, struct random_data *restrict` *buf*`)`
stdlib.h (GNU): *BSD Rasgele Sayı İşlevleri* (sayfa: 499).

`int` **`settimeofday`** `(const struct timeval *`*tp*`, const struct timezone *`*tzp*`)`
sys/time.h (BSD): *Yüksek Çözünürlüklü Zaman* (sayfa: 543).

`int` **`setuid`** `(uid_t` *newuid*`)`
unistd.h (POSIX.1): *Kullanıcı Kimliğinin Belirtilmesi* (sayfa: 745).

`void` **`setutent`** `(void)`
utmp.h (SVID): *Kullanıcı Hesapları Veritabanına Erişim* (sayfa: 752).

`void` **`setutxent`** `(void)`
utmpx.h (XPG4.2): *XPG Kullanıcı Hesapları Veritabanı İşlevleri* (sayfa: 757).

`int` **`setvbuf`** `(FILE *`*stream*`, char *`*buf*`, int` *mode*`, size_t` *size*`)`
stdio.h (ISO): *Tamponlama Çeşidinin Seçimi* (sayfa: 294).

`SHRT_MAX`

sys/stat.h (POSIX): *Bir Dosyanın Türünün Sınanması* (sayfa: 375).

`int` **S_ISCHR** (mode_t *m*)
sys/stat.h (POSIX): *Bir Dosyanın Türünün Sınanması* (sayfa: 375).

`int` **S_ISDIR** (mode_t *m*)
sys/stat.h (POSIX): *Bir Dosyanın Türünün Sınanması* (sayfa: 375).

`int` **S_ISFIFO** (mode_t *m*)
sys/stat.h (POSIX): *Bir Dosyanın Türünün Sınanması* (sayfa: 375).

S_ISGID
sys/stat.h (POSIX): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

`int` **S_ISLNK** (mode_t *m*)
sys/stat.h (GNU): *Bir Dosyanın Türünün Sınanması* (sayfa: 375).

`int` **S_ISREG** (mode_t *m*)
sys/stat.h (POSIX): *Bir Dosyanın Türünün Sınanması* (sayfa: 375).

`int` **S_ISSOCK** (mode_t *m*)
sys/stat.h (GNU): *Bir Dosyanın Türünün Sınanması* (sayfa: 375).

S_ISUID
sys/stat.h (POSIX): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

S_ISVTX
sys/stat.h (BSD): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

S_IWGRP
sys/stat.h (POSIX.1): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

S_IWOTH
sys/stat.h (POSIX.1): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

S_IWRITE
sys/stat.h (BSD): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

S_IWUSR
sys/stat.h (POSIX.1): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

S_IXGRP
sys/stat.h (POSIX.1): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

S_IXOTH
sys/stat.h (POSIX.1): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

S_IXUSR
sys/stat.h (POSIX.1): *Erişim İzinleri için Kip Bitleri* (sayfa: 378).

size_t
stddef.h (ISO): *Önemli Veri Türleri* (sayfa: 820).

`unsigned int` **sleep** (unsigned int *seconds*)
unistd.h (POSIX.1): *Uyku* (sayfa: 570).

`int` **snprintf** (char *s, size_t *size*, const char *template, ...)

string.h (GNU): *Kopyalama ve Birleştirme* (sayfa: 94).

`int` **STREAM_MAX**
limits.h (POSIX.1): *Genel Sınırlar* (sayfa: 784).

`char *` **strerror** (`int` *errnum*)
string.h (ISO): *Hata İletileri* (sayfa: 41).

`char *` **strerror_r** (`int` *errnum*, `char *`*buf*, `size_t` *n*)
string.h (GNU): *Hata İletileri* (sayfa: 41).

`char *` **strfry** (`char *`*string*)
string.h (GNU): *strfry* (sayfa: 119).

`size_t` **strftime** (`char *`*s*, `size_t` *size*, `const char *`*template*, `const struct tm *`*brokentime*)
time.h (ISO): *Zaman Değerlerinin Biçimlenmesi* (sayfa: 550).

`size_t` **strlen** (`const char *`*s*)
string.h (ISO): *Dizge Uzunluğu* (sayfa: 92).

`int` **strncasecmp** (`const char *`*s1*, `const char *`*s2*, `size_t` *n*)
string.h (BSD): *Dizi/Dizge Karşılaştırması* (sayfa: 104).

`char *` **strncat** (`char *restrict` *to*, `const char *restrict` *from*, `size_t` *size*)
string.h (ISO): *Kopyalama ve Birleştirme* (sayfa: 94).

`int` **strncmp** (`const char *`*s1*, `const char *`*s2*, `size_t` *size*)
string.h (ISO): *Dizi/Dizge Karşılaştırması* (sayfa: 104).

`char *` **strncpy** (`char *restrict` *to*, `const char *restrict` *from*, `size_t` *size*)
string.h (ISO): *Kopyalama ve Birleştirme* (sayfa: 94).

`char *` **strndup** (`const char *`*s*, `size_t` *size*)
string.h (GNU): *Kopyalama ve Birleştirme* (sayfa: 94).

`char *` **strndupa** (`const char *`*s*, `size_t` *size*)
string.h (GNU): *Kopyalama ve Birleştirme* (sayfa: 94).

`size_t` **strnlen** (`const char *`*s*, `size_t` *maxlen*)
string.h (GNU): *Dizge Uzunluğu* (sayfa: 92).

`char *` **strpbrk** (`const char *`*string*, `const char *`*stopset*)
string.h (ISO): *Arama İşlevleri* (sayfa: 111).

`char *` **strptime** (`const char *`*s*, `const char *`*fmt*, `struct tm *`*tp*)
time.h (XPG4): *Düşük Seviyede Çözümleme* (sayfa: 556).

`char *` **strrchr** (`const char *`*string*, `int` *c*)
string.h (ISO): *Arama İşlevleri* (sayfa: 111).

`char *` **strsep** (`char **`*string_ptr*, `const char *`*delimiter*)
string.h (BSD): *Bir Dizgeyi Dizgeciklere Ayırma* (sayfa: 115).

`char *` **strsignal** (`int` *signum*)
string.h (GNU): *Sinyal İletileri* (sayfa: 611).

`size_t` **strspn** (`const char *`*string*, `const char *`*skipset*)

string.h (ISO): *Arama İşlevleri* (sayfa: 111).

`char * strstr (const char *haystack, const char *needle)`
string.h (ISO): *Arama İşlevleri* (sayfa: 111).

`double strtod (const char *restrict string, char **restrict tailptr)`
stdlib.h (ISO): *Gerçek Sayıların Çözümlenmesi* (sayfa: 533).

`float strtof (const char *string, char **tailptr)`
stdlib.h (ISO): *Gerçek Sayıların Çözümlenmesi* (sayfa: 533).

`intmax_t strtoimax (const char *restrict string, char **restrict tailptr, int base)`
inttypes.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`char * strtok (char *restrict newstring, const char *restrict delimiters)`
string.h (ISO): *Bir Dizgeyi Dizgeciklere Ayırma* (sayfa: 115).

`char * strtok_r (char *newstring, const char *delimiters, char **save_ptr)`
string.h (POSIX): *Bir Dizgeyi Dizgeciklere Ayırma* (sayfa: 115).

`long int strtol (const char *restrict string, char **restrict tailptr, int base)`
stdlib.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`long double strtold (const char *string, char **tailptr)`
stdlib.h (ISO): *Gerçek Sayıların Çözümlenmesi* (sayfa: 533).

`long long int strtoll (const char *restrict string, char **restrict tailptr, int base)`
stdlib.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`long long int strtoq (const char *restrict string, char **restrict tailptr, int base)`
stdlib.h (BSD): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`unsigned long int strtoul (const char *retrict string, char **restrict tailptr, int base)`
stdlib.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`unsigned long long int strtoull (const char *restrict string, char **restrict tailptr, int base)`
stdlib.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`uintmax_t strtoumax (const char *restrict string, char **restrict tailptr, int base)`
inttypes.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`unsigned long long int strtouq (const char *restrict string, char **restrict tailptr, int base)`
stdlib.h (BSD): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`struct aiocb`
aio.h (POSIX.1b): *Eşzamansız G/Ç* (sayfa: 327).

`struct aiocb64`

signal.h (GNU): *Basit Sinyal İşleme* (sayfa: 611).

B.20. T

double **tan** (double *x*)
math.h (ISO): *Trigonometrik İşlevler* (sayfa: 476).

float **tanf** (float *x*)
math.h (ISO): *Trigonometrik İşlevler* (sayfa: 476).

double **tanh** (double *x*)
math.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

float **tanhf** (float *x*)
math.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

long double **tanhl** (long double *x*)
math.h (ISO): *Hiperbolik İşlevler* (sayfa: 483).

long double **tanl** (long double *x*)
math.h (ISO): *Trigonometrik İşlevler* (sayfa: 476).

int **tcdrain** (int *filedes*)
termios.h (POSIX.1): *Hat Denetim İşlevleri* (sayfa: 460).

tcflag_t
termios.h (POSIX.1): *Uçbirim Kipi Veri Türleri* (sayfa: 444).

int **tcflow** (int *filedes*, int *action*)
termios.h (POSIX.1): *Hat Denetim İşlevleri* (sayfa: 460).

int **tcflush** (int *filedes*, int *queue*)
termios.h (POSIX.1): *Hat Denetim İşlevleri* (sayfa: 460).

int **tcgetattr** (int *filedes*, struct termios **termios–p*)
termios.h (POSIX.1): *Uçbirim Kipi İşlevleri* (sayfa: 445).

pid_t **tcgetpgrp** (int *filedes*)
unistd.h (POSIX.1): *Denetim Uçbirimine Erişim İşlevleri* (sayfa: 731).

pid_t **tcgetsid** (int *fildes*)
termios.h (Unix98): *Denetim Uçbirimine Erişim İşlevleri* (sayfa: 731).

TCSADRAIN
termios.h (POSIX.1): *Uçbirim Kipi İşlevleri* (sayfa: 445).

TCSAFLUSH
termios.h (POSIX.1): *Uçbirim Kipi İşlevleri* (sayfa: 445).

TCSANOW
termios.h (POSIX.1): *Uçbirim Kipi İşlevleri* (sayfa: 445).

TCSASOFT
termios.h (BSD): *Uçbirim Kipi İşlevleri* (sayfa: 445).

int **tcsendbreak** (int *filedes*, int *duration*)

netdb.h (BSD): *Konak İsimleri* (sayfa: 412).

`void * tsearch (const void *key, void **rootp, comparison_fn_t compar)`
search.h (SVID): *Ağaç Arama İşlevi* (sayfa: 210).

`char * ttyname (int filedes)`
unistd.h (POSIX.1): *Uçbirimlerin Tanımlanması* (sayfa: 442).

`int ttyname_r (int filedes, char *buf, size_t len)`
unistd.h (POSIX.1): *Uçbirimlerin Tanımlanması* (sayfa: 442).

`void twalk (const void *root, __action_fn_t action)`
search.h (SVID): *Ağaç Arama İşlevi* (sayfa: 210).

`char * tzname [2]`
time.h (POSIX.1): *Zaman Dilimi Değişkenleri ve İşlevleri* (sayfa: 566).

`int TZNAME_MAX`
limits.h (POSIX.1): *Genel Sınırlar* (sayfa: 784).

`void tzset (void)`
time.h (POSIX.1): *Zaman Dilimi Değişkenleri ve İşlevleri* (sayfa: 566).

B.21. U

UCHAR_MAX
limits.h (ISO): *Bir Tamsayı Türün Aralığı* (sayfa: 821).

ucontext_t
ucontext.h (SVID): *Bütünsel Bağlam Denetimi* (sayfa: 596).

uid_t
sys/types.h (POSIX.1): *Bir Sürecin Aidiyetinin Okunması* (sayfa: 744).

UINT_MAX
limits.h (ISO): *Bir Tamsayı Türün Aralığı* (sayfa: 821).

`int ulimit (int cmd, ...)`
ulimit.h (BSD): *Özkaynak Kullanımın Sınırlanması* (sayfa: 575).

ULONG_LONG_MAX
limits.h (ISO): *Bir Tamsayı Türün Aralığı* (sayfa: 821).

ULONG_MAX
limits.h (ISO): *Bir Tamsayı Türün Aralığı* (sayfa: 821).

`mode_t umask (mode_t mask)`
sys/stat.h (POSIX.1): *Dosya İzinlerinin Atanması* (sayfa: 380).

`int umount (const char *file)`
sys/mount.h (SVID, GNU): *Bağlama, Ayırma, Yeniden Bağlama* (sayfa: 778).

`int umount2 (const char *file, int flags)`
sys/mount.h (GNU): *Bağlama, Ayırma, Yeniden Bağlama* (sayfa: 778).

`int uname (struct utsname *info)`

B.22. V

B.23. W

wchar.h (ISO): *Kopyalama ve Birleştirme* (sayfa: 94).

`int` **`wcsncmp`** `(const wchar_t *`*ws1*`, const wchar_t *`*ws2*`, size_t` *size*`)`
wchar.h (ISO): *Dizi/Dizge Karşılaştırması* (sayfa: 104).

`wchar_t *` **`wcsncpy`** `(wchar_t *restrict` *wto*`, const wchar_t *restrict` *wfrom*`, size_t` *size*`)`
wchar.h (ISO): *Kopyalama ve Birleştirme* (sayfa: 94).

`size_t` **`wcsnlen`** `(const wchar_t *`*ws*`, size_t` *maxlen*`)`
wchar.h (GNU): *Dizge Uzunluğu* (sayfa: 92).

`size_t` **`wcsnrtombs`** `(char *restrict` *dst*`, const wchar_t **restrict` *src*`, size_t` *nwc*`, size_t` *len*`, mbstate_t *restrict` *ps*`)`
wchar.h (GNU): *Dizge Dönüşümleri* (sayfa: 137).

`wchar_t *` **`wcspbrk`** `(const wchar_t *`*wstring*`, const wchar_t *`*stopset*`)`
wchar.h (ISO): *Arama İşlevleri* (sayfa: 111).

`wchar_t *` **`wcsrchr`** `(const wchar_t *`*wstring*`, wchar_t` *c*`)`
wchar.h (ISO): *Arama İşlevleri* (sayfa: 111).

`size_t` **`wcsrtombs`** `(char *restrict` *dst*`, const wchar_t **restrict` *src*`, size_t` *len*`, mbstate_t *restrict` *ps*`)`
wchar.h (ISO): *Dizge Dönüşümleri* (sayfa: 137).

`size_t` **`wcsspn`** `(const wchar_t *`*wstring*`, const wchar_t *`*skipset*`)`
wchar.h (ISO): *Arama İşlevleri* (sayfa: 111).

`wchar_t *` **`wcsstr`** `(const wchar_t *`*haystack*`, const wchar_t *`*needle*`)`
wchar.h (ISO): *Arama İşlevleri* (sayfa: 111).

`double` **`wcstod`** `(const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`)`
wchar.h (ISO): *Gerçek Sayıların Çözümlenmesi* (sayfa: 533).

`float` **`wcstof`** `(const wchar_t *`*string*`, wchar_t **`*tailptr*`)`
stdlib.h (ISO): *Gerçek Sayıların Çözümlenmesi* (sayfa: 533).

`intmax_t` **`wcstoimax`** `(const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*`)`
wchar.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`wchar_t *` **`wcstok`** `(wchar_t *`*newstring*`, const char *`*delimiters*`)`
wchar.h (ISO): *Bir Dizgeyi Dizgeciklere Ayırma* (sayfa: 115).

`long int` **`wcstol`** `(const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*`)`
wchar.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`long double` **`wcstold`** `(const wchar_t *`*string*`, wchar_t **`*tailptr*`)`
stdlib.h (ISO): *Gerçek Sayıların Çözümlenmesi* (sayfa: 533).

`long long int` **`wcstoll`** `(const wchar_t *restrict` *string*`, wchar_t **restrict` *tailptr*`, int` *base*`)`
wchar.h (ISO): *Tamsayıların Çözümlenmesi* (sayfa: 528).

`size_t` **`wcstombs`** `(char *`*string*`, const wchar_t *`*wstring*`, size_t` *size*`)`

WRDE_DOOFFS
wordexp.h (POSIX.2): *Sözcük Yorumlama Seçenekleri* (sayfa: 227).

WRDE_NOCMD
wordexp.h (POSIX.2): *Sözcük Yorumlama Seçenekleri* (sayfa: 227).

WRDE_NOSPACE
wordexp.h (POSIX.2): *wordexp çağrısı* (sayfa: 225).

WRDE_REUSE
wordexp.h (POSIX.2): *Sözcük Yorumlama Seçenekleri* (sayfa: 227).

WRDE_SHOWERR
wordexp.h (POSIX.2): *Sözcük Yorumlama Seçenekleri* (sayfa: 227).

WRDE_SYNTAX
wordexp.h (POSIX.2): *wordexp çağrısı* (sayfa: 225).

WRDE_UNDEF
wordexp.h (POSIX.2): *Sözcük Yorumlama Seçenekleri* (sayfa: 227).

`ssize_t` **write** `(int` *filedes*`, const void *`*buffer*`, size_t` *size*`)`
unistd.h (POSIX.1): *Girdi ve Çıktı İlkelleri* (sayfa: 308).

`ssize_t` **writev** `(int` *filedes*`, const struct iovec *`*vector*`, int` *count*`)`
sys/uio.h (BSD): *G/Ç'yı Hızlı Dağıtıp Toplama* (sayfa: 318).

`int` **wscanf** `(const wchar_t *`*template*`, ...)`
wchar.h (ISO): *Biçimli Girdi İşlevleri* (sayfa: 284).

`int` **WSTOPSIG** `(int` *status*`)`
sys/wait.h (POSIX.1): *Süreç Tamamlanma Durumu* (sayfa: 692).

`int` **WTERMSIG** `(int` *status*`)`
sys/wait.h (POSIX.1): *Süreç Tamamlanma Durumu* (sayfa: 692).

B.24. X

`int` **X_OK**
unistd.h (POSIX.1): *Dosya Erişim İzinlerinin Sınanması* (sayfa: 382).

_XOPEN_SOURCE
(X/Open): *Özellik Sınama Makroları* (sayfa: 25).

_XOPEN_SOURCE_EXTENDED
(X/Open): *Özellik Sınama Makroları* (sayfa: 25).

B.25. Y

`double` **y0** `(double` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`float` **y0f** `(float` *x*`)`
math.h (SVID): *Özel İşlevler* (sayfa: 484).

`long double` **y0l** `(long double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`double` **y1** `(double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`float` **y1f** `(float` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`long double` **y1l** `(long double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`double` **yn** `(int n, double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`float` **ynf** `(int n, float` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

`long double` **ynl** `(int n, long double` *x*`)`
`math.h` (SVID): *Özel İşlevler* (sayfa: 484).

C. GNU C Kütüphanesinin Kurulması

Hiçbir şey yapmadan, önce kaynak ağacının kök dizinindeki `FAQ` dosyasını okumalısınız. Bu dosyada kurulum ve derleme ile ilgili olarak bazı sorular ve bu sorulara verilmiş yanıtlar bulacaksınız. Bu dosya, bu kılavuzdan daha sık güncellenir.

GNU Libc'ye eklenebilen özellikler *add–on* paketleri ile eklenebilir. Bu paketler kaynak ağacının kök dizininde açılarak kaynak ağacına eklenirler. Bundan sonra **`configure`** betiğini **`--enable-add-ons`** seçeneği ile çalıştırarak bunları etkinleştirebilir ve bunları kütüphane ile birlikte derleyebilirsiniz.

Ayrıca GNU araçlarının en son sürümlerine de ihtiyacınız olacak: GCC, GNU Make ve olası diğerleri. Bkz. *Derleme için Önerilen Araçlar* (sayfa: 954).

C.1. GNU Libc'nin Yapılandırılması ve Derlenmesi

GNU libc kaynak ağacının kök dizininde derlenemez. Ayrı bir `build` dizininde derlemelisiniz. Örneğin, glibc kaynak paketini `/usr/src/gnu/glibc-2.4` dizinine açtıysanız, `/usr/src/gnu/glibc-build` adında bir dizin daha oluşturup derlenen nesne dosyalarının bu dizinde bulunmasını sağlamalısınız. Böylece, derleme sırasında bir hata oluşursa basitçe bu dizini içindekilerle birlikte silip yeniden temiz bir derleme yapabilirsiniz.

`configure` betiğini `/usr/src/gnu/glibc-build` dizininden aşağıdaki komutla çalıştırırsanız, derleme sırasında nesne dosyaları bu dizinde yer alacaktır:

```
$ ../glibc-2.4/configure argümanlar...
```

Burada anlatıldığı gibi paketi ayrı bir dizinde derlerken bile kaynak dizindeki bazı dosyaların derleme sırasında değişeceğini bilmelisiniz. Özellikle `manual` dizinindeki dosyalar bundan etkilenecektir.

`configure` betiği çok sayıda seçenek kabul eder, ancak burada önemli olan tek bir seçenek üzerinde duracağız: **`--prefix`**. Bu seçenek ile glibc'nin kurulmasını istediğiniz yeri belirtebilirsiniz. Bu seçenek için `/usr/local` öntanımlıdır fakat standart sistem kütüphanesi olarak bu seçenek GNU/Linux sistemleri için **`--prefix=/usr`**, GNU/Hurd sistemleri için ise **`--prefix=`** (boş önekle) biçiminde belirtilmelidir.

Ayrıca, **configure** betiği çalıştırılırken CC ve CFLAGS değişkenlerini de ortama dahil etmek yararlı olabilir. CC değişkeni ile kullanılacak C derleyicisi, CFLAGS ile derleyicinin kullanacağı eniyilemeler belirtilebilir.

configure betiğinde kullanılabilecek komut satırı seçeneklerinin listesi:

--prefix=*dizin*
: Makinaya bağımlı veri dosyaları *dizin* dizininin alt dizinlerine kurulur. Öntanımlı olarak /usr/local dizinidir.

--exec-prefix=*dizin*
: Kütüphane ve makinaya bağımlı diğer veri dosyaları *dizin* dizininin alt dizinlerine kurulur. Öntanımlı değeri eğer belirtilmişse **--prefix** seçeneğinde belirtilen dizindir, aksi takdirde /usr/local dizinidir.

--with-headers=*dizin*
: Çekirdek başlık dosyaları /usr/include dizininde değil, *dizin* dizininde aranır. Glibc normalde /usr/include dizine bakar, ancak bu seçenekle farklı bir dizin belirtirseniz *dizin* dizinine bakar.

  Bu seçenek öncelikli olarak /usr/include içindeki başlık dosyaları glibc'nin daha eski bir sürümünden gelen bir sistemde yararlı olur. Bu durumda bazan çelişkiler oluşur. Linux libc5'in glibc'nin daha eski bir sürümünü nitelediğine dikkat edin. Bu seçeneği ayrıca glibc'yi /usr/include içindeki dosyalardan daha yeni bir çekirdeğin başlık dosyalarıyla derlemek isterseniz de kullanabilirsiniz.

--enable-add-ons [=*liste*]
: Derleme sırasında dahil edilecek eklentiler belirtilir. Eğer bu seçenek listesiz olarak belirtilirse, ana kaynak dizininde bulunan bütün eklentiler etkinleştirilir; bu öntanımlı davranıştır. Etkin olmasını istediklerinizin listesini aralarına boşluk veya virgül koyarak belirtebilirsiniz (boşluk kullanacaksanız onları kabuktan korumak için tırnak içine almayı unutmayın). *liste* içindeki her eklenti bir mutlak dizin ismi olabileceği gibi ana kaynak dizinine göreli bir dizin ismi, hatta derleme dizinine (yani, o anki çalışma dizinine) göreli bir dizin ismi olabilir. Örnek: **--enable-add-ons=nptl,../glibc-libidn-2.4**.

--enable-kernel=*sürüm*
: Bu seçenek şimdilik sadece GNU/Linux sistemlerinde kullanışlıdır. *sürüm* parametresi X.Y.Z şeklinde belirtilmeli ve üretilen kütüphanenin destekleyeceği en küçük Linux çekirdeğinin sürümü olmalıdır. Daha yüksek *sürüm* numarası daha az uyumluluk kodu ekler ve kod daha hızlı olur.

--with-binutils=*dizin*
: C derleyicinin öntanımlı kullandığı değil, *dizin* ile belirtilen yerdeki binutils (çevirici ve ilintileyici) kullanılır. Eğer GNU C kütüphanesindeki oluşumlar sisteminizdeki öntanımlı binutils ile çalışamayacaksa bu seçeneği kullanabilirsiniz. Bu durumda, **configure** sorunu saptar ve bu oluşumları bastırır, böylece kütüphane hala kullanılabilir kalabilir, ancak işlevsellik kaybolabilir; örneğin, eski binutils ile bir paylaşımlı libc derleyemezsiniz.

--without-fp
: Makinanın donanımında kayan noktalı sayılar için destek yoksa ve işletim sistemi de bu desteği yazılımsal olarak içermiyorsa bu seçeneği kullanmalısınız.

--disable-shared
: Mümkün olsa bile paylaşımlı kütüphaneler oluşturulmaz. Tüm sistemlerde paylaşımlı kütüphane desteği yoktur; paylaşımlı kütüphaneler için ELF desteği ve GNU ilintileyici gerekir.

--disable-profile
: Kütüphaneler profil desteği ile derlenmez. Profil desteğine ihtiyacınız olmayacağını düşünüyorsanız bu seçeneği kullanabilirsiniz.

--enable-omitfp

Normal (durağan ve paylaşımlı) kütüphaneler için en yüksek eniyileme kullanılır ve durağan kütüphaneler ayrı olarak hata ayıklama bilgileri ile eniyilemesiz derlenir. Bunu yapmanızı tavsiye etmiyoruz. Fazladan bir eniyileme size fazla birşey kazandırmadığı gibi derleyici hatalarını uyandırabilir ve C kütüphanesi üzerinden hataları izlemek mümkün olmayabilir.

--disable-versioning
Paylaşımlı kütüphaneleri sembol sürüm bilgileri ile derlemez. Böyle yaparak derlenen kütüphane eski ikilik dosyalarla uyumsuz olacaktır, bu bakımdan tavsiye edilmez.

--enable-static-nss
NSS kütüphanelerinin durağan sürümleri derlenir. Bu NSS'in amacını bozacağından tavsiye edilmez; NSS kütüphaneleriyle durağan ilintili bir yazılım farklı bir isim veritabanını kullanmak için yeniden özdevimli olarak yapılandırılamaz.

--without-tls
Öntanımlı olarak C kütüphanesi evreye özel saklama alanı (thread–local storage) desteği ile derlenir. Bu seçenek kullanılarak bunun olmaması sağlanabilir ama uyumluluk sorunlarına yol açacağından genellikle böyle birşeye gerek yoktur.

--build=*derleme-sistemi*
--host=*çalışma-sistemi*
Bu seçenekler çapraz derleme için kullanılır.Her iki seçeneği de belirtirseniz ve *derleme–sistemi* ile *çalışma–sistemi* farklıysa, **configure** betiği glibc'yi *çalışma–sistemi* üzerinde *derleme–sistemi* olarak kullanılacak şekilde çapraz derlemeye hazırlar. Büyük ihtimalle **--with-headers** seçeneğine de ihtiyacınız olacaktır, ayrıca betiğin derleyici ve/veya binutils seçimlerini değiştirmek zorunda kalabilirsiniz.

Sadece **--host** seçeneğini kullanırsanız, betik sizin sisteminiz yerine belirttiğiniz sistem için sistemdeki mevcut derleme araçlarıyla derlenecek şekilde glibc'yi derlemeye hazırlar Bu daha alt seviyede bir işlemci belirtmek için yararlıdır. Örneğin, **configure** betiği sizin makinanızı **i586-pc-linux-gnu** olarak belirlemişse ve siz bunun yerine kütüphaneyi 386'lar üzerinde çalışacak şekilde derlemek isterseniz seçeneği **--host=i386-pc-linux-gnu** ya da sadece **--host=i386-linux** olarak belirtebilir ve CFLAGS değişkeninde derleyiciye **--mcpu=i386** belirtebilirsiniz.

Sadece **--build** seçeneğinin belirtilmesi betiğin şaşırmasına sebep olacaktır.

Kütüphaneyi ve ilgili uygulamaları derlemek için **make** komutunu kullanın. Derleme sırasında epey bir çıktı üretilir, bazıları da hata gibi görünebilir ama değildir. **make**'in ürettiği iletiler ******* içeriyorsa bunlar hata iletileridir. Bunlar bazı şeylerin yanlış gittiğini belirtirler.

Derleme işlemi yapılandırmanıza ve makinenizin hızına bağlı olarak biraz uzun sürebilir. Bazı karmaşık modüllerin derlenmesi çok uzun zaman alabilir, daha yavaş makinalar için bu daha da artar. Eğer derleyici çökmüş gibi görünürse paniklemeyin, işini bitirmesini bekleyin.

Paralel derleme yapmak isterseniz **make**'e **-j** seçeneği ile uygun sayısal parametreleri vermeniz yeterlidir. Bunun için en son GNU **make** sürümüne ihtiyacınız var.

Bazı kütüphane oluşumlarının çalışıp çalışmadığını sınamak isterseniz sınama yazılımlarını derlemek ve çalıştırmak için **make check** komutunu verebilirsiniz. Bu işlem başarıyla tamamlanmazsa derlenen kütüphaneyi kullanmayın ve sorunun bilinenlerden biri olmadığını saptarsanız hatayı raporlayın. Hata raporlama hakkında daha fazla bilgi için *Yazılım Hatalarının Raporlanması* (sayfa: 956) bölümüne bakınız. Bazı sınamaların yapılması **root** tarafından çalışıtırılmış olmamayı gerektirir. Glibc'yi derlerken ve sınarken ayrıcalıksız kullanıcı olmanızı öneririz.

Hataları raporlamadan önce sorunun sizin sisteminizden kaynaklanmadığından emin olmalısınız. Sınamalar (daha sonra da kurulum) `/etc/passwd`, `/etc/nsswitch.conf`, vb. sistem dosyalarını kullanır. Bu dosyaların hepsi doğru içeriğe sahip olmalıdır.

[GNU C Kütüphanesi Kılavuzu]'nu yazdırmaya hazırlamak isterseniz **`make dvi`** komutunu verin. Bunu yapabilmek için çalışan bir TeX kurulumuna ihtiyacınız vardır. Kılavuzun Info dosyaları pakette biçimlenmiş olarak zaten vardır. Bunları yeniden üretmek isterseniz **`make info`** komutunu verebilirsiniz ama bu gerekmez.

Kütüphane, **`Makeconfig`** içinde bulabileceğiniz bir miktar özel amaçlı yapılandırma parametresine sahiptir. Bunlar **`configparms`** dosyası ile değiştirilebilir. Bunları değiştirmek isterseniz derleme işlemini yapacağınız dizinde bir **`configparms`** dosyası oluşturun ve içine sisteminize uygun değerleri ekleyin. Bu dosya **`make`** tarafından derleme ortamına dahil edilir.

GNU C kütüphanesini çapraz derleme için yapılandırmak **`configparms`** içine bir kaç tanım ekleyerek kolayca yapılabilir. `CC` değişkenine kütüphaneyi kendisi için yapılandıracağınız hedef derleyiciyi atayabilirsiniz; bu değerin **`configure`** betiğini çalıştırırken kullanılacak `CC` değeri ile aynı olması önemlidir. Örnek: `CC=`*`hedef`*`-gcc` **`configure`** *`hedef`*. Derlenen kütüphanenin parçası olacağı sistemde çalışacak yazılımlar için kullanılacak derleyiciyi `BUILD_CC` değişkenine atayın. Eğer yerel araçlar hedefteki nesne dosyaları ile çalışacak şekilde yapılandırılmamışsa, hedef sistemde çalışacak **`ar`** ve **`ranlib`** araçlarını `AR` ve `RANLIB` değişkenlerinde belirtmeniz gerekebilir.

C.2. C Kütüphanesinin Kurulması

Kütüphaneyi, başlık dosyalarını ve kılavuzun Info dosyalarını kurmak için **`env LANGUAGE=C LC_ALL=C make install`** komutunu verin. Eğer daha önce derleme yapılmamışsa, bu komut önce paketi derleyecek sonra da kuracaktır. Ancak kurulumdan önce derleme işlemini bitirmiş ve gerekli sınamaları yapmış olmanız daha iyidir. Eğer glibc'yi birincil C kütüphanesi olarak kuruyorsanız, önce sistemi tek kullanıcılı kipe almanız ve kurulumdan sonra sistemi yeniden başlatmanız önerilir. Bu işlem, kütüphaneyi temelden değiştirirken bazı bozucu şeylerin oluşturacağı riskleri en aza indirir.

Linux libc5'den yükseltme ya da başka bir C kütüphanesinden geçiş yapıyorsanız **`/usr/include`** dizininin boşaltılması önem kazanır. Yeni **`/usr/include`** Linux başlık dosyalarından başka birşey içermemelidir.

Bunun için önce kütüphaneyi derlemelisiniz (**`make`**), isterseniz sınayabilirsiniz (**`make check`**) sonra başlık dosyalarını içeren dizini değiştirin ve kütüphaneyi kurun (**`make install`**). İşlem bu sırayla yapılmalıdır. Kurulumdan önce başlık dosyalarını içeren dizinin değiştirilmemesi, eski kütüphanenin başlık dosyalarıyla istenmeyen bir karışımın oluşmasına yol açacaktır. Ancak bu değiştirme işleminin kurulumdan hemen önce yapılması gerekir; çünkü yapılandırma, derleme ve sınama işlemleri bu eski başlık dosyaları kullanılarak yapılacaktır.

Glibc 2.0 veya 2.1 sürümünden yükseltme yapıyorsanız, **`make install`** komutu bütün işi kendisi yapacaktır. Eski başlık dosyalarını kaldırmanıza gerek kalmayacaktır. Ama yapmak isterseniz bir mahsuru yok, yukarıda anlatılan adımları uygulayabilirsiniz.

Ayrıca GCC'yi yeni kütüphane ile çalışacak şekilde yeniden yapılandırmanız gerekebilir. Bunu yapmanın en kolay yolu tekrar çalışır hale getirmek için derleyici seçeneklerini yapılandırıp (GNU/Linux sistemlerde **`-Wl,--dynamic-linker=/lib/ld-linux.so.2`** çalışmalıdır) gcc'yi yeniden derlemektir. Ayrıca `specs` dosyasını (`/usr/lib/gcc-lib/`*`HEDEF`*`/`*`SÜRÜM`*`/specs`) yeniden düzenlemeniz gerekebilir, ancak bu biraz kara sanattır.

Glibc'yi yapılandırdığınız yerden farklı bir yere kurmak isterseniz kuracağınız yeri **`make install`** komut satırında **`install_root`** değişkeni ile belirtebilirsiniz. Bu değişkenin değeri tüm kurulum yollarını önceleyerek değiştirir. Bu bağlı bir dosya sistemindeki başka sisteme ait bir kök dosya sistemine (chroot) kurulum yapmak ya da ikilik bir dağıtım hazırlamak için yararlıdır. Belirtilen dizin bir mutlak dosya yolu olarak belirtilmelidir.

Glibc 2.2 **nscd** olarak bilinen bir artalan yazılımı içerir. **nscd** isim hizmeti aramalarında arabellekleme yapar. NIS+ ile başarımı oldukça yükseltir ve DNS'ye de yardımcı olur.

Bir yardımcı yazılım, `/usr/libexec/pt_chown`, setuid **root** olarak kurulmalıdır. Yazılım **grantpt** işlevi tarafından çalıştırılır; işlev çağrıldığı süreç için uçbirimsiler üzerindeki izinleri ayarlar. Böylelikle, **xterm** ve **screen** gibi yazılımların setuid olmaları gerekmez. (Bunların ayrıcalıklara ihtiyaç duyma sebepleri olabilir.) Eğer 2.1 veya daha yeni bir Linux çekirdeğini **devptsfs** veya **devfs** desteği ile kullanıyorsanız bu yazılıma ihtyacınız olmaz; aksi takdirde ihiyacınız olacaktır. **pt_chown** yazılımının kodu `login/programs/pt_chown.c` dosyasındadır.

Kurulumdan sonra sisteminizde yerel ve zaman dilimi yapılandırmasını yapmak isteyebilirsiniz. GNU C kütüphanesi **localedef** ile yapılandırılabilen bir yerel veritabanı ile gelir. Örneğin Türkçe yerelini **tr_TR** ismiyle ayarlamak isterseniz **localedef -i tr_TR -f ISO-8859-9 tr_TR** komutu yeterli olur. Glibc tarafından desteklenen tüm yerelleri yapılandırmak isterseniz, kütüphaneyi derlediğiniz dizinde **make localedata/install-locales** komutunu verin.

Zaman dilimini yapılandırmak için **TZ** ortam değişkenini kullanabilirsiniz. **tzselect** betiği doğru değeri seçmenize yardımcı olur. Örneğin Türkiye için **tzselect**, **TZ='Europe/Istanbul'** ataması yapmanızı önerecektir. Sistem çapında yapılandırma için **/usr/share/zoneinfo** içindeki zaman dilimi dosyasından **/etc/localtime** dosyasına bir sembolik bağ yapın. Türkiye için, **ln -s /usr/share/zoneinfo/Europe/Istanbul /etc/localtime** komutu yeterli olacaktır.

C.3. Derleme için Önerilen Araçlar

GNU C Kütüphanesini derlemeye başlamadan önce aşağıdaki GNU araçlarını kurmanızı öneririz:

GNU `make` 3.79 veya üstü

GNU **make**'in en son sürümüne ihtiyacınız olacak. GNU C Kütüphanesini başka bir **make** için değiştirmek zor olacağından GNU **make** kullanmanızı öneririz. Tavsiyemiz, GNU **make**'in 3.79 sürümü olacaktır. Daha eski tüm sürümleri çeşitli hatalar içerir ya da kütüphanenin özellikleri ile uyumlu değildir.

GCC 3.4 veya üstü, GCC 4.1 önerilir

GNU C Kütüphanesi sadece GNU C derleyici ailesi ile derlenebilir. 2.3 sürümlerinin derlenmesi için GCC 3.2 veya üstü bir sürüm gerekir; 2.3 sürümlerini derlemek için biz GCC 3.4'ü öneriyoruz. 2.4 sürümlerinin derlenmesi için GCC 3.4 veya üstü bir sürüm gerekir; bu sürümü yazarken GCC 4.1'i kullandık ve şimdiki sürümler için onu öneriyoruz. **powerpc64** iceren bazı makinalarda GCC 4.0 öncesi derleyicilerin C kütüphanesinin 2.4 sürümlerini derlemekten kaçınmanızı gerektirecek sorunları vardır. Diğer makinalarda, doğru **long double** türü biçim destekli C kütüphanesi derlemek için GCC 4.1 gerekir; `powerpc` (32 bit), `s390` ve `s390x` dahil.

GNU libc'yi kullanacak yazılımları derlerken istediğiniz derleyiciyi kullanabilirsiniz. Ancak, GCC 2.7 ve 2.8 sürümlerinde kayan noktalı sayılara destek sorunludur. Matematik kütüphanesi doğru çalışmaz. Buna dikkat edin.

Kütüphaneyi kullanacağınız platforma özel derleyici sorunları için FAQ dosyasına bakabilirsiniz.

GNU `binutils` 2.15 veya üstü

GNU C kütüphanesini derlemek için GNU **binutils** (as ve ld) kullanmalısınız. Şu an gerekli işlevselliği sağlayacak başka bir çevirici ve ilintileyici yoktur.

GNU `texinfo` 3.12f

Texinfo belgeleri düzgün olarak dönüştürmek ve kurmak için **texinfo** paketinin bu sürümü gerekir. Daha eski sürümleri belgelerde kullanılan tüm yaftaları anlayamaz ve info dosyaları için kurulum mekanizması ya mevcut değildir ya da farklı çalışır.

GNU `awk` 3.0 veya üstü
: **`awk`** çeşitli yerlerde dosya üretmek için kullanılmıştır. **`gawk`** 3.0 işe yaramaktadır.

Perl 5
: Perl derleme için gerekmez, ama kurulumun sınanması sırasında kullanılmıştır. İlerde kullanmayı düşünebiliriz.

GNU `sed` 3.02 veya üstü
: **`sed`** çeşitli yerlerde dosya üretmek için kullanılmıştır. Betiklerin çoğu **`sed`**'in herhangi bir sürümü ile çalışır. Sınama amaçlı kullanılan **`msgs.h`** dosyasını üreten **`intl`** altdizinindeki **`po2test.sed`** betiği bunun dışındadır. Bu betik sadece GNU **`sed`** 3.02 ile düzgün çalışır. Kurulumu sınamayı düşünüyorsanız **`sed`**'i mutlaka güncellemelisiniz.

`configure.in` dosyalarında bazı değişiklikler yapmak niyetindeyseniz,

- GNU **`autoconf`** 2.53 veya üstü

gerekir. Eğer ileti çeviri dosyalarını değiştirmek isterseniz,

- GNU **`gettext`** 0.10.36 veya üstü

gerekir. Eğer kaynak ağacını yamalarla yükseltecekseniz bu paketlere yine de ihtiyacınız olacaktır ama bundan kaçınmayı deneseniz daha iyi olur.

C.4. GNU/Linux Sistemlere Özgü Tavsiyeler

GNU libc'yi bir GNU/Linux sisteme kurucaksanız, sisteminizdeki Linux çekirdeğinin başlık dosyalarının 2.2 sürümü ya da daha yeni bir çekirdeğin başlık dosyaları olmalıdır. ia64, sh ve hppa gibi bazı mimarilerde ise bu en az 2.3.99 (sh ve hppa) ya da 2.4.0 (ia64) olmalıdır. Böyle bir çekirdeği kullanmaya ihtiyacınız yoksa, sadece glibc'nin onları bulabilmesini sağlamanız yeterlidir. Bunu yapmanın en kolay yolu `/usr/src/linux-2.2.1` gibi bir dizine çekirdek kaynak kodunu yerleştirdikten sonra **`make config`** komutunu verin ve öntanımlı yapılandırmayı kabul edin, sonra da **`make include/linux/version.h`** komutunu verin. Son olarak glibc'yi **`--with-headers=/usr/src/linux-2.2.1/include`** seçeneği ile yapılandırın ve derleyin. Bu işlemi en son çıkan çekirdekle yapmaya çalışın.

Bir diğer taktik ise yukarıdaki gibi 2.2 veya üstü bir çekirdeği yine yükeyin ve aynı şekilde **`make config`** yapın. Sonra `/usr/include` dizinini ya silin ya da ismini değiştirin ve yeni `/usr/include` dizini içinde çekirdek kaynak koduna `/usr/include/linux` ve `/usr/include/asm` sembolik bağlarını yapın. Bu taktikte glibc'yi yapılandırırken özel bir seçenek kullanmanıza gerek yoktur. Eğer libc5'ten yükseltme yapıyorsanız bu taktiği kullanmalısınız, çünkü artık eski başlık dosyalarına hiç ihtiyacınız olmayacaktır.

GNU libc'yi sisteme kurduktan sonra `/usr/include/linux` ve `/usr/include/asm` dizinlerini ya silin ya da isimlerini değiştirin ve bunların içeriklerini glibc kurulumunda kullandığınız çekirdek kaynak kodundaki `include/linux` ve **`include/asm-$`***MİMARİ* dizinlerinin içerikleri ile değiştirin. Burada *MİMARİ* kütüphanenin kurulduğu makinanın mimarisi (**`i386`** veya **`alpha`** gibi) olacaktır. Glibc'yi yapılandırırken **`--with-headers`** seçeneğini kullanmadıysanız bu işlemi yapmak zorunda değilsiniz. Yalnız, burada dikkat edeceğiniz husus işlemin sembolik bağlarla değil kopyalama ile yapılacağıdır.

`/usr/include/net` ve `/usr/include/scsi` dizinlerinin içerikleri çekirdek kaynak koduna sembolik bağ olmamalıdır. Bu dosyalar için GNU libc kendi sürümlerini içerir.

GNU/Linux'ta libc'nin bazı bileşenlerinin `/lib` içinde bazılarının da `/usr/lib` içinde olması gerekir. Eğer glibc'yi **`--prefix=/usr`** ile yapılandırırsanız kurulumda bu zaten böyle olur. Eğer öntanımlı yapılandırmayı tercih ederseniz bu dosyalar `/usr/local` altına kurulacaktır.

Libc5'ten yükseltme yapıyorsanız, her paylaşımlı kütüphanenin yeni kütüphaneye göre yeniden derlenmesi gerekir, ancak eski çalıştırılabilirlerin kullandığı eski kütüphaneleri de tutmanız gerekir. Bu işlem biraz karmaşık ve zordur. Bu konuda daha ayrıntılı bilgiyi Glibc2 HOWTO[B3454]'da bulabilirsiniz.

nscd'yi çekirdekteki evre desteği sorunlu olduğundan 2.0 çekirdeklerle kullanamayabilirsiniz. Belki, **nscd** bu sorunlara zor da olsa uyum sağlayabilir, ama diğer evreli yazılımlarla sorunlar çıkabilir.

C.5. Yazılım Hatalarının Raporlanması

GNU C kütüphanesinde bazı kodlama hataları olabilir. Bu kılavuzda da hatalar ya da gözden kaçmış konular olabilir. Bunları raporlarsanız düzeltilecektir. Yapmazsanız, bundan kimsenin haberi olmayacağından düzeltilmeden kalacaktır.

Sorunu raporlamadan önce zaten raporlanmış olup olmadığına bakmak daha iyidir. Bunlar iki yerde belgelenmiştir: `BUGS` dosyasında bilinen sorunlar listelenmiştir. Ayrıca, `http://sources.redhat.com/bugzilla/` adresinde bir hata izleme sistemi bulunur. Bu adreste henüz açık bulunan ya da kapanmış raporları bulabilirsiniz. Kapatılmış raporlarda bir yama ya da sorunu çözen bir ipucu vardır.

Bir sorunu raporlamak için önce onu bulmalısınız. Bu işin zor kısmıdır. Sorunu saptadıktan sonra bunun gerçekten bir hata olup olmadığından emin olmalısınız. Bunun en kolay yolu diğer C kütüphanelerinin böyle bir durumda nasıl davrandıklarına bakmaktır. Eğer davranışlar aynıysa, siz birşeyleri yanlış yapmışsınızdır ve kütüphaneler doğrudur. Değilse, kütüphanelerden biri bir ihtimal yanlış olabilir. Hatta yanlış olan GNU kütüphanesi de olmayabilir. Unix C kütüphanelerinin çoğu geçmişten gelerek bazı şeylere izin verirler, biz vermeyiz, örneğin bir dosyanın iki kere kapatılması gibi.

Eğer GNU C kütüphanesinin bazı şeyleri *ISO ve POSIX standartlarına* (sayfa: 20) uygun olarak yapmadığını düşünüyorsanız, bu bir hatadır. Onu raporlayın!

Bir hata bulduğunuza emin olduktan sonra, sorunu üreten en küçük sınama şartını oluşturun. Eğer mümkünse bir işlev çağrısına kadar sorunu küçültün. Bu çok zor olmasa gerek.

Son adım hatayı raporlamaktır. Bunu `http://sources.redhat.com/bugzilla/` arayüzünden yapın.

Eğer bir işlevin nasıl davranması gerektiğinden emin olamıyorsanız ve bu kılavuz da bunu size söylemiyorsa, bu kılavuzdaki bir hatadır. Onu da raporlayın! Eğer işlev bu kılavuzda yazıldığı gibi davranmıyorsa ya kütüphane ya da kılavuz yanlıştır. Bu kılavuzda herhangi bir hata ya da eksik bulursanız bunu `http://sources.redhat.com/bugzilla/` arayüzünden raporlayın. Sorunu raporlarken hatanın hangi bölümün neresinde olduğunu açıkça belirtmeye çalışın.

(Sorunu kılavuzun türkçe çevirisine göre saptamaya çalışmamanızı öneririm. Çevirmen hatalarından kütüphanenin yazarları sorumlu olamaz, bu bakımdan kılavuzun İngilizce özgün sürümünde sorun varsa bu adrese bunu raporlayın. Hata bir çeviri hatası ise bunu lütfen bana (<nilgun (at) belgeler·gen·tr>) bildirin. Çevirinin güncel olmasını sağlamaya çalışacağımdan özgün sürüme yapılan eklemeler ve düzeltmeler çeviriye er ya da geç yansıtılacaktır.)

D. Kütüphanenin Sürdürülmesi

D.1. Yeni İşlevsellik Eklenmesi

Kütüphanenin kurgulanması GNU **make**'in özel oluşumlarının ağırlıkla kullanıldığı `Makefile` dosyaları ile yapılır. `Makefile` dosyaları oldukça karmaşıktır ve büyük ihtimalle onları anlamaya çalışmayı istemezsiniz. Fakat, sadece bir kaç değişkeni bile doğru yerde tanımlayabilmeniz için onların ne yaptıklarını anlamanız oldukça önemlidir.

Kütüphane kaynak kodu konulara göre gruplanmış alt dizinlere bölünmüştür.

Örneğin, `string` alt dizininde dizge işleme işlevleri, `math` alt dizininde matematiksel işlevler bulunur.

Her dizinde birer basit `Makefile` dosyası bulunur. İçinde birkaç **make** değişkeni ve üst dizindeki `Makefile` dosyasından bu dosyaya dahil edilenler bulunur. Örneğin, **Rules** şöyle bir satırla dahil edilir:

```
include ../Rules
```

Alt dizinlerde bulunan `Makefile` dosyalarında tanımlanmış temel değişkenler şunlardır:

subdir
: Alt dizinin ismi, örneğin `stdio`. Bu değişken tanımlı *olmalıdır*.

headers
: Kütüphanenin bu bölümündeki başlık dosyalarının isimleri, örneğin `stdio.h`.

routines
aux
: Kütüphanenin bu bölümündeki modüllerin (kaynak dosyalarının) isimleri. Bunlar **strlen** gibi basit isimler olmalıdır (`strlen.c` gibi bir dosya ismi değil). Kütüphanede tanımlı işlevlerin modülleri için **routines**, veri tanımları gibi şeyleri içeren modüller için ise **aux** kullanın. Fakat, **routines** ve **aux** değerleri birleşik olmalıdır, yani aslında pratik olarak bir farkları yoktur.

tests
: Kütüphanenin bu bölümündeki sınama yazılımlarının isimleri. Bunlar **tester** gibi basit isimler olmalıdır (`tester.c` gibi bir dosya ismi değil). **make tests** tüm sınama yazılımlarını derleyecek ve çalıştıracaktır. Eğer bir sınama yazılımı girdiye ihtiyaç duyuyorsa sınama verisini *sınama–yazılımı*`.input` dosyasına koyun; bu dosyanın içindeki veriler yazılıma standart girdi üzerinden verilecektir. Eğer bir sınama yazılımının argümanlar alması gerekiyorsa bunları (hepsi tek bir satırda olmak üzere) *sınama–yazılımı*`.args` dosyasına koyun. Sınama yazılımları sınamalar başarılı olursa sıfır durumu ile sınama sırasında kütüphanede ya da derlemede bir hata bulmuşsa sıfırdan farklı bir durum ile çıkmalıdır.

others
: Kütüphanenin bu bölümündeki "diğer" yazılımlarının isimleri. Bunlar sınama yazılımları değildir, ama kütüphane ile bir takım küçük uygulamalar dağıtılabilir. Bunlar **make others** ile derlenirler.

install-lib
install-data
install
: **make install** ile kurulacak dosyalar. **install-lib** ile belirtilen dosyalar `configparms` veya `Makeconfig` içinde **libdir** ile belirtilen dizine kurulurlar (bkz. *GNU C Kütüphanesinin Kurulması* (sayfa: 950)). **install-data** ile belirtilen dosyalar `configparms` veya `Makeconfig` içinde **datadir** ile belirtilen dizine kurulurlar. **install** ile belirtilen dosyalar ise `configparms` veya `Makeconfig` içinde **bindir** ile belirtilen dizine kurulurlar.

distribute
: Dağıtılacak bir tar dosyasına bu alt dizinden konacak dosyalar. Burada `Makefile` dosyasını ve diğer standart kütüphanelerin içinde listelenen kaynak ve başlık dosyaları listelenmemelidir. Sadece dağıtıma bir şekilde girmeyecek dosyaları belirtin.

generated
: Bu alt dizinde `Makefile` dosyası tarafından üretilecek dosyalar. Bu dosyalar **make clean** ile silinecek ve bir dağıtıma asla dahil edilmeyecektir.

extra-objs
Bu alt dizinde Makefile dosyası tarafından derlenecek ek nesne dosyaları. Bunlar foo.o gibi dosyalardan oluşan bir listedir. Bu dosyalar derleme sonucunda oluşurlar ve **make clean** ile silinirler. Bu değişken **others** veya **tests** derlemelerinde elde edilecek ikincil nesne dosyaları için kullanılır.

D.2. GNU C Kütüphanesinin Uyarlanması

GNU C Kütüphanesi bir makina ve işletim sistemine kolayca uyarlanabilecek şekilde yazılmıştır. Makina ve işletim sistemi bağımlı işlevler yeni bir makina veya işletim sistemi için gerçeklenimler ayrı dizinler halinde kolayca eklenebilir. Bu bölümde kütüphane kaynak kodu ağacının yerleşimi ile kullanılacak makina bağımlı kodu seçecek mekanizmalar anlatılacaktır.

Tüm makina ve işletim sistemi bağımlı dosyalar kütüphane kaynak kodunun ana dizini altındaki sysdeps dizininde bulunurlar. Bu dizinin kendi alt dizinlerinden oluşan bir hiyerarşisi vardır (bkz. *Hiyerarşi Uzlaşımları* (sayfa: 960)).

sysdeps dizininin altındaki her dizin belli bir makina veya işletim sistemi için kaynak dosyaları içerir (örneğin, belli bir üreticiye ait sistemler veya IEEE 754 kayan nokta biçimini kullanan tüm makinalar). Bir yapılandırma bu alt dizinlerin sıralı bir listesini içerir. Her alt dizin kendi üst dizinini bu listeye dolaylı olarak ekler. Örneğin listeyi unix/bsd/vax olarak belirtirseniz bu, unix/bsd/vax unix/bsd unix listesine eşdeğerdir. Ayrıca bir alt dizin doğrudan dizin hiyerarşisinde bulunmayan başka bir alt dizine uygulanabilir. Bir alt dizinde Implies adında bir dosya varsa ve içinde sysdeps dizinindeki Implies dosyasını içeren dizinin altındaki dizinlerden oluşan bir liste varsa, bunlar listeye eklenirler. Implies dosyasında **#** ile başlayan satırlar açıklamalar olarak yoksayılırlar. Örneğin, unix/bsd/Implies dosyasında şunlar olsun:

```
# BSD has Internet-related things.
unix/inet
```

ve **unix/Implies** dosyası da şunu içersin:

```
posix
```

Bu durumda son liste unix/bsd/vax unix/bsd unix/inet unix posix olur.

sysdeps dizininde generic isminde "özel" bir dizin bulunur. Daima dolaylı olarak altdizin listesine eklenir, böylece onu bir Implies dosyasına koymanız gerekmez. Ancak onun altına yeni bir özel kategori olacağını düşünerek bir alt dizin eklememelisiniz. generic dizini iki amaca hizmet eder. İlki; Makefile dosyaları, generic dizininde olmayan bir dosyanın sistem bağımlı bir sürümüne bakma endişesi duymazlar. Yani bir sistem bağımlı kaynak dosyası, diğer platformalarda gerçeklenmeyen yordamlar içeriyorsa bile, generic içinde benzeri olmalıdır. İkincisi; Makefile dosyaları bir sistem bağımlı dosyanın sisteme özel bir sürümünü derlemek için bulamıyorsa bu dosyanın generic sürümü kullanılır.

Bir generic dosyasında makineden bağımsız C yordamlarını sadece kütüphanedeki makineden bağımsız işlevleri kullanarak gerçekleştirebiliyorsanız, öyle yapmalısınız. Aksi takdirde onları içiboş işlevler yapın. Bir *içiboş* (stub) işlev belli bir makina ya da işletim sistemi için gerçeklenemeyen bir işlevdir (yani adı var kendi yok). İçiboş işlevler daima bir hata döndürür ve **errno** değişkenine **ENOSYS** (İşlev gerçeklenmedi) hatasını atarlar. Bkz. *Hata Bildirme* (sayfa: 31). Bir içiboş işlev tanımlayacaksanız, işlevin tanımından sonra, *işlev* bu işlevin ismi olmak üzere stub_warning(*işlev*) deyimini eklemelisiniz; ayrıca bu dosyaya stub-tag.h başlık dosyasını dahil etmelisiniz. Bu işlem, işlevin gnu/stubs.h dosyasında listelenmesini ve bu işlev kullanıldığında GNU ld'nin uyarı vermesini sağlar.

Nadiren bazı işlevler sadece belli sistemlerde kullanılabilir, diğerlerinde tanımlanmazlar; bunlar sistem bağımsız kaynak kodun bir yerlerinde ya da Makefile dosyalarında (generic dizini dahil) bulunmazlar, sadece o belli sistemlerin altdizinlerindeki Makefile dosyalarında bulunurlar.

Eğer ana kaynak dizininlerinden birindeki bir dosyanın makina ya da işletim sistemine bağlı sürümünü yazmak isterseniz o dosyayı `sysdeps/generic` içine taşımalı ve kendi gerçeklemenizi sisteme özel altdizinde yazmalısınız. Yalnız, sistem bağımlı bir dosyanın ana kaynak dizinlerinden birinde *bulunmaması* gerektiğine dikkat edin.

`sysdeps` altdizinlerinin her birinde mevcut olması gereken bir kaç özel dosya vardır:

`Makefile`
: O makina ya da makina sınıfı ve işletim sistemi için bir **`make`** dosyası. Bu dosya, ana kaynak dizinindeki ve onun alt dizinlerindeki `Makefile` dosyalarının **`Makerules`**'u tarafından içerilir. İçerildiği `Makefile` dosyalarının değişken kümesini değiştirebilir ya da yeni kurallar ekleyebilir. Farklı değişken kümelerinin ve kütüphanenin farklı bölümlerinin kurallarının seçilmesi için **`subdir`** değişkenini temel alan GNU **`make`** koşullu yönergelerini kullanabilir. Ayrıca, kütüphaneye dahil edilecek ek modülleri **`make`** değişkeni **`sysdep-routines`**'e atayabilir. Modülleri eklemek için **`routines`** değil **`sysdep-routines`** değişkenini kullanmalısınız çünkü birincisi ana kaynak dizinindeki alt dizinlerden nelerin dağıtıma gireceğini belirtmek için kullanılır.

  Her alt dizinin `Makefile` dosyası, alt dizin listesinde bulundukları sırayla aranırlar. Birden fazla sistem bağımlı `Makefile` içerileceğinden her birinin basitçe atanması yerine **`sysdep-routines`**'e eklenebilir:

```
sysdep-routines := $(sysdep-routines) foo bar
```

`Subdirs`
: Bu sistemin dosyaları için ana dizin olan dizinin altındaki tüm dizinlerin isimlerini içeren dosya. Bu altdizinler, kütüphane kaynak ağacındaki sistem bağımsız altdizinler gibi işlem görür.

  Bu `sysdeps` altdizinindeki sistem için gerçeklenen kütüphane tamamen yeni işlevler ve başlık dosyalarından oluşacaksa bunu kullanın. Örneğin, `sysdeps/unix/inet/Subdirs` dosyası **`inet`** içerir; `inet` dizininde sadece interneti destekleyen sistemlerdeki kütüphane içinde yer alacak çeşitli ağ yönlenimli işlemler bulunur.

`configure`
: Bu dosya yapılandırma için kullanılan bir kabuk betiğidir. Ana kaynak dizinindeki **`configure`** betiği seçilen her sistem bağımlı dizindeki **`configure`** betiğini kabuğun `.` komutunu kullanarak okur. **`configure`** betikleri genelde `configure.in` dosyalarından Autoconf tarafından üretilir.

  Bir sistem bağımlı **`configure`** betiği genellikle **`DEFS`** ve **`config_vars`** kabuk değişkenlerine birşeyler ekler; ayrıntılar için ana kaynak dizinindeki **`configure`** betiğine bakınız. Bu betik, ana kaynak dizinindeki **`configure`** betiğinin **`--with-`***paket* seçeneğiyle çalıştırılması sonucunda etkin olur. **`--with-`***`paket=değer`* gibi bir seçenekle **`configure`** betiği **`with_`***paket* değişkenine (*paket* ismindeki tire işaretlerini altçizgiye dönüştürülerek) *değer* değerini atar; eğer seçenek **`--with-`***paket* şeklinde argümansız olarak belirtilirse, **`with_`***paket* değişkenine **`yes`** değerini atar.

`configure.in`
: Bu dosya Autoconf tarafından bu dizin içinde **`configure`** betiğini oluşturmakta kullanılır. Autoconf ile ilgili açıklamalar için Autoconf Kılavuzundaki Giriş (Introduction) bölümüne bakınız. Ya **`configure`** betiğini ya da `configure.in` dosyasını yazmalısınız, ikisini birden değil. `configure.in` dosyasındaki ilk satırda bir **`m4`** makrosu olan **`GLIBC_PROVIDES`** çağrılmalıdır. Bu makro ana kaynak dizinindeki **`configure`** betiğinin kullandığı Autoconf makroları için çeşitli **`AC_PROVIDE`** çağrıları yapar; bunsuz bu makrolar Autoconf tarafınfan yine de gerekmediği halde çağrılabilir.

Böylece genel sistem, sistem bağımlılıklarından yalıtılır.

D.2.1. Hiyerarşi Uzlaşımları

Bir GNU yapılandırma ismi üç parçadan oluşur: İşlemci türü, üretici ismi ve işletim sistemi. **configure** betiği bunları sistem bağımlı dizinlerin hangilerine bakacağını bilmek için kullanır. Eğer **configure** betiğine **--nfp** seçeneği belirtilmemişse, ayrıca, *makina*/fpu dizini de kullanılır. Bir işletim sisteminin genellikle bir *temel işletim sistemi* vardır; örneğin, **Linux** işletim sistemi için bu temel işletim sistemi **unix/sysv**'dir. Dizinlere bakılmasını sağlayacak algoritma basittir: **configure** betiği, temel işletim sistemi, üretici, işlemci türü ve işletim sissemi sırasıyla bir liste oluşturur. Bunların aralarına bölü işaretleri yerleştirerek dizinleri elde eder; örneğin, yapılandırma **i686-linux-gnu** ise bu unix/sysv/linux/i386/i686 ile sonuçlanır. **configure** betiği sırayla dizinleri kaldırarak diğer dizileri de dener, böylece diğerlerinin yanında unix/sysv/linux ve unix/sysv dizinleri de ayrıca denenir. İşletim sistemlerinde sürüm numaraları her zaman çok önemli olmaz; örneğin irix6.2 ve irix6.3 dizinlerinin isimlerinde **configure** noktadan sonraki kısımları atarak da deneme yapar.

Bir örnek olarak, **i686-linux-gnu** yapılandırmasında denenecek dizinlerin bir listesine yer verilmiştir (**crypt** ve **linuxthreads** eklentileriyle):

```
sysdeps/i386/elf
crypt/sysdeps/unix
linuxthreads/sysdeps/unix/sysv/linux
linuxthreads/sysdeps/pthread
linuxthreads/sysdeps/unix/sysv
linuxthreads/sysdeps/unix
linuxthreads/sysdeps/i386/i686
linuxthreads/sysdeps/i386
linuxthreads/sysdeps/pthread/no-cmpxchg
sysdeps/unix/sysv/linux/i386
sysdeps/unix/sysv/linux
sysdeps/gnu
sysdeps/unix/common
sysdeps/unix/mman
sysdeps/unix/inet
sysdeps/unix/sysv/i386/i686
sysdeps/unix/sysv/i386
sysdeps/unix/sysv
sysdeps/unix/i386
sysdeps/unix
sysdeps/posix
sysdeps/i386/i686
sysdeps/i386/i486
sysdeps/libm-i387/i686
sysdeps/i386/fpu
sysdeps/libm-i387
sysdeps/i386
sysdeps/wordsize-32
sysdeps/ieee754
sysdeps/libm-ieee754
sysdeps/generic
```

Farklı makina mimarileri teamülen ana kaynak dizinindeki sydeps dizin ağacındaki altdizinlerdir. Örneğin,

`sysdeps/sparc` ve `sysdeps/m68k`. Bunlar bu makina mimarilerine özgü dosyaları içerir, bu mimarilerde kullanılan işletim sistemlerine özgü dosyaları içermezler. Bu mimarilerde özelleştirilmiş alt dizinler de olabilir; örneğin, `sysdeps/m68k/68020`. Belli bir makinada kullanılan aritmetik işlemciye özel kod `sysdeps/`*`makina`*`/fpu` dizinlerinde bulunur.

`sysdeps` dizininin altında belli bir makina mimarisine ait olmayan birkaç dizin vardır.

`generic`
: Evvelce açıklandığı gibi (*GNU C Kütüphanesinin Uyarlanması* (sayfa: 958)), bu alt dizindekiler tüm diğerlerinden sonra her yapılandırma tarafından dolaylı olarak kullanılır.

`ieee754`
: IEEE 754 kayan nokta biçimi (IEEE 754 tek hassasiyetli biçim için **`float`** türü ve IEEE 754 çift hassasiyetli biçim için **`double`** türü) kullanan kodun bulunduğu dizin. Genellikle bu dizin makina mimarisine özel dizin içindeki `Implies` (`m68k/Implies` gibi) dosyasında belirtilir.

`libm-ieee754`
: IEEE 754 uyumlu kayan nokta biçimi kullanılan platformların matematik kütüphanesinin gerçeklenimini içerir.

`libm-i387`
: Bu özel bir durumdur. Aslında kodun ideal yeri `sysdeps/i386/fpu` olmalıydı ama çeşitli sebeplerle ayrı tutulmaktadır.

`posix`
: Bu dizin POSIX.1 işlevlerine konu olan kütüphanedeki şeylerin gerçeklenimini içerir. Bazı POSIX.1 işlevlerinin kendilerini içerir. Şüphesiz, POSIX.1 kendi kurallarına göre tamamen gerçeklenemez, bu bakımdan sadece `posix` kullanan bir yapılandırma tamam olamayacaktır.

`unix`
: Bu dizin Unix benzeri şeyler içindir. Bkz. *GNU C Kütüphanesinin Unix Sistemlerine Uyarlanması* (sayfa: 961). `unix`, `posix`'i uygular. `unix` dizininin altında özel amaçlı bazı dizinler bulunur:

  `unix/common`
  : Bu dizin hem BSD hem de System V 4. sürümünde ortak olan şeyleri içerir. `unix/bsd` ve `unix/sysv/sysv4` her ikisi de `unix/common` uygular.

  `unix/inet`
  : Bu dizin Unix sistemlerindeki `socket` ve ilgili işlevler içindir. `unix/inet/Subdirs` dosyasında `inet` üst seviyeli alt dizindir. **`unix/common`**, `unix/inet` uygular.

`mach`
: Bu dizin, CMU'daki Mach mikroçekirdeğini (GNU işletim sistemi içerir) temel alan şeyleri içerir. Diğer temel işletim sistemlerinin (örneğin, VMS) `sysdeps` hiyerarşisi içinde `unix` and `mach` ile aynı seviyede kendi dizinleri vardır.

D.2.2. GNU C Kütüphanesinin Unix Sistemlerine Uyarlanması

Unix sistemlerinin çoğu temelde birbirlerine çok benzerler. Farklı makinalar için çeşitleri olduğu gibi çekirdek tarafından sağlanan oluşumlara bağlı çeşitleri de vardır. Fakat işletim sistemi oluşumları ile arayüz çoğu parçası bakımında neredeyse tektip ve basittir.

Unix sistemleri için kod `sysdeps` dizini altındaki `unix` dizininde bulunur. Bu dizin Unix çeşitlemelerine özel alt dizinler hatta alt dizin ağaçları içerir.

Unix sistemlerindeki sistem çağrılarını oluşturan işlevler `syscalls.list` isimli dosyadaki belirtimlerden özdevinimli üretilen makina kodları şeklinde gerçeklenmiştir. `sysdeps/unix` altında ve bunun alt dizinlerinde böyle çeşitli dosyalar vardır. Bazı sistem çağrıları soneki **`.S`** olan dosyalarla gerçeklenmiştir; örneğin, `_exit.S`. **`.S`** uzantılı dosyalar makina kodu çeviricisinden geçmeden önce C önişlemcisi tarafından işlenirler.

Bu dosyaların hepsi `sysdep.h` dosyasında tanımlanmış olan makroları kullanırlar. `sysdeps/unix` içindeki `sysdep.h` bunları kısmen tanımlar; başka bir dizindeki `sysdep.h` dosyası ile belli bir makina ve işletim sistemi çeşidi için olanlar tanımlanarak seri tamamlanmalıdır. `sysdeps/unix/sysdep.h` dosyasına ve makinaya özel **`sysdep.h`** gerçeklemelerine bakarsanız bu makroları ve ne yaptıklarını görebilirsiniz.

`unix` dizinindeki sisteme özel `Makefile` dosyası (`sysdeps/unix/Makefile`), kütüphanenin derlenmesini istediğiniz hedef Unix sistemindeki dosyalardan çeşitli dosyaların üretilmesini sağlayan kurallar içerir. Bu şekilde üretilen tüm dosyalar derlenmiş nesne dosyalarının tutulduğu dizinlere konur; yani kaynak ağacını etkilemezler. Üretilen dosyalar `ioctls.h`, `errnos.h`, `sys/param.h` ve `errlist.c`'dir (kütüphanenin **`stdio`** bölümü için).

E. GNU C Kütüphanesini Yazanlar

GNU C kütüphanesi Roland McGrath tarafından yazılmaya başlandı, şu an bakımı Ulrich Drepper tarfından yapılmaktadır. Kütüphanenin bazı parçaları başkaları tarafından yazılmış ya da üzerinde başka kişiler çalışmıştır.

- **`getopt`** ve ilgili işlevler Richard Stallman, David J. MacKenzie ve Roland McGrath tarafından yazılmıştır.
- Katıştırarak sıralama işlevi **`qsort`** Michael J. Haertel tarafından yazılmıştır.
- **`qsort`** tarafından son çare olarak kullanılan hızlı sıralama işlevi Douglas C. Schmidt tarafından yazılmıştır.
- Bellek ayırma işlevleri **`malloc`**, **`realloc`** ve **`free`** ve bunlarla ilişkili kodlar Michael J. Haertel, Wolfram Gloger ve Doug Lea tarafından yazılmıştır.
- Çoğu dizge işlevinin (**`memcpy`**, **`strlen`**, vs.) hızlı çalışmasının gerçeklenmesi Torbjörn Granlund tarafından yapılmıştır.
- **`tar.h`** başlık dosyası David J. MacKenzie tarafından yazılmıştır.
- Ultrix 4'ü çalıştıran MIPS DECStation'a uyarlama (**`mips-dec-ultrix4`**) Brendan Kehoe ve Ian Lance Taylor tarafından sağlanmıştır.
- DES şifreleme işlevi **`crypt`** ile bununla ilişkili işlevler Michael Glad tarafından yazılmıştır.
- **`ftw`** ve **`nftw`** işlevleri Ulrich Drepper tarafından yazılmıştır.
- SunOS paylaşımlı kütüphanelerine destek için başlangıç kodu Tom Quinn tarafından yazılmıştır.
- **`mktime`** işlevi Paul Eggert tarafından yazılmıştır.
- Dynix version 3'ü çalıştıran Sequent Symmetry uyarlaması (**`i386-sequent-bsd`**) Jason Merrill tarafından sağlanmıştır.
- Zaman dilimi destek kodu, kamuya açık zaman dilimi paketinden Arthur David Olson ve arkadaşları tarafından uyarlanmıştır.
- OSF/1'i çalıştıran DEC Alpha uyarlaması (**`alpha-dec-osf1`**) Roland McGrath tarafından yazılan kodlar kullanılarak Brendan Kehoe tarafından yazılmıştır.
- Irix 4 çalıştıran SGI makinalara uyarlama (**`mips-sgi-irix4`**) Tom Quinn tarafından yapılmıştır.

- MIPS mimarisine Mach ve Hurd kodunun uyarlanması (**mips-***birşey*-**gnu**) Kazumoto Kojima tarafından yapılmıştır.
- **printf** ve arkadaşları tarafından kullanılan gerçek sayı yazma ve **scanf** ve arkadaşları tarafından kullanılan gerçek sayı okuma işlevi **strtod** ve arkadaşları Ulrich Drepper tarafından yazılmıştır. Bu işlevler tarafından kullanılan çok hassasiyetli tamsayı işlevleri Torbjörn Granlund tarafından yazılan GNU MP'den alınmıştır.
- Kütüphanedeki uluslararasılaştırma desteği ile **locale** ve **localedef** uygulamaları Ulrich Drepper tarafından yazılmıştır. Ulrich Drepper ayrıca, kendi yazdığı GNU **gettext** paketinden ileti katalogları desteğini kütüphaneye uyarlamıştır. Ayrıca **catgets** desteği ile çokbaytlı karakter ve geniş karakter desteğini sağlayan işlevlerin tamamını yazmıştır (**wctype.h**, **wchar.h**, vs.).
- **nsswitch.conf** mekanizmasının, dosyalarının ve DNS arkaplanının gerçeklenimi Peter Eriksson tarafından tanımlanan arkaplan arayüzü temel alınarak Ulrich Drepper ve Roland McGrath tarafından tasarlanmış ve yazılmıştır.
- Linux i386/ELF (**i386-***birşey*-**linux**) uyarlaması, Hongjiu Lu'nun GNU C kütüphanesi Linux sürümünde yaptığı çalışmalar temel alınarak Ulrich Drepper tarafından yapılmıştır.
- Linux/m68k (**m68k-***birşey*-**linux**) uyarlaması Andreas Schwab tarafından yapılmıştır.
- Linux/ARM (**arm-***birşey*-**linuxaout**) ve tekbaşına ARM (**arm-***birşey*-**none**) uyarlaması, IPv6 destek kodunu da yazan Philip Blundell tarafından yapılmıştır.
- Richard Henderson ELF özdevimli ilintileme kodunu ve Alpha işlemci için diğer destek kodlarını yazdı.
- David Mosberger–Tang Linux/Alpha (**alpha-***anything*-**linux**) uyarlamasını yaptı.
- Linux'un PowerPC'e uyarlaması (**powerpc-***birşey*-**linux**) Geoffrey Keating tarafından yapıldı.
- Miles Bader argp argüman çözümleme paketini ve argz/envz arayüzlerini yazdı.
- Stephen R. van den Berg oldukça eniyilenmiş bir **strstr** işlevi yazdı.
- Ulrich Drepper **hsearch** ve **drand48** ailesi işlevleri; **random** ailesinin evresel sürümlerini (**…_r**); System V paylaşımlı bellek ve IPC destek kodunu; ve i*x*86 işlemciler için en yüksek seviyede eniyilenmiş dizge işlevlerini yazdı.
- Sun Microsystems'ın **fdlibm-5.1**'inden alınan matematik işlevleri J.T. Conklin, Ian Lance Taylor, Ulrich Drepper, Andreas Schwab ve Roland McGrath tarafından değiştirilerek uyarlandı.
- **stdio** işlevlerinin gerçeklendiği **libio** kütüphanesi bazı platformlar için Per Bothner tarafından yazıldı ve sonra Ulrich Drepper tarafından üzerinde değişiklikler yapıldı.
- Eric Youngdale ve Ulrich Drepper nesnelerin sembol seviyesinde sürümlenmesini gerçekleştirdi.
- Thorsten Kukuk NIS (YP) ve NIS+, güvenlk seviyesi 0, 1 ve 2 için bir gerçekleme sağladı.
- Andreas Jaeger matematik kütüphanesi için sınama yordamlarını yazdı.
- Mark Kettenis utmpx arayüzünü ve bir utmp artalan uygulaması gerçekleştirdi.
- Ulrich Drepper karakkter dönüşüm işlevlerini (**iconv**) ekledi.
- Thorsten Kukuk NSS (nscd) için bir arabellekleme artalan uygulaması gerçekleştirdi.
- Tim Waugh POSIX.2 wordexp işlev ailesi için bir gerçekleme sağladı.

- Mark Kettenis Hesiod NSS modülünü üretti.
- Internet ile ilgili kod (**inet** altdizinini çoğu) ve diğer muhtelif işlevlerle başlık dosyaları bir kaç küçük değişiklikle ya da değiştirilmeksizin 4.4 BSD'den alındı. Bu kod için kopyalama izinleri kaynak paketin ana dizinindeki LICENSES dosyasında bulunabilir.
- Rasgele sayı üretim işlevleri **random**, **srandom**, **setstate** ve **initstate** ile **rand** ve **srand** işlevleri Berkeley'deki Kaliforniya Üniversitesi için Earl T. Cohen tarafından yazıldı. Kodun telif hakkı Regents of the University of California'ya aittir. GNU C kütüphanesi ve ISO C standardına uyum için çok küçük bazı değişiklikler yapılmış olsa da işlevsel kod Berkeley'indir.
- DNS çözümleyici kod doğrudan BIND 4.9.5'ten alınmıştır; telif hakkı UC Berkeley ve from Digital Equipment Corporation'a aittir. DEC lisansının metni için LICENSES dosyasına bakınız.
- Sun RPC destek kodu tıpatıp Sun'ın rpcsrc–4.0 dağıtımından alınmıştır; lisans metni için LICENSES dosyasına bakınız.
- Mach için destek kodu kısmen CMU'nun Mach 3.0'ından alınmıştır, fakat farklı bir lisans altında; lisans metni için LICENSES dosyasına bakınız.
- IA64 matematik işlevlerinin birçoğu Intel'in bir özgür lisans altında kullanılabilir kıldığı "Highly Optimized Mathematical Functions for Itanium"'dan alınmıştır; ayrıntılar için LICENSES dosyasına bakınız.
- **getaddrinfo** ve **getnameinfo** işlevleri ve destekleme kodu Craig Metz tarafından yazılmıştır; lisans metni için LICENSES dosyasına bakınız.
- IEEE 64 bitlik çift hassasiyetli matematik işlevlerinin birçoğu (sysdeps/ieee754/dbl-64 alt dizinindekiler) IBM tarafından yazılmış IBM Accurate Mathematical Library'den alınmıştır.

F. Özgür Kılavuzlar

Bugün özgür yazılım topluluğundaki en büyük eksiklik yazılım değildir; özgür yazılımların iyi özgür belgeler içermemesidir. Bizim en önemli yazılımlarımızın çoğu özgür başvuru kılavuzları ve tanıtıcı metinler ile gelmemektedir. Belgeleme her yazılım paketinin en önemli parçasıdır; önemli bir özgür yazılım paketi bir özgür kılavuz ve bir özgür öğretici ile gelmiyorsa bu büyük bir boşluktur. Günümüzde böyle bir sürü boşluk var.

Örnek olarak Perl'i ele alalım. Öğretici kılavuzların kullanımı normalde özgür değildir. Bu nasıl oldu? Bu kılavuzların yazarları özgür yazılım dünyasından onları dışlayan bazı sınırlamalar getirdiler: kopyalanamaz, değiştirilemez, kaynak dosyaları kullanılamaz.

Bu ne ilkti ne de son olacaktı. Bazan, bir GNU kullanıcısı bir kılavuzu büyük bir istekle yazmaya başladığını, özgür toplumun desteğini istediğini duyuruyor, ancak sonradan bir yayıncıyla anlaşma imzalayarak kılavuzu özgür olmaktan çıkardığını herşeyi mahvettiğini duyuyoruz.

Özgür belgeleme, özgür yazılım gibidir, bir fiyat konusu değil, bir özgürlük konusudur. Sorun özgür olmayan kılavuzlara yayıncının koyduğu fiyat değildir, bu iyi birşeydir. (Free Software Foundation'da kılavuzların basılı kopyalarını satıyor.) Asıl sorun kılavuzun kullanımına getirilen sınırlamalardır. Özgür kılavuzların özgür kaynak kodlar gibi, kopyalama ve değiştirme izinleri vardır. Özgür olmayan kılavuzlar buna izin vermez.

Bir özgür kılavuzun özgürlük kuralları özgür yazılımla hemen hemen aynıdır. Yeniden dağıtımına izin verilir (ticari yeniden dağıtım dahil), böylece kılavuz yazılımın her kopyasına eşlik edebilir, hem internetten indirilerek hem de kağıt üzerinde.

Teknik içeriğin değiştirilmesine izin vermek son derece önemlidir de. Birileri yazılımda değişiklik yaptığında veya birşeyler eklediğinde vicdanları varsa kılavuzu da değiştireceklerdir. Bir yazılımın değiştirilmiş bir sürümünü

belgelemek için yeni bir kılavuzun yazılması bakımından sizi seçimsiz bırakacak bir kılavuz bizim toplumumuzda aslında yoktur.

Bazı değişklikler için sınırlamalar olması kabul edilebilir. Örneğin, özgün belge yazarının telif hakkı uyarısının, dağıtım kurallarının ya da yazar listesinin korunması istenebilir. Ayrıca değiştirilen sürüm dağıtılırken bunları içermesi sorun olmaz. Hatta bazı bölümerin değiştirilmemesi hatta kaldırılmaması isteği de kabul edilebilir, ancak bunların teknik bilgi içermemesi gerekir (bu bölüm gibi). Bu çeşit sınırlamalar kabul edilebilir, çünkü bunlar kılavuzun özgür toplumca normal kullanımını engellemez.

Ancak, kılavuzun tüm *teknik* içeriğinin değiştirilebilmesi mümkün olmalıdır. Bunun sonucu olan kılavuzunda tüm bilinen ortamlarda bilinen kanallardan dağıtımı mümkün olmalıdır. Aksi takdirde sınırlamalar kılavuzun kullanımını engeller, artık özgür olmaz ve bizim o kılavuzu başka bir kılavuzla değiştirmemiz gerekir.

Lütfen bu konu hakkında sözünüzü sakınmayın. Toplumumuz hala sahipli yayıncılığa kılavuz kaptırmaya devam ediyor. Özgür yazılımların özgür kılavuzları ve özgür öğreticileri gerektirdiğini savunuyorsak, şüphesiz çok geç olmadan özgür belgelerin yazılmasınını destekleyenler de böyle yapacak ve özgür yazılım toplumuna sadece özgür kılavuzlarla katılacaklartır.

Eğer belge yazıyorsanız, onun lütfen GNU Özgür Belgeleme Lisansı (GFDL) ya da başka bir özgür belgeleme lisansı altında yayınlanması için ısrarcı olunuz. Bu kararın sizin onayınızı gerektirdiğini unutmayın—yayıncının kararlarına uymak zorunda değilsiniz. Eğer ısrarcı olursanız bazı ticari yayıncılar bir özgür lisansı kullanırken bir seçenek teklif etmez; bu tamamen sizin ne istediğinizi ne kadar sebatla belirttiğinize bağlıdır. Eğer yayıncı sizi reddederse, başka yayıncıları deneyin. Eğer size önerilen bir lisansın özgür olup olmadığı konusunda kararsızlığa düşerseniz bize (tabii, İngilizce) yazın: <licensing (at) gnu.org>.

Daha özgür, telif hakkını belgeyi özgürleştirmek amacıyla kullanmış yazarların kılavuzlarını ve öğretici belgelerini satın alarak, yayıncıları bunlardan daha fazla satmaya teşvik edebilirsiniz, özellikle yayıncı bu yayınların hazırlanması veya kapsamlı olarak iyileştirilmesi için ücret ödemişse. Bunun yanında özgür olmayan belgeleri satın almaktan kesinlikle kaçının. Bir belgeyi satın almadan önce dağıtım kurallarını inceleyin. İnceleyin ve sizinle iş yapmak isteyen herkesten (bu şartnameyi inceleme) özgürlüğünüze saygı göstermesini isteyin. Kitabın tarihçesini inceleyin ve hazırlanması için ücret ödemiş olan yayıncıları veya kitap üzerinde çalışmış olan yazarları ödüllendirin.

Free Software Foundation sitesinde özgür belgeleri yayınlayanların bir listesini bulabilirsiniz: `http://www.fsf.org/doc/other-«free-«books.html`.

Çevirmenden: Çeviri burada bitti. Bundan sonraki bölümde bu bölümün ingilizcesini bulacaksınız. Belgenin sonundaki dizinler, belge içeriğinden programlama sonucu kendiliğinden üretilen bölümlerdir.

Sonraki üç bölüm bu "özgür belge"nin dil bakımından bir çeviri için dahi özgür olmayan bölümleri. Türkçe bir belgede bu ingilizce içeriği belgenin "özgür lisansı" nedeniyle bulundurmak zorunda kalmaktan üzgünüz. İnşallah bir gün bu belgenin yazarları özgür belgelerinin lisansı olarak belgelerinin başka dillere çevirilerine kendi dillerinde yazılmış bölümleri sokuşturmaya çalışmayan daha özgür bir lisans seçerler. Bu belgeyi okuyacak olanlar, ingilizce bölümlerden birşey anlamayacaktır, onun için burada kısa bir açıklama yapmam iyi olacak. Diğer iki bölümden biri glibc paketinin yazılım lisansı (bir çevirisi[B3466] var) diğeri ise bu belgenin "özgür lisansı" (çevirilerle ilgili zorlamaları özgürlüğümüzün bir parçası olarak haketmediğimizden gfdl'yi dilimize çevirmedik).

G. Free Software Needs Free Documentation

The biggest deficiency in the free software community today is not in the software—it is the lack of good free documentation that we can include with the free software. Many of our most important programs do not come with free reference manuals and free introductory texts. Documentation is an essential part of any software package; when an important free software package does not come with a free manual and a free tutorial, that is a major gap. We have many such gaps today.

Consider Perl, for instance. The tutorial manuals that people normally use are non–free. How did this come about? Because the authors of those manuals published them with restrictive terms—no copying, no modification, source files not available—which exclude them from the free software world.

That wasn't the first time this sort of thing happened, and it was far from the last. Many times we have heard a GNU user eagerly describe a manual that he is writing, his intended contribution to the community, only to learn that he had ruined everything by signing a publication contract to make it non–free.

Free documentation, like free software, is a matter of freedom, not price. The problem with the non–free manual is not that publishers charge a price for printed copies—that in itself is fine. (The Free Software Foundation sells printed copies of manuals, too.) The problem is the restrictions on the use of the manual. Free manuals are available in source code form, and give you permission to copy and modify. Non–free manuals do not allow this.

The criteria of freedom for a free manual are roughly the same as for free software. Redistribution (including the normal kinds of commercial redistribution) must be permitted, so that the manual can accompany every copy of the program, both on–line and on paper.

Permission for modification of the technical content is crucial too. When people modify the software, adding or changing features, if they are conscientious they will change the manual too—so they can provide accurate and clear documentation for the modified program. A manual that leaves you no choice but to write a new manual to document a changed version of the program is not really available to our community.

Some kinds of limits on the way modification is handled are acceptable. For example, requirements to preserve the original author's copyright notice, the distribution terms, or the list of authors, are ok. It is also no problem to require modified versions to include notice that they were modified. Even entire sections that may not be deleted or changed are acceptable, as long as they deal with nontechnical topics (like this one). These kinds of restrictions are acceptable because they don't obstruct the community's normal use of the manual.

However, it must be possible to modify all the *technical* content of the manual, and then distribute the result in all the usual media, through all the usual channels. Otherwise, the restrictions obstruct the use of the manual, it is not free, and we need another manual to replace it.

Please spread the word about this issue. Our community continues to lose manuals to proprietary publishing. If we spread the word that free software needs free reference manuals and free tutorials, perhaps the next person who wants to contribute by writing documentation will realize, before it is too late, that only free manuals contribute to the free software community.

If you are writing documentation, please insist on publishing it under the GNU Free Documentation License or another free documentation license. Remember that this decision requires your approval—you don't have to let the publisher decide. Some commercial publishers will use a free license if you insist, but they will not propose the option; it is up to you to raise the issue and say firmly that this is what you want. If the publisher you are dealing with refuses, please try other publishers. If you're not sure whether a proposed license is free, write to <licensing (at) (at) gnu.org>.

You can encourage commercial publishers to sell more free, copylefted manuals and tutorials by buying them, and particularly by buying copies from the publishers that paid for their writing or for major improvements. Meanwhile, try to avoid buying non–free documentation at all. Check the distribution terms of a manual before you buy it, and insist that whoever seeks your business must respect your freedom. Check the history of the book, and try reward the publishers that have paid or pay the authors to work on it.

The Free Software Foundation maintains a list of free documentation published by other publishers, at ***http://www.fsf.org/doc/other–free–books.html***.

H. GNU Lesser General Public License

Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

H.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two–step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non–free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the *Lesser* General Public License because it does *Less* to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non–free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de–facto standard. To achieve this, non–free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non–free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non–free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non–free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

4. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

   A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

   "Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

5. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate

copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

6. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. The modified work must itself be a software library.

   b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

      (For example, a function in a library to compute square roots has a purpose that is entirely well–defined independent of the application. Therefore, Subsection 2d requires that any application–supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

   In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

7. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

   This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

8. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine–readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

9. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

10. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

    You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

    a. Accompany the work with the complete corresponding machine–readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine–readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

    b. Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface–compatible with the version that the work was made with.

c. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

11. You may place library facilities that are a work based on the Library side–by–side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

    a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

    b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

12. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

13. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

14. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

15. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty–free redistribution of the Library by all those who receive copies

directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

16. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

17. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

18. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

19. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

20. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE

LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

H.2. How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the library's name and an idea of what it does.
Copyright (C) year  name of author

This library is free software; you can redistribute it and/or modify it
under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 2.1 of the License, or (at
your option) any later version.

This library is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307,
USA.
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the library
'Frob' (a library for tweaking knobs) written by James Random Hacker.

signature of Ty Coon, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

I. GNU Free Documentation License

Version 1.1, March 2000

```
Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front–matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front–Cover Texts or Back–Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine–readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ascii without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard–conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine–generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which

do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

3. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

4. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front–Cover Texts on the front cover, and Back–Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine–readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly–accessible computer–network location containing a complete Transparent copy of the Document, free of added material, which the general network–using public has access to download anonymously at no charge using public–standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

5. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

   B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front–matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front–Cover Text, and a passage of up to 25 words as a Back–Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front–Cover Text and one of Back–Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

7. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self–contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

9. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

I.1. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front–Cover Texts, write "no Front–Cover Texts" instead of "Front–Cover Texts being *list*"; likewise for Back–Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Kavramlar Dizini

Semboller

A

B

C

Ç

D

E

F

G

K

L

M

S

Ü

V

W

Y

Z

Veri Türleri Dizini

R

S

T

U

V

W

İşlevler Dizini

A

B

C

D

E

F

G

H

I

İ

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Değişkenler Dizini

A

B

C

D

E

F

G

H

I

İ

L

M

N

O

P

R

S

T

U

V

W

X

Dosyalar Dizini

T

U

V

W

Notlar

Belge içinde dipnotlar ve dış bağlantılar varsa, bunlarla ilgili bilgiler bulundukları sayfanın sonunda dipnot olarak verilmeyip, hepsi toplu olarak burada listelenmiş olacaktır.

(B4) http://haluk.buguner.name.tr

(B5) http://www.arayan.com/da

(B6) http://nilgun.buguner.name.tr/

(1) GCC seçeneklerinin bir listesini **gcc --help** ile görebilirsiniz.

(B65) file:/usr/include/errno.h

(B67) file:/usr/include/errno.h

(B96) file:/usr/include/string.h

(B97) file:/usr/include/string.h

(B99) file:/usr/include/stdio.h

(B134) file:/usr/include/obstack.h

(2) Ç.N. – Örneğin, Türkçedeki I ve İ harfleri gibi, I harfinin karakter numarası U+0049 iken İ harfininki U+0130 J hafinin karakter kodu ise U+004A'dır. Yani IİJ alfabetik sıralaması ile U+0049, U+0130, U+004A karakter kodu sıralaması farklıdır. O kadar ki, bazı karakterlerimizi sırf karakter kodlamasına göre sıralasaydık, sıralamada sona kalacaklardır ki bu sorunlar geçmişte yazılımcıları çok uğraştırmıştı. GNU C kütüphanesi bu sorunları kökten çözmüştür.

(3) Ç.N. — Böyle bir dizgeyi Türkçe'ye çevirirken şöyle yapardık:

```
"%d dosya silindi. %s"
```

Ve siz iletiyi genellikle şöyle görürdünüz: "5 dosya silindi. s"

(4) Bu listeye eklenmesini istediğiniz bilgileri <bug–glibc–manual (at) gnu.org> adresine bekliyoruz.

(5) Çevirenin Notu:
Her ne kadar Türkçe'de miktar belirtirken tekil/çoğul farkı yoksa da bir uygulama nedeniyle tekil ve çoğul biçimler için ayrı iletiler gerekli olmaktadır. Ben (NBB) GNU **gettext** paketinin ileti kataloğu dosyasının da çevirmeni olduğumdan o dosyadaki bir örneği vererek ne demek istediğimi açıklamaya çalışayım.

```
#: src/msgfmt.c:1095
#, c-format
msgid "...but some messages have one plural form"
msgid_plural "...but some messages have %lu plural forms"
msgstr[0] "...ama bazı iletiler tek çoğul biçim içeriyor"
msgstr[1] "...ama bazı iletiler %lu çoğul biçim içeriyor"
```

Burada dikkat ederseniz tekil biçimli ileti içinde miktar %lu belirtimi ile değil, yazıyla, **one** yazılmış. Bu kılavuzun orjinalinde belirtildiği gibi tek çoğul biçim kullanma şansımız böyle bir örnek karşısında kalmıyor. Çünkü Türkçe iletiye, İngilizce iletide bulunmayan %lu belirtecini koyarsak ve dosyayı **msgfmt** ile derlemeye çalışırsak bunun bir hata olduğunu görürüz. Böyle bir durumda her iki ileti için birer çeviri olmak zorunda, dolayısıyla bir tekil ve bir çoğul biçim belirtmek zorundayız:

```
Plural-Forms: nplurals=2; plural=n != 1;
```

Bunun bize bir zararı yok, en kötü durumda çoğul biçimli iletilerde birbirinin aynı iki ileti olur. Ama dosya böyle özel bir duruma sahip tek bir ileti dahi içermedikçe tekil biçim de kullanılabilir.

(B241) ../bashref/bashref.pdf

(B242) ../regex/regexinfo.pdf

(B248) ../regex/regexinfo.pdf

(B256) ../bashref/bashref_shell.features.pdf#bashref_shell.expansions

(6) Aslında, uçbirimlere özel işlevler, çoğu platformda IOCTL'lerle gerçeklenir.

(7) Ç.N.: Benim, "benzeş formül" diye adlandırdığım formülden özgün metinde "congruential formula" ismiyle bahsedilmiş. Bunun kullanımda olan bir türkçe karşılığı var mı bilmiyorum ama tam tarifi şöyle: özgün formülle tıpatıp aynısı olmasa da benzer sonucu veren (ve belki hesaplama kolaylığı getiren) bir başka formül. — [NBB]

(8) Ç.N.: Bu işlevleri içeren bir kodu **gcc** ile derlerken `-lm` seçeneğini vermeyi unutmayın.

(9) Ç.N.: Süre, iki mutlak zaman değeri arasındaki miktardır, zaman ise belli bir olaya göreli olarak tanımlanmış bir başlangıca göre geçen süredir; "miladi zaman", "hicri zaman", Uzay Yolu'nu izlediyseniz kaptanın günlüğünde bahsettiği "yıldız zamanı" `:-)` gibi.

(10) Jülyen takviminde M.S. 1582 de Papa III. Gergory zamanında Ekim'in 5'i ile 15'i arasındaki günler takvimlerden çıkarılmış ve bu suretle bozuk zaman düzeltilerek artık yılları hesaba karan Gregoryen takvimi oluşturulmuştur. Bu takvim ilk olarak 1752'de İngiliz ve Amerikan kolonilerinde uygulanmaya başlamıştır. Bu bakımdan eksik günlerin kimse farkına varmamıştır `:))`. Halen bu sistemi kullanmaktayız.

(11) Mutlak zaman başlangıcı (epoch), sistemden sisteme değişiklik gösterir. Unix için belirtildiği gibi 1 Ocak 1970, 00:00:00 GMT iken; VMS için 17 Kasım 1858, 00:00:00; Macintosh için 1 Ocak 1904 geceyarısıdır.

(12) Ç.N.:İngilizce düşünen acemiler için evet de, anadili ingilizce olmayanlar için uzun seçeneklerin de diğer iletiler gibi yerel dile çevrilmesi sağlansa iyi olacak.

(B965) ../man/man8/man8-«mount.pdf

(B966) ../man/man8/man8-«mount.pdf

(B974) ../termcap/termcap.pdf#termcap-«Find

(13) [Ç.N.] mutex (mutual exclusion'dan kısaltma): Bir karşılıklı red (muteks) nesnesi paylaşımlı özkaynaklara erişimi eşzamanlayan çoklu evreleri mümkün kılar. Bir muteks iki durumda olabilir: kilitli, kilitsiz. Bir muteks bir evre tarafından bir kere kilitlendi mi onu kilitlemeye çalışan diğer evreler engellenir. Kilitli evre muteksi bıraktığında (kilidini kaldırdığında) engellenen bloklardan biri onu kilitler ve işlem böyle sürüp gider.

(14) ABI: (Application Binary Interface kısaltması) – Bir uygulama ile işletim sistemi ya da hizmetler arasında erişimi sağlayan arayüz.

(15) Şimdi soracaksınız, niçin bu bilgi işlevlerin de ismine sokuluyor, modülün isminde zaten var. Yanıtı, biz bunun paylaşımlı nesenelerle birlikte ilintilenmesinin mümkün olmasını istiyoruz.

(16) İkinci bir açıklama da şu: Makefile'ları, **lib** ile başlamayan paylaşımlı nesnelerin oluşturulması için değiştirmeye üşendik, ama bunu kimseye söylemeyin.

(B1157) file:/usr/include/stdarg.h

(B3454) http://www.imaxx.net/~thrytis/glibc

(B3466) ../howto/lgpl.pdf

Bu dosya (glibc.pdf), belgenin XML biçiminin TEXLive ve belgeler-xsl paketlerindeki araçlar kullanılarak PDF biçimine dönüştürülmesiyle elde edilmiştir.

17 Ocak 2007