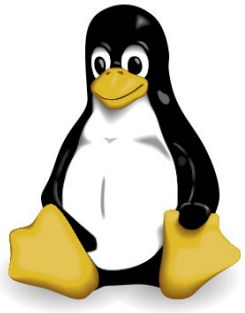
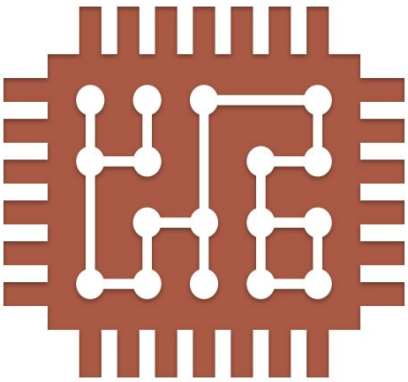


LINUX SİSTEM PROGRAMLAMA



Murat Demirten
Serkan Eser

İçindekiler

Önsöz

Linux Sistem Programlama kitabı, bu konuda verdiğimiz eğitim içeriklerinin kitap formatında bir araya getirilmesi ve güncellenmesi fikriyle ortaya çıktı. İçerik güncellemeleri halen devam etmekte olduğundan haftada veya ayda bir değişiklikleri kontrol etmeniz önerilir. Kitap listesine abone olup, kapsamlı değişikliklerden e-posta yoluyla haberdar olabilirsiniz.

Sorularınız İçin

Kitapla ilgili öneri ve düşüncelerinizi mdemirten@yh.com.tr adresinden bizimle paylaşabilirsiniz.

Ek olarak teknik sorularınızı http://linux-tips.org sitesinde özellikle **Programming** tartışma başlığı altında iletmeniz halinde, yanıtlamaya çalışacağımızı belirtmek isteriz. Katkı sağlayabilecek daha çok kişiye ulaşmak için site İngilizce olarak hazırlanmaktadır.

Yardımcı Kitap

- **Linux Sistem Yönetimi**
- **Linux Yazılım Notları**
- **Linux Yazılım Güvenliği**

kitaplarının da incelenmesinde fayda vardır. Bu kitapların içerikleri görece daha az olmakla birlikte, içerik girişleri devam etmektedir.

İlginizi çekmesi durumunda **Gömülü Linux** kitabımızı da inceleyebilirsiniz. Özellikle Linux tabanlı gömülü sistemler üzerine çalışanlar için faydalı olacaktır.

Telif Hakkı

Teşekkür

İçeriğe katkılarından dolayı Serkan Eser'e teşekkürlerimi sunuyorum, katkılarının devamını bekliyoruz :)

Önsöz

Unix/Linux platformlarında verimli uygulamalar geliştirebilmek için uygulama geliştiricinin kullandığı sistemin genel özelliklerinden ve sistem programlama disiplini altında değerlendirilen temel kavramlarla ilgili daha fazla bilgi sahibi olması gereklidir.

Bunu tek başına olumsuz bir faktör olarak değerlendirmemek gerekir. Esasen iyi bir uygulama geliştiricinin yolu, elbette sistem programlamadan da geçmelidir. Eğer *Java, Ruby on Rails* vb. gibi çok üst düzey soyutlamalar sunan gelişmiş bir dil/platform kullanıyorsanız, sistem programlama bilgisi günlük çalışmalarınız için elzem değildir. Bununla birlikte sistem programlama bilgisi, bu platformlarda dahi ortaya çıkan iş kalitesini artıran bir etken olacaktır. Ne kadar üst seviye bir dil olursa olsun, daha karmaşık konularda sistem programlamaya başvuru kaçınılmazdır.

Örnek olarak *Ruby 2.1 Garbage Collection* iyileştirmesi ve bir takım *sihirli* ortam değişkenlerinin etkisini anlayabilmek için, sistem programlama konularına önemli oranda aşina olmanız gerekir.

> - Ruby 2.1 Garbage Collection: ready for production
> - Ruby 2.1: Out-of-Band GC

Bu kitapta biz öncelikli olarak Linux Sistem Programlama konularına değinecek, yanısıra doğrudan sistem programlama ile ilgisi olmamakla birlikte ilginizi çekebilecek bazı ek başlıklara da yer vereceğiz.

Sistem Programlamaya Giriş

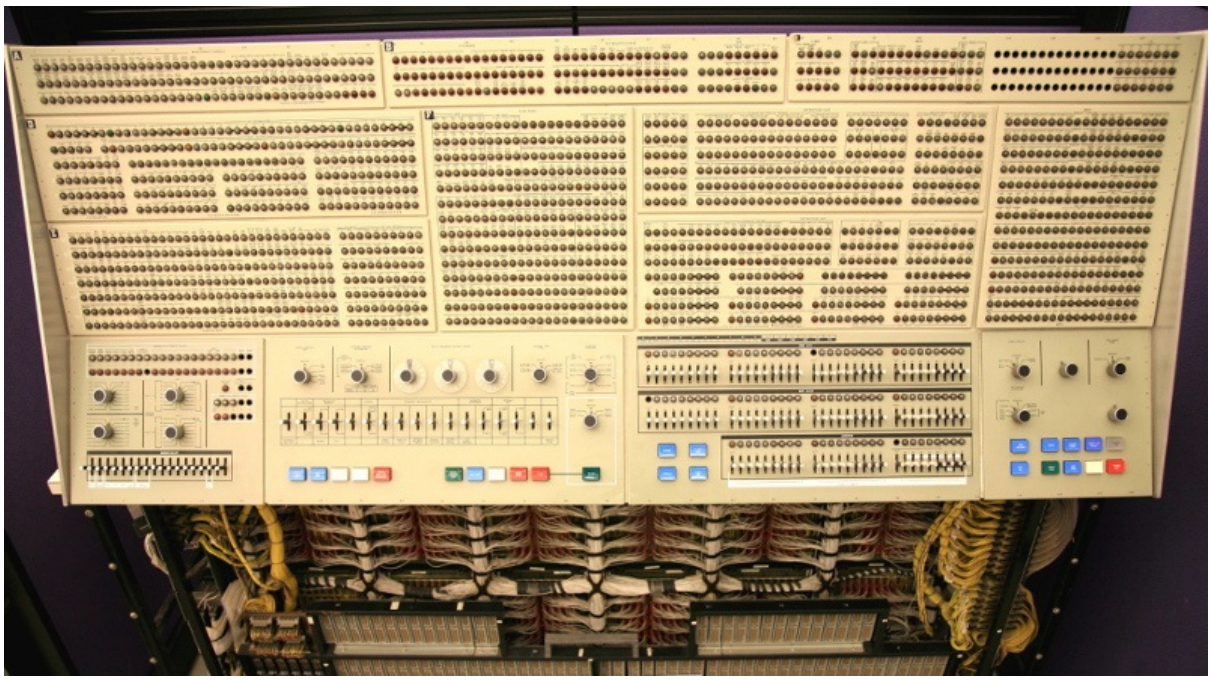
Sistem programlama kavramı temel olarak, uygulama geliştiricinin sistemle daha alt seviye bir katman üzerinden iletişim kurduğu dolayısıyla geliştirmelerin sisteme daha yakın bir noktada yapıldığı yazılım geliştirme modelini tarif eder.

Bununla birlikte herkesin bu kavramı ilk duyduğunda kafasında canlanan senaryo az çok farklılık gösterebilir ve muhtemelen bunların tümü de doğrudur.

2000 yılında üniversitede 3. sınıfa devam ederken, yeni dönemdeki ders seçeneklerinin birinin *Sistem Programlama* olduğunu gördüğümde çok sevindiğimi hatırlıyorum. O zamanlar da Linux ile yoğun biçimde ilgileniyor olduğumdan, şimdi bu kitabın konusu olan başlıkların işleneceğini düşünerek hemen bu dersi almıştım.

Fakat atladığım bir nokta vardı. Dersi veren hocamız emekliliğine çok az kalmış, yurt dışında yıllarca mainframe'ler üzerinde çalışmış, çok zaman delikli kartlarla (punched card) uygulama geliştirmiş, deneyimli biriydi. Farklı bir dersini de almıştım ve ders sırasında o eski *punched card* günlerinden örnekler verirken gözlerindeki parlamayı görmüş; *vaktiyle nelerle uğraşıyorduk şimdi bu genç tıfıllara ne anlatacağız* edasını sezmiştim.

Birikimli birinden mutlaka öğrenilecek çok şey vardır. Severek takip ettiğim nadir derslerdendi. Tahmin edilebileceği üzere sistem programlama dersinin içeriği, düşündüğümden oldukça farklı çıktı. Kendimizi bir anda IBM'in 1964 yılında piyasaya sürdüğü *System/360* mainframe ailesi üzerinde makine dili çalışırken bulduk.



IBM System/360 Model 91

Elbette bu da sistem programlama kapsamındaydı. Fakat biz bu kitapta bunlardan bahsetmeyeceğiz. Bizim bahsedeceğimiz konulara da belki bundan 50 yıl sonra *punched card* muamelesi yapılıyor olabilir belki ama günümüzde oldukça önemliler ve dikkatle üzerlerine gidilmesi gerekmektedir.

Sistem programlamadan günümüzde pratik olarak, yüksek seviyeli dillerde uygulama geliştirmek yerine (Java, Ruby vb.) daha alt seviye dillerde (C vb.), işletim sistemi çekirdeği ile standart C kitaplığındaki fonksiyonlar ve sistem çağrıları yoluyla iletişim kurulmasını, nadiren de olsa daha spesifik ve performans gerektiren konular için işin içerisine biraz da platform spesifik makine dili kodlarının karıştırıldığı programlama metodunu algılıyoruz.

Sistem programlama eforunun daha alt seviyeli dillerde yapılıyor olması, bu konuda kazanılan deneyimlerin yüksek seviyeli dillerde işe yaramayacağı anlamına gelmez. Aksine pekiştirici bir etki yaratır. Bununla birlikte yüksek seviyeli bir dille çalışırken sistem programlama kısımları bizim için çoğunlukla görünmez veya zor görünür olur.

Anlatılacak konuların bir kısmı Linux platformuna spesifik olsa da büyük çoğunluğu tüm UNIX türevlerinde ve kısmen diğer platformlarda da geçerlidir.

Konuların daha iyi anlaşılması için öncelikle Unix/Linux platformlarının zaman içerisinde gelişimine, geliştirilen standartlara ve genel olarak sistem çağrıları konularına değineceğiz.

Tarihçe

Unix ve C

1969 yılında *Ken Thompson* Bell laboratuarlarında ilk Unix versiyonunu geliştiriyorken aynı yıl kilometrelerce uzakta *Linus Benedict Torvalds* hayata gözlerini açıyordu.

İlk Unix versiyonu **Digital PDP-7** sistemi üzerinde assembler ile yazılmıştı. Bir yıl sonra da Unix bu defa **Digital PDP-11** için yeniden yazıldı.

Ken Thompson işletim sistemini tasarlarken pek çok prensibi **MULTICS**'den almıştı. Bu vesileyle genel olarak MULTICS'e de kısaca değinmekte fayda görüyoruz.

MULTICS (Multiplexed Information and Computing Service) projesi 1964 yılında *Cambridge, Massachusetts*'de başlamıştır. Ardından önce *General Electric* sonrasında da *Honeywell* firmalarında bir ürüne dönüştürülmeye çalışılmış fakat ticari açıdan tarihe başarısız projelerden biri olarak geçmekten kurtulamamıştır.

Yapılan kritiklerde MULTICS'in zamanının ötesinde, fazla karmaşık bir işletim sistemi olduğu değerlendirilmiştir. Bununla birlikte MULTICS, işletim sistemleri dünyasında bazı temel yöntemlerin ilk uygulandığı ve geliştirildiği sistemlerden biri olmuş, kendisinden sonra yapılan dizaynları etkilemiştir.

İlk defa MULTICS'te görülen başlıca özellikler aşağıdaki gibi sıralanabilir:

- **Dosya Sistemi:** Dosya ve dizinlerden oluşan ağaç yapısındaki hiyerarşi geliştirilmiştir. Sembolik link desteği gibi özelliklere de yer verilmiştir.
- **Per-Process Stack:** Çekirdek seviyesinde her *process* için ayrı bir *stack* alanı kullanılmıştır.
- **Komut İşleyici - Kabuk:** İlk defa komut işleyici (command processor) kullanıcı kipinde çalışan bir uygulama ile yapılmıştır. Bu fikir daha sonra kabuk şekliyle diğer işletim sistemlerinde ortaya çıkmıştır.
- **Process Memory:** Dosya işlemleri için IO katmanı yerine, dosyaların process'in adres alanı içerisine haritalanması ve dosya işlemlerinin standart bellek operasyonlarıyla yapılması, alt katmandaki işlemlerin ise işletim sistemi çekirdeği tarafından yerine getirilmesi sağlanmıştır. Günümüzdeki *POSIX* standartlarıyla konuşacak olursak, sistemdeki tüm dosyaların Memory-Mapped IO *mmap* ile kullanıldığındakine benzer bir uygulama söz konusudur.
- **Dinamik Linkleme:** Bazı yönleriyle günümüzdeki kullanımdan bile ileride olduğu söylenebilecek dinamik link desteği geliştirilmiştir.

- **Çoklu İşlemci:** Birden fazla CPU kullanımı desteklenmektedir.
- **Hot-Swap:** Çalışma zamanında sistemi kapatma ihtiyacı olmaksızın bellek, disk ve CPU eklenip çıkarılabilmektedir.

Ken Thompson ve Dennis Ritchie

Unix ismi de **MULTICS** üzerinde yapılan kelime oyunlarından ortaya çıkmıştır. MULTICS zamanında büyük ve kompleks bir işletim sistemi olarak çok eleştirilmişti ve Unix ismindeki **U**, MULTICS'in *Multiplexed*'ine antitez olması niyetiyle *Uniplex*'den gelmektedir.

MULTICS'ten esinlenilmiş olan pek çok fikre karşın, ondan farklı olarak Unix'te her zaman sadelik ve tüm sistemin belirli bir işlevi iyi yapan küçük parçalardan oluşması prensibi herşeyin önünde tutulmuştur.

Bir süre sonra Thompson'ın Bell laboratuarlarından arkadaşı olan *Dennis Ritchie* **C** dilini geliştirdi. C dili öncesinde Thompson tarafından interpreted modda çalışmak üzere geliştirilen **B** dilinin devamı niteliğindeydi. Thompson & Ritchie işbirliği bununla kalmadı ve 1973'e gelindiğinde UNIX kodu C dilinde yeniden yazıldı. O tarih için ilk defa bir işletim sistemi, C gibi yüksek seviyeli bir dille yazılmış oluyordu.

> O gün için **C** dili yüksek seviyeli dillere örnek olarak gösterilirken, kitabımızın devamında daha çok düşük seviyeli bir dil olarak adlandıracağız.

C dili o zamanki FORTRAN veya COBOL gibi belirli bir iş alanını adresleyen daha büyük ve karmaşık dillerin aksine, çok küçük bir ekiple, güçlü, verimli, taşınabilir, modüler ve sade biçimde tasarlanmıştır. Unix'in C ile yazılması aynı zamanda işletim sisteminin diğer donanım ve mimarilerde de çalışır hale getirilebilmesini kolaylaştırmıştır. Bunu da büyük oranda C dilinin iyi tasarımı ve kolay taşınabilirliğine borçludur.

1974 yılına gelindiğinde 50'den fazla UNIX kurulu sistem AT&T bünyesinde çalışmaktaydı. Ancak AT&T'nin A.B.D. içerisinde telefon sistemleri konusudaki tekel durumu ve devlet tarafından bağlayıcı maddeler nedeniyle, Unix'in bir ürün olarak ticari piyasaya sunulması mümkün olamıyordu.

Bir ara çözüm olarak, Unix'in 5. versiyonunun yayınlandığı 1974 yılından başlayarak, kaynak kodlar ve dokümantasyonun sembolik bir bedel ile üniversitelerin kullanımına açıldı. Bu tarihte Unix kaynak kodu yaklaşık 10 bin satırdan oluşmaktaydı.

O zamana kadar alternatif çok kullanıcılı - zaman paylaşımlı (multi-user & time-sharing) sistemler oldukça pahalıydı. Unix'in kaynak kodlarıyla birlikte üniversitelerin kullanımına sunulması bir anda hızlı bir büyüme ivmesine de yol açtı. 1977 yılına geldiğinde toplamda 125'i üniversitelerde olmak üzere 500'den fazla sistemde Unix çalışır hale gelmişti.

BSD ve SystemV Ekolleri

Ken Thompson 1975-76 yıllarını mezunu olduğu *University of California* Berkeley kampüsünde ziyaretçi profesör olarak geçirir. Bu süre zarfında öğrencilerle yaptığı çalışmaların neticesinde Unix'e çok sayıda yeni özellik eklenmesi de sağlanmış oldu.

> Bu öğrencilerden biri olan *Bill Joy* **vi** editörünün mimarı olduğu gibi 1982 yılında *Sun Microsystems*'in kurucuları arasında yer alacaktır.

Bu süre zarfında geliştirilen başlıca yenilikler arasında *C shell*, *vi* editörü, *sendmail* e-posta sunucusu, Digital VAX sistemi için bir *virtual memory management* desteği ve yeni bir dosya sistemi *Berkeley Fast File System* sayılabilir.

Geliştirilen araçlar önceleri *Berkeley Software Distribution* adıyla dağıtılıyorken, 1979 yılında yayınlanan **3BSD** versiyonuyla birlikte Unix çekirdeğini de içeren tam bir işletim sistemi olarak dağıtılmaya başlanmış ve kısa zamanda özellikle üniversiteler arasında yoğun olarak kullanılmaya başlanmıştır.

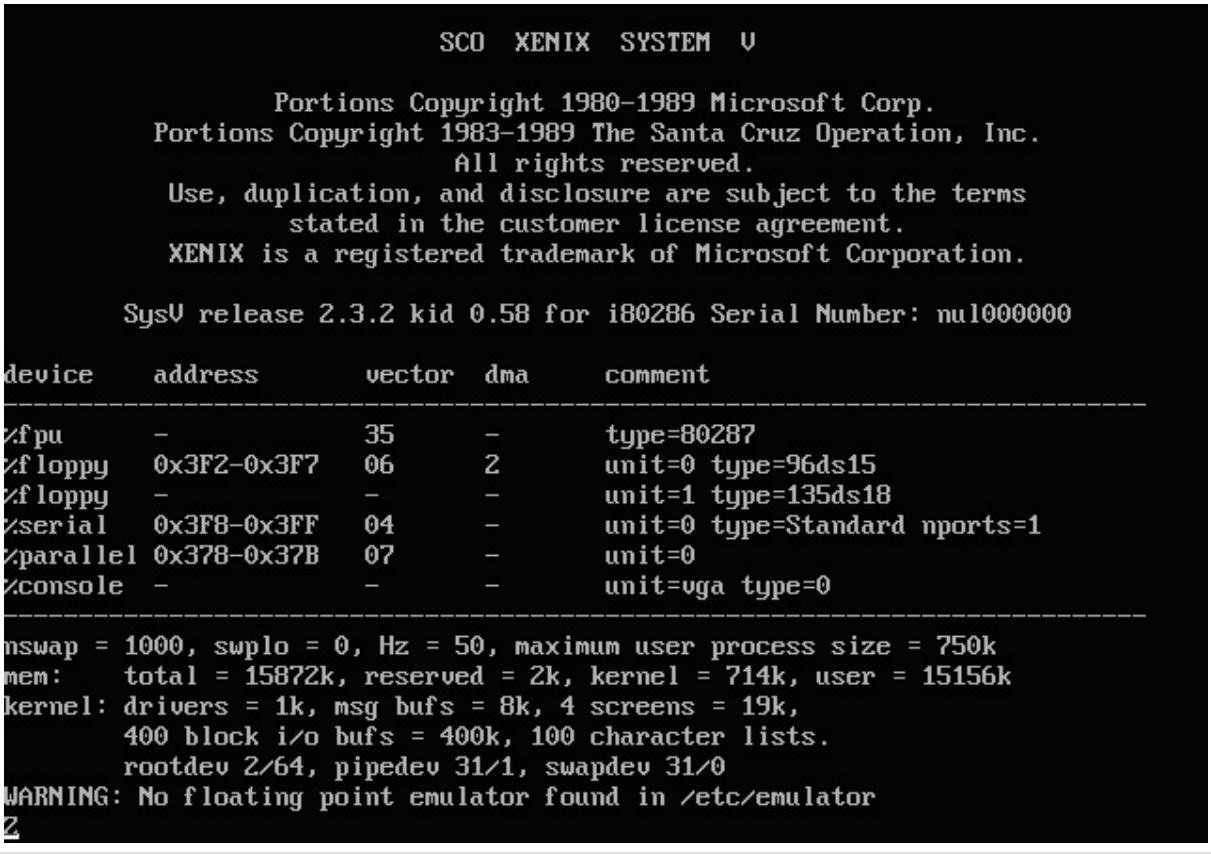
1983 yılına gelindiğinde *University of California*'da **4.2BSD** versiyonu yayınlandı. Bu versiyonda ilk defa tam bir **TCP/IP** stack implementasyonu bulunuyor ve çeşitli network araçlarının yanı sıra soket programlama için gerekli kullanıcı kipi fonksiyonları da sağlanmaya başlanmış oldu.

1986 yılında **4.3BSD** ve son olarak 1993 yılında **4.4BSD** versiyonu yayınlandı.

Bir tarafta özellikle *University of California*'nın başını çektiği bu gelişmeler yaşanırken, diğer yandan da AT&T'nin telefon tekeli ortadan kalktı ve AT&T'nin Unix'i ticari olarak pazarlayabilmesinin önündeki engeller ortadan kalkmış oldu.

Bu durumu takiben AT&T tarafından öncelikle 1981 yılında **System III**, 1983 yılında **System V** ve 1989 yılında **System V Release 4 (SVR4)** yayınlandı. AT&T ticari olarak kendi özel Unix versiyonlarını üretmek isteyen firmalara System V'i uygun lisans ile satmaya başladı. Bunun sonucunda SunOS, Digital Ultrix, IBM AIX, HP-UX, NeXTStep, XENIX (Microsoft) vb. ticari Unix versiyonları ortaya çıktı.

Sonuç olarak BSD versiyonu üniversitelerde yoğun kullanım alanı bulup genişlerken, System V versiyonları ticari dünyada gelişim imkanı buldu ve bu versiyonlar çeşitli noktalarda birbirlerinden ufak tefek farklar içermeye başladılar. Zamanla daha belirgin hale gelen bu farklar, BSD ve System V ekolü olarak isimlendirildi ve sonrasındaki işletim sistemlerinin tanımlamasında da bu ifadeler kullanılmaya başlandı.



Microsoft XENIX Açılış Ekranı

GNU Projesi

27 Eylül 1983 tarihinde, *Massachusetts Institute of Technology*'de (MIT) çalışan *Richard Stallman* tarafından GNU Projesi'nin duyurusu yapıldı.

Stallman'in amacı, legal açıdan kullanıcıyı hiç bir şekilde kısıtlamayan, "özgür" (free) bir işletim sistemi oluşturulmasını sağlamaktı. Buradaki özgürlük, ücretsiz anlamına gelmemekle birlikte, hedeflenen idealler doğrultusunda geliştirilecek olan sistemin de ücretsiz veya herkesin kolay erişimini mümkün kılacak makul bir ücretle sunulması da dolaylı hedeflerden biriydi.

GNU'nun açılımı, *GNU is Not Unix* şeklinde olup, diğer özgür olmayan Unix işletim sistemlerine karşı alternatif bir işletim sistemini ifade ediyordu.

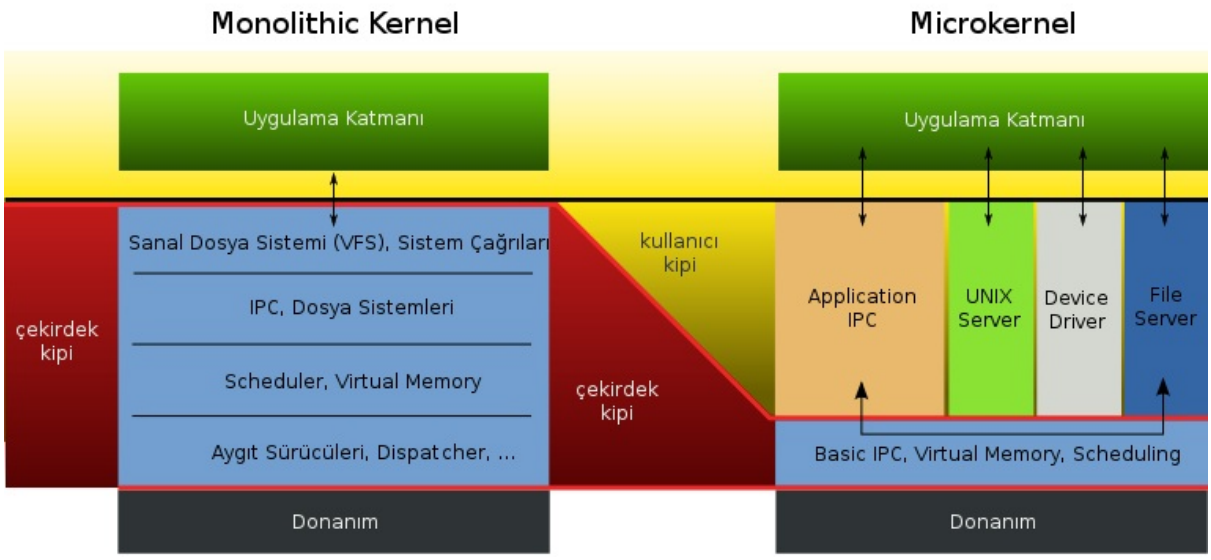
Bir işletim sistemi için çekirdek önemli bir bileşen olmakla birlikte, üzerinde çalışacak uygulamalar olmaksızın tek başına çekirdek çok fazla anlam teşkil etmez. O yıllarda akademik dünyada yoğun kullanım alanı bulmuş olan **BSD** işletim sistemi (ve çekirdeği) de Stallman'ın tariflediği anlamda "özgür" bir işletim sistemi değildi zira kullanmak için AT&T'den lisans almak zorunluydu ve içerisinde yer alan AT&T tarafından yazılmış kodları özgürce değiştirmek ve dağıtmak mümkün değildi.

Bu noktadan hareketle GNU projesinin tam bir işletim sistemi ortaya çıkarabilmesi için bir özgür yazılımlar topluluğu ve özgür bir işletim sistemi çekirdeği üretmesi gerekliydi.

1987 yılına gelindiğinde, GNU Projesi kapsamında Emacs editörü, gcc C derleyicisi, linker, çeşitli yardımcı araçlar (awk, make, grep vb.) üretilmiş durumdaydı. Üretilen bu yazılımlar ise halihazırda "özgür" olmayan Unix işletim sistemi çekirdekleri üzerinde çalışıyordu.

Proje halen özgür işletim sistemi çekirdeği bileşeninden yoksundu. 1985 yılında *Carnegie Mellon* Üniversitesi'nde, *monolithic* Unix çekirdek mimarisinden farklı olarak *microkernel* mimarisinin ilk örnekleri arasında gösterilen **Mach** çekirdeği projesine başlanmıştı. Stallman çekirdeğin *microkernel* mimarisinde geliştirilmesi gerektiğine inanıyordu ve 1987 - 1990 yılları, Carnagie Mellon Üniversitesi'nin Mach çekirdeğini özgür bir lisansla yayınlayıp yayınlamayacağının netleşmesini beklemekle geçti.

Bu beklemeden sonuç çıkmayınca, 1990 yılında eksik olan işletim sistemi çekirdeği parçasının tamamlanabilmesi için, **Hurd** projesi başlatıldı. Hurd projesinde temel amaç Unix çekirdeği ile uyumlu ancak *monolithic* Unix mimarisinden farklı olarak *microkernel* mimarisinde modern bir çekirdek üretilmesiydi.



Monolithic ve Microkernel Mimarileri

Maalesef Hurd projesinin gelişim süreci, Linux'un da etkisiyle oldukça yavaş ilerledi. 2002 yılında Stallman'ın yılın sonunda kararlı versiyonunu çıkartmayı planlıyoruz mesajına rağmen, günümüzde Hurd halen tamamen kararlı bir çekirdek haline gelebilmiş değildir.

Linux

Önceki başlıklarda Unix ve GNU'nun gelişim sürecini kısaca incelemiş olduk. Unix kullanımı zamanın gelişmiş bilgisayar sistemleri ve mainframe'ler ile mümkün oluyordu, kişisel bilgisayarlar için Unix çok pahalıydı.

1985 yılında **Intel** *Memory Management Unit* ve *paging* desteğine sahip ilk **32bit** *x86* ailesi işlemcisini çıkardı: **80386**

1986 yılında AT&T'den **Maurice J. Bach**'ın, *The Design of the UNIX Operating System* kitabı yayınlandı. Bu kitapla birlikte daha geniş bir kesim Unix ile ilgilenmeye başladı.

Bir yıl sonra 1987'de Andrew S. Tanenbaum'un *Operating Systems: Design and Implementation* kitabı ve üniversitedeki derslerini desteklemek için geliştirilen **MINIX** işletim sistemi duyuruldu.

25 Ağustos 1991'de Helsinki Üniversitesi'nden **Linux Torvalds**, *comp.os.minix* haber grubuna hobi amaçlı olarak **386/486** ailesi için *MINIX* benzeri bir işletim sistemi çekirdeği geliştirdiğini ve halihazırda **bash** (1.08) kabuğu ve **gcc** (1.40) derleyicisini port ettiğini duyuran bir mesaj gönderdi.

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki


Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and
professional like gnu) for 386(486) AT clones.  This has been brewing
since april, and is starting to get ready.  I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat
(same physical layout of the file-system (due to practical reasons)
among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work.
This implies that I'll get something practical within a few months, and
I'd like to know what features most people would want.  Any suggestions
are welcome, but I won't promise I'll implement them :-)

           Linus (torvalds@kruuna.helsinki.fi)

PS.  Yes - it's free of any minix code, and it has a multi-threaded fs.
It is NOT protable (uses 386 task switching etc), and it probably never
will support anything other than AT-harddisks, as that's all I have :-(.
```

Torvalds daha sonraki bir ropörtajında, eğer GNU Hurd çekirdeği kullanılabilir olsaydı veya 32 bit 386 ailesi için uygun lisanslı BSD çekirdekleri bulunsaydı muhtemelen yeni bir çekirdek yazmayı hiç düşünmeyecektim diyecektir.

MINIX'in dezavantajı ise 16 bit'lik bir dizayna sahip olması ve kaynak kod değişiklikleri ve dağıtımları için lisansında sorunlu maddeler bulunmasıydı.

Tüm bu şartlar neticesinde Linux doğdu ve hızla gelişti.

Linux İsmi

Linus Torvalds projesine *free*, *freak* ve *x*'in birleşiminden oluşan **Freax** ismini uygun görmüştü. Projeye isim bulma safhasında *Linux* ismi de aklına gelmiş ancak bunu fazla "egoist" bulduğundan vazgeçmişti. 6 ay kadar süre boyunca Torvalds tüm kaynak kodları "Freax" adlı bir dizinde tutuyordu.

Eylül 1991'de projenin kaynak kodları *ftp.funet.fi* FTP sunucusuna yüklendi. Torvalds'ın Helsinki Üniversitesi'nden de iş arkadaşı olan FTP Sunucusunun sistem yöneticisi **Ari Lemmke** Freax'ın iyi bir isim olduğunu düşünmüyordu. Bu nedenle Torvalds'a danışmadan

FTP sunucu üzerinde projeyi "Linux" ismiyle açtı ve sonradan Torvalds da buna razı oldu.

> Linux telaffuzu için bakınız: https://www.youtube.com/watch?v=5IfHm6R5le0

Lisans Modeli

İlk versiyonlarda Linux, Torvalds'ın her tür ticari çalışmayı engelleyen özel bir lisans ile dağıtılıyordu.

1992 yılında Linux'un lisans modeli Torvalds tarafından **GPL** olarak değiştirildi ve Aralık 1992'de **0.99** versiyonu GPL lisansıyla dağıtılmaya başlandı.

Sonraki yıllarda Torvalds, "Linux'un lisansını GPL'e çevirmek hayatım boyunca verdiğim en iyi karardı" diyecektir.

GPL lisansına geçiş ve internetin de yaygınlaşmasıyla birlikte Linux projesi çok hızlı bir gelişme ivmesi yakalamış oldu.

GNU/Linux

Linux sadece işletim sisteminin çekirdeğini oluşturur. İşe yarar bir işletim sistemi için pek çok uygulamaya ihtiyaç bulunuyordu ve bu uygulamalar GNU projesinden sağlanıyordu. Linux çekirdeği ve GNU yazılımları topluluğundan oluşan işletim sistemi ise **GNU/Linux** olarak adlandırılır.

Richard Stallman 1996 yılında GNU ve Linux kelimelerinden yola çıkarak bu işletim sistemi için **Lignux** ismini önermiş ve bunu **emacs** editöründe sistem ismi olarak kullanmış olsa da kabul görmedi ve GNU/Linux şeklindeki kullanım devam etti.

Günümüzde Linux dağıtımlarının da GNU/Linux şeklinde isimlendirilmesi gereklidir. Debian dağıtımı bu isimlendirmeye özen gösterir ve Debian GNU/Linux ismini kullanır. Ancak çoğu zaman sadece Linux isminin kullanıldığı ve bazen işletim sisteminin bazen de çekirdeğin kastedildiği görülmektedir.

Standartlar

> Önceki bölümde Unix ve C'den başlayarak Linux'un gelişimine kadarki genel tarihçe üzerinden özet olarak geçmeye çalıştık. Kitaplarda yer alan tarihçe bölümleri çoğunlukla atlanır. Siz de öyle yaptı iseniz bu noktada önceki bölüme geri dönüp, olabildiğince özetlemeye çalıştığımız tarihçeye göz atmanızı öneriririz.

1980'li yıllarda BSD veya SystemV tabanlı Unix versiyonlarının sayısı hızla artmaktaydı. Çoğu ticari Unix versiyonu da bazı özellikleri BSD'den bazılarını SystemV'den alıp üstüne bir de kendileri kimi yeni özellikler eklemekteydi. GNU projesi, Minix ve Linux da bu açıdan değerlendirildiğinde, 90'lı yıllara gelindiğinde ortada epey karmaşık bir durum olduğu rahatlıkla söylenebilir.

Bir uygulamayı çalıştığı Unix sisteminden diğerine aktarmakta zorluklarla karşılaşıldığı gibi, bir Unix sistemine aşinalık kazanmış bir kullanıcının da diğer bir sisteme alışması zor oluyordu.

Bu durumların üstesinden gelebilmek adına C dili ve Unix sistemleri arasında bir standardizasyonuna gitmek gerekliydi ve öyle de oldu.

C ve C++

1970'li ve 80'li yıllarda Unix'in gelişimine paralel olarak C dilinin de kullanımı hızla yaygınlaşmaktaydı. Farklı sistemlerdeki implementasyonları arasında özellikle bu konuda ana kaynak teşkil eden **Kernighan** ve **Ritchie**'nin 1978 yılında yayınlamış oldukları *The C Programming Language* kitabında detaylı bir şekilde değinilmemiş olan kısımlar, farklılıklar gösterebiliyordu.

1979'da AT&T Bell Laboratuarları'nda çalışan Danimarka'lı bilgisayar bilimcisi **Bjarne Stroustrup**, "C with Classess" ismini verdiği çalışmalarına başlamıştı. Stroustrup özellikle Unix platformlarında dağıtık programlama ile ilgilenmiş ve büyük projelerde **Simula** dilinin avantajlarını iyi değerlenmişti. Bunun sonucunda C diline Simula benzeri özelliklerin eklenmesi için çalışmalara başladı. 1983 yılında dilin ismi C'deki 1 artırma operatörünün eklenmesiyle, C++ haline geldi. 1985 yılında *The C++ Programming Language* kitabı yayınlandığında henüz ortada herhangi bir standart yoktu.

C++ dilinin gelişim sürecinin C diline de katkıları oldu ve fonksiyon prototipleri, *void* kelimesi, *struct* atamaları, *enum* tipi vb. yeni özellikleri içeren **ANSI C** standardı 1989 yılında yayınlandı. Böylelikle C dili için sözdizimi ve kullanımıyla birlikte, içerdiği başlık dosyaları ve

fonksiyonları da bir standarda bağlanmış oldu. ANSI C, bazen **Standard C** veya **C89** biçiminde de adlandırılmaktadır.

ANSI C standardının ardından 10 yıl sonra 1999'da **C99** standardı yayınlanmıştır. C99 ile birlikte *inline* fonksiyonlar, *long long int* gibi yeni veri tipleri, değişken uzunlukta dizi tanımlama imkanı, belirsiz sayıda parametre alabilen makro desteği, daha iyi **IEEE 754** kayar nokta desteği, // tek satırlık yorum desteği gibi özellikler eklenmiştir.

2011 yılında **C11** standardı yayınlanmıştır. Bu versiyonda eklenen bir çok özellik arasında başlıcaları *multi-threading* desteği, unicode iyileştirmeleri, *gets()* fonksiyon ailesinin dilden çıkartılması, *_Atomic* tip niteleyeciler, anonim *struct* ve *union* tanımları sayılabilir. Bununla birlikte henüz derleyiciler tarafından tüm özellikleri tam olarak gerçeklenmemiştir.

Son olarak 2018 yılında **C18** standardı yayınlanmıştır. Bu versiyonda önemli bir değişiklik bulunmamakta, sadece *C11*'e yönelik bazı düzeltmelere yer verilmektedir.

POSIX

Portable Operating System Interface (POSIX) standartları, **IEEE**'nin gözetiminde uygulamaların kaynak kod seviyesinde platformlar arası taşınabilirliğini sağlamak için geliştirilmiştir.

> POSIX ismi *Richard Stallman* tarafından önerilmiş olup, sonundaki **X** de *Unix like* sistemler için genel olarak yapılan ekleme alışkanlığından kaynaklanır.

POSIX.1 standardı 1988 yılında duyuruldu. POSIX.1 standardı UNIX sistem çağrıları ve C kütüphanesi arayüzünü tanımlamaktadır. Bu arayüzü gerçekleyen işletim sistemleri *POSIX.1 Uyumlu* olarak nitelendirilirler ve UNIX dışındaki bir işletim sistemi için de arayüzü sağladığı müddetçe POSIX uyumluluğundan bahsetmemiz mümkündür.

POSIX.1 standardı, POSIX.1b, POSIX.1c ... ve son olarak realtime işlemlere yönelik destekle birlikte 2000 yılında **POSIX.1j** olarak genişletilmiştir.

POSIX.2 standardı ise, kabuk uygulamasının özellikleri, çeşitli sistem araçları ve C derleyicisinin kullanımı ve parametrelerini standardize etmeye çalışmakta olup, 1992 yılında duyurulmuştur.

SUS & X/Open & The Open Group

1984 yılında başta Bull, ICL, Siemens, Olivetti, Nixdorf, Philips ve Ericsson olmak üzere, çeşitli Avrupa firmaları biraraya gelerek **X/Open Company Ltd.** konsorsiyomunu kurdular. Bu yapının amacı UNIX tabanlı işletim sistemlerinde uygulamaların taşınabilirliğinin

sağlanması için gereken standartları oluşturmaktı. Oluşturulan standartlar **X/Open Portability Guide** (XPG) adıyla yayınlandı. 1992 yılına gelindiğinde Avrupa kıtasının dışından da katılımlarla konsorsiyomdaki firma sayısı 21'e ulaşmıştı.

1993 yılında Unix iş departmanı **Novell** tarafından **AT&T**'den satın alındı. Hemen ardından Novell, Unix tarafından geri çekilmeye karar verdi ve Unix ticari markasının haklarını X/Open konsorsiyomuna devretti. Bu noktada mevcut XPG4 standartları **Single UNIX Specification** (SUSv1) ismiyle birleştirildi ve yayınlandı. Ardından 1997'de SUSv2 ve 1999'da SUSv3 standardı yayınlandı. SUSv3, POSIX standardına göre daha detaylı bir belgedir (3700 sayfa, 1742 interface) ve syslog arayüzü, memory-mapped io, SysV IPC, login accounting, pseudoterminals vb. pek çok başlığı adreslemektedir.

Son olarak 2008 yılında **SUSv4** yayınlanmış olup, bazı arayüzler çıkartılmış, bazı yeni eklemeler de yapılarak son hali verilmiştir. Standardın güncel haline http://www.unix.org/online.html adresinden erişebilirsiniz.

1996 yılında X/Open oluşumu *Open Software Foundation* (OSF) çatısı altında **The Open Group** olarak birleşti. Günümüzde UNIX sistemleri üzerinde çalışan kurum ve organizasyonlar *The Open Group*'a üyedirler.

Linux Standard Base

Linux'un gelişim sürecine baktığımızda POSIX ve Single Unix Specification'larına çok önem verildiği ve geliştirmelerin bu yönde yapıldığı görülür. Ancak buna rağmen henüz herhangi bir Linux dağıtımı, *The Open Group* tarafından bir UNIX sistemi olarak görülmemektedir. Geleneksel UNIX üretici firmalarından farklı olarak Linux'un gelişim modeli farklılığı ve her versiyonda UNIX olarak markalama yapabilmek için gereken test süreçleri ve bu sürecin maliyetleri nedeniyle, Linux dağıtımlarının UNIX olarak görülmeye fazla önem vermediğini söyleyebiliriz.

Linux dağıtımları temelde resmi bir Linux kernel versiyonunu baz alır; üzerine yapılan bazı yamalar veya kernel içerisine dahil edilmiş öntanımlı seçenekler açısından ise birbirinden farklılıklar gösterebilir. Dağıtımların kernel üzerinde bu şekilde yaptıkları iyileştirme çabaları, genellikle dağıtımın stratejik hedefleriyle ilgili olur. Örneğin temel hedefi sunucu sistemleri olan bir Linux dağıtımı, maksimum sayıda Host Bus Adapter kartı için sürücüyü sisteme dahil edebilir, hatta ana Linux kaynak kodu içerisine henüz girememiş bazı sürücüleri de dağıtım içerisinde gelen kernel ile uyumlu hale getirebilir.

Dağıtımların temel stratejileri doğrultusunda yaptığı değişiklikler sadece kernel seviyesiyle de sınırlı değildir. Bazı dağıtımlar grafik arayüzde ve çok kolay kurulumu hedeflerken bazıları en fazla sayıda mimaride çalışmayı, bazıları sadece sunucu sistemleri hedeflerken bazıları da güvenlik önlemleri artırılmış sistemlerde kullanılmayı vb. hedefleyebilir. Tüm bu kararlar

Linux dağıtımlarının kolaylıkla birbirinden farklılaşması olasılığını doğurur ve bu noktada bir standardizasyon ihtiyacı ortaya çıkar. İşte **Linux Standard Base** oluşumu bu süreci adresler. Günümüzde son **LSB** versiyonu **5.0** olup, Haziran 2015'de yayınlanmıştır. POSIX ve SUS standartlarından farklı olarak, sadece kaynak kod seviyesinde taşınabilirliği değil, çalıştırılabilir dosyaların da dağıtımlar arasında taşınabilirliğine dair tanımları içerir.

Güncel duruma https://wiki.linuxfoundation.org/en/LSB adresinden erişilebilir.

Sistem Çağrıları

Modern işletim sistemlerinde, çekirdek kipinde çalışma ve kullanıcı kipinde çalışma modları, sert bariyerlerle birbirinden ayrılmış durumdadır.

Bu şekildeki bir tasarım, sistemin güvenli ve sağlıklı çalışması için elzemdir.

Kullanıcı kipinde çalışan bir uygulamanın, sistem çağrıları aracılığıyla işletim sistemi çekirdeğinden ihtiyaç duyduğu servisleri alabilmesi sağlanır.

Her uygulama, mutlaka sistem çağrısı yapmak durumundadır. Yapılan sistem çağrıları disk üzerinden okuma, yazma gibi daha "fiziksel" ve donanıma yakın olabileceği gibi o anki sistem zamanını alma `gettimeofday`, çalışan uygulamanın process id değerini öğrenme `getpid`, uygulamaya öncelikler atama `setpriority` gibi doğrudan çekirdek içerisindeki belirli mekanizmalarla ilgili de olabilir. Bir uygulamanın hayata gelmesi ve çalışmaya başlayabilmesi için öncesinde bir başka uygulamanın `fork()` sistem çağrısıyla yeni bir *process* üretmesi de gereklidir.

Linux çekirdeğinde X86 mimarisi için yaklaşık **380** civarında sistem çağrısı bulunur.

Sistem Çağrısı Nasıl Gerçekleşir?

Sistem çağrılarının çekirdek tarafındaki gerçekleştirimi mimariden mimariye değişkenlik gösterir. Linux çekirdeğinin farklı bir mimariye port edilirken yapılan temel işlem adımlarından biri, sistem çağrılarının en verimli şekilde yapacak şekilde uygun bir kodlamanın, mimari spesifik olarak yapılmasıdır.

Her sistem çağrısının 1, 5, 27 gibi ilişkili bir numarası vardır. Bu numaralar da mimariden mimariye değişkenlik göstermektedir. Temel sistem çağrıları tüm mimarilerde bulunmakla birlikte, tüm mimarilerde eşit sayıda sistem çağrısı bulunmaz.

> Konunun devamında aksi belirtilmedikçe verilen örnekler 32 bit Intel mimarisi için geçerlidir.

Kullanıcı kipindeyken herhangi bir sistem çağrısı yapıldığında `INT 0x80` makine dili kodu ile *trap* oluşturulur.

Aynı zamanda talep edilen sistem çağrısının numarası, `EAX` yazmacına yazılır.

Talep edilen sistem çağrısının parametreleri var ise, bu parametrelerin diğer yazmaçlar kullanılarak belirtilmesi gerekir. Ancak her mimaride bu amaçla kullanılabilecek yazmaç sayısı limitlidir. Bazılarında daha çok genel amaçlı yazmaç var iken bazılarında daha az

olduğu görülmektedir.

32 bitlik Intel platformu için Linux çekirdek versiyonu **2.3.31** ve sonrası, maksimum **6** sistem çağrısı parametresini desteklemektedir. Bu parametreler sırasıyla `EBX`, `ECX`, `EDX`, `ESI`, `EDI` ve `EBP` yazmaçlarında saklanır.

Sistem çağrısı için 6'dan fazla parametre gerekli olduğunda, bellekteki bir veri yapısı hazırlanarak parametrelar burada saklanır, sonrasında ilgili bellek adresi sistem çağrısına parametre olarak geçirilir.

Mimari Bağımlılığı

Sistem çağrılarının doğrudan işlemci mimarisine bağımlı olduğuna değinmiştik.

Örnek olarak Intel 32 bitlik işlemcilerde `INT 0x80` ile *trap* oluşturulurken, **ARM** mimarisinde aynı işlem **supervisor call** `SVC` ile yapılır

Benzer şekilde Intel mimarisinde yapılan sistem çağrısının numarası için `EAX` yazmacı kullanılırken, ARM mimarisinde `R8` yazmacı kullanılır. ARM mimarisinde sistem çağrısına ait 4 adede kadar parametre, `R9`, `R10`, `R11` ve `R12` yazmaçlarına aktarılır. 4 adetten fazla parametre geçilmesi gerektiğinde, bellek üzerinde veri yapısı hazırlanarak bu bölümün adresi geçirilir.

Genel olarak sistem çağrıları performansının *ARM* mimarisinde *X86*'ya göre daha düşük olduğunu söyleyebiliriz (yazmaç/register sayısının azlığı bunda etken olabilir mi düşününüz).

Sistem Çağrısı Nasıl Yapılır?

Sistem çağrılarını daha zor bir yoldan doğrudan yapmak mümkün olsa da bu önerilen bir durum değildir.

Sistem çağrıları, `glibc` kütüphanesindeki *wrapper* fonksiyonlar üzerinden kullanılır.

`glibc` kütüphanesi, üzerinde çalıştığı çekirdek versiyonuna göre, hangi Linux sistem çağrısını yapacağını belirler.

> Bazı durumlarda ise bundan daha fazlasını yaparak, üzerinde çalışılan çekirdek versiyonunda hiç desteklenmeyen bir özelliği de sunuyor olabilir. Örnek olarak, Linux **2.6** versiyonuyla birlikte gelen *POSIX Timer API*'nin olmadığı Linux **2.4** versiyonu üzerinde çalışan ve aynı anda pek çok *timer* kullanan bir uygulamanız var ise, *glibc* çekirdek tarafından alamadığı desteği kullanıcı kipinde her *timer* için bir *thread* açarak sağlar. Elbette timer sayınız fazla ise bu çok yavaş bir çözüm olur ancak uygulamanın çalışmasını da mümkün kılar.

Sistem Çağrıları → Glibc Fonksiyonları İlişkisi

Pek çok sistem çağrısı, aynı isimdeki `glibc` *wrapper* fonksiyonları üzerinden çağrılmaktadır.

> Not: Bu duruma `strace` çıktılarını okurken de dikkat etmemiz gereklidir.

Örnek olarak sistem çağrılarını takip etmede sık kullanacağımız **strace** aracının çıktısındaki `open()` çağrısına bakalım:

```
open("/tmp/index.jpeg", O_RDONLY)       = 3
```

Burada kastedilen `glibc` içerisindeki `open()` fonksiyonu değil, `open` sistem çağrısıdır.

Strace üzerinden sistem çağrısına geçirilen argümanları ve geri dönüş değerini (3) görmekteyiz.

> Normalde bu yöntemle sistem çağrılarının ismi değil numarası izlenebilir. Strace uygulaması elde ettiği sistem çağrısı numarasını kendi veritabanında arayıp bizler için daha okunabilir bir formda gösterir. Strace uygulamasının bu işlemi hangi yöntemle gerçekleştirdiğine dair detaylı bilgileri ilerleyen bölümlerimizde bulabilirsiniz.

Performans

Sistem çağrıları normal fonksiyon çağrılarına oranla oldukça yüksek maliyetli işlemlerdir.

Her sistem çağrısında uygulamanın o anki durumunun saklanması, çekirdeğin işlemcinin kontrolünü ele alması ve ilgili sistem çağrısı ile çekirdek kipinde talep edilen işlemleri gerçekleştirmesi, sonra ilgili uygulamanın tekrar çalışma sırası geldiğinde, uygulamanın saklanan durumunun yeniden üretilip işlemlerin kaldığı yerden devamının sağlanması gereklidir.

2002 yılında Linux Kernel eposta listelerine *Mike Hayward*'ın şaşkınlığını içeren bir eposta düştü. *Hayward* elindeki **Pentium 3 - 850 Mhz** dizüstü bilgisayarıyla **Pentium 4 - 2 Ghz** ve **Xeon - 2.4 Ghz** sistemlerinin, sistem çağrıları açısından performansını ölçmek için bir test uygulaması yazdı ve 1K'lık buffered dosya okuma testinde aşağıdaki şaşırtıcı sonuçları elde etti:

Sistem	Saniyedeki IO
Pentium 3 - 850 Mhz	149
Pentium 4 - 2 Ghz	108
Xeon - 2.4 Ghz	69

Aynı testi dosya okuma yerine farklı sistem çağrılarıyla da test ettiğinde benzer sonuçların alındığını tespi etti.

Bunun sebebi, bazı **x86** serisi işlemcilerde çekirdek kipine daha hızlı geçiş için `SYSENTER/SYSEXIT` özel instruction'ının bulunmasıydı. Pentium 3 serisinde varolan bu destek, Pentium 4 ve Xeon işlemcilere yeterince olgunlaşmadığından konulmamıştı. Pentium 3'teki bu imkanı iyi değerlendiren Linux çekirdeği, kendisinden daha üstün Xeon işlemcilerden bile daha iyi performans göstermekteydi.

Benzer zamanlarda **AMD** de benzer şekilde `SYSCALL/SYSRET` özel instruction'ınını sunmaya başlamıştır.

Linux çekirdeği de bu yeni imkanları kullanarak geleneksel `INT 0x80` kesme yöntemine göre önemli oranda performans iyileşmesi sağlanmıştır.

Günümüz **x86** ve **x86_64** işlemcilerinde bu mekanizma tümüyle desteklenmektedir.

Performans Problemi - Detaylı Bakış

Özellikle **x86** tabanlı mimarilerde `SYSENTER` özel yolu sayesinde sistem çağrılarının hızlanmasını sağladık. Ancak bu yeterli olacak mıdır?

Bir çok uygulamada, özellikle `gettimeofday()` gibi sistem çağrılarının çok sık kullanıldığını görürürüz.

Uygulamalarınızı `strace` ile incelediğinizde, bilginiz dahilinde olmayan pek çok farklı `gettimeofday()` çağrısını yapıldığını görebilirsiniz.

`glibc` kütüphanesinden kullandığınız bazı fonksiyonlar, *internal* olarak bu fonksiyonaliteyi kullanıyor olabilir.

Java Virtual Machine gibi bir **VM** üzerinde çalışan uygulamalar için de benzer bir durum söz konusudur.

Görece basit bir işlem olmasına rağmen sık kullanılan bu operasyon yüzünden sistemlerde nasıl bir sistem çağrısı yükü oluşmaktadır? sorusunu kendimize sorabiliriz.

Linux Dynamic Linker/Loader: `ld.so`

Paylaşımlı kütüphaneler kullanan uygulamaların çalıştırılması sırasında, **Linux Loader** tarafından gereken kütüphaneler yüklenerek uygulamanın çalışacağı ortam hazırlanır. En basit **Hello World** uygulamamız bile `libc` kütüphanesine bağımlı olacaktır.

`ldd` ile uygulamanın linklenmiş olduğu kütüphanelerin listesini alabiliriz:

```
$ ldd hello
    linux-vdso.so.1 (0x00007fff7d88a000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2fb43a0000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f2fb476d000)
```

Görüldüğü üzere `libc` ve `ld.so` bağımlılıkları listelendi. Fakat `linux-vdso.so.1` kütüphanesi nedir?

`find` komutu ile tüm sistemimizi arattığımızda neden bu kütüphaneyi bulamıyoruz?

Virtual DSO: `linux-vdso.so.1`

linux-vdso.so.1 sanal bir **D**namically Linked **S**hared **O**bject dosyasıdır. Gerçekte böyle bir kütüphane dosya sistemi üzerinde yer almaz. Linux çekirdeği, çok sık kullanılan bazı sistem çağrılarını, bu şekilde bir hile kullanarak kullanıcı kipinde daha hızlı gerçekleştirmektedir.

Örnek olarak, sistem saati her değişiminde sonucu tüm çalışan uygulamaların adres haritalarına da eklenmiş olan özel bir bellek alanına koyarsa, `gettimeofday()` işlemi gerçekte bir sistem çağrısına yol açmadan kullanıcı kipinde tamamlanabilir.

Şimdi bu konuları biraz daha detaylandıralım.

> Not: Konunun bundan sonrası meraklıları için olup, çok gerekli olmayan bu bölümün yeni başlayan kullanıcılar için atlanması önerilir.

`/proc/self/maps`

Linux `proc` dosya sisteminde `/proc/<PID>/maps` dosyasında ilgili `PID` (process id) için çekirdek tarafından yapılmış olan adres haritalaması gösterilir.

Özel bir durum olarak, `<PID>` yerine `self` ibaresi kullanıldığında, o an bu dosya erişimini yapan process ile ilgili dizinde işlem yapılmış olur.

```
$ cat /proc/self/maps
    00400000-0040c000 r-xp 00000000 08:02 1703938   /bin/cat
    0060b000-0060c000 r--p 0000b000 08:02 1703938   /bin/cat
    0060c000-0060d000 rw-p 0000c000 08:02 1703938   /bin/cat
    024de000-024ff000 rw-p 00000000 00:00 0         [heap]
    7ff7033c5000-7ff70369f000 r--p 00000000 08:02 2362900 /usr/lib/locale/locale-archi
ve
    7ff70369f000-7ff70383e000 r-xp 00000000 08:02 393230  /lib/x86_64-linux-gnu/libc-2
.19.so
    7ff70383e000-7ff703a3d000 ---p 0019f000 08:02 393230  /lib/x86_64-linux-gnu/libc-2
.19.so
    7ff703c69000-7ff703c6a000 rw-p 00000000 00:00 0
    7fff8cd95000-7fff8cdb6000 rw-p 00000000 00:00 0 [stack]
    7fff8cdfc000-7fff8cdfe000 r-xp 00000000 00:00 0 [vdso]
    7fff8cdfe000-7fff8ce00000 r--p 00000000 00:00 0 [vvar]
    ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0   [vsyscall]
```

`vsyscall`

`/proc/self/maps` dosyasına `cat` uygulaması ile bir kaç defa baktığınızda, `vsyscall` (Virtual System Call Page) haricindeki bölümlerin başlangıç adreslerinin değiştiğini görmekteyiz.

`vsyscall` bölümü, sadece bir uygulama için değil, sistemdeki tüm uygulamalar için aynı statik yeri göstermektedir.

Bu sayede dinamik linkleme (dolayısıyla paylaşımlı kütüphane) kullanmayan, tamamen statik uygulamaların da bu **statik** adres üzerinden `vsyscall` bölümüne erişimi mümkün olmaktadır.

Bu bölgenin uzunluğu kısıtlı olduğundan, sadece belirli sayıda girdiye sahiptir: *vgettimeofday()*, *vtime()*, *vgetcpu()*

Tüm uygulamalar için aynı adrese haritalanması, özellikle **return to libc** türü ataklarıyla sistem çağrısı yapılabilmesine neden olmaktadır.

Linux **3.0** versiyonuna kadar **vsyscall** tablosu kullanılmış olmakla birlikte, **3.1** ve sonrasında bu yöntem artık önerilmiyor. **vDSO** mekanizması hem daha güvenli hem daha hızlı.

`vdso` Bölümünü Dışarı Çıkartmak

İnceleme amacıyla uygulamanın adres haritasındaki `[vdso]` biçiminde işaretlenmiş alanı diske çıkartmaya çalışalım.

Örneğimizde bu bölümün `7fff8cdfc000` ile `7fff8cdfe000` adresleri arasında, 2 adet `Page` büyüklüğünde olduğunu görüyoruz.

Acaba `dd` komutu ile bu bölümü dışarı çıkartabilir miyiz:

```
$ dd if=/proc/self/mem of=dso.out bs=1 skip=$((0x7fff8cdfc000)) count=8192
```

Maalesef bu yöntem artık çalışmıyor. Bunun 2 nedeni var:

1. `/proc` sanal dosya sistemi altındaki girdiler normal bir dosya gibi görünmesine karşılık, `stat()` ile bakıldığında `st_size` değeri **0** olmaktadır. Bu durum `dd` uygulamasının ilgili offset adresine **seek** yapılamayacağını söylemesine neden oluyor. Çözüm için ufak bir yama gerekiyor
2. Yeni Linux çekirdek versiyonlarında buradaki başlangıç değer adresi, `7fff8cdfc000` her uygulama için aynı değildir. **Return to libc** tarzı atakları zorlaştırmak için bu değer ancak çalışan uygulama içerisinden öğrenilebilir. Bunun için `dd` kodunun değiştirilmesi veya ufak bir test uygulaması yazılması gerekiyor.

`extract_region.c`

Bu işlemi yapabilmek için `extract_region` adını verdiğimiz aşağıdaki gibi bir uygulama hazırlayalım:

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

#define _FILE_OFFSET_BITS 64

int main (int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr,
            "Kullanım: %s <cikti> <section>\n"
            "\t<cikti>\t\t: export edilecek dosya\n"
            "\t<section>\t: export edilecek map region\n\n", argv[0]);
        return 1;
    }

    off_t start_addr, end_addr;
    char buf[4096];
    const char *out = argv[1];
    const char *region = argv[2];

    int found = 0;
```

```
    FILE *fp = fopen("/proc/self/maps", "r");
    while (fgets(buf, sizeof(buf), fp)) {
        printf("%s", buf);
        if (strstr(buf, region)) {
            found = 1;
            break;
        }
    }
    fclose(fp);
    if (!found) {
        fprintf(stderr, "%s bölümü bulunamadı\n", region);
        return 1;
    }
    end_addr = strtoull((strchr(buf, '-') + 1), NULL, 16);
    *(strchr(buf, '-')) = '\0';
    start_addr = strtoull(buf, NULL, 16);

    printf("\nÇıkartılacak Alan Başlangıç: 0x%llx, Bitiş: 0x%llx\n\n", start_addr, end
_addr);

    FILE *dst = fopen(out, "w+");
    if (dst == NULL) {
        fprintf(stderr, "%s açılamadı\n", out);
        return 1;
    }
    FILE *src = fopen("/proc/self/mem", "r");
    char *tmp = malloc(end_addr - start_addr);
    fseeko(src, start_addr, SEEK_SET);
    fread(tmp, end_addr - start_addr, 1, src);
    fwrite(tmp, end_addr - start_addr, 1, dst);
    fclose(src);
    fclose(dst);
    return 0;
}
```

Test Uygulamamızı Çalıştıralım

```
$ ./extract_region vdso.out vdso
...
7ff6db951000-7ff6dbaf0000 r-xp 00000000 08:02 393230    /lib/x86_64-linux-gnu/libc-2.1
9.so
7ff6dbf19000-7ff6dbf1a000 r--p 00020000 08:02 393236    /lib/x86_64-linux-gnu/ld-2.19.
so
7ff6dbf1a000-7ff6dbf1b000 rw-p 00021000 08:02 393236    /lib/x86_64-linux-gnu/ld-2.19.
so
7ff6dbf1b000-7ff6dbf1c000 rw-p 00000000 00:00 0
7fffc477d000-7fffc479e000 rw-p 00000000 00:00 0         [stack]
7fffc47fc000-7fffc47fe000 r-xp 00000000 00:00 0         [vdso]

Çıkartılacak Alan Başlangıç: 0x7fffc47fc000, Bitiş: 0x7fffc47fe000
```

İşlem bitiminde **8192** byte uzunluğunda **vdso.out** dosyası oluşacaktır.

`file` komutu ile dosyanın tipine baktığımızda standart bir kütüphane gibi görünecektir:

```
$ file vdso.out
vdso.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
BuildID
[sha1]=538bea2738a229413dcc98af8f4f7127f9bca874, stripped
```

`vdso` İçine Bakalım

`objdump` ile dışarı çıkarttığımız bu bölüme bir bakalım:

```
$ objdump -T vdso.out

vdso.out:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000418 l    d  .rodata    0000000000000000              .rodata
0000000000000970  w   DF .text    000000000000057d  LINUX_2.6   clock_gettime
0000000000000000 g    DO *ABS*    0000000000000000  LINUX_2.6   LINUX_2.6
00000000000000ef0 g    DF .text    00000000000002b9  LINUX_2.6   __vdso_gettimeofday
00000000000011d0 g    DF .text    000000000000003d  LINUX_2.6   __vdso_getcpu
00000000000000ef0  w   DF .text    00000000000002b9  LINUX_2.6   gettimeofday
00000000000011b0  w   DF .text    0000000000000015  LINUX_2.6   time
00000000000011d0  w   DF .text    000000000000003d  LINUX_2.6   getcpu
0000000000000970 g    DF .text    000000000000057d  LINUX_2.6   __vdso_clock_gettime
00000000000011b0 g    DF .text    0000000000000015  LINUX_2.6   __vdso_time
```

Basit Bir Test Uygulaması

100.000 defa `gettimeofday()` fonksiyonunu çağıran ve işlem bitiminde başlangıç ve bitiş zamanlarını gösteren örnek bir uygulama yapalım:

```
#include <stdio.h>
#include <sys/time.h>

int main ()
{
    int i;
    struct timeval now;
    struct timeval before;
    struct timeval after;
    gettimeofday(&before, NULL);
    for (i = 0; i < 100000; i++) gettimeofday(&now, NULL);
    gettimeofday(&after, NULL);
    printf("Before: %li.%li\n", before.tv_sec, before.tv_usec);
    printf("After : %li.%li\n", after.tv_sec, after.tv_usec);
    return 0;
}
```

`X86_64` ve `ARM` Üzerinde Test

100.000 defa `gettimeofday` çağrısı yapan `time_test` örnek uygulamasını, 1 Ghz hızına düşürülmüş **X86_64** *i5* işlemcili platform ile 1 Ghz saat frekansındaki **ARM** *BeagleBoneBlack* platformunda karşılaştıralım

```
(X86_64) $ ./time_test
Before: 1419786575.719463
After : 1419786575.722560

(ARM) $ ./time_test
Before: 1419786909.186960
After : 1419786909.252160
```

Görüldüğü üzere X86_64'te 3-4 milisaniyede gerçekleşen işlem, ARM sistemimizde 70 milisaniyelerde gerçekleşmektedir.

Şimdi test uygulamamızı bir de `strace` kontrolünde her iki platformda çalıştıralım:

```
(X86_64) $ strace ./time_test
...
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff4ee1f1
000
write(1, "Before: 1419787456.897162\n", 26Before: 1419787456.897162) = 26
write(1, "After : 1419787456.904669\n", 26After : 1419787456.904669) = 26
exit_group(0)

##########################

(ARM) $ strace ./time_test
...
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x400b3000
gettimeofday({1419787571, 234967}, NULL) = 0
gettimeofday({1419787571, 235064}, NULL) = 0
gettimeofday({1419787571, 235142}, NULL) = 0
gettimeofday({1419787571, 235257}, NULL) = 0
gettimeofday({1419787571, 235394}, NULL) = 0
...
...
write(1, "Before: 1419787571.234967\n", 26) = 26
write(1, "After : 1419787571.976285\n", 26) = 26
exit_group(0) = ?
```

- `X86_64` platformunda `strace` ile yaptığımız incelemede, herhangi bir `gettimeofday()` sistem çağrısı göremedik
- Beklediğimiz şekilde `linux-vdso.so` mekanizması sayesinde, işlem tamamen kullanıcı kipinde gerçekleştirildi, herhangi bir sistem çağrısı yapılmadı
- Sadece `printf()` fonksiyonu nedeniyle `write()` sistem çağrısı kullanılarak son çıktı konsola gönderildi
- `ARM` mimarisindeki örneğimize baktığımızda, benzeri bir mekanizma olmadığı için, her defasında karşılık gelen bir `gettimeofday()` sistem çağrısı yapıldığını görüyoruz (örnek ekran çıktımızda ... olarak belirttiğimiz bölümde 100000 adet benzer çağrı bulunmaktadır)

Sistem Çağrılarının Kesintiye Uğraması

Kendimize şu soruyu soralım: uygulamamız bir sistem çağrısı yaparak çekirdek kipinde kod işletiliyor durumunda iken sinyal (software interrupt) gelirse ne olur?

Bu durumda sistem çağrısı sona erecek ve **EINTERRUPTED** hatası dönecektir.

Sistem çağrılarını `glibc` üzerinden kullandığımız için, glibc tarafında sistem çağrısından `EINTR` hatası geldiğinde, uygulamaya geri dönüş değeri olarak `-1` dönülür fakat `errno`

global değişkeni `EINTR` şeklinde ayarlanır.

Bu aslında hata olmayan istisnai durum, zaman zaman pek çok uygulama kodunda gözardı edilmektedir.

Bazı kullanım senaryolarında yukarıdaki senaryo istisnai olmaktan çıkıp, ilgili yazılımın doğası gereği sürekli veya sıklıkla da (read, write, open, connect vb.) oluşabilir.

Uygulama perspektifinden baktığımızda tüm sistem çağrılarını sarmalayan fonksiyonlar için aşağıdaki kural geçerlidir:

- Eğer bir sistem çağrısının geri dönüş değeri `0` 'dan küçükse ve `errno` değişkeni `EINTR` sabitine eşitse, herhangi bir hata söz konusu değildir.

Bahsedilen senaryo oluştuğunda ilgili fonksiyonun (yani sistem çağrısının) yeniden çağrılması gerekir.

Bu süreç, ilgili sinyallerin oluşturulmasında `SA_RESTART` bayrağının işaretlenmesi suretiyle otomatik hale getirilebilir. Peki neden öntanımlı olarak bu şekilde değil?

Esasen bir zamanlar öyleydi. Ancak sistem çağrısının otomatik olarak yeniden başlatılmasını istemeyeceğimiz durumlar da olabilir. Bu yüzden öntanımlı olarak bir aksiyon alınmıyor.

API ve ABI

Yazılım geliştiriciler açısından uygulamaların tüm Linux dağıtımlarında ve desteklenen tüm mimarilerde çalışması temel hedeflerden biridir. İdeal durumda, geliştirilen yazılımın belirli bir dağıtım veya mimariye bağımlı olmayıp, taşınabilir olması beklenir.

Sistem seviyesinden bakıldığında taşınabilirlikle ilgili 2 farklı tanım bulunur. Bunlardan ilki *Application Programming Interface* (API) diğeri ise *Application Binary Interface* (ABI) şeklindedir.

API

API, uygulamaların kaynak kod seviyesinde birbirleriyle iletişim kurabilmelerine imkan sağlayan, önceden anlaşılmış arayüzler olarak tanımlanabilir.

Genellikle her bir API, daha karmaşık ve alt seviye detaylar içeren bir sürecin, çeşitli arayüzlerle (fonksiyon çağrıları vb.) soyutlanması sağlar.

Bu şekildeki bir soyutlama üzerinden kullanılan API'yi sağlayan yazılım bileşenleri güncellense ve alt tarafta yapılan işlemlerle ilgili yöntemler değiştirilse dahi, API seviyesinde aynı arayüz sağlandığı müddetçe diğer uygulamalar için kaynak kod seviyesinde bir değişiklik yapılmasına gerek olmayacaktır.

ABI

API kaynak kod seviyesinde bir arayüz tanımlarken ABI yazılım bileşenleri arasında belirli bir mimari için obje kod (binary) arayüzlerini tanımlamaktadır.

ABI uygulama bileşenleri arasında *binary compatibility*'yi sağlar. Bu uyum korunduğu müddetçe aralarında etkileşim bulunan yazılım bileşenlerinin versiyonları değişse de yeniden derlenmeye ihtiyaç duymaksızın eskisi gibi çalışmaya devam ederler.

Fonksiyonların nasıl çağrılacağı, parametrelerin nasıl geçirileceği, yazmaçların kullanım şekilleri, sistem çağrılarının gerçekleştirilme biçimi, uygulamların linklenmesi, binary obje dosya formatı gibi konular ABI içerisinde incelenir. Tüm mimariler için ortak bir ABI oluşturmak mümkün değildir (Java Virtual Machine gibi çözümler ayrı bir başlıkta değerlendirilebilir).

Yazılım geliştirme sürecinde ABI varlığını pek hissettirmez. Kullanılan toolchain içerisindeki araçlar (derleyici, linker vb.) hedef platform için uygun ABI ile kod üretirler.

Linux Çekirdeği

Her işletim sistemin en temel bileşeni, çekirdeğidir (kernel).

Kabuk

Dosya Sistemi

Genel olarak UNIX tabanlı sistemler için şöyle bir özdeyiş vardır: *UNIX sistemlerinde her şey birer dosyadır; eğer bir şey dosya değilse o zaman process'dir.*

Bu yaklaşım Linux için de geçerlidir. Bazı dosyaların diğerlerinden daha özel anlamlarının da olabilmesiyle birlikte (aygıt dosyaları, domain soketler, pipe, link vb.), pek çok işlem dosya sistematiği içerisinde kalınarak gerçekleşmektedir.

Dosya sistemi arayüzüne verilen önem, **proc** yalancı dosya sistemi gibi (*pseudo file system*) örneklerde de kendini gösterir. Bu dosya sistemini bir dizine bağlayıp (*mount*) kullanmaya başladığınızda içerisinde görülen dizin ve dosyalar, çalışan kernel içeriğinin bir yansımasından başka bir şey değildir.

Aygıt Dosyaları

Unix sistemlerde fiziksel aygıtlara erişmek için aygıt dosyaları (device files) kullanılır. Aygıt dosyaları sayesinde çekirdek içerisinde hangi aygıt sürücüsünün kullanılacağı belirlenmiş olur.

Bu yöntemle disk, seri port gibi fiziksel aygıtlara erişilebildiği gibi, */dev/random* gibi fiziksel bir karşılığı olmayan, sanal aygıt dosyaları üzerinden erişimler de mümkündür.

Aygıt dosyalarının ilişkili olduğu aygıt sürücüleri, çekirdek içerisindeki belirli bir kod kümesinden ibarettir. Örnek olarak SCSI veriyolu üzerindeki birinci disk üzerinde işlem yapmak için */dev/sda* dosyası kullanıldığında çekirdek içerisinde SCSI katmanıyla ilgili bir agıt sürücüsü devreye girerken, sisteme takılan ilk USB Seri Çevirici cihazına erişim için */dev/ttyUSB0* dosyası kullanıldığında USB - Seri çeviricilere ait sürücü kodları devreye girecektir. Peki acaba çekirdek içerisinde, erişilen aygıt dosyasının ismini metin karşılaştırma yöntemiyle kontrol edip, *sda* ise şu fonksiyonu çalıştır şeklinde bir tablo mu bulunmaktadır?

Yukarıdaki sorumuza vereceğimiz yanıt *hayır* olacaktır. Esasen */dev* dizini altındaki aygıt dosyalarının isminin ne olduğunun hiç bir önemi bulunmamaktadır. Aşağıdaki komutun çıktısını inceleyelim:

```
$ ls -l /dev/sda
brw-rw---- 1 root disk 8, 0 Mar  7 21:52 /dev/sda
```

Yukarıdaki çıktıda normal bir dosyadan farklı olarak 3 farklı bilgi daha yer almaktadır. Bunlar:

- Satırın ilk karakteri olan **b**, blok tabanlı bir aygıt dosyası olduğunu gösterir. Karakter tabanlı aygıt dosyaları için **c** kullanılır.
- İlgili aygıta ait **Major** numarası olarak **8** rakamı görünmektedir.
- İlgili aygıta ait **Minor** numarası olarak **0** rakamı görünmektedir.

Özetle aygıt dosyaları üzerindeki bu ek bilgilerle, dosyanın karakter tabanlı mı yoksa blok tabanlı erişim gerektirdiği, çekirdek içerisinde kullanılacak **Major** ve **Minor** numaralarının neler olduğu bilgisi saklanır.

Çekirdek açısından bu 3 bilginin aynı olduğu tüm aygıt dosyaları üzerinden kullanıcı kipinden yapılan her türlü erişim, çekirdek içerisinde aynı sürücü tarafından karşılanır. Yani aygıt tipi, major ve minor numarası */dev/sda* ile aynı olacak şekilde */dev/birinci-scsi-disk*, */etc/sda*, */mnt/disk* vb. herhangi isimlerde aygıt dosyaları üretsek de bu dosyalar üzerinden yapılacak erişimler hep aynı etkiyi üretir.

Peki çekirdek için aygıt dosya isimlerinin bir anlam taşımadığını öğrendiğimize göre isimlendirme kurallarına hatta aygıt dosyalarına neden ihtiyaç vardır?

Kullanıcı, Grup ve Erişim Yetkileri

Linux, çok kullanıcılı bir işletim sistemidir. Çalışan her uygulama mutlaka bir kullanıcı ve grup no ile ilişkilidir; kullanıcı veya grubu olmayan bir uygulamanın olması söz konusu değildir. Uygulama içerisinden yapılan tüm erişimler, bu kullanıcı ve grup hakları doğrultusunda gerçekleştirilir.

Benzer şekilde dosya sistemi katmanında da her dosya ve dizin mutlaka bir kullanıcı ve grupla ilişkili durumdadır. Linux sanal dosya sistemi katmanı (*Virtual Filesystem Switch*) ile dosya sistemlerinin (*ext4, xfs, btrfs* vb.) alt seviye implementasyon detayları soyutlanmış ve ayrı bir katman oluşturulmuştur.

Dosyaların kullanıcı, grup sahiplikleri ve erişim yetkileri, VFS katmanında tanımlanmış bir özelliktir. FAT32 gibi kendi içinde bu özelliklere sahip olmayan bir dosya sistemine Linux altında erişim yapılırken, kullanıcı, grup sahiplikleri ve erişim yetkileri öntanımlı değerlerle simüle edilir.

Okuma, Yazma ve Çalıştırma Yetkileri

Dosya ve dizinler üzerindeki yetki kontrolleri temel olarak okuma (read - **r**), yazma (write - **w**) ve çalıştırma (execute - **x**) bitleri üzerinden kontrol edilir. Bu bilgiler dosya sistemi üzerinde **inode** yapılarında saklanmaktadır.

Bu yetki bitlerinin dosya ve dizinler üzerindeki anlamları aşağıdaki gibidir:

Yetki	Açıklama
r	Dosyalar için okuma yetkisi, dosya içeriğine ulaşılabilmesi anlamını taşır. Dizinlerde olduğunda ise, dizin içeriğinin listelenebilmesini (dizinlerin okunması gibi düşünülebilir) sağlar.
w	Dosyalar için yazma yetkisi, dosya içeriğinin değiştirilebilmesini kontrol eder. Dizinler içinse, dizin içerisinde yeni dosya ve dizin oluşturma/silme operasyonlarını kontrol eder.
x	Dosyalar için çalıştırma yetkisi anlamı taşır. Dizinler için ilgili dizine geçilip geçilemeyeceğini belirtir.

Standart read, write ve execute yetki bitleri haricinde SUID, SGID ve Sticky Bit olmak üzere 3 farklı bit daha bulunmaktadır.

SUID (Set User ID) Yetkisi

SUID yetkisi dosyanın sahibiyle ilgili erişim yetkileri arasında yer alıp, SUID biti üzerinden kontrol edilir.

Bir uygulamada SUID biti aktif ise, o uygulamayı hangi kullanıcı çalıştırırsa çalıştırsın, uygulama dosyasının sahibi kim ise, onun haklarıyla çalışır.

En tipik örnek `passwd` uygulamasıdır. Bu uygulamayı kullanarak sistemdeki her kullanıcı kendi parolasını değiştirebilir. Ancak parola değişikliğinin yapılabilmesi için sadece **root** kullanıcısının yazma yetkisi olan `/etc/shadow` dosyasında değişiklik yapılması gereklidir. Sistemdeki `/usr/bin/passwd` dosyasının sahibi **root** kullanıcısı olmakla birlikte, biz herhangi bir normal kullanıcı olarak bu dosyayı çalıştırdığımızda, ilgili process kendi kullanıcı ID değerimizle ve bu değerin sistem ve dosya sistemi üzerindeki haklarıyla kısıtlı olarak faaliyet gösterecektir. Dolayısıyla parola değişimi için gereken `/etc/shadow` dosyasına yazma yetkisi olmadığından işlemi gerçekleştiremeyecek, hata alacaktır.

Bu problemin çözümü için, bazı uygulamaların çalıştıran kullanıcının kim olduğundan bağımsız olarak sistemde belirli bir kullanıcının yetkileriyle (örneğimizde root) çalışabilmesi için bir olanak sunulması gereklidir. SUID biti bu noktada devreye girer ve `/usr/bin/passwd` uygulamasına eklenen SUID biti sayesinde, uygulama çalıştırıldığı andan itibaren uygulama dosyasının dosya sistemindeki sahibi olan **root** kullanıcısının haklarıya çalışır.

Peki bu bir güvenlik zaafiyeti yaratmıyor mu, şeklinde bir soru aklınıza gelebilir. Elbette yaratıyor. Ama durum göründüğü kadar kötü değil. SUID bitine sahip uygulamaların geliştirilmesinde çok daha fazla özen gösterilir ve bu uygulamalar kısa bir zaman diliminde net bir şekilde belirlenmiş bir işlevi gerçekleştirip sona ererler. Örneğimizdeki `passwd` uygulaması, herhangi bir zaafiyet içermediği yıllardır kanıtlanmış olan güvenilir bir uygulamadır. Ancak SUID bitinin çok daha büyük ve kompleks uygulamalarda kullanılması sonucu, uygulama içi herhangi bir buffer overflow sorunu olması halinde, uygulamaya yapılacak başarılı bir saldırı sonrası sistemde **root** erişimi elde edilmesi gibi istenmeyen durumlar oluşabilir. 2000'li yıllardan önce bu tarz sorunlar çok daha fazla görülmekteydi ancak günümüzde hem bu tarz saldırıları zorlaştırıcı pek çok önlem mevcut hem de güvenlik tarafında artık belirli bir kültür oluştuğu için, bu bit daha bilinçli ve dikkatli biçimde kullanılmaktadır.

SGID (Set Group ID) Yetkisi

SUID biti ile benzer mantıkta, bir uygulamanın, kimin çalıştırdığına bakılmaksızın uygulama dosyasının grup sahibinin grup erişim yetkileri doğrultusunda çalıştırılmasını sağlamaktadır. SGID biti, SUID bitine oranla pratikte daha az kullanım alanı bulmaktadır.

Sticky Bit

Unix tabanlı işletim sistemlerinde `/tmp` dizini geçici dosya oluşturmak için tüm kullanıcıların ve uygulamaların kullanımına açılmıştır. Root kullanıcısının haklarıyla çalışan bir uygulama da, farklı kullanıcıların haklarıyla çalışan diğer uygulamalar da herhangi bir anda kısa veya uzun süreliğine geçici bir dosya oluşturma ihtiyacı duyabilir. Sistemde yazılabilir bir dizinin deneme/yanılma ile aranması makul bir süreç olmadığından, `/tmp` dizini bu işler için ayrılmıştır.

Bu noktada karşımıza önemli bir problem çıkmaktadır: `/tmp` dizini tüm kullanıcılar tarafından yazılabilir durumda ise, **A** kullanıcısının oluşturduğu `/tmp/test.txt` dosyasına **B** kullanıcısı tarafından yazılması veya dosyanın tamamen silinmesi nasıl engellenecektir?

İşte sticky bit (**t** olarak gösterilir) bu özel durumun çözümünde kullanılır.

Bir dizin üzerinde sticky bit aktif ise, o dizin altında her kullanıcı yeni dosya oluşturabilir ve kendi oluşturduğu dosyaları silebilir. Diğer kullanıcıların oluşturduğu dosyaları ise silemez.

> Sticky bit kullanımı dizinler için anlamlı olmakla birlikte, dosyalar için de kullanımı geçerlidir. Sticky bit aktif olan çalıştırılabilir bir uygulama dosyasının, işletim sistemi tarafından ilk çalıştırıldığı andan sonra bellekte tutulmaya devam edilmesi ve sonraki yüklemelerinin hızlı gerçekleştirilmesi istenir. Çok eskiden bu anlamlı bir kullanım iken, günümüzde modern işletim sistemlerindeki gelişmiş paylaşımlı kütüphane, sanal bellek ve page cache stratejileri sayesinde bu kullanım ihtiyacı ortadan kalkmıştır.

Erişim Yetkilerinin Görüntülenmesi

Bir dosya üzerindeki erişim yetkilerinin görüntülenmesi için `ls` komutu `-l` parametresi ile çalıştırılmalıdır:

```
$ ls -l /etc/debian_version
-rw-r--r-- 1 root root 11 May  5  2013 /etc/debian_version
```

Dizinler için aynı parametrelerle çalıştırıldığında, ilgili dizinin içeriği listelenecektir. Dizinin kendisiyle ilgili erişim yetkilerini görmek içinse komut `-d` parametresi eklenerek kullanılmalıdır:

```
$ ls -ld /etc/init.d
drwxr-xr-x 2 root root 4096 Feb  2 22:36 /etc/init.d
```

C kütüphanesi üzerinden erişim yetkilerini test edebilmek için `stat()` fonksiyonu kullanılır. Ayrıntılı kullanımı için man sayfasına bakabilirsiniz: man 2 stat

`ls` komutunun çıktısındaki erişim yetkileriyle ilgili ilk karakter aşağıdaki tablo doğrultusunda anlamlandırılır:

Karakter	Anlam
-	standart dosya (*regular file*)
d	dizin
c	karakter tabanlı aygıt dosyası (*/dev/console* vb.)
b	blok tabanlı aygıt dosyası (*/dev/sda3* vb.)
s	özel dosya (unix domain soket vb.)

Erişim yetkileriyle ilgili ilk karakterden sonraki 3'lü blok, dosya/dizin sahibinin dosya/dizin üzerindeki erişim yetkilerini gösterir.

Bir sonraki 3'lü blok, dosya/dizin'in grup sahibinin dosya/dizin üzerindeki erişim yetkilerini gösterir.

Sonraki ve son 3'lü blok ise, dosya/dizin için sahibi veya grup sahibi kısmına girmeyen kullanıcılar için (*other*) tanımlanmış olan erişim yetkilerini gösterir.

Erişim Yetkilerinin Düzenlenmesi

Dosya sisteminde yer alan bir dosya/dizin için erişim yetkilerinin düzenlenmesi işlemi `chmod` komutu ile yapılır. Değiştirilmek istenen yetkilerin neler olduğu ve üçlü erişim yetki bloklarından hangisi veya hangileriyle ilişkili olduğu parametre olarak belirtilir. Parametrelerde erişim yetki grubu belirtildikten sonra, **+** veya **-** ile hangi yetkilerin ekleneceği veya çıkartılabileceği belirtilebileceği gibi, **=** ile tam olarak hangi yetkilere sahip olacağı da belirtilebilmektedir.

`chmod` Erişim Yetki Grupları

Karakter	Etkilediği Grup
u	Dosya/dizin sahibi
g	Dosya/dizin grup sahibi
o	Dosya/dizin sahibi ve grup sahibi dışında kalanlar (others)
a	Dosya/dizin sahibi, grup sahibi ve bunların dışında kalanlar dahil her üç blok

`chmod` Erişim Yetki Parametreleri

Karakter	Yetki
r	Okuma
w	Yazma
x	Çalıştırma
X	Sadece dizinler için çalıştırma yetkisi
s	Çalışma zamanında **user id** veya **grup id** değişimi (suid bit)
S	Suid biti aktif fakat çalıştırma yetkisi yok
t	Sticky bit
u	Dosyanın kullanıcı (user) olarak sahibinin yetkileri
g	Dosyanın grup sahibi yetkileri
o	Dosya ile ilgili diğer (other) kullanıcıların yetkileri

Örnekler

Aşağıdaki `file1` dosyası ve `dir1` dizini için erişim yetki değişikliği örneklerini inceleyiniz:

Komut	Açıklama
`chmod u+rwx file1`	Dosyanın sahibine okuma, yazma, çalıştırma yetkisi ver
`chmod u-wx file1`	Dosyanın sahibinden yazma ve çalıştırma yetkisini kaldır
`chmod u+w-r file1`	Dosyanın sahibine yazma yetkisini ver, okuma yetkisini kaldır
`chmod u=rw file1`	Dosyanın sahibine okuma ve yazma yetkisi ver, çalıştırma yetkisi verme
`chmod u+x,g+wx file1`	Dosyanın sahibine çalıştırma yetkisi ekle, aynı zamanda grup sahibine yazma ve çalıştırma yetkisi ekle
`chmod g=rw file1`	Dosyanın grup sahibine okuma ve yazma yetkisi ver, çalıştırma yetkisini kaldır
`chmod o=r file1`	Sahip veya grup sahibi dışında kalan diğer kullanıcılara sadece okuma yetkisi ver
`chmod +x file1`	Tüm erişim gruplarına (sahip, grup ve diğer), çalıştırma yetkisi ekle
`chmod -x file1`	Tüm erişim gruplarından çalıştırma yetkisini kaldır
`chmod a=r file1`	Tüm erişim gruplarına (a = all) okuma yetkisi ver, yazma ve çalıştırma yetkisini kaldır
`chmod u+s file1`	SUID yetkisini ekle
`chmod g+s file1`	SGID yetkisini ekle
`chmod +t file1`	Sticky bitini aktifleştir
`chmod -R g-X dir1/`	Dizin altında recursive olarak dolaş, sadece bulunan alt dizinlerin grup sahiblerinden dizin içine geçme (x) yetkisini kaldır

Sekizli Sistemle Gösterim

`chmod` komutu yukarıdaki bölümde anlattığımız şekilde kullanılabileceği gibi, `chmod 755 dir` örneğindeki gibi erişim yetkilerinin bir sayı ile ifade edilmesi yöntemiyle de kullanılabilir.

Erişim yetkilerinin sayısal değerlerle ifade edildiği bu modelin kullanımı kulağa daha zor gelse de, en sık kullanılan 3-5 erişim yetki deseni için bir süre sonra numaralarının öğrenilmesi neticesinde pratikte sayısal değerlerin kullanımının oldukça yaygın olduğunu söyleyebiliriz.

Örnek olarak, dosyanın sadece sahibinin okuma/yazma yapıp, grup sahibi ve geri kalanların sadece okuma yaptığı **644** sayısı veya tüm kullanıcı ve gruplara tüm hakların verilmesini ifade eden **777** sayısı pek çok kullanıcı tarafından ezbere bilinir.

İstediğimiz erişim yetki bitleri için doğru sayıyı bulabilmek için aşağıdaki tablodan faydalanabilirsiniz. Soldan sağa doğru, read, write ve execute şeklinde sıralanan erişim bitlerinin aktif olma veya olmama durumuna göre (1 veya 0), elde edilen rakam önemlidir. Her 3 erişim yetki grubu için bu 3 bitin durumundan elde edilen sayı yazılıp birleştirildiğinde `chmod` için kullanılacak değer öğrenilmiş olur. Yani **644** olarak sayıyı görmek yerine, 1. üçlü erişim biti kombinasyonu değerinin **6** (rw-), 2. üçlü erişim biti kombinasyonu değerinin **4** (r--) ve son olarak 3. erişim biti kombinasyonu değerinin de **4** (r--) olduğunu düşünmeliyiz. Bu rakamları yanyana yazdığımızda **644**'ü elde ediyoruz.

8'li değer	Read	Write	Execute
7	r	w	x
6	r	w	-
5	r	-	x
4	r	-	-
3	-	w	x
2	-	w	-
1	-	-	x
0	-	-	-

Peki SUID, SGID ve Sticky Bit'inin rakam olarak ifadesini nasıl yapacağız?

3 adet olan bu özel bitleri de, `ls` komutu çıktısında gösterildiği sırasıyla, **sst** (rwx) biçiminde görebiliriz.

SUID	SGID	Sticky Bit
s	s	t

Bu özel bitlerin aktif olup olmama durumuna göre gene 0-7 arasında bir sayı değeri elde edilecektir. Örneği SUID ve SGID biti işaretlenmemiş ancak Sticky Bit işaretlenmişse bu bloktan elde edilecek sayı değeri **1** olacaktır. Dosya veya dizinin geri kalan erişim bitleri de **755** şeklinde ise özel bitlerin kombinasyonundan gelen rakam da bu değerin soluna eklenir ve **1755** şeklinde ifade edilir: `chmod 1755 mydir`

Benzer şekilde **2755**, **4755** değerlerinin nasıl bir erişim yetkisini ifade ettiğini düşününüz.

Artık neden bazı fonksiyon dokümantasyonlarında erişim modlarının **0644**, **0755** gibi ifade edildiğini de öğrenmiş olduk. Çoğu zaman bu özel 3 bit aktif olmadığından hep **0** değerini üretecektir.

Process Kavramı

Dosya İşlemleri

Unix tabanlı sistemlerde genel olarak, bir şey okunabiliyor veya yazılabiliyorsa, dosya arayüzü üzerinden de kullanılabilir. Örnek olarak ağ iletişiminde kullanılan soketler, seri port, dizinler, pipe, timer, sinyal vb. pek çok bileşen üzerinde dosya arayüzü ile işlem yapmak mümkündür.

Dosya arayüzünü açmak için öncelikle ilgili dosyayı açmak ve dosya betimleyiciyi *file descriptor* elde etmek gereklidir.

File descriptor elde etme yolları kullanılan sisteme göre değişir. Örneğin standart bir dosya için `open()` sistem çağrısı kullanılırken soket açmak istiyorsak `socket()`, timer oluşturmak istiyorsak `timerfd_create()` gibi ilgili sisteme özgü özel bir sistem çağrısı kullanılır.

Tüm bu çağrıların ortak özelliği, başarılı olunması durumunda geriye `int` tipinde bir file descriptor dönülüyor olmasıdır. File descriptor elde edildikten sonra artık bu dosya ile ilgili tüm işlemlerde, dosya kapatılana kadar ilgili `int` değeri kullanılıyor olacaktır.

Proses Dosya Tablosu

İşletim sistemindeki proses tablosunda, her proses ile ilgili açık dosyaların listesi tutulur. Bu liste yapısı içerisinde açık dosyalarla ilgili ek bir takım bilgiler de saklanır (örneğin normal bir dosya ise *inode* numarası, erişim yetkileri, son erişim zamanı, geçerli file position vb.)

Çekirdek içerisinde yer alan bu proses tablosunun daha basit bir haline **/proc** dosya sistemi üzerinden erişmek de mümkündür. Her bir proses id (PID) için aşağıdaki bilgilerin alınması mümkündür:

`/proc/PID/fd`

İlgili PID için açık dosyaların `int` descriptor numaralarını içeren sembolink linkler bulunur. Bu sembolink linklerin bulunduğu dizine `ls -l` komutuyla bakılacak olursa veya uygulamanızda `readlink()` fonksiyonuyla sembolik linkin işaret ettiği yerle ilgili bilgileri okumaya çalıştığınızda `type:[inode]` formatında bir yanıt gelir.

Dosya sistemi üzerinde fiziksel bir *inode* olmadığı senaryolar için de inode sistematiği içinde kalınarak özgün bir numara verilir.

Örnek olarak açık soket bağlantıları için `readlink()` sonrasında `socket:[205846]` şeklinde bir yanıt döner. Burada tip olarak **soket** bağlantısını olduğunu görmekteyiz, inode bilgisi olarak da **205846** numarası dönülmüştür. Bu numaranın soketler için anlamlandırılması `/proc/net` altındaki ilgili dosyaların okunmasıyla yapılır. Eğer bu soket bağlantısı bir TCP soketine aitse, `/proc/net/tcp` dosyasında, UDP soketine aitse `/proc/net/udp` dosyasında vb. burada görülen inode numarasıyla eşleşecek şekilde kayıtlar bulunduğu görülecektir.

Eğer inode konseptiyle henüz ilişkilendirilemeyen `epoll_create()`, `inotify_init()`, `signalfd()` gibi bir fonksiyonla elde edilmiş file descriptor söz konusu ise bu durumda `anon_inode:<file type>` formatında bir çıktı verecektir.

`/proc/PID/fdinfo`

Linux **2.6.22** versiyonu ve sonrasında bu dizin yapısı üzerinden ek bilgiler elde edilmesi mümkündür. Dizin içerisinde her bir `int` file descriptor ile aynı isimde dosyalar bulunur. Herhangi bir dosyanın içerisinde baktığımızda aşağıdaki bilgiler elde edilir:

```
pos:    0
flags:  02
mnt_id: 7
```

Bu dosyada file pointer pozisyonu, dosya açılırken kullanılan `flag`'ler, dosyanın ilişkili olduğu *mount point* referansı yer almaktadır.

Açık Dosya Limitleri

Unix tabanlı sistemlerde, bir prosesin aynı anda açabileceği dosya sayısının bir limiti bulunmaktadır.

Bu limitler **soft limit** ve **hard limit** adında 2 başlıkta toplanır. Açık dosya limitini `ulimit` uygulamasıyla veya `setrlimit()` fonksiyonuyla artırmak mümkündür. Ancak bu değer **hard limit**'i aşamaz. Bu limit ancak **root** kullanıcısı tarafından değiştirilebilir.

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 15234
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 15234
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited

$ ulimit -n 2048
$ ulimit -n
2048
```

Aşağıdaki örnek uygulama ile parametre vererek aynı anda kaç dosya açabileceğini test edebilir, `ulimit -n` komutuyla limitleri değiştirip tekrar testi gerçekleştirebilirsiniz.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include "../common/debug.h"

int main (int argc, char *argv[])
{
    int opened = 0;
    int wanted;
    int i;

    if (argc != 2) {
        printf("Usage: %s NumberOfFiles\n", argv[0]);
        exit(1);
    }

    wanted = atoi(argv[1]);
    for (i = 0; i < wanted; i++) {
        if (open("/dev/zero", O_RDONLY) < 0) {
            errorf("Couldn't open file");
            break;
        }
        opened++;
    }
    debugf("Total of %d files opened", opened);

    return 0;
}
```

Proses başına ayarlanan bu limitin haricinde, sistem genelinde maksimum açık dosya sayısının da bir limiti mevcuttur. Bu değer `/proc/sys/fs/file-max` dosyasından veya `fs.file-max` parametresi ile `sysctl` üzerinden okunabilir ve değiştirilebilir:

```
$ cat /proc/sys/fs/file-max
389794
$ sudo sysctl fs.file-max=400000
fs.file-max = 400000
$ cat /proc/sys/fs/file-max
400000
```

IO Modelleri

Bir process eğer zamanın çoğunu işlem ve hesaplamalardan ziyade IO (Input/Output) işlemlerinde harcıyorsa *IO bound*, tersi durumda ise *CPU bound* olarak tanımlanır.

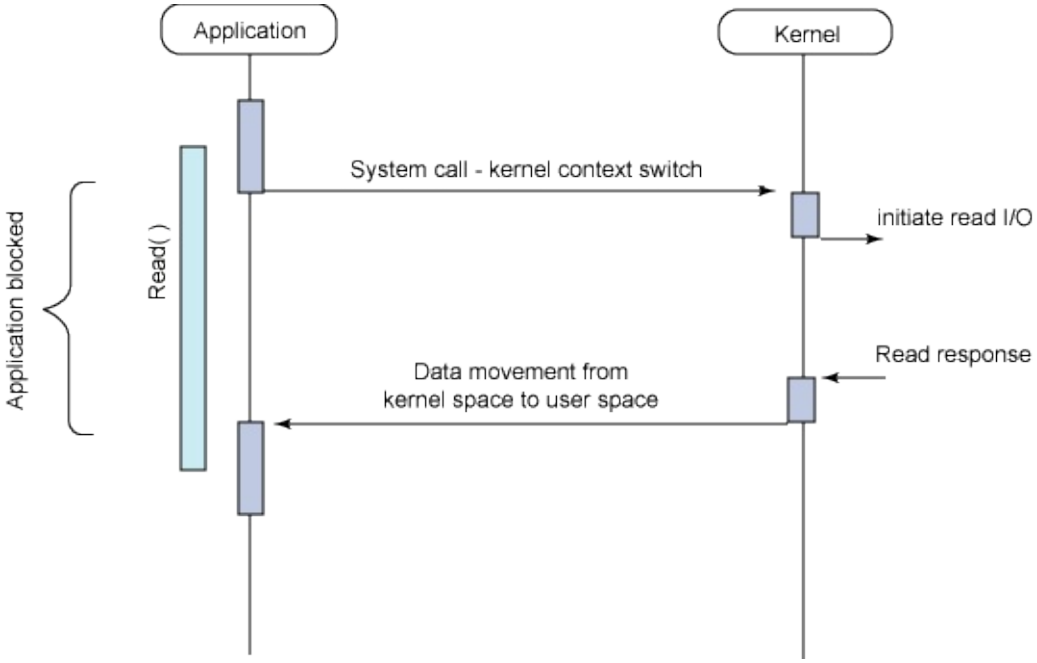
IO Modeli	Blocking	Non-Blocking
Senkron	Read/Write	Read/Write (O_NONBLOCK)
Asenkron	Multiplex IO (select/poll)	AIO

Linux platformunda IO işlemlerini ihtiyaç duyulan senaryo doğrultusundan verimli şekilde yapabilmek için IO modellerinin sunduğu imkanların incelenmesi gerekir.

Senkron IO

Senkron Blocking

IO işlemlerinde ilk seçeneğimiz senkron - Blocking IO modelidir. Bu modelde bir process IO talebinde bulunduğunda, işlem tamamlanana kadar ilgili process bloklanır. Linux çekirdeği bu durumda IO bekleme sürecinde bloklanan process'den CPU kaynağını geri alarak, uygun başka bir process'e tahsis eder. IO işlemi tamamlana kadar *IO Wait* durumunda bloklanan process'e bir daha asla CPU kaynağı tahsis edilmez. Böylelikle IO beklemelerinin sistemin geri kalanına etkisi minimum olur.



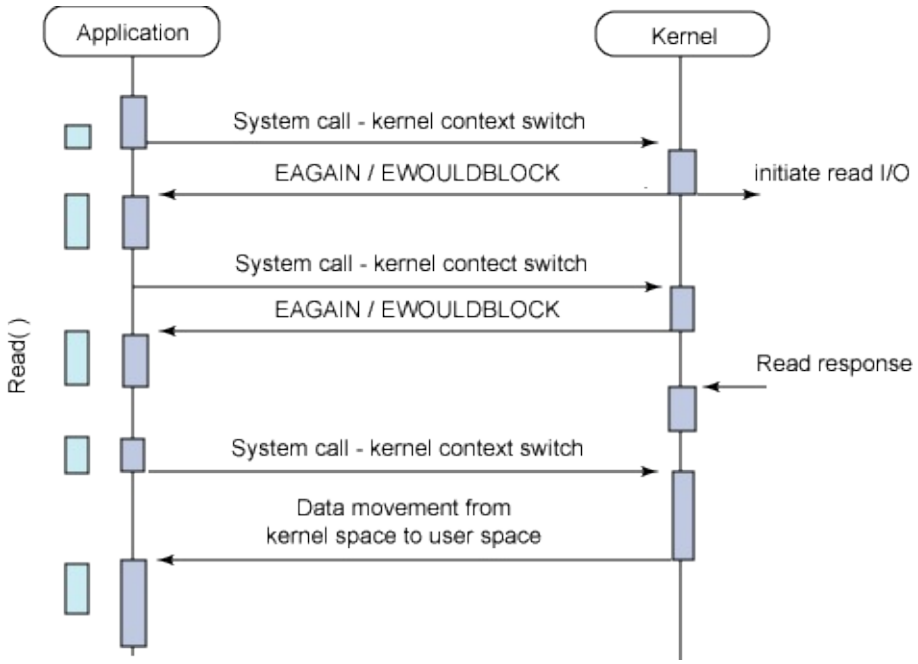
Tipik bir örnek verecek olursak, bir process *read()* fonksiyonuyla sistemden veri talebinde bulunduğunda ilgili sistem çağrısıyla kullanıcı kipinden kernel kipine *context switch* gerçekleştirilir ve sistem çağrısı tamamlanana kadar bloklanır. Çağrı tamamlandığında (okunan veri kullanıcı kipinde erişilebilecek bir tampona kopyalandığında veya herhangi bir hata oluştuğunda) ilgili process çalışmaya kaldığı yerden devam eder.

Bu geleneksel modelde her tür IO işlemi uygulamayı blokladığından, aynı anda uygulamada farklı işlemlerin yapılması gerekiyorsa *thread* kullanımı gerekecektir. Örnek olarak aynı anda 2 seri portu ve 20 farklı TCP soketini dinleyen bir uygulama yapmak istiyorsak, sadece IO işlemleri için toplamda 22 farklı thread kullanmamız gerekecektir. Aksi takdirde herhangi birinde IO nedeniyle karşılaşacağımız bloklanma nedeniyle, diğerlerinde veri gelmiş olsa dahi verinin işlenememesine, dolayısıyla uygulamanın toplamda olması gerekenden yavaş ve verimsiz çalışmasına yol açacaktır.

Senkron Non-Blocking

Senkron IO işleminin diğer bir varyasyonu, Non/Blocking IO modelidir.

Bu modelde IO işlemi yapılacak aygıt `O_NONBLOCK` bayrağı ile Non/Blocking modda açılır. IO kaynağı bu durumda iken herhangi bir *read()* işlemi yapıldığında bloklanmak yerine, verinin varsa hazır olan kısmı kernel kipininden kullanıcı kipindeki tampon alanına kopyalanır, veri henüz hazır değilse `EWOULDBLOCK` hatası döner, sistem çağrısı başka bir sinyal vb. yüzünden kesintiye uğramışsa da standart `EAGAIN` hatası ile döner. Böylelikle her halikarda bloklanmaksızın *read()* işlemi gerçeklenmiş olur.



Non/Blocking IO modelinde ihtiyaç duyduğumuz veriye tek seferde ulaşamayız. Bloklanmıyor olmanın negatif yönü, parça parça elde edeceğimiz verileri bir döngü ile tamamlanana kadar okumak zorunda oluşumuzdur. Bu aynı zamanda kod karmaşıklığını da artıracaktır. Ancak bloklanılmadığı için, verinin tamamını okumak amacıyla kurulan döngüde sadece okuma işlemi değil, diğer başka kontrol ve işlemlerin de yapılması mümkün olacaktır.

Benzer bir süreç yazma *write()* fonksiyonları için de geçerlidir.

Bu modelin asıl dezavantajı kod karmaşıklığını artırmasından ziyade, daha fazla sistem çağrısı gerektiriyor olmasıdır. Aynı işlem bloklanan bir modelde olsa idi tek bir sistem çağrısı ile yapılabilecekken, Non/Blocking IO modelinde bir döngü içerisinde tamamlana kadar, belirsiz sayıda sistem çağrısı yapılmak zorunda kalınmaktadır. Sistem çağrılarının genel maliyeti ve kullanıcı kipi -> kernel kipi *context switch*'lerinin sürekli yapılıyor olması sistem performansını önemli ölçüde düşürecektir.

Non/Blocking IO modu standart dosyalar üzerinde pek fazla anlam taşımaz. Asıl olarak soket işlemleri, FIFO ve pipe kullanımı, terminal, seri port vb. gibi aygıtlardan okuma ve yazmalarda kullanımı anlamlıdır.

Standart bir dosyayı Non/Blocking modda `O_NONBLOCK` bayrağı ile açsanız dahi, gerçekte değişen hiç bir şey olmaz. Linux tüm bu tarz dosya işlemlerini tampon kullanarak *buffered* biçimde gerçekleştirir. Herhangi bir yazma işlemi yapıldığında kullanıcı kipinden alınan veri doğrudan depolama birimine yazılmak yerine, kernel tarafından tutulan *page cache* tablolarına yazılır. Bu şekilde write çağrıları çok daha hızlı dönecektir. Ancak *page cache* tablolarında yer kalmadığında, yenilerini açmak için yapılan işlemler nedeniyle *write()* çağrıları uygulamayı bloklar (standart bir dosya Non/Blocking modda açılmış olsa dahi)

Benzer şekilde standart bir dosya bu modda okunmaya çalışıldığında kernel tarafından veriler disk üzerinden *page cache* içerisine alınır ve buradan kullanıcı kipine kopyalanır. Hatta özellikle sıralı erişim yapılan dosyalarda kernel, uygulama henüz talep etmeden, daha önden giderek o anki dosya ofsetinden ileriye doğru okuma yapıp *page cache* tabloları içerisine veri okumaya devam eder. Bu işlem **read-ahead** modu olarak da adlandırılır. Böylece uygulama katmanında sonradan yapılacak *read()* işlemleri doğrudan *page cache* içerisinden karşılanabilmektedir. İstenen verinin cache'te olmaması durumunda verinin diskten okunup getirilmesi için gereken süre zarfında *read()* fonksiyonu da bloklanacaktır.

> **Not:** 2014 Eylül'ünde **Milosz Tanski** tarafından hazırlanan bir patch ile `readv2()` sistem çağrısının `O_NONBLOCK` bayrağı ile standart dosyalar üzerinde de tam bir Non/Blocking çalışma desteği sunulmaya başlanmıştır. Detaylı testler sonrası ileride güvenle kullanılabilir hale geleceği öngörülmektedir. http://lwn.net/Articles/612483 adresinden ek bilgilere ulaşabilirsiniz.

Asenkron IO

Sinyaller

Sinyal mekanizması, sistemde yeni bir olay (**event**) oluştuğunda çalışan uygulamaların asenkron biçimde haberdar edilebilmesine olanak verir. Bu yapısı nedeniyle sinyal mekanizması genellikle yazılım tabanlı kesmeler (*software interrupt*) şeklinde de adlandırılır. Tıpkı donanım kesmelerinde olduğu gibi (*hardware interrupt*) sinyaller de bir uygulamanın normal akışını kesintiye uğratır ve uygulamanın ne zaman bir sinyal alacağı önceden bilinemez.

Sinyaller 3 temel durumda oluşturulur:

- Donanım tarafında istisnai bir durum oluştuğunda sinyal üretilir. Örnek olarak uygulamanın kendi izin verilen adres alanının dışındaki bir bölgeye erişmeye çalışması (**Segmentation Fault**), sıfıra bölme işlemi içeren bir makine kodu üretilmesi vb. gösterilebilir.
- Kullanıcı tarafından konsolda **CTRL-C** veya **CTRL-Z** gibi tuş kombinasyonlarının kullanımı, konsol ekranının yeniden boyutlandırılması ya da `kill` uygulaması ile sinyal gönderim isteğinin oluşturulması
- Uygulama içerisinde kurulan bir timer'ın dolması, uygulamaya verilen CPU limitinin aşılması, açık bir file descriptor'e veri gelmesi vb.

Sinyal kavramı Unix'in ilk versiyonlarından beri bulunmaktadır. Önceleri sinyal işlemleriyle ilgili Unix versiyonları arasında çeşitli farklılıklar mevcuttu. Sonradan sinyal yönetimi için POSIX standardizasyonu yapıldı ve Linux dahil diğer Unix türevleri de bu standartları takip etti. Bu nedenle kimi dokümanlarda karşılaşabileceğiniz Unix Sinyalleri ve POSIX Sinyalleri kavramları aradaki farklılıklara işaret etmektedir.

Temel Kavramlar

Sinyal Numaraları

Sinyaller sistemde **1**'den başlamak suretiyle nümerik değerlerle gösterilirler.

Örnek olarak **1** nolu sinyal, hemen her sistemde **HUP** sinyali olarak değerlendirilirken **9** nolu sinyal de **KILL** sinyali olarak bilinir.

Bununla birlikte uygulamalarımızda sinyal kullandığımızda bu rakamların kullanılması kesinlikle önerilmez. POSIX sinyalleri için uygulama içerisinde `<signal.h>` dosyası include edilmeli ve `SIGHUP`, `SIGKILL` gibi ilgili numaralara ait sabit tanımları kullanılmalıdır.

Sisteminizdeki `/usr/include/signal.h` dosyasını inceleyecek olursanız, dosya içerisinde `__USE_POSIX`, `__USE_XOPEN`, `__USE_POSIX199309` gibi değerlerin tanımlarına bakılarak yapılan ek işlemleri ve include edilen diğer dosyaları görebilirsiniz. Linux sistemlerdeki sinyal numaralarını uygulama kodunuzda doğrudan include etmeniz gerekmeyen `/usr/include/asm-generic/signal.h` dosyasında bulabilirsiniz.

Sinyal Üretimi ve Gönderimi

Sinyal üretimi, bir olay nedeniyle gerçekleşir. Ancak sinyalin ilgili uygulamaya gönderilmesi (**deliver**) işlemi, sinyalin üretimi ile aynı anda olmaz.

Sinyalin uygulamaya gönderilebilmesi için uygulamanın o an CPU kaynağına sahip ve *Running* durumda çalışıyor olması gereklidir. Dolayısıyla belirli bir uygulamaya gönderilmek üzere üretilen bir sinyalin gönderimi, ilgili uygulamanın *context-switch* sonrası tekrar çalışmaya başlamasıyla birlikte gerçekleşir.

Bekleyen (Pending) Sinyal Kavramı

Sinyalin üretim anından gönderilme işlemi gerçekleşene kadarki süre zarfında, sinyaller **pending** olarak işaretlenir ve beklemede tutulur.

Bir process için bekleme durumundaki sinyal sayısı ve izin verilen *pending* sinyal sayısına, **proc** dosya sisteminden ilgili process id (PID) dizini altındaki `/proc/PID/status` dosyasında **SigQ: 2/15235** örneğindeki gibi erişilebilir.

Sinyal Maskeleri ve Bloklama

Sinyallerin asenkron doğası nedeniyle uygulama tarafından ne zaman geleceğinin öngörülemez olması, uygulamada kritik bir işlem yapılıyorken kesintiye uğrama riskini de beraberinde getirir.

Bu şekildeki istenmeyen durumların önüne geçebilmek için sinyal maskeleri kullanarak kritik bir işlem öncesi bazı sinyalleri bloklama, kritik olan kısım tamamlandıktan sonra da tanımlanmış blokları kaldırma imkanı sağlanmıştır. Bu işlem uygulama geliştiricinin sorumluluğundadır.

Uygulama tarafından bir sinyal bloklandığında, blok kaldırılana kadar üretilen aynı tipteki diğer sinyaller bekleme (*pending*) durumunu korur. Uygulamada blok kaldırıldığın anda bekleyen sinyallerin gönderimi de sağlanır.

Bu şekilde blok anında beklemeye geçen sinyallerden aynı tipte olanlar, normal kullanımda blok kaldırıldıktan sonra uygulamaya sadece 1 defa gönderilir. Realtime sinyallerde ise durum farklıdır.

Sinyal Türleri

Sinyal türlerine göre öntanımlı eylemler değişkenlik gösterebilmektedir. Eğer uygulama içerisinde ilgili sinyali karşılayan bir *signal handler* fonksiyonu tanımlanmamış ise, öntanımlı eylem gerçekleşecektir. Bu bazen uygulamanın sonlandırılması bazen de sinyalin gözardı edilmesi anlamına gelir.

Bazı sinyallerin ise uygulama katmanında yakalanması mümkün değildir, bu tarz sinyaller her zaman öntanımlı eylemi gerçekleştirirler (**KILL** sinyali gibi).

Uygulamanın sonlanmasına yol açan bazı eylemlerde ek olarak **core dump** dosyası da üretilir. Core dump dosyaları, ilgili process'in sanal bellek tablosunun diske yazılması suretiyle, sonraki aşamalarda debug araçları ile process'in sonlanmadan önceki durum bilgisinin incelenebilmesi amacıyla oluşturulur.

Sinyal	No	Öntanımlı Eylem	Yakalanabilir mi?
SIGHUP	1	Uygulamayı sonlandır	Evet
SIGINT	2	Uygulamayı sonlandır	Evet
SIGQUIT	3	Uygulamayı sonlandır (core dump)	Evet
SIGILL	4	Uygulamayı sonlandır (core dump)	Evet
SIGTRAP	5	Uygulamayı sonlandır (core dump)	Evet
SIGABRT	6	Uygulamayı sonlandır (core dump)	Evet
SIGFPE	8	Uygulamayı sonlandır (core dump)	Evet
SIGKILL	9	Uygulamayı sonlandır	Hayır
SIGBUS	10	Uygulamayı sonlandır (core dump)	Evet
SIGSEGV	11	Uygulamayı sonlandır (core dump)	Evet
SIGSYS	12	Uygulamayı sonlandır (core dump)	Evet
SIGPIPE	13	Uygulamayı sonlandır	Evet
SIGALRM	14	Uygulamayı sonlandır	Evet
SIGTERM	15	Uygulamayı sonlandır	Evet
SIGUSR1	16	Uygulamayı sonlandır	Evet
SIGUSR2	17	Uygulamayı sonlandır	Evet
SIGCHLD	18	Yoksay	Evet
SIGTSTP	20	Durdur	Evet
SIGURG	21	Yoksay	Evet
SIGPOLL	22	Uygulamayı sonlandır	Evet
SIGSTOP	23	Durdur	Hayır
SIGCONT	25	Durdurulmuşsa sürdür	Evet
SIGTTIN	26	Durdur	Evet
SIGTTOU	27	Durdur	Evet
SIGVTALRM	28	Uygulamayı sonlandır	Evet
SIGPROF	29	Uygulamayı sonlandır	Evet
SIGXCPU	30	Uygulamayı sonlandır (core dump)	Evet
SIGXFSZ	31	Uygulamayı sonlandır (core dump)	Evet

Sinyal Yakalama ve Gönderme

Sinyal Yakalama

Uygulama tarafından yakalanabilmesine izin verilen sinyaller, ilgili fonksiyonlar kullanılarak uygulama içerisinde yakalanıp işlenecek hale getirilir. Bunun için temel olarak, yakalamak istediğimiz sinyalleri belirlemek ve bu sinyalleri işleyecek *callback* fonksiyonlarını hazırlamamız gerekir.

Sinyal işleme sürecinde aynı callback fonksiyonunu birden fazla sinyal için kullanmamız da mümkündür. Asenkron olarak çağırılacak callback fonksiyonunun prototipi aşağıdaki gibidir:

```
typedef void (*sighandler_t)(int);
```

Parametre olarak gelen `int` değeri, sinyalin numarasını gösterecektir.

Yakalamak istediğimiz sinyal ve karşılayacak callback fonksiyonunu belirtmek için `signal` veya `sigaction` fonksiyonları kullanılır. Kullanım prototipleri aşağıdaki gibidir:

```
sighandler_t signal(int signum, sighandler_t handler);

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

struct sigaction {
    void        (*sa_handler)(int);
    void        (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t    sa_mask;
    int         sa_flags;
    void        (*sa_restorer)(void);
};
```

`signal` kullanımı artık önerilmemektedir, bunun yerine daha fazla kontrol sunan `sigaction` kullanımı tercih edilmelidir. Zaten günümüzde *glibc* içerisindeki *signal* gerçekleştirimi de *sigaction* fonksiyonunu sarmalayan bir yapıdan ibarettir.

Aşağıdaki örnek uygulamayı `sample.c` adında kaydediniz.

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

static int exit_flag = 0;

static void handler (int signum)
{
    if (signum == SIGTERM) {
        printf("TERM signal received, exiting program\n");
        exit_flag = 1;
    } else {
        printf("Not a TERM signal: %d (%s)\n", signum, strsignal(signum));
    }
}

int main (int argc, char *argv[])
{
    struct sigaction act;

    memset (&act, '\0', sizeof(act));
    act.sa_handler = &handler;
    if (sigaction(SIGTERM, &act, NULL) < 0) {
        perror ("sigaction");
        return 1;
    }
    if (sigaction(SIGINT, &act, NULL) < 0) {
        perror ("sigaction");
        return 1;
    }

    while (!exit_flag) usleep(100000);

    return 0;
}
```

Sonrasında aşağıdaki gibi derleme işlemini yapıp çalıştıralım:

```
$ gcc -o sample sample.c
$ ./sample
```

Uygulamayı çalıştırdıktan sonra **CTRL-C** tuş kombinasyonu ile **INT** sinyali ürettiğimizde `sigaction` ile belirttiğimiz callback fonksiyonumuzun devreye girdiğini, **TERM** sinyali dışında bir sinyal geldiği için uyarı mesajı verip çalışmasına devam ettiğini görmekteyiz.

Başka bir konsoldan uygulamamıza `kill` komutu ile **TERM** sinyalini gönderdiğimizde ise beklediğimiz gibi `exit_flag` değişkeninin değerinin **1** yapılmak suretiyle sonlandığını görmekteyiz. Bu şekilde uygulama sonlandığında çıkış değerinin de **$?** kabuk değişkeninde beklediğimiz üzere **0** olduğunu görüyoruz:

```
$ ./sample
$ (diger konsoldan) kill -s SIGTERM $(pidof sample)
TERM signal received, exiting program
$ echo $?
0
```

Bu noktada şu soruyu kendimize soralım: Test uygulamamız çalışıyorken diğer konsoldan uygulama içerisinde işlediğimiz TERM ve INT sinyalleri dışında, örneğin USR2 sinyalini gönderirsek ne olur?

```
$ ./sample
$ (diger konsoldan) kill -s SIGUSR2 $(pidof sample)
User defined signal 2
$ echo $?
140
```

Yukarıda görüldüğü üzere, uygulamamız **USR1** sinyalini aldığında, *User defined signal 2* mesajıyla sonlandı. Bu bizim uygulama içerisinden bastırdığımız bir mesaj değil. Ayrıca uygulamanın çıkış değeri önceki örnekte olduğu gibi **0** değil, **140** oldu.

Uygulamada nasıl işleneceği düzenlenmiş olmayan bir sinyal geldiğinde, ilgili sinyalin önceki bölümde vermiş olduğumuz tabloda yer alan öntanımlı eylemi gerçekleşir. **SIGUSR2** sinyalinin öntanımlı eylemi de uygulamayı durdurmak olduğu için uygulamamız sonlandı. Eğer öntanımlı eylemi yoksay şeklinde olan örneğin **SIGURG** sinyalini göndermiş olsaydık, uygulamamız çalışmasına bir şey olmamış gibi devam edecekti:

```
$ ./sample
$ (diger konsoldan) kill -s SIGURG $(pidof sample)
```

Bu bilgiler ışığında, uygulamalarımızda sinyallere karşı önlem almadığımızda, beklenmeyen bir sinyalin bize ulaşması nedeniyle kontrolsüz sonlanmalara yol açacağını rahatlıkla söyleyebiliriz.

Sinyal Gönderme

Uygulama içerisinden sinyal göndermek için prototipi aşağıda belirtilen `kill` fonksiyonu kullanılır:

```
int kill(pid_t pid, int sig);
```

Fonksiyon basitçe parametre olarak verilmiş olan PID değerine sahip uygulamaya istenen sinyalin gönderimini sağlamaktadır. İşlemin gerçekleşebilmesi için ilgili uygulamaya sinyal gönderebilme yetkisinin olması gerekir. Örnek olarak A kullanıcısı B kullanıcısının uygulamalarına sinyal gönderemez, ancak kendi sahip olduğu uygulamalara sinyal gönderebilir.

Sistemdeki **root** kullanıcısı veya Linux Capabilities API üzerinden **CAP_KILL** kabiliyetine sahip olan uygulamalar ise herhangi bir uygulamaya sinyal gönderebilirler.

> Uygulama dışından genel amaçlı sinyal gönderme işlemleri için **kill** komutunu kullanabilirsiniz.

Uygulama içerisinden bir başka uygulama yerine, çalışan uygulamanın kendisine sinyal gönderilmek istendiğinde `raise` ve `abort` fonksiyonları kullanılır.

Signal-Safe Kavramı

Unix türevi sistemlerde iyi uygulama yazmanın temel kurallarından biri **signal safety** ve **thread safety** kavramlarının anlaşılmasıdır.

> Signal-Safe kavramı yoğunlukla **Async-Signal-Safe** biçiminde de ifade edilir.

Reentrancy

Konuya geçmeden önce **reentrant** kod kavramını incelememizde fayda var. Bir fonksiyon eğer aynı anda birden fazla *task* içinden çağrılsa da içerde kullanılan/üretilen verilerde herhangi bir bozulmaya yol açmayacak şekilde dizayn edilmiş ise, reentrant olduğunu söyleyebiliriz.

Dolayısıyla reentrant bir fonksiyonun karakteristik özelliklerini aşağıdaki gibi sıralayabiliriz:

- Farklı çağrımlarında statik bir alan üzerinde değişiklik yapmaya çalışmaz
- Sadece yerel değişkenler kullanır veya global bir değişken kullanımı zorunlu ise, yerel bir kopyasını çıkartarak işlemlerde yerel kopyasını kullanır
- Geri dönüş değeri olarak asla herhangi bir statik alanın adresini dönmez
- Fonksiyon içerisinden reentrant olmayan başka bir fonksiyon çağrılmaz

Bir **thread-safe** fonksiyon ise aynı anda birden fazla thread içerisinden, paylaşımlı bir alan kullanılsa dahi güvenle çağrılabilir. Paylaşımlı alan kullanımı söz konusu ise, ilgili alana erişimin **thread-safe** fonksiyonlarda gerekli kilit mekanizmalarıyla sıralı halde yapılması garanti edilir.

Bir fonksiyon eğer **thread-safe** ise aynı zamanda **reentrant**'dır.

Sinyal Kesmeleri

Sinyaller donanım kesmelerine benzer şekilde yazılımın akışını bir anda kesip ayrı bir yere dallanma (sinyal işleyici fonksiyonu) sonra kaldığı yerden devam etme özelliğine sahiptir. Tek işlemcili ve tek bir *thread* ile çalışan uygulamalarda dahi, signal-safety önemli bir problemdir.

Konunun daha iyi anlaşılması için görece zararsız örneklerden yola çıkalım. Örnek olarak uygulamanızın belirli bir bölümünde `printf` fonksiyonu ile konsola bir çıktı gönderdiğinizi varsayalım. Eğer *printf* fonksiyonu henüz çalışmasını bitirmemişken işlemin yarısında

uygulamaya sinyal gelirse ve sinyalin işlendiği fonksiyon içerisinde de ayrı bir *printf* fonksiyonu çalışıyorsa, sinyalin işlemi tamamlanıp asıl uygulamaya geri dönülüp ilerlendiğinde konsoldaki çıktılar birbirine karışacaktır.

Şimdi daha tehlikeli bir senaryoya göz atalım. Uygulamanızda `malloc()` ile **heap** alanından bir miktar bellek talep ettiğinizi düşünelim. İşlem henüz tamamlanmamışken bir sinyal ile kesintiye uğrar ve sinyal işleyici fonksiyonumuzda doğrudan veya dolaylı olarak `malloc()` fonksiyonunu çağırırsak ne olur?

Malloc işlemi performans açısından uygulama içerisindeki allokasyonları bağlı liste yapıları ile tutar. Bu liste yapısı güncelleniyorken sinyal nedeniyle yeniden çağrılması kritik hatalara yol açabilecektir.

Aşağıdaki kod parçacığını ve 64bit mimarideki karşılığını inceleyelim:

```
int y;
int x = 3;
y = x * 15;

/* derleyicinin urettiği kod */
movl    $3, -4(%rbp)
movl    -4(%rbp), %edx
movl    %edx, %eax
sall    $4, %eax
subl    %edx, %eax
movl    %eax, -8(%rbp)
```

Yukarıdaki örnekte programcı açısından tek satırlık `y = x * 15` satırı işlemcide 5 adımda gerçekleştirilmektedir (sayı yazmaca yükleniyor, `sall` ile 4 bit kaydırılıp 16 ile çarpılmış oluyor, sonra elde edilen değerden başlangıçtaki sayı çıkartılıp 15 ile çarpım değerine ulaşılıyor). Bu 5 adımın herhangi birinde sinyal nedeni ile kesinti gerçekleşebilir. Dolayısıyla kaynak kod seviyesindeki tek satırlık basit bir işlem dahi çokça işlem adımına yol açıyor olabilir.

Sinyal Maskeleme

Signal-safe olmayan fonksiyonların kullanımına yönelik 2 temel çözüm bulunur:

- signal-safe olmayan fonksiyonların kullanıldığı bölümlerde sinyalleri maskelemek
- sinyallerin işlendiği *callback* fonksiyonlarında signal-safe olmayan fonksiyonların kullanımından kaçınmak

Maskeleme işlemleri için sinyal kümeleri ve bloklama konularına geçelim.

Sinyal Kümeleri

Sinyal kümeleri, sinyal bloklama gibi işlemlerde tek bir küme ile birden fazla sinyali gösterebilmek için kullanılır.

Kümeler `glibc` içerisinde `sigset_t` tipiyle tutulur.

Bir sinyal kümesini ilklendirmek için 2 yol bulunur:

- `sigemptyset(sigset_t *set)` fonksiyonu ile hiç sinyal bulunmayan boş bir küme oluşturmak
- `sigfillset(sigset_t *set)` fonksiyonu ile tüm sinyallerin dahil olduğu bir küme oluşturmak

Sinyal kümeleri ile çalışırken mutlaka bu iki fonksiyondan biri kullanılarak kümenin ilklendirilmesi gerekir.

Bir sinyal kümesine belirli bir sinyali eklemek için `sigaddset (sigset_t *set, int signum)` fonksiyonu kullanılır.

Bir sinyal kümesinden belirli bir sinyali çıkarmak içinse `sigdelset (sigset_t *set, int signum)` fonksiyonu kullanılmaktadır.

Belirli bir sinyalin küme içindeki varlığını test etmek içinse `sigismember (const sigset_t *set, int signum)` fonksiyonu kullanılmaktadır.

Sinyal Bloklama

Sinyalleri bloklama ihtiyacı temel olarak aşağıdaki 2 durumda ortaya çıkmaktadır:

- Belirli bir sinyal işleyici fonksiyon içerisinde kritik bir işlem yapılıyorsa, farklı bir sinyal ile kesintiye uğramamak için
- Uygulama içerisinde aynı zamanda sinyal işleyici fonksiyon üzerinden de değiştirilmesine izin verilen bir global değişkeni değiştiriyorsak, sinyal ile kesintiye uğramamak için

> **Not:** İşletim sistemi sinyal nedeniyle ilgili uygulamadaki sinyal için tanımlı fonksiyonu çalıştırmayı başlatırken süreç tamamlanmadan aynı tipte gelen sinyallerde, sinyal işleyici fonksiyonu kesintiye uğratmaz. Ancak fonksiyon çalışmasına devam ederken farklı tipte bir sinyal gelirse, sinyal işleme fonksiyonu kesintiye uğrayacaktır.

Sinyal İle Birlikte Veri Gönderimi

Sinyal ve Core Dump

Thread Kullanımı

Linux altında thread kullanımı **C** kütüphanesinin bir parçası olan `pthread` kütüphanesi ile yapılmaktadır.

Diğer işletim sistemlerinin aksine, Linux'te thread ile process arasında çok az fark vardır.

Bu nedenle thread'ler, LightWeightProcess olarak adlandırılmaktadır.

Linux versiyon **2.6** öncesindeki thread implementasyonu **LinuxThreads** şeklinde adlandırılıyordu. Bu implementasyon performans ve senkronizasyon işlemleri açısından önemli limitlere sahipti. Maksimum çalışabilecek thread sayısı 1000'li rakamlarla sınırlıydı.

2003 yılında özellikle **IBM** ve **RedHat** firmasındaki geliştiricilerin başını çektiği ekip, **Native POSIX Thread Library** (NPTL) projesini kullanılabilir duruma getirmeyi başardı ve ilk olarak enterprise dünyanın Linux üzerinde Java Virtual Machine performansı şikayetlerini çözüme kavuşturmak için RedHat Enterprise versiyon 3'te kullanılmaya başlandı. Ardından GNU C kütüphanesindeki yeni NPTL implementasyonu gelmeye başladı. Günümüzde de her iki implementasyon glibc içerisinde yer almakta, kullanılan çekirdek versiyonu tarafından uygun implementasyon seçilmektedir.

Daha detaylı bilgi için en önemli ilk 5 Linux gurularından olan **Ulrich Drepper** ve **Ingo Molnar**'ın dizayn dokümanını inceleyebilirsiniz: http://www.akkadia.org/drepper/nptl-design.pdf

> Çalışan thread implementasyonunu `getconf GNU_LIBPTHREAD_VERSION` komutuyla öğrenebilirsiniz.

Her iki implementasyon da thread yönetiminin tamamen kullanıcı kipinde çalışan bir Virtual Machine üzerinde yapıldığı **green threads** uyarlaması değildir.

Her program başlatıldığında, bir kernel thread'i de otomatik olarak oluşturulur.

Çalışan herhangi bir process'in thread spesifik bilgileri `/proc/<PID>/task` dizini altında yer alır. Single-thread uygulamalar için de bu dizin altında **PID** ile aynı değere sahip bir **task** kaydı olduğu görünecektir.

Thread Oluşturma

Yeni bir thread oluşturmak için `pthread_create` fonksiyonu kullanılır.

Thread fonksiyonlarının kullanımı için `pthread.h` başlık dosyası include edilmelidir.

Threadler ana program ile aynı adresleme alanını ve aynı file descriptor'ları kullanırlar.

Pthread kütüphanesi aynı zamanda senkronizasyon işlemleri için gerekli **mutex** ve **conditional** işlemleri için gerekli desteği de içermektedir.

Pthread kütüphanesi fonksiyonları kullanıldığında uygulama `pthread` kütüphanesi ile de linklenmelidir:

```
$ gcc -o example example_thread.c -lpthread
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routin
e) (void *), void *arg)
```

Başarılı durumda **0** döner. Hata durumunda ise geriye bir hata kodu dönecektir.

`thread` parametresi `pthread_t` türünde olup önceden tanımlanması gerekir. Oluşan thread'e bu referansla her zaman erişilebilecektir.

`attr` parametresi thread spesifik olarak `pthread_attr_` ile başlayan fonksiyonlarla ayarlanmış, scheduling policy, stack büyüklüğü, detach policy gibi kuralları gösterir.

`start_routine` thread tarafından çalıştırılacak olan fonksiyonu gösterir.

`arg` ise thread tarafından çalıştırılacak fonksiyona geçirilecek `void*` 'a cast edilmiş genel bir veri yapısını göstermektedir.

Örnek Uygulama

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *worker(void *data)
{
    char *name = (char*)data;
    for (int i=0; i<120; i++) {
        usleep(50000);
        printf("Hello from thread %s\n", name);
    }
    printf("Thread %s done...\n", name);
    return NULL;
}

int main(void) {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, worker, "A");
    pthread_create(&th2, NULL, worker, "B");
    sleep(5);
    printf("Exiting from main program\n");
    return 0;
}
```

Joinable ve Detachable Thread Türleri

Thread kullanılan bir uygulamada `main()` fonksiyonundan *return* edilirse, tüm thread'ler de sonlandırılır ve kullanılan tüm kaynaklar sisteme geri verilir.

Aynı şekilde herhangi bir thread içerisinden `exit()` benzeri bir komutla çıkış yapılması halinde gene tüm thread'ler sonlandırılacaktır.

`pthread_join` fonksiyonu ile, bir thread'in sonlanmasını bekleyebiliriz. Bu fonksiyonun kullanıldığı thread, sonlanması beklenen thread sonlanana kadar bloklanacaktır.

Normal (*joinable*) thread'ler, sonlanmış olsa dahi `pthread_join` ile *join* işlemine tabi tutulmazlar ise, CPU tarafından tekrar schedule edilmeseler de sistemden kullandığı kaynaklar **geri verilmez**

Detachable Thread

Bazen `pthread_join` ile join işlemi yapmanın anlamlı olmadığı, thread'in ne zaman sonlanacağının öngörülemediği durumlar olabilir.

Bu durumda thread return ettiği noktada tüm kaynakların sisteme otomatik olarak geri verilmesini sağlayabiliriz.

Bunu sağlamak için ise, ilgili thread'leri, **DETACHED** durumu ile başlatmamız gerekmektedir.

Bir thread başlatılırken thread attribute değerleri üzerinden veya `pthread_detach` fonksiyonu ile *DETACH* durumu belirtiebilir:

```
int pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
int pthread_detach(pthread_t thread);
```

`pthread_join` Örneği

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>


void *worker(void *data)
{
        char *name = (char*)data;
        for (int i=0; i<120; i++) {
                usleep(50000);
                printf("Hello from thread %s\n", name);
        }
        printf("Thread %s done...\n", name);
        return NULL;
}

int main(void) {
        pthread_t th1, th2;
        pthread_create(&th1, NULL, worker, "A");
        pthread_create(&th2, NULL, worker, "B");
        sleep(5);
        printf("Exiting from main program\n");
        pthread_join(th1, NULL);
        pthread_join(th2, NULL);
        return 0;
}
```

Thread Sonlandırma

Bir thread, başka bir thread tarafından, ilgili `pthread_t` id değeri verilmek suretiyle *cancel* edilebilir:

```
int pthread_cancel (pthread_t thread);
```

Örnek: thread_cancel.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *worker(void *data)
{
    char *name = (char*)data;
    for (int i=0; i<120; i++) {
        usleep(50000);
        printf("Hello from thread %s\n", name);
    }
    printf("Thread %s done...\n", name);
    return NULL;
}

int main(void) {
    pthread_t th1, th2;
    pthread_create(&th1, NULL, worker, "A");
    pthread_create(&th2, NULL, worker, "B");
    sleep(1);
    printf("## Cancelling thread B\n");
    pthread_cancel(th2);
    usleep(100000);
    printf("## Cancelling thread A\n");
    pthread_cancel(th1);
    printf("Exiting from main program\n");
    return 0;
}
```

Mutex Kullanımı

pthread kütüphanesi **mutual exclusive** kilit mekanizmalarını **pthread_mutex** fonksiyonları ile yönetir.

Mutex tanımı ve ilklendirmesi şu şekilde yapılır:

```
pthread_mutex_t control_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Çalışma zamanında oluşturup sonradan ilklendirmek için:

```
    pthread_mutex_t control_mutex;
    ...
    pthread_mutex_init(&control_mutex, NULL);
    ...
    pthread_mutex_destroy(&control_mutex);
```

`pthread_mutex_init` 'in ikinci argümanı `pthread_mutexattr_` ailesi fonksiyonlarla mutex spesifik özelliklerle doldurulabilinir.

Mutex'i ele almaya çalışalım:

```
pthread_mutex_lock(&control_mutex);
```

Eğer mutex o an başka bir thread tarafından alınmış ise, mutex tipine göre aşağıdaki 3 durumdan biri gerçekleşir:

- Fonksiyon mutex'i elde edene kadar bekler (öntanımlı durum)
- Fonksiyon **-EDEADLK** hatasıyla döner
- Fonksiyon başarılı bir şekilde döner (recursive mutex kullanımı)

Mutex'i serbest bırakmak için:

```
pthread_mutex_unlock(&control_mutex);
```

SpinLock & Mutex Karşılaştırması

Mutex kullanımlarının performans problemi yaratabileceği senaryolar mevcuttur.

Örnek olarak A ve B thread'lerinin bir mutex kaynağını kilitleyip, çok hızlı biçimde işlemi gerçekleştirip kilidi kaldırdığını, bu işlemi de saniyede binlerce veya yüzbinlerce defa yaptığını düşünelim.

Mutex operasyonları *context-switch* gerektirdiğinden kilitli kalma süresi çok az olsa da bir yerden sonra her bir kilitleme işleminde thread'in Sleep durumuna geçmesi sonra tekrar uyandırılması sürecinin kendisi zaman alıcı bir işleme dönüşür.

Böyle bir durumda mutex kullanmak yerine spinlock kullanımı daha avantajlı olur.

Spinlock kullanıcı kipinde *busy wait* ile gerçekleştirilir. Dolayısıyla ilgili thread Sleep moduna geçmez, *context-switch* gerçekleşmez ve kullanıcı kipinde kendisine tahsis edilen cpu zamanını sonuna kadar kullanabilmiş olur.

Ancak eğer ilgili uygulama cpu kullanmaya devam ederek beklediği spinlock'a çok hızlı bir şekilde erişemez ise bu defa tersi bir etki yaratır ve Sleep moduna geçmesi halinde aynı zaman diliminde CPU ile başka bir işlem yapılabilecekken, bu imkan ortadan kalkar ve CPU yükünün artmasına neden olur.

Dolayısıyla spinlock kullanımının tüm senaryolarda daha iyi sonuç verecek olduğunu söylemek çok yanlış olur.

Dahası spinlock kullanımı özellikle Linux mutex performansının çok iyileştirilmiş olmasından ötürü, pek çok senaryoda mutex kullanımının daha iyi sonuç vermesini sağlar. Emin değilseniz, risk almayın, mutex kullanın.

Probleminizi iyi tanıdıysanız ve spinlock kullanımını iyice test ettiyseniz kullanabilirsiniz.

Aşağıdaki örnek uygulama kodunu `locktest.c` adıyla kaydedin. Normal mutex versiyonunu ek bir parametre vermeden, Spinlock versiyonunu `-DUSE_SPINLOCK` parametresiyle aşağıdaki gibi derleyip 100 milyon loop parametresi ile çalıştırıp işlem süresini ölçün.

Teste başlamadan önce Cpu Frequency Scaling aktif ise işlemci hızını aşağıdaki gibi maksimuma ayarlayıp, bu sebeple sonuçlarda oluşacak ek dalgalanmayı bertaraf etmeniz de önerilir (işlemin tüm cpu çekirdekleri için tekrar edilmesi gereklidir):

```
# echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

```
/* locktest.c */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <errno.h>
#include <sys/time.h>
#include "../common/utlist.h"
#include "../common/debug.h"

#define DEFAULT_LOOP_COUNT 10000000

struct list {
    int no;
    struct list *next;
};

struct timespec timespec_diff (struct timespec before, struct timespec after)
{
    struct timespec res;
    if ((after.tv_nsec - before.tv_nsec) < 0) {
        res.tv_sec    = after.tv_sec - before.tv_sec - 1;
        res.tv_nsec    = 1000000000 + after.tv_nsec - before.tv_nsec;
    } else {
        res.tv_sec    = after.tv_sec - before.tv_sec;
        res.tv_nsec    = after.tv_nsec - before.tv_nsec;
    }
    return res;
}


#ifdef USE_SPINLOCK
pthread_spinlock_t spinlock;
#else
pthread_mutex_t mutex;
#endif

struct list *mylist;

pid_t gettid() { return syscall( __NR_gettid ); }

void *worker (void *args)
{
    (void) args;
    struct list *tmp;

    infof("Worker thread %lu started", (unsigned long )gettid());

    while (1) {
#ifdef USE_SPINLOCK
        pthread_spin_lock(&spinlock);
```

```
#else
        pthread_mutex_lock(&mutex);
#endif

        if (mylist == NULL) {
#ifdef USE_SPINLOCK
            pthread_spin_unlock(&spinlock);
#else
            pthread_mutex_unlock(&mutex);
#endif
            break;
        }
        tmp = mylist;
        LL_DELETE(mylist, tmp);

#ifdef USE_SPINLOCK
        pthread_spin_unlock(&spinlock);
#else
        pthread_mutex_unlock(&mutex);
#endif
    }

    return NULL;
}

int main (int argc, char *argv[])
{
    int i;
    int loop_count;
    pthread_t thr1, thr2;
    struct timespec before, after;
    struct timespec result;
    struct list *el;

#ifdef USE_SPINLOCK
    pthread_spin_init(&spinlock, 0);
#elif USE_HYBRID
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ADAPTIVE_NP);
#else
    pthread_mutex_init(&mutex, NULL);
#endif

    if (argc == 2) {
        loop_count = atoi(argv[1]);
    } else {
        loop_count = DEFAULT_LOOP_COUNT;
    }
    infof("Preparing list");
    struct list *container = calloc(loop_count, sizeof(struct list));
    for (i = 0; i < loop_count; i++) {
        el = &container[i];
```

```
        el->no = i;
        LL_PREPEND(mylist, el);
    }
    clock_gettime(CLOCK_MONOTONIC, &before);

    pthread_create(&thr1, NULL, worker, NULL);
    pthread_create(&thr2, NULL, worker, NULL);

    pthread_join(thr1, NULL);
    pthread_join(thr2, NULL);

    clock_gettime(CLOCK_MONOTONIC, &after);

    result = timespec_diff(before, after);
    debugf("Elapsed time with %d loops: %li.%06li", loop_count, result.tv_sec,
            result.tv_nsec / 1000);

#ifdef USE_SPINLOCK
    pthread_spin_destroy(&spinlock);
#else
    pthread_mutex_destroy(&mutex);
#endif

    return 0;
}
```

Şimdi normal mutex ve spinlock versiyonlarını derleyelim:

```
$ gcc -o locktest_normal locktest.c -lrt -lpthread
$ gcc -DUSE_SPINLOCK -o locktest_spinlock locktest.c -lrt -lpthread
```

Her iki uygulamayı 100 milyon loop için aşağıdaki gibi çalıştırdığımızda performans farklılıklarını görebiliriz:

```
$ ./locktest_normal 100000000
debug: Elapsed time with 100000000 loops: 19.487106 (main locktest.c:120)

$ ./locktest_spinlock 100000000
debug: Elapsed time with 100000000 loops: 8.115992 (main locktest.c:120)
```

Testleri bir kaç defa daha üstüste çalıştırdığımızda bir miktar değişiklikler görebiliriz ancak normal mutex versiyonu, spinlock kullanan versiyondan en az 2 kat daha yavaş çalışmaktadır.

Hibrid Yaklaşım

Bu senaryo için spinlock'un daha iyi olduğunu gördük. Bir çok senaryo için de mutex kullanımının toplam sistem performansı için daha olumlu olacağını söyledik.

POSIX standardında her iki yaklaşımı birden kullanan hibrid bir modele de yer verilmiş olup Linux NPTL implementasyonunda böyle bir kullanım da desteklenmektedir.

Hibrid modelde mutex lock işlemi önce spinlock kullanılarak (try_lock mekanizmalarıyla) maksimum N defa denenenir (bu değerin hesaplaması glibc içerisinde mevcuttur). Ardından spinlock ile elde edilemiyorsa bu defa geleneksel mutex lock modeline geri dönülür.

Elbette hibrit modelin de tüm senaryolara uygulanabileceğini söylemek doğru olmaz. Spinlock kullanım avantajlarını hiç üretemeyen bir yazılım akışınız var ise, hibrid mod da performans kaybına yol açacaktır.

Hibrid modun kullanılabilmesi için, mutex initialize işleminde `PTHREAD_MUTEX_ADAPTIVE_NP` tipinin seçilmiş olması gereklidir.

Yukarıdaki örnek uygulamamızın `-DUSE_HYBRID` parametresiyle üçüncü bir versiyonunu derleyip test edelim:

```
$ gcc -DUSE_HYBRID -o locktest_hybrid locktest.c -lrt -lpthread
$ ./locktest_hybrid 100000000
debug: Elapsed time with 100000000 loops: 11.823482 (main locktest.c:120)
```

Görüldüğü üzere doğrudan spinlock kullandığımız versiyona oranla biraz daha kötü ama ona çok yakın bir değer elde etmiş olduk.

Semafor Kullanımı

POSIX semaforlar, farklı thread veya process'ler tarafından kullanılan ortak bir kaynağa erişim için gerekli senkronizasyon altyapısını sunar.

Semaforlar üzerinde kilitleme ve kilidi kaldırma yerine, semafor değerini artırma ve azaltma şeklinde işlemler yapılabilir.

Semafor ve Mutex Karşılaştırması

- Semaforların pthread mutex'lerden temel farkı, sadece aynı uygulamanın thread'leri arasında değil, farklı process'ler arasındaki senkronizasyon işlemleri için de kullanılabilmesidir.
- Aynı uygulama içerisinde semafor veya mutex kullanımı açısından performans farkı görülmese de, mutex kullanımı tercih edilmelidir. Mutex kullanımı sayesinde, ancak mutex'i lock etmiş thread'in tekrar unlock yapabilmesini garanti etmiş oluruz.
- Semafor kullanılması durumunda, bir thread tarafından artırılan semafor değeri, başka bir thread tarafından düşürülebilir. Bu nedenle semaforlar, eşzamanlı uygulama geliştirme sürecindeki `goto` deyimi olarak da nitelendirilir.
- Bununla birlikte, farklı process'ler arasındaki senkronizasyon probleminin çözümü ve async-signal safe yapısı nedeniyle multi-threaded uygulamalarda signal handler içerisinden kullanılabilmesi, semafor kullanımının avantajlı olduğu noktalardır.

Semafor Türleri

- İsimlendirilmiş (Named) Semafor
- Bu tipteki semaforlar, tanımlanan bir isim üzerinden farklı process'ler arasında kullanılabilirler.
- İsimlendirilmemiş (Un-Named) Semafor
- Atanmış bir ismi olmayan ancak ortak bir bellek bölgesinde yer alan semafor tipidir.
- Thread'ler arasında kullanılmak istendiğinde, heap'tan alınmış veya global olarak tanımlanmış değişkenler içerisinde tutulur.
- Process'ler arasında kullanılmak istendiğinde "shared memory" yöntemleriyle process'ler arasında erişilebilen ortak bellek bölgesinde tutulur.

İsimlendirilmiş (Named) Semafor Kullanımı

Tipik kullanımı şu şekildedir:

- `sem_open()` ile semafor oluşturulur veya kullanım için açılır
- `sem_post()` ile semaforun değeri artırılırken `sem_wait()` ile semaforun azaltılır
- `sem_getvalue()` ile o anki değeri öğrenilir
- `sem_close()` ile semafore erişim iptal edilir
- `sem_unlink()` ile semafor tamamen silinmek üzere işaretlenir, semaforu kullanan tüm process'ler `sem_close()` yaptıktan sonra silme gerçekleşir.

`sem_open`

- Yeni bir isimli semafor oluşturmak veya varolana erişebilmek için kullanılır.
- Eğer yeni bir semafor oluşuyorsa, semafora erişim izni için gerekli bit maskesi ve semaforun ilk değeri de parametre olarak verilir.
- Burada kullanılan bit maskesi, dosya işlemlerinde kullanılanlarla aynıdır.
- Semaforun oluşturulması ve ilk değerinin atanması işlemi atomik olarak gerçekleşir.
- Fonksiyon geri dönüş değeri olarak `sem_t *` şeklinde bir adres döndürür.

- Hata durumunda bu değer `SEM_FAILED` 'e eşit olur.
- Başarılı durumda `/dev/shm` dizini altında semaforun referansı oluşur.

`sem_close`

- Bir süreç semafor oluşturduğunda, çekirdek tarafından ilgili sürecin semafor ilişkisi kurulur.
- `sem_close` fonksiyonu ile çağrıldığı sürecin ilgili semaforla ilişkisi kesilir.
- İlgili semafor için kullanıldığı süreç sayısı sayacı bir azaltılır.
- Tüm isimlendirilmiş semaforlar, ilgili süreç sonlandığında veya `exec` çağrısıyla yeni bir uygulama belleğe yüklendiğinde, yukarıdaki işlem çekirdek tarafından otomatik yapılır.

`sem_unlink`

- Bir semaforu yok etmek için `sem_unlink` fonksiyonu kullanılır.
- Bu fonksiyonla ilgili semaforun, kullanımda olan tüm süreçler tarafından kapatıldığında otomatik olarak silinmesi istendiğini belirtmiş oluruz.
- İlgili tüm süreçler tarafından semafor kullanımı sonlandırıldığında, semafor yok edilir ve `/dev/shm` altındaki dosya referansı da silinir.

Örnek: Semafor Oluşturma

```c
/* sem_open.c */
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <getopt.h>
#include "common.h"

static void usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] name [octal-perms [value]]\n", progName);
    fprintf(stderr, "-c Create semaphore (O_CREAT)\n");
    fprintf(stderr, "-x Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    unsigned int value;
    sem_t *sem;
    flags = 0;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
        case 'c':
            flags |= O_CREAT;
            break;
        case 'x':
            flags |= O_EXCL;
            break;
        default:
            usageError(argv[0]);
        }
    }
    if (optind >= argc) usageError(argv[0]);
    /* Default permissions are rw-------; default semaphore initialization value is 0
*/
    perms = (argc <= optind + 1) ? (S_IRUSR | S_IWUSR) : getInt(argv[optind + 1], GN_B
ASE_8, "octal-perms");
    value = (argc <= optind + 2) ? 0 : getInt(argv[optind + 2], 0, "value");

    sem = sem_open(argv[optind], flags, perms, value);
    if (sem == SEM_FAILED)    errExit("sem_open");

    exit(EXIT_SUCCESS);
}
```

Semafor Operasyonları

Semaforları, işaretsiz integer değerler olarak düşünebiliriz.

Bir semafor üzerinde, artırma ve azaltma yönünde iki temel operasyon gerçekleştirilebilir.

Operasyonel Semafor Fonksiyonları

`sem_wait`

- Semaforun değerini 1 azaltmak için kullanılır.
- Eğer ilgili semafor değeri 1'den büyükse azaltma işlemi anında gerçekleştirilir ve fonksiyon geri döner.
- Semafor değeri zaten 0'a eşitse, `sem_wait` fonksiyonu, semaforun değerinin 1 artmasını bekler. Artış olduğunda hemen tekrar 0'a çeker ve fonksiyon geri döner.
- Yukarıdaki bloklanmış bekleme durumunda bir sinyal yakalanması halinde, sinyale ilişkin **SA_RESTART** bayrağının ayarlanmış olup olmamasından bağımsız olarak, **EINTR** hatasıyla fonksiyondan çıkılır.

Örnek: Semafor Bekleme

```c
/* sem_wait.c */
#include <semaphore.h>
#include "common.h"

int main(int argc, char *argv[])
{
    sem_t *sem;
    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sem-name\n", argv[0]);
    sem = sem_open(argv[1], 0);
    if (sem == SEM_FAILED)
        errExit("sem_open");
    if (sem_wait(sem) == -1)
        errExit("sem_wait");
    printf("%ld sem_wait() succeeded\n", (long) getpid());
    exit(EXIT_SUCCESS);
}
```

`sem_trywait` ve `sem_timedwait`

- Semafor bekleme operasyonu bir sinyal gelmediği müddetçe işlem gerçekleşene kadar çağıran süreci bloklamaktadır.
- Bloklama yapması istenmediği durumda, `sem_trywait` kullanılabilir. Bu modelde eğer semafor değeri *1* ise azaltılıp başarılı döner, ancak değer zaten *0* ise bloklama yapılmaz ve **EAGAIN** hatası ile dönülür.
- Bekleme işlemi için maksimum zaman aralığını seçmek istersek, `sem_timedwait` fonksiyonunu kullanabiliriz. Bu durumda semafor değeri hemen azaltılamıyorsa, maksimum olarak verilen parametre süresi kadar bekleyecektir. Bu süre zarfında da işlemin gerçekleşmemesi halinde **ETIMEDOUT** hatası ile döner.

`sem_post`

- Semaforun değerini 1 artırmak için kullanılır.
- Eğer semaforun değeri zaten 0 ise ve başka bir süreç aynı semaforu beklediği için bloklanmış durumdaysa, ilgili süreç uyandırılır.
- Yukarıdaki durumda birden fazla sürecin aynı semaforu beklerken bloklanması sözkonusu olursa, hangi sürecin uyandırılacağı garanti edilemez.

Örnek: Semafor Değerini Artırma

```c
/* sem_post.c */
#include <semaphore.h>
#include "common.h"

int main(int argc, char *argv[])
{
    sem_t *sem;
    if (argc != 2)
        usageErr("%s sem-name\n", argv[0]);

    sem = sem_open(argv[1], 0);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    if (sem_post(sem) == -1)
        errExit("sem_post");

    exit(EXIT_SUCCESS);
}
```

`sem_getvalue`

- Mevcut bir semaforun o anki değerini almak için kullanılır.

Örnek

```c
/* sem_get.c */
#include <semaphore.h>
#include "common.h"

int main(int argc, char *argv[])
{
    int value;
    sem_t *sem;
    if (argc != 2)
        usageErr("%s sem-name\n", argv[0]);

    sem = sem_open(argv[1], 0);

    if (sem == SEM_FAILED)
        errExit("sem_open");

    if (sem_getvalue(sem, &value) == -1)
        errExit("sem_getvalue");

    printf("%d\n", value);
    exit(EXIT_SUCCESS);
}
```

İsimsiz Semaforlar

- İsimsiz semaforlar, ortak bir bellek alanına erişimin mümkün olduğu tüm senaryolarda kullanılabilir.
- İsimlendirilmiş semaforlardaki fonksiyonlar aynen kullanılabilir.
- Bu fonksiyonlara ek olarak isimsize semaforlarda `sem_init` ve `sem_destroy` fonksiyonları da kullanılmaktadır.

`sem_init`

- İlgili semaforun ilklendirilmesi işlemlerini gerçekleştirir.

- Fonksiyonun 2. parametresi olan `pshared` değeri **0** olduğu takdirde, semafor sadece aynı sürecin thread'leri içerisinde kullanılabilir. Bu nedenle shared memory kullanımına gerek kalmadan, global bir değişkende veya heap üzerinde ayrılan bir alanda tutulabilir.
- `pshared` değeri 0'dan farklı olursa, semafor farklı süreçler arasında kullanılabilir. Bu durumda 1. parametredeki adres, shared memory üzerindeki bir yeri göstermelidir.
- Bu fonksiyonun aynı semafor için birden fazla çağrılması durumunda sistem kararlı davranamaz. Bu nedenle, her semaforun sadece bir defa ilklendirilmesi garanti edilmelidir.

`sem_destroy`

- İsimsiz bir semaforun sonlandırılmasını sağlar.
- Semafor sonlandırılmadan, tutulduğu bellek bölgesi free edilmemelidir.

Örnek: İsimsiz semafor kullanımı

```
/* sem_unnamed.c */
#include <semaphore.h>
#include <pthread.h>
#include "common.h"

static int glob = 0;
static sem_t sem;

static void *threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        if (sem_wait(&sem) == -1)
            errExit("sem_wait");
        loc = glob;
        loc++;
        glob = loc;
        if (sem_post(&sem) == -1)
            errExit("sem_post");
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;
```

```
    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    /* Initialize a thread-shared mutex with the value 1 */
    if (sem_init(&sem, 0, 1) == -1)
        errExit("sem_init");

    /* Create two threads that increment 'glob' */
    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExit("pthread_create");

    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExit("pthread_create");

    /* Wait for threads to terminate */
    s = pthread_join(t1, NULL);
    if (s != 0)
        errExit("pthread_join");

    s = pthread_join(t2, NULL);
    if (s != 0)
        errExit("pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

Shared Memory Kullanımı

- POSIX shared memory kullanımı, System V shared memory modeline göre avantajları nedeniyle özellikle yeni uygulamalarda daha fazla tercih edilmektedir.
- Linux 2.4 çekirdeği ile birlikte desteklenmeye başlanmıştır.
- Semafor kullanımına benzer şekilde, POSIX shared memory objeleri de Linux altında `tmpfs` dosya sistemi ile **/dev/shm** dizini altına bağlanmış halde tutulur.
- Bu dizin tmpfs türünde öntanımlı olarak, mevcut sistem belleğinin maksimum yarısını kullanacak şekilde bağlanır ancak `mount` işleminde parametre vererek bu değeri değiştirmek mümkündür.

`shm_open`

- `shm_open` fonksiyonu, standart kütüphanedeki `open` fonksiyonuyla aynı arayüze sahiptir.
- Parametre olarak dosya ismi, işleme dair ayarlanan opsiyonlar ve açılacak kaynak üzerindeki erişim yetkilerini alır.
- `open` fonksiyonundakine benzer şekilde **O_CREAT**, **O_EXCL**, **O_RDONLY**, **O_RDRW** ve **O_TRUNC** opsiyonlarının bir veya birkaçı birleştirilerek parametre olarak verilebilir.
- Geriye dönen file descriptor referansı üzerinden `fstat` fonksiyonuyla yapacağımız kontroller ile, shared memory alanının büyüklüğünü öğrenebiliriz. Yeni açılan shared memory alanlarının boyutu başlangıçta sıfırdır.
- Memory map işlemini yapmadan önce, `ftruncate` fonksiyonu ile shared memory alanının boyutunu istediğimiz değere ayarlayabiliriz. Bu işlem üretilen alandaki bellek bölgesini `'\0'` değerleriyle doldurur.

Örnek: Shared Memory Oluşturma

```
/* shm_create.c */
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "common.h"
```

```c
static void usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] name size [octal-perms]\n", progName);
    fprintf(stderr, "-c Create shared memory (O_CREAT)\n");
    fprintf(stderr, "-x Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
    int flags, opt, fd;
    mode_t perms;
    size_t size;
    void *addr;
    flags = O_RDWR;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
        case 'c':
            flags |= O_CREAT;
            break;
        case 'x':
            flags |= O_EXCL;
            break;
        default:
            usageError(argv[0]);
        }
    }
    if (optind + 1 >= argc)
        usageError(argv[0]);

    size = getLong(argv[optind + 1], GN_ANY_BASE, "size");
    perms = (argc <= optind + 2) ? (S_IRUSR | S_IWUSR) :
    getLong(argv[optind + 2], GN_BASE_8, "octal-perms");

    /*  Create shared memory object and set its size */
    fd = shm_open(argv[optind], flags, perms);
    if (fd == -1)
        errExit("shm_open");

    if (ftruncate(fd, size) == -1)
        errExit("ftruncate");

    /*  Map shared memory object */
    addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
            errExit("mmap");

    exit(EXIT_SUCCESS);

}
```

Örnek: Shared Memory Yazma

```c
/* shm_write.c */
#include <fcntl.h>
#include <sys/mman.h>
#include "common.h"

int main(int argc, char *argv[])
{
    int fd;
    size_t len;
    char *addr;
    /*  Size of shared memory object */

     if (argc != 3 || strcmp(argv[1], "--help") == 0)
       usageErr("%s shm-name string\n", argv[0]);

    fd = shm_open(argv[1], O_RDWR, 0);

    if (fd == -1)
        errExit("shm_open");

    /*  Open existing object */
    len = strlen(argv[2]);
    if (ftruncate(fd, len) == -1)
        /*  Resize object to hold string */
        errExit("ftruncate");

    printf("Resized to %ld bytes\n", (long) len);
    addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");
    if (close(fd) == -1)
        errExit("close");
    /*  'fd' is no longer needed */
    printf("copying %ld bytes\n", (long) len);
    memcpy(addr, argv[2], len);
    /*  Copy string to shared memory */
    exit(EXIT_SUCCESS);
}
```

Kullanım

- İlk örnekteki uygulamayı kullanarak, 4 byte uzunluğunda *deneme1* adında bir shared memory alanı oluşturalım:

  ./shm_create -c deneme1 4

- Ardından hexdump ile ilgili dosyaya bakalım:

  hexdump -C /dev/shm/deneme1 00000000 00 00 00 00

- Şimdi bu shared memory alanını genişleterek başka bir süreçten içerisine veri aktarıp hexdump ile kontrol edelim:

```
./shm_write deneme1 "Örnek bir metin"
Resized to 16 bytes
copying 16 bytes

hexdump -C /dev/shm/deneme1
00000000  c3 96 72 6e 65 6b 20 62  69 72 20 6d 65 74 69 6e  |..rnek bir metin|
```

`shm_unlink`

- Shared memory alanını kaldırmak istediğimizde kullanılır.
- `shm_unlink` işlemi sonrası, ilgili alanını memory-map yöntemiyle kullanmakta olan diğer süreçlerin mapping bilgilerini etkilemez. İlgili süreçlerde ayrıca `munmap` fonksiyonu çağrılmalıdır.
- Shared memory alanını kullanan tüm süreçler `shm_unlink` yaptıktan veya sonlandıktan sonra, işletim sistemi tarafından ilgili alan tamamen kaldırılır.

Örnek

```c
/* shm_remove.c */
#include <fcntl.h>
#include <sys/mman.h>
#include "common.h"

int main(int argc, char *argv[])
{
    if (argc != 2 || strcmp(argv[1], "--help") == 0)
            usageErr("%s shm-name\n", argv[0]);
    if (shm_unlink(argv[1]) == -1)
            errExit("shm_unlink");
    exit(EXIT_SUCCESS);
}
```

Memory Mapped IO

- Memory Mapped IO işlemleriyle temel olarak, `read` , `write` fonksiyonlarıyla yapılacak bir IO operasyonunun, ilgili sürecin adreslediği bellek alanı içerisinde doğrudan bellek erişim yöntemleriyle gerçeklenmesini sağlar.
- Birden fazla süreç, bir dosyayı kendi bellek alanlarına bu şekilde adreslediklerinde, aynı dosya üzerinde aynı anda bellek operasyonlarıyla işlem yapabilirler.
- Eğer mapping işlemi, **MAP_PRIVATE** bayrağı ile oluşturulmuş ise, ilgili sürecin yaptığı değişiklikleri diğer süreçler göremezler. Bu şekildeki bir kullanım aynı zamanda, yapılan değişikliklerin dosyaya da tekrar yazılmayacağı anlamına gelmektedir.
- **MAP_SHARED** bayrağı ile oluşturulmuş mapping'lerde ise, diğer süreçler de yapılan değişiklikleri görebilirler. Bu değişiklikler işletim sistemi tarafından dosyaya da yansıtılır.

`mmap`

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- `mmap` sistem çağrısı ile ilgili sürecin adresleme aralığında yeni bir mapping oluşturulur.
- `addr` alanı **NULL** ise sistem tarafından uygun bir adres otomatik olarak atanır.
- İşlemin başarılı olması durumunda, ilgili bellek adresi geri dönülür, diğer durumda **MAP_FAILED** makrosuyla test edebileceğimiz bir değer döner.
- `length` değeri mapping yapmak istediğimiz uzunluğu gösterir.
- `prot` değeri ilgili alan üzerindeki erişim yetkilerini gösterir.
- **PROT_NONE**: ilgili alana erişim engellenir
- **PROT_READ**: okunabilir
- **PROT_WRITE**: yazılabilir
- **PROT_EXEC**: kod çalıştırılabilir
- Bu değerler veya operatörüyle birleştirilebilir.
- `flags` parametresi ile alanın kullanım şeklinde dair opsiyonlar belirtilir.

- **MAP_PRIVATE** olması durumunda, yapılan değişiklikler sadece mapping'i yapan süreç içerisinde görünebilir. Sistem bu noktada **copy-on-write** mekanizmasını kullanır ve değişiklik yapılmadığı müddetçe ek bir bellek kaybı söz konusu olmaz.
- **MAP_SHARED** olması durumunda, yapılan değişiklikler aynı mapping işlemini yapmış diğer süreçlerde de görünür. Aynı zamanda sistem tarafından dosyaya da yansıtılır. Burada dosyaya yazma işlemini garanti etmek amacıyla `msync` sistem çağrısı da kullanılabilir.

Örnek: mmap kullanımı

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "../common/debug.h"

int main(int argc, char *argv[])
{
        char *addr;
        int fd;
        struct stat sb;
        if (argc != 2 || strcmp(argv[1], "--help") == 0)
                usageErr("%s file\n", argv[0]);
        fd = open(argv[1], O_RDONLY);
        if (fd == -1)
                errExit("open");

        /*  Obtain the size of the file and use it to specify the size of
         *  the mapping and the size of the buffer to be written */
        if (fstat(fd, &sb) == -1)
           errExit("fstat");

        addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
        if (addr == MAP_FAILED)
                errExit("mmap");

        if (write(STDOUT_FILENO, addr, sb.st_size) != sb.st_size)
                fatal("partial/failed write");

        exit(EXIT_SUCCESS);
}
```

`munmap`

- `mmap` işleminin tersini yaparak ilgili sürecin bellek alanındaki mapping'i kaldırır.

- Bellek üzerinde yapılan operasyonların dosya üzerinde de gerçekleştiğinden emin olmak için, `munmap` öncesinde `msync` fonksiyonunu kullanmak gerekir.

Dosya Mapping İşlemleri

- **MAP_PRIVATE** yöntemiyle dosyalar üzerinde mapping uygulamak, özellikle çalıştırılabilir dosyalar ve paylaşımlı kütüphane kullanımı için önemlidir.
- Bu sayede aynı uygulamanın veya kütüphanenin *text* segmenti salt-okunur olarak açılabilir ve sistemdeki tüm süreçler tarafından ortak kullanılabilirken, *data* segmenti her süreç için özel olarak mapping yapıldığından, ilgili sürece ait değişiklikleri taşıyabilir.
- **MAP_SHARED** yöntemiyle uygulanan mapping işlemlerinde ise tüm süreçler bellekteki aynı fiziksel alanı kullandığından önemli bir performans kazancı oluşur.

Memory Mapped IO Avantajları

- `read` , `write` operasyonlarına yerine bellek üzerinde işlem yapmak, daha kolay okunur ve kısa kodlar üretebilmemizi sağlar.
- Tanımlanmış bir `struct` ile, bellek üzerinde geziniyormuş gibi işlemler yapılabilir. Özellikle rastgele erişimli dosyalarda ciddi bir kolaylık sağlanır.
- Bu yöntemle yapılan operasyonlar daha performanslı çalışır.
- Normal `read` ve `write` işlemlerinde, sistemde 2 adet transfer işlemi oluşur. Bunlardan birincisi verinin dosyadan kernel içerisindeki tampon alanına taşınması, ikincisi ise tampon alandan ilgili userspace sürecin bellek alanına taşınması işlemidir.
- Memory mapped IO modelinde ikinci işleme gerek olmadığından, hem taşıma işleminden kurtulunur, hem de toplamda ihtiyaç duyulan bellek miktarı azalır. Özellikle birden fazla sürecin aynı (büyük) dosya üzerinde rastgele erişim yaptığı modelde önemli oranda avantaj sağlamaktadır.

Dezavantajlar

- Küçük çaplı IO işlemlerinde, memory mapped IO yönteminin performansı daha düşük olacaktır.
- Bunun temel nedeni, başlangıçta gereken ekstra mapping işlemi, page fault, unmap işlemleri vb.)

- Bir diğer problem de bellek üzerinde yapılan işlemlerin dosyaya yansıtılmasını garanti etmek için ek işlem yapılma ihtiyacı olarak gösterilebilir.
- Bu nedenlerle basit dosya işlemlerinde kullanımı genellikle önerilmez.

Soket Kullanımı

Soketler, prosesler arası haberleşmede (IPC) kullanılan özel dosyalardır. Aynı makina üzerindeki veya ağ bağlantısına sahip farklı makinalardaki prosesler, birer uç noktası (endpoint) olarak tanımlanan soketler üzerinden, çift yönlü olarak, haberleşebilmektedir.

Bu bölümde soket arayüzünü, haberleşme protokollerini (TCP, UDP) ve istemci sunucu uygulamalarını inceleyeceğiz.

Soket API

Bu bölümde, sonraki bölümlere alt yapı oluşturmak amacıyla, temel soket fonksiyonlarına sırasıyla değinmeye çalışacağız.

socket

Haberleşecek her iki taraf da ilk olarak birer soket oluşturmalıdır. Soketler, *socket* sistem çağrısı ile oluşturulmaktadır.

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

socket fonksiyonu başarılı olması durumunda bir soket oluşturur ve sonraki çağrılarda kullanılmak üzere, sokete ilişkin bir dosya betimleyicisi (file descriptor) döner. Başarısızlık durumunda ise -1 değerine geri dönmektedir. Oluşan soketin gerçekte ne olduğunu ve geri dönüş değerinin neyi gösterdiğini merak edebilirsiniz.

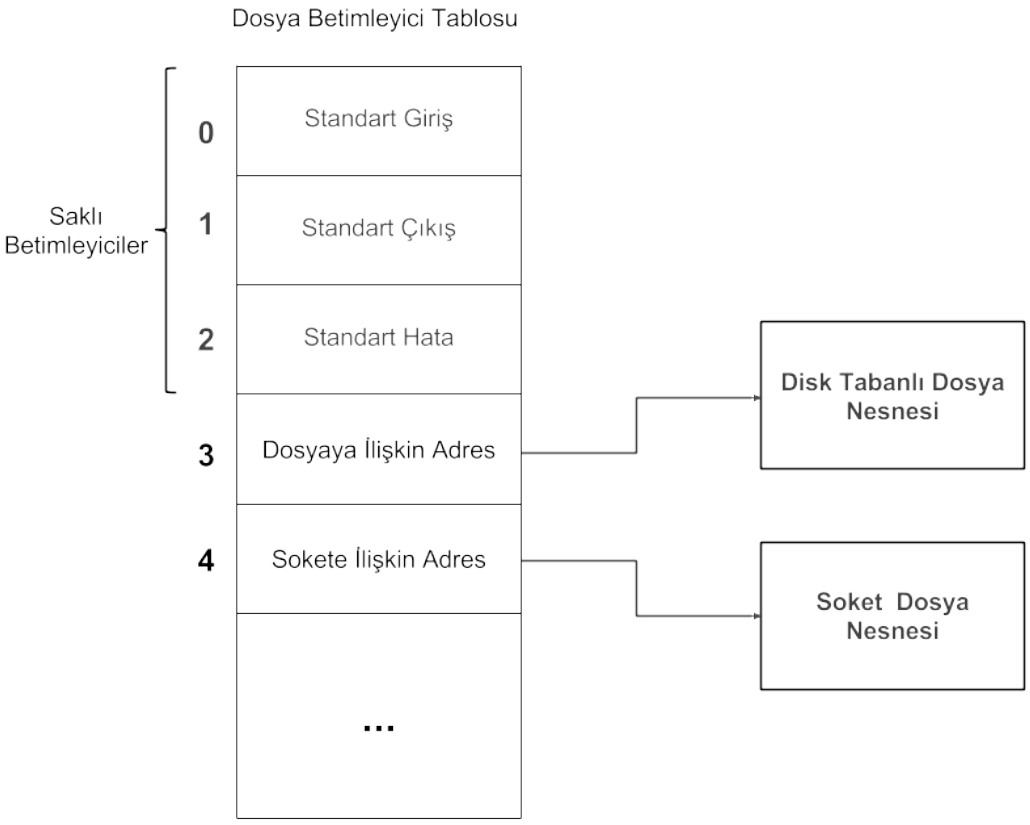
Unix sistemlerinde tüm Giriş/Çıkış işlemleri dosya betimleyicileri üzerinden, okuma ve yazma işlemleri şeklinde yapılmaktadır. Dosya betimleyicileri açık dosyalara erişimi sağlayan birer tamsayı değeridir. Örneğin, disk tabanlı bir dosya açıldığında, işletim sistemi bellekte o dosyaya ilişkin, adına dosya nesnesi (file object) denilen, bir alan oluşturur. Dosya ile ilgili daha sonraki okuma ve yazma işlemleri bu dosya nesnesi üzerinden yapılmaktadır. Bu dosya nesnesine ulaşmak için işletim sistemi her proses özelinde ayrıca bir dosya betimleyici tablosu (file descriptor table) denilen bir alan daha tutmaktadır. Gösterici dizisi şeklindeki bu tablonun her elemanı bir dosya nesnesinin adresini tutmakta ve dosya betimleyicileri de bu tabloda bir indeks değeri göstermektedir. *open* fonksiyonuyla bir disk tabanlı dosya açtığımızda fonksiyonun, başarılı olması durumunda, döndüğü değer aslında dosya betimleyici tablosundaki indeks değeridir.

soket fonksiyonunun argümanlarına geçmeden bu durumu aşağıdaki gibi bir örnek üzerinden gözleyebiliriz. Örneği *test.c* adıyla saklayıp aşağıdaki gibi derleyip çalıştırabilirsiniz.

```c
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
    int fd;

    fd = open("./test.c", O_CREAT);
    printf("%d\n", fd);

    fd = socket(PF_INET, SOCK_STREAM, 0);
    printf("%d\n", fd);

    return 0;
}
```

```
$ gcc -otest test.c

$ ./test
3
4
```

Bir proses başlatıldığında dosya betimleyici tablosundaki ilk `3` giriş, standart okuma ve yazma işlemleri için, saklı durumdadır. Bu aşamadan sonra açılacak her dosya için dosya betimleyici tablodaki en düşük indeksli alan ayrılmaktadır. Örneğimizde disk tabanlı dosya için 3 numaralı girişin, soket için ise 4 numaralı girişin ayrıldığını görmekteyiz. Bu durumu görsel olarak aşağıdaki gibi gösterebiliriz.



Ortada gerçek anlamda bir dosya olmamasına karşın, soketler programcı tarafından bakıldığında gerçek birer dosyaymış gibi görünmektedir. Gerçek dosyalara ve soketlere aynı dosya betimleyici tablosu üzerinden erişildiğinden soketler üzerinde de dosya fonksiyonlarını kullanmak mümkün olmaktadır. Soketlerle çalışmak normal dosyalara göre daha karmaşık olduğundan okuma ve yazma işlemleri için ayrıca özel fonksiyonlar da bulunmaktadır.

> **Not:** Soketlere ilişkin dosya sisteminde girişler oluşturulabilir. Bu tür soketlere (Unix Domain Socket) daha sonra değineceğiz.

socket fonksiyonunun aldığı argümanlar ve ilgilendiğimiz başlıca değerleri aşağıdaki gibidir.

- **domain:** Kullanılacak olan protokol ailesini (protocol family) gösterir. *bits/socket.h* dosyasında sembolik sabitler şeklinde tanımlanmışlardır. Kullanacağımız başlıca değerler aşağıdaki gibidir.

Alan (Domain)	İletişim Ortamı	Adres Biçimi	Adres Yapı Türü
AF_UNIX	Aynı makina	Dosya yolu	sockaddr_un
AF_INET	IPv4 yoluyla ağ üzerinden	32-bit IPv4 adresi + 16-bit port numarası	sockaddr_in
AF_INET6	IPv6 yoluyla ağ üzerinden	128-bit IPv4 adresi + 16-bit port numarası	sockaddr_in6

socket fonkiyonuna, AF_UNIX geçirilmesi durumunda bir UNIX alan soketi oluşturulurken, diğer iki argüman için Internet alan soketi oluşturulmaktadır. Tablodaki diğer bilgilere yeri geldiğince değineceğiz.

- **type:** *type* argümanı ile soketin tipi bildirilmektedir. *bits/socket_type.h* bu amaçla tanımlanmış sembolik sabitler bulunmaktadır. Başlıcaları aşağıdaki gibidir.

Tip	Anlamı
SOCK_STREAM	Güvenli, bağlantı temelli protokol (TCP)
SOCK_DGRAM	Güvensiz, mesaj temelli protokol (UDP)
SOCK_RAW	IP veya Ethernet gibi alt seviye protokol

SOCK_STREAM ile tanımlanan soketler *stream* soketi, SOCK_DGRAM ile tanımlananlar ise *datagram* soketi olarak isimlendirilmektedir.

- **protocol:** Bu argümanın değeri diğer ikisine göre değişmektedir, çoğunlukla `0` değeri kullanılmaktadır.

bind

bind fonksiyonu bir soketi bir adresle ilişkilendirmek için kullanılmaktadır.

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Parametre	Görevi
sockfd	*socket* çağrısından elde edilen dosya betimleyicisi
addr	Soketin bağlanacağı adres bilgilerinin tutulduğu alana ait gösterici
addrlen	Adres bilgilerinin tutulduğu alanın uzunluğu

socket fonksiyonunu incelerken, soketlerin birden çok alan için (AF_UNIX, AF_INET, AF_INET6) tanımlanabildiklerini görmüştük. Bu durumda farklı uzunluklardaki adres alanlarına ihtiyaç duyulmaktadır. Örneğin IPv4 ve IPv6 için soket adresleri sırasıyla, *netinet/in.h* dosyasında tanımlı, aşağıdaki yapı türlerinde saklanmaktadır. IPv4 için IP adresi 32 bit ile tutulurken IPv6 için 128 bit kullanıldığını hatırlayınız.

```
struct sockaddr_in {
    short int           sin_family; // AF_INET
    unsigned short int  sin_port; // Port numarası
    struct in_addr      sin_addr; // IPv4 adresi
    unsigned char       sin_zero[8];
}
```

```
struct sockaddr_in6 {
    u_int16_t           sin6_family; // AF_INET6
    u_int16_t           sin6_port; // Port numarası
    u_int32_t           sin6_flowinfo;
    struct in6_addr     sin6_addr; // IPv6 adresi
    u_int32_t           sin6_scope_id;
}
```

bind fonksiyonunu bu türlerden bağımsız yazabilmek için ayrıca genel bir adres türü olarak *sockaddr* türü tanımlanmıştır.

```
struct sockaddr {
    sa_family_t     sa_family; // Adres ailesi (AF_*)
    char            sa_data[14]; // Soket alanına göre değişen adres
};
```

bind fonksiyonu çağrılırken gerçek adres türünden *sockaddr* türüne tür dönüşümü yapılmakta ve gerçek adres alanının uzunluğu *addrlen* parametresine geçirilmektedir. *bind* fonksiyonu ayrıca *sa_family* alanının değerine bakarak yapının geri kalan kısmının uzunluğunu ve adres formatını belirleyebilmektedir.

Şimdi soket ile ilişkilendireceğimiz adres alanını nasıl oluşturacağımıza bakalım. Daha basit olduğundan IPv4 adresini kullanacağız. *sockaddr_in* içindeki *sin_addr* alanına ait yapı ve tür tanımı aşağıdaki gibidir.

```
typedef uint32_t in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};
```

16 byte uzunluğundaki IPv4 için soket adres alanını (sockaddr_in) görsel olarak aşağıdaki gibi gösterebiliriz.

0x0	Adres Ailesi (AF_INET)	Port
0x4	IP Adresi	
0x8	0	
0xC	0	

Birden çok byte'tan oluşan tam sayılar işlemci mimarisine göre bellekte iki farklı şekilde tutulabilmektedir. Örneğin, x86 mimarisinde sayının düşük anlamlı byte'ı düşük adreste tutulurken başka bir mimaride yerleşim tam tersi olabilir. x86 gibi düşük anlamlı byte'ın düşük adreste tutulduğu mimariler *little endian* olarak isimlendirilirken, sayının düşük anlamlı byte'ını yüksek adreste saklayan mimariler *big endian* olarak isimlendirilmektedir. Haberleşen sistemlerin byte sıralamaları farklı olabilmektedir bu yüzden ağ üzerinden gönderilecek olan bilgilerin byte sıralaması önem taşımaktadır. Ağ üzerinden gönderilen bilgiler geleneksel olarak *big endian*'dır (network byte order). Bu durumda soket adresine ilişkin alan içerisine IP adresinin ve port numarasının yazılma biçimi önem taşımaktadır. Çalışılan sistem big endian ise, port ve IP değerleri olduğu gibi yazılırken tersi durumda byte sıralamalarının değiştirilmesi gerekmektedir. Soket arayüzü, çalışılan sistem (host) ile ağ (network) arasındaki byte sıralaması uygunluğunu sağlamak için aşağıdaki yardımcı fonksiyonları barındırmaktadır.

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

listen

Stream soketi üzerinde dinleme yapmak için *listen* fonksiyonu kullanılır.

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Soketler bir bağlantıyı başlatmak için kullanılabildikleri gibi gelen bağlantıyı kabul etmek için de kullanılmaktadırlar. Bağlantıyı başlatmak için kullanılan soketler *aktif*, bağlantıyı kabul edenler ise *pasif* olarak isimledirilmektedir. Bir soket *socket* fonksiyonuyla oluşturulduğunda

aktiftir, *listen* fonksiyonu ile soket pasif hale geçirilerek bağlantıyı kabul eden tarafta olduğu belirtilir.

Parametre	Görevi
sockfd	*socket* çağrısından elde edilen dosya betimleyicisi
backlog	Cevap verilmeyi bekleyen kuyruktaki istek sayısı

backlog bekleyen bağlantı isteklerini sınırlamak için kullanılmaktadır.

accept

Gelen bağlantı istekleri *accept* ile kabul edilir. *accept* çağrıldığında bekleyen bir bağlantı isteği yoksa, normal şartlarda, yeni bir bağlantı isteği gelene kadar kod bloklanmaktadır.

> **Not:** *socket* ile bir soket blokeli modda oluşturulmaktadır. Sonrasında soket *fcntl* ile blokesiz moda aşağıdaki gibi geçirilebilir. Bu durumda normalde bloklayan *accept* ve soketten okuma için kullanılan *recv* gibi fonksiyonlar blokesiz olarak çalışacaktır.
>
> fcntl(sockfd, F_SETFL, O_NONBLOCK);

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr , socklen_t *addrlen);
```

Parametre	Görevi
sockfd	*socket* çağrısından elde edilen dosya betimleyicisi
addr	Karşı taraftaki soket adresinin yerleştirileceği alanın adresi
addrlen	addr uzunluğunun tutulduğu adres

accept ile karşı tarafın adres bilgilerine ulaşılabilmektedir. İstekte bulunan soketin adresiyle ilgilenmiyorsanız addr ve addrlen için NULL ve 0 değerlerini geçirebilirsiniz. *addrlen* parametresi *accept* tarafından hem okuma hem de yazma amaçlı olarak kullanılmaktadır. *accept* çağrıldığında *addrlen*, *addr* için ayrılan alanın uzunluğunu gösterirken, *accept* döndüğünde o alana yazılan gerçek byte sayısını göstermektedir.

> **Not:** Karşı tarafın adres bilgilerine daha sonra *getpeername* yardımcı fonksiyonuyla da ulaşılabilir.

accept bir bağlantı isteğini kabul ettiğinde yeni bir soket ile geri dönmekte ve haberleşme bu yeni soket üzerinden yapılmaktadır.

connect

Karşı tarafa bağlanma isteği *connect* ile gönderilir.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr * addr, socklen_t addrlen);
```

Parametre	Görevi
sockfd	*socket* çağrısından elde edilen dosya betimleyicisi
addr	Karşı taraftaki soket adresinin tutulduğu alana ait gösterici
addrlen	Adres bilgilerinin tutulduğu alanın uzunluğu

close

Soket bağlantısı *close* ile sonlandırılır.

```
#include <unistd.h>
int close(int fd);
```

shutdown

shutdown ile, geçirilen son argümanın değerine göre iletişim kanalı tek veya çift yönlü olarak kapatılabilir.

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
```

how	Durum
SHUT_RD	Soket okuma işlemine kapatılır
SHUT_WR	Soket yazma işlemine kapatılır
SHUT_RDWR	Soket okuma ve yazmaya kapatılır

TCP Soketleri

TCP, veri güvenliğinin garanti altına alındığı, bağlantı temelli (connection-oriented) bir iletişim protokülüdür. TCP protokolü *stream* soketleri üzerinden sağlanmaktadır. Bu amaçla *socket* fonksiyonuna ilk argüman olarak SOCK_STREAM sembolik sabitini geçireceğiz.

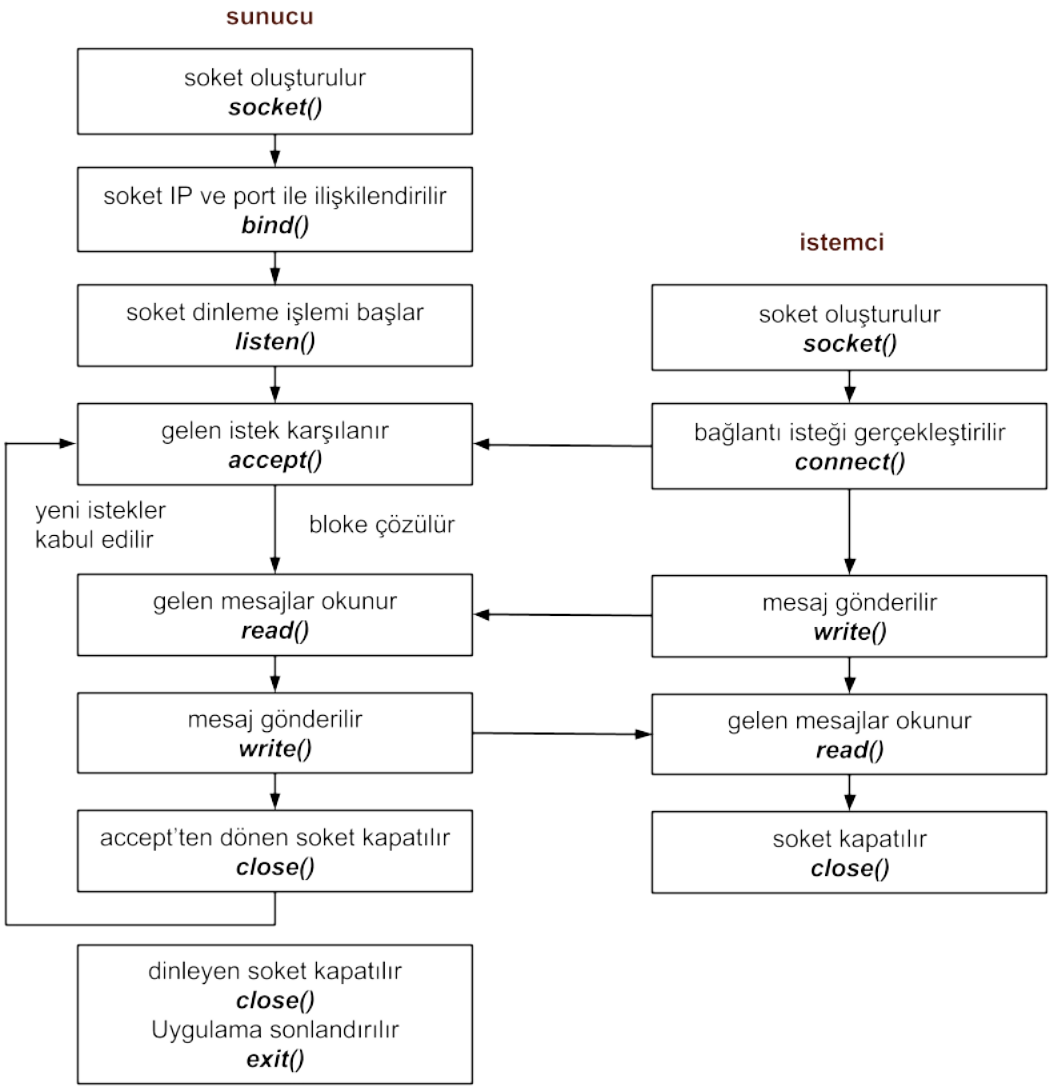
Haberleşecek taraflardan bir tanesi gelecek istekleri dinlemekte diğer taraf ise istekte bulunmaktadır. Dinleyen taraf sunucu (server), istekte bulunan taraf ise istemci (client) olarak isimlendirilmektedir. TCP için sunucu istemci haberleşmesini kabaca telefon sistemine benzetebiliriz. Bu benzetim üzerinden gidecek olursak sırasıyla sunucu ve istemci tarafında gerçekleşen olaylar aşağıdaki gibi olacaktır.

1. *socket* ile haberleşme alt yapısı oluşturulur. Bu durumu telefon cihazının kendisine sahip olmaya benzetebiliriz.
2. *bind* ile soket bir IP adresi ve port ile ilişkilendirilir. Bu durumu, kendi adımıza tahsis edilmiş, bir telefon numarasına sahip olmaya benzetebiliriz.
3. *listen* ile soket üzerinde dinleme işlemi yapılacağı belirtilir. Bu durumu telefonun şebekeye bağlanmasına ve başında cevap vermek üzere beklemeye benzetebiliriz.
4. *accept* ile gelen bağlantı isteği kabul edilir. Bu durumu telefon çaldığında cevap verilmesine benzetebiliriz.
5. *read* ve *write* fonksiyonlarıyla çift yönlü okuma ve yazma işlemleri yapılır. Bu durum karşılıklı konuşmaya denk gelmektedir.
6. *close* ile bağlantı sonlandırılır. Bu durum telefonun kapatılmasına karşılık gelmektedir.

İstekte bulunan tarafta ise geçen olaylar kabaca şöyledir:

1. Sunucu tarafında olduğu gibi ilk olarak *socket* ile haberleşme alt yapısı oluşturulur.
2. *connect* ile karşı tarafa bağlanma isteği gönderilir. Bu durum birini telefon numarasını çevirerek aramaya denk gelmektedir.
3. *read* ve *write* fonksiyonlarıyla çift yönlü okuma ve yazma işlemleri yapılır. Bu durum karşılıklı konuşmaya denk gelmektedir.
4. *close* ile bağlantı sonlandırılır. Bu durum telefonun kapatılmasına karşılık gelmektedir.

Sunucu ve istemci arasında geçen olayları görsel olarak aşağıdaki gibi gösterebiliriz.



Sunucunun yaşam döngüsü boyunca birden çok istemciye yanıt verebilmek için bir döngü içerisinde yeniden *accept* fonksiyonunu çağırdığını görmekteyiz. Genel işleyişi gösterdiğimiz bu örnekte, sunucu aynı anda bir tek istemciye hizmet vermektedir, çoklu istemciyle eş zamanlı çalışma şekline daha sonra bakacağız.

Bu noktada port numaralarıyla ilgili birkaç şey söylemek istiyoruz. IP numarasını bir şirketin telefon numarasına benzetirsek, port numarası konuşmak istediğimiz kişinin dahili numarasına denk gelmektedir. Port numaraları sistemdeki servisleri birbirinden ayırmak için kullanılmaktadır. Port numaraları için adres alanlarında 2 byte yer ayrıldığını hatırlayınız, ilk 1024 tanesi *root* kullanıcısına ayrılmış 65535 port (0. port saklı durumdadır) kullanılabilmektedir. */etc/services* dosyasından sisteminizdeki servislere ve ilgili port numaralarına bakabilirsiniz. İstemci tarafında herhangi bir port numarası belirtmememiz dikkatinizi çekmiş olabilir, işletim sistemi bizim için geçici bir port ayarlayacaktır. Bu tür portlar *ephemeral port* olarak isimlendirilmektedir.

Şimdi basit bir örnek üzerinden istemci sunucu haberleşmesine bakalım. Sunucu ve istemci kodları aşağıdaki gibidir. *client* uygulaması bağlanacağı tarafın IP adresini komut satırından almaktadır. Her iki tarafında bilmesi gereken port numarasını 8080 olarak seçtik. Buradaki test kendi makinamızda gerçekleştireceğimizden istemciye *loopback* adresini (127.0.0.1) geçirdik.

server.c

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 8080
#define BACKLOG 4

int main() {
    /*istemcileri dinleyen soket*/
    int s;
    /*iletişim soketi*/
    int c;
    int b;
    struct sockaddr_in sa;
    FILE *client;
    int count = 0;

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return 1;
    }

    bzero(&sa, sizeof sa);

    sa.sin_family = AF_INET;
    sa.sin_port   = htons(PORT);

    if (INADDR_ANY)
        sa.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(s, (struct sockaddr *)&sa, sizeof sa) < 0) {
        perror("bind");
        return 2;
    }

    listen(s, BACKLOG);

    for (;;) {
        b = sizeof sa;
```

```
        if ((c = accept(s, (struct sockaddr *)&sa, &b)) < 0) {
            perror("accept");
            return 4;
        }

        if ((client = fdopen(c, "w")) == NULL) {
            perror("fdopen");
            return 5;
        }

        fprintf(client, "%d. cevap\n", ++count);

        fclose(client);
    }
}
```

client.c

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 8080

int main(int argc, char **argv) {
  int s;
  int bytes;
  struct sockaddr_in sa;
  char buffer[BUFSIZ+1];

  if (argc != 2) {
      printf("kullanım:  client <IP adresi>\n");
      return 1;
  }

  if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    return 1;
  }

  bzero(&sa, sizeof sa);

  sa.sin_family = AF_INET;
  sa.sin_port = htons(PORT);
  sa.sin_addr.s_addr = inet_addr(argv[1]);
  if (connect(s, (struct sockaddr *)&sa, sizeof sa) < 0) {
    perror("connect");
    close(s);
    return 2;
  }

  while ((bytes = read(s, buffer, BUFSIZ)) > 0)
    write(1, buffer, bytes);

  close(s);
  return 0;
}
```

Her iki kodu derleyelim ve sunucu uygulamasını çalıştıralım.

```
$ gcc -oserver server.c
$ gcc -oclient client.c
$ ./server
```

Bu aşamada *server* uygulamasının *accept* fonksiyonunda bloklandığını görmekteyiz. *client* uygulamasını çalıştırdığımızda ekrana sunucu tarafından gönderilen bir mesaj yazdığını ve ardından sonlandığını görmekteyiz.

```
$ ./client 127.0.0.1
1. cevap
$ ./client 127.0.0.1
2. cevap
$ ./client 127.0.0.1
3. cevap
```

Sunucu ve istemci kodlarında şimdiye kadar görmediğimiz bir sembolik sabit ve birer fonksiyon bulunmakta, kısaca ne olduklarına bakalım.

INADDR_ANY: Bu sembolik sabit, soketi spesifik bir IP adresine bağlamak yerine makinanın tüm internet arayüzlerinin kullanımına olanak sağlar.

fdopen: Sokete ilişkin düşük seviyeli dosya betimleyicisini almakta ve standart C kütüphanesindeki fonksiyonların tanıdığı FILE türünden adrese dönmektedir. Bu sayede soket üzerinde *fprintf* gibi fonksiyonları kullanmak mümkün olmaktadır.

> FILE *fdopen(int fildes, const char* mode);

inet_addr: Sayı ve noktalarla ifade edilen IPv4 adreslerini ağ byte sıralamasına uygun sayı formuna dönüştürür.

Soketlere Özgü G/Ç Fonksiyonları

Soketler üzerinde G/Ç işlemleri için geleneksel *read*, *write* yerine ek özellikler sunan, soket spesifik *recv* ve *send* fonksiyonları da kullanılabilir.

```
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buffer, size_t length, int flags);
ssize_t send(int sockfd, const void *buffer, size_t length, int flags);
```

Fonksiyonların geri dönüş değerleri ve aldıkları ilk 3 argüman *read* ve *write* ile aynıdır. Son argüman *flags* ise G/Ç işleminin davranışını değiştiren bitsel bir maskedir, örnek bazı değerler aşağıdaki gibidir.

Değer	Özellik
MSG_DONTWAIT	Soket, *fcntl* ile blokesiz moda geçirilmek zorunda kalınmaksızın, G/Ç işlemleri blokesiz modda yapılır.
MSG_PEEK	Soketten okuma yaptıktan sonra tamponu silmez. Bu sayede soket üzerinden aynı bilgilere sonraki okumalarda da ulaşılabilir.
MSG_WAITALL	Okuma işleminde, sokette *length* kadar bilgi birikene kadar bloklar.

UDP Soketleri

UDP, TCP'nin aksine, veri güvenliğinin garanti altına alınamadığı mesaj temelli bir iletişim protokolüdür. Mesajlaşacak taraflar arasında öncesinde bir iletişim kanalının oluşturulması gerekmemektedir. UDP protokolü *datagram* soketleri üzerinden sağlanmaktadır. Bu amaçla *socket* fonksiyonuna ilk argüman olarak SOCK_DGRAM sembolik sabitini geçireceğiz.

UDP protokülündeki haberleşmeyi posta sistemine benzetebiliriz. Bu benzetim üzerinden gidecek olursak sırasıyla sunucu ve istemci tarafında gerçekleşen olaylar aşağıdaki gibi olacaktır.

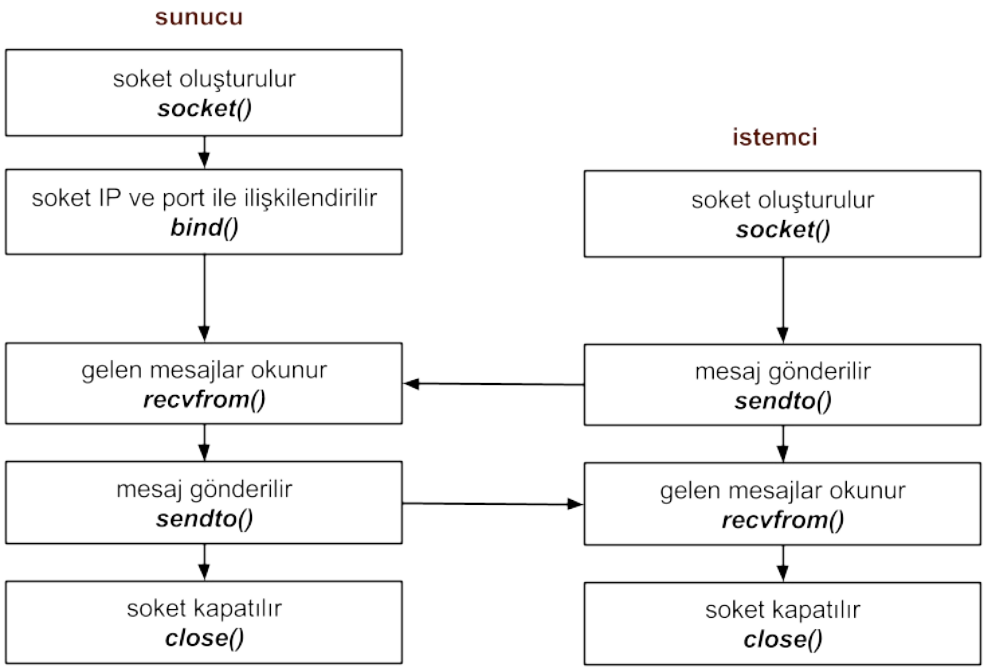
1. *socket* ile haberleşme alt yapısı oluşturulur. Bu durumu fiziksel bir posta kutusuna sahip olmaya benzetebiliriz.
2. *bind* ile soket bir IP adresi ve port ile ilişkilendirilir. Bu durumu, evimizin adresinin posta işletmesi tarafından biliniyor olmasına benzetebiliriz.
3. *recvfrom* ile gelen mesaj, bloklamalı modda, okunur. Bu durumu posta kutumuzun başında postacıyı beklemeye ve mektup ulaştığında okumaya benzetebiliriz.
4. *sendto* ile karşı tarafa mesaj gönderilir. Bu durumu alıcının adresini yazdığımız bir mektubu postalamaya benzetebiliriz.
5. *close* ile soket sonlandırılır. Artık bir posta kutunuz olmadığından mektup alamayacaksınız.

Tipik bir istemci tarafında gerçekleşen olaylar ise *bind* işlemi hariç aynıdır.

1. Sunucu tarafında olduğu gibi ilk olarak *socket* ile haberleşme alt yapısı oluşturulur.
2. *recvfrom* ile gelen mesaj, bloklamalı modda, okunur.
3. *sendto* ile karşı tarafa mesaj gönderilir.
4. *close* ile soket sonlandırılır.

Gerçekte istemci tarafında da *bind* yapılabilir. Bu durumu gönderdiğiniz bir mektubun üzerine kendi adresinizi yazmanıza benzetebiliriz. Bu durumda alıcı taraf başka bir ön bilgiye ihtiyaç duymadan size cevap verebilir. Ayrıca sunucu tarafta *accept* kullanılabilmektedir, bu duruma değineceğiz.

UDP istemci sunucu haberleşmesini en basit haliyle görsel olarak aşağıdaki gibi gösterebiliriz.



UDP'de veri alışverişi, genellikle, *recvfrom* ve *sendto* isimli özel fonksiyonlarla yapılır. Bu fonksiyonlara daha yakından bakalım.

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,
                 const struct sockaddr *dest_addr, socklen_t addrlen);
```

Her iki fonksiyonun da geri dönüş değeri ve ilk 3 argümanı *read* ve *write* fonksiyonlarıyla aynıdır. *flag* argümanı soket spesifik G/Ç özelliklerini değiştirmek için kullanılmaktadır. Diğer iki argüman ise karşıdaki tarafın adresini belirtmek veya elde etmek için kullanılmaktadır.

recvfrom fonksiyonundaki *src_addr* uzaktaki soketin adresini elde etmek için kullanılmaktadır. Karşı taraf *bind* işlemini yapmışsa gönderen tarafın adresine bu şekilde erişilebilir. Bu argümanların kullanımı *accept* fonksiyonundaki kullanıma benzemektedir. Gönderen tarafın adresiyle ilgilenilmiyorsa bu argümanlara NULL değeri geçilebilir.

sento fonksiyonundaki *dest_addr* ve *addrlen* parameterleri karşı tarafın adresini belirtmek için kullanılmaktadır. Bu argümanların kullanımı *connect* fonksiyonundaki kullanıma benzemektedir.

Datagram soketleri bağlantısız olmalarına karşın *connect* fonksiyonu bu soketler için de kullanılabilir. Bu durumda bağlantılı datagram soketleri (connected datagram socket) oluşturulmuş olur. *connect* işleminden sonra *sendto* fonksiyonundaki *dest_addr* ve *addrlen*

argümanlarına gerek kalmamaktadır, bu sebeple *sendto* yerine *write* fonksiyonu kullanılabilir. İşletim sistemi karşıdaki soketin adresini bizim için saklamaktadır. Ayrıca bu şekilde yalnız bağlantı yapılan taraftan gelen paketler ulaşmaktadır. Bu durumu mektubumuzun gideceği adresi mektuba yazmak yerine postacımıza söylememize benzetebiliriz. Ayrıca postacımız ilgilendiğimiz dışındaki mektupları bize ulaştırmayacaktır.

Basit bir örnek üzerinden UDP istemci sunucu haberleşmesine bakalım. Sunucu ve istemci kodları aşağıdaki gibidir. *client* uygulaması bağlanacağı tarafın IP adresini ve göndereceği mesajı komut satırından almaktadır. Sunucu istemciden gelen mesajı geri göndermekte ve sonrasında sonlanmaktadır. Her iki tarafında bilmesi gereken port numarasını yine 8080 olarak seçtik. Testi yine kendi makinamızda gerçekleştireceğimizden istemciye *loopback* adresini (127.0.0.1) geçireceğiz.

server.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>

#define BUF_SIZE 1024
#define PORT 8080

int main() {
    char buf[BUF_SIZE];
    struct sockaddr_in self, other;
    int len = sizeof(struct sockaddr_in);
    int n, s, port;

    if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1) {
        perror("socket");
        return 1;
    }

    memset((char *) &self, 0, sizeof(struct sockaddr_in));
    self.sin_family = AF_INET;
    self.sin_port = htons(PORT);
    self.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *) &self, sizeof(self)) == -1) {
        perror("bind");
        return 1;
    }

    while ((n = recvfrom(s, buf, BUF_SIZE, 0, (struct sockaddr *) &other, &len)) != -1
) {
        printf("Received from %s:%d: ", inet_ntoa(other.sin_addr), ntohs(other.sin_por
t));
        fflush(stdout);
        write(1, buf, n);
        write(1, "\n", 1);

        sendto(s, buf, n, 0, (struct sockaddr *) &other, len);
    }

    close(s);
    return 0;
}
```

client.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

#define BUF_SIZE 1024
#define PORT 8080

int main(int argc, char *argv[]) {
    struct sockaddr_in server;
    int len = sizeof(struct sockaddr_in);
    char buf[BUF_SIZE];
    int n, s;

    if (argc != 3) {
        printf("kullanım: client <IP adresi> <mesaj>\n");
        return 1;
    }

    if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1) {
        perror("socket");
        return 1;
    }

    memset((char *) &server, 0, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_port = htons(PORT);
    server.sin_addr.s_addr = inet_addr(argv[1]);

    if (sendto(s, argv[2], strlen(argv[2]), 0, (struct sockaddr *) &server, len) == -1
) {
        perror("sendto()");
        return 1;
    }

    n = recvfrom(s, buf, BUF_SIZE, 0, (struct sockaddr *) &server, &len);
    buf[n] = 0;
    printf("Received from %s:%d: ", inet_ntoa(server.sin_addr), ntohs(server.sin_port)
);
    fflush(stdout);
    write(1, buf, n);
    write(1, "\n", 1);

    close(s);
    return 0;
}
```

Kodları aşağıdaki gibi derleyebiliriz.

```
$ gcc -oserver server.c
$ gcc -oclient client.c
```

Sırasıyla sunucu ve istemci taraflarını aşağıdaki gibi çalıştırabiliriz.

```
$ ./server
```

```
$ ./client 127.0.0.1 "Mary had a little lamb"
Received from 127.0.0.1:8080: Mary had a little lamb

$ ./client 127.0.0.1 "All the place you have been trying..."
Received from 127.0.0.1:8080: All the place you have been trying...
```

Sunucu uygulamasının çıktısı ise aşağıdaki gibi olmaktadır.

```
$ ./server
Received from 127.0.0.1:50593: Mary had a little lamb
Received from 127.0.0.1:47827: All the place you have been trying...
```

İstemci tarafa ilişkin 50593 ve 47827 olarak gösterilen port numaraları daha önce bahsettiğimiz işletim sistemi tarafından atanan geçici port numaralarıdır.

UNIX Soketleri

UNIX alan soketleri (UNIX domain socket), Internet soketlerinin aksine, yalnız aynı makina üzerindeki proseslerin haberleşmesine imkan tanımaktadır. Internet soketlerinde olduğu gibi TCP ve UDP üzerinden haberleşme sağlanabilir.

UNIX alan soketi oluşturmak için *socket* fonksiyonuna ilk argüman olarak AF_UNIX sembolik sabitini geçirmeliyiz. Alan soketleri dosya sisteminde bir giriş oluşturmakta ve buna ilişkin bir yol ifadesi (pathname) almaktadır. Alan soketlerinin adresleri *sockaddr_un* yapı türünde saklanmaktadır.

```
struct sockaddr_un {
    sa_family_t sun_family; // AF_UNIX
    char sun_path[108]; // Null karakter ile sonlandırılmış soket yol ismi
};
```

Soket *bind* edildiğinde dosya sisteminde sokete ilişkin bir giriş oluşturulacaktır. Aşağıdaki örneği çalıştırıp bu durumu gözleyebiliriz.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define UNIXSOCKNAME "/tmp/unixsock"

int main() {
    int sfd;
    struct sockaddr_un addr;
    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        return 1;

    memset(&addr, 0, sizeof(struct sockaddr_un));

    addr.sun_family = AF_UNIX;

    strncpy(addr.sun_path, UNIXSOCKNAME, sizeof(addr.sun_path) - 1);
    if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
        return 1;

    return 0;
}
```

```
$ gcc -oserver server.c

$ ./server

$ ls -l /tmp/unixsock
srwxrwxr-x 1 serkan serkan 0 Mar 25 17:24 /tmp/unixsock
```

ls çıktısının başındaki *s* karakteri dosyanın bir sokete ilişkin olduğunu göstermektedir.

Basit bir örnek üzerinden, *stream* soketlerini kullanarak, sunucu ve istemci haberleşmesine bakalım. İstemci klavyeden girilen yazıyı sunucuya göndermektedir.

server.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define UNIXSOCKNAME "/tmp/unixsock"
#define BUF_SIZE 100
#define BACKLOG 5

int main() {
    struct sockaddr_un addr;
    int sfd, cfd;
    ssize_t numRead;
    char buf[BUF_SIZE];
    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        return 1;

    unlink(UNIXSOCKNAME);

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, UNIXSOCKNAME, sizeof(addr.sun_path) - 1);

    if (bind(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
        return 1;

    if (listen(sfd, BACKLOG) == -1)
        return 1;

    for (;;) {
        cfd = accept(sfd, NULL, NULL);
        if (cfd == -1)
            return 1;

        while ((numRead = read(cfd, buf, BUF_SIZE)) > 0) {
            if (write(1, buf, numRead) != numRead)
                return 1;
        }
        close(cfd);
    }
    return 0;
}
```

client.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define UNIXSOCKNAME "/tmp/unixsock"
#define BUF_SIZE 100

int main()
{
    struct sockaddr_un addr;
    int sfd;
    ssize_t numRead;
    char buf[BUF_SIZE];
    sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sfd == -1)
        return 1;

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, UNIXSOCKNAME, sizeof(addr.sun_path) - 1);

    if (connect(sfd, (struct sockaddr *) &addr, sizeof(struct sockaddr_un)) == -1)
        return 1;

    while ((numRead = read(0, buf, BUF_SIZE)) > 0) {
        if (write(sfd, buf, numRead) != numRead)
            return 1;
    }
}
```

```
$ gcc -oserver server.c
$ gcc -oclient client.c
```

Birden Çok İstemciyle Çalışma

Çoğu durumda, bir sunucunun aynı anda birden çok istemciye hizmet vermesi beklenmektedir. Örneğin, *ssh* sunucusu üzerinden, çok sayıda uzak makina kullanıcısı sisteme giriş yapabilmektedir. ssh sunucusu, yeni bağlantıları kabul etmekte, aynı zamanda istemcilerden gelen komutları çalıştırarak sonuçları geri dönmektedir.

Belli bir anda, tek bir istemciyle çalışan ve istemcileri sırayla kabul eden sunucular tekrarlamalı sunucu (iterative server) olarak adlandırılırken, birden çok istemciyle aynı anda çalışabilen sunucular eş zamanlı sunucu (concurrent server) olarak adlandırılmaktadır. Buradaki eş zamanlılık tüm istemcilerin makul bir süre içinde sunucudan hizmet alabilmesini ifade etmektedir, hiçbir istemci diğerinin hizmet almasını engellememektedir.

Daha önce, *socket* fonksiyonu ile soketlerin blokeli modda oluşturulduğunu ve sonrasında *fcntl* ile blokesiz moda geçirilebildiklerini söylemiştik. Blokeli modda çalışılması durumunda yeni bağlantıları kabul eden *accept* ve soketten okuma yapan *read, recv* gibi fonksiyonlar program kodunu bloklayacaktır. Dolayısıyla, birden çok istemciyle eş zamanlı çalışmak mümkün olmayacaktır. Soketlerin blokesiz moda geçirilmesi durumunda ise fonksiyonlar bloklamayacak ve bir döngü içerisinde isteklerden haberdar olmak için sürekli bir yoklama (polling) işlemi yapılacaktır. Bu tür döngüler meşgul döngü (busy loop) olarak adlandırılmakta ve işlemci zamanının gereksiz yere harcanmasına neden olmaktadır.

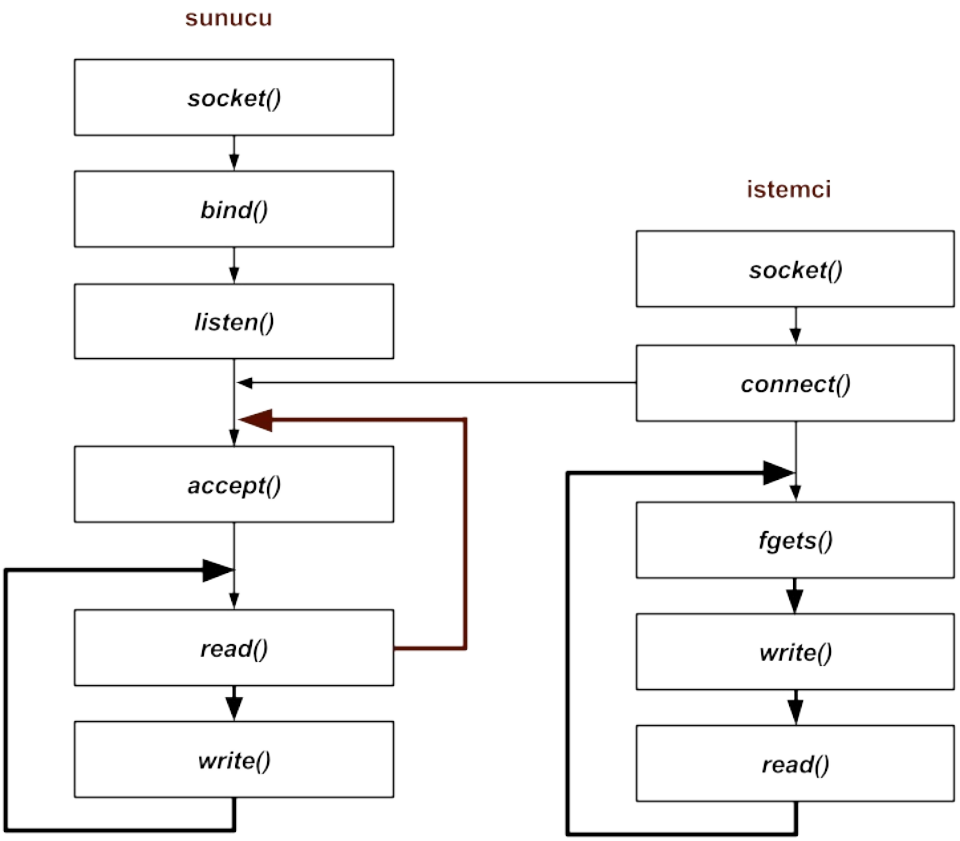
Özetleyecek olursak, blokeli çalışma durumunda beklediğimiz olay gerçekleşene kadar prosesimiz uyutulup, olay gerçekleştiğinde uyandırılarak, işlemci zamanı etkin bir şekilde kullanılmakta fakat eş zamanlı çalışma mümkün olmamaktadır. Blokesiz modda ise eş zamanlı çalışma mümkün olmasına karşın işlemci zamanı boş yere harcanmaktadır. Bu durumda temel olarak aşağıdaki yöntemler kullanılmaktadır.

- Her bir istemci için sucunu tarafında yeni bir *thread* veya *proses* yaratılır.
- *select* ve *poll* sistem çağrıları ile soketlere ilişkin dosya betimleyicileri izlenerek (*I/O multiplexing*) *accept* ve *read* işlemleri gerektiğinde yapılır.

Bu bölümdeki incelemelerimizi, istemciden gelen mesajları istemciye geri gönderen basit bir TCP sunucu (echo server) üzerinden yapacağız. İlk olarak istemcilerle sıralı çalışma şeklini sonrasında ise, yeni prosesler oluşturarak ve *select* çağrısıyla, eş zamanlı çalışma şekillerini inceleyeceğiz.

İstemcilerle Sıralı Çalışma

Bu çalışma şeklinde istemciler sunucudan sırayla hizmet alabilmektedir. Bir istemci sunucuya bağlandıktan sonra, bağlantı sonlandırılana kadar, yeni bir istemci bağlanamamaktadır. Bu durumu aşağıdaki şekille gösterebiliriz.



Sunucu istemciyi kabul ettikten sonra blokeli modda *read* işleminde beklemektedir. Sunucudan mesaj geldiğinde mesaj okunmakta, okunan mesaj istemciye geri gönderilmekte ardından yeniden *read* ile beklenmektedir. İstemci bağlantıyı sonlandırdığında sunucu tekrar *accept* fonksiyonunu çağırarak yeni gelecek bağlantıyı bekleyecektir. Örnek istemci ve sunucu kodları aşağıdaki gibidir.

client.c :

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 8080
#define BUF_SIZE 100

int main(int argc, char **argv) {
    int sfd;
    int bytes;
    struct sockaddr_in sa;
```

```
    char buf[BUF_SIZE];
    ssize_t numRead;

    if (argc != 2) {
        printf("usage:  client <IP adresi>\n");
        return 1;
    }

    if ((sfd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return 1;
    }

    bzero(&sa, sizeof sa);

    sa.sin_family = AF_INET;
    sa.sin_port = htons(PORT);
    sa.sin_addr.s_addr = inet_addr(argv[1]);

    if (connect(sfd, (struct sockaddr *)&sa, sizeof sa) < 0) {
        perror("connect");
        close(sfd);
        return 2;
    }

    for (;;) {
        printf("Mesaj: ");
        fgets(buf, BUF_SIZE, stdin);

        if (write(sfd, buf, strlen(buf)) < 0) {
            return 1;
        }

           if(numRead = read(sfd, buf, BUF_SIZE) < 0) {
            break;
        }

        write(1, "Yanıt: ", 8);
        puts(buf);
    }

    close(sfd);
    return 0;
}
```

server.c :

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

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 8080
#define BUF_SIZE 100
#define BACKLOG 4

char buf[BUF_SIZE];
int numRead;

void echo(int cfd) {
    while ((numRead = read(cfd, buf, BUF_SIZE)) > 0) {
        if (write(cfd, buf, numRead) != numRead)
            return;
    }
}

int main() {
    int sfd;
    int cfd;
    int b;
    struct sockaddr_in sa;
    FILE *client;
    fd_set readfds, masterfds;
    int nready;
    int i;
    int maxfd;

    if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return 1;
    }

    bzero(&sa, sizeof sa);

    sa.sin_family = AF_INET;
    sa.sin_port   = htons(PORT);

    if (INADDR_ANY)
        sa.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sfd, (struct sockaddr *)&sa, sizeof sa) < 0) {
        perror("bind");
        return 2;
    }

    if (listen(sfd, BACKLOG) == -1)
        return 1;

    for (;;) {
        cfd = accept(sfd, NULL, NULL);
        if (cfd == -1)
```

```
            return 1;
        echo(cfd);
        close(cfd);
    }
}
```

Sunucu, istemci bağlantıyı sonlandırana kadar *echo* fonksiyonunda bloklanmaktadır. İstemci bağlantıyı sonlandırdığında *read* 0 değerine dönmekte ve echo fonksiyonundan çıkılmaktadır. Örneğimiz üzerinden bu durumu inceleyelim.

Sunucu ve istemci uygulamalarımız derleyelim.

```
$ gcc -oserver server.c
$ gcc -oclient client.c
```

Sunucuyu, ardından istemciyi çalıştırıp bir mesaj gönderelim.

```
$ ./server

$ ./client 127.0.0.1
Mesaj: istemci konuşuyor
Yanıt: istemci konuşuyor

Mesaj:
```

İstemcinin mesajının kendisine geri gönderildiğini görüyoruz. Şimdi başka bir terminale geçerek ikinci bir istemci çalıştıralım ve bir mesaj göndermeyi deneyelim.

```
$ ./client 127.0.0.1
Mesaj: ikinci istemci konuşuyor
```

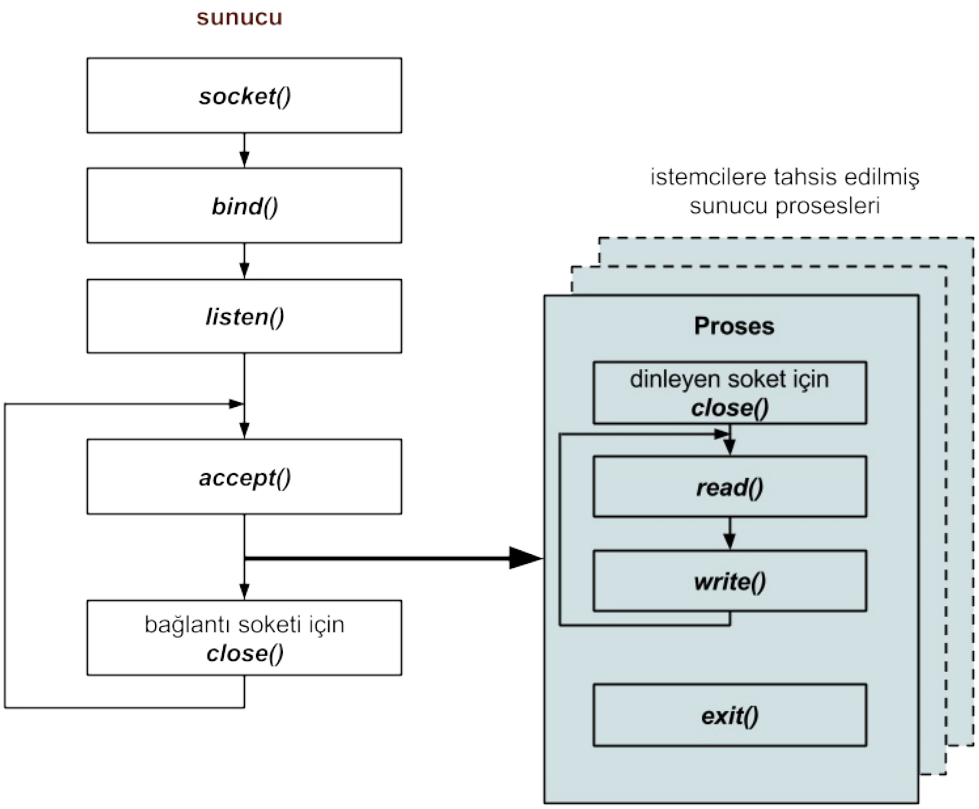
İkinci istemci, bağlantı isteği sunucu tarafından kabul edilmediğinden, *write* işleminde bloklanmaktadır. İlk istemciyi sonlandırdığımızda, ikinci istemcinin bağlantı isteği sunucu tarafından kabul edilecek ve ikinci istemci sunucu ile haberleşebilecektir.

İstemcilerle Eş Zamanlı Çalışma

Birden çok istemciyle eş zamanlı çalışmanın ilk olarak proseslerle sonrasında ise *select* sistem çağrısıyla nasıl gerçeklendiğine bakalım.

Yeni Proses Oluşturma

Bu yöntemde her bir istemci için yeni bir proses oluşturulmakta ve haberleşme, blokeli modda, bu yeni proses üzerinden sağlanmaktadır. Bu durumu aşağıdaki şekil ile gösterebiliriz.



Bu durumda, kodun diğer kısımlarına dokunmaksızın, sunucunun yaşam döngüsünü oluşturan *for* bloğunu aşağıdaki gibi değiştireceğiz.

```
for (;;) {
    cfd = accept(sfd, NULL, NULL);
    if (cfd == -1)
        return 1;
    if (fork() == 0) {
        /*istemciye tahsis edilmiş alt proses kodu*/
        close(sfd);
        echo(cfd);
        close(cfd);
        exit(0);
    }
    close(cfd);
}
```

Sunucu kodu ilk çalışmaya başladığında var olan proses (üst proses) temel olarak yeni bağlantıları kabul etmek ve yeni alt prosesler oluşturmaktan sorumludur. Haberleşme bu alt prosesler üzerinden sağlanmakta ve bu sayede çok sayıda istemciyle paralel olarak

çalışılabilmektedir. Üst proses ve bir alt proseste geçen olayları sırasıyla aşağıdaki gibi sıralayabiliriz.

Üst proses:

- Blokeli modda yeni bir bağlatıyı bekle
- Bağlantı isteği olması durumunda kabul et
- Yeni bir alt proses oluştur
- Alt prosesin haberleşme için kullanacağı soketi kapat
- Tekrardan bağlantı isteklerini bekle

Alt proses:

- Üst proses tarafından yeni bağlantıları dinlemek için kullanılan soketi kapat
- Blokeli modda okuma ve yazma işlemlerini yap
- Sunucu bağlantıyı kapatmışsa haberleşme soketini kapat, ardından prosesi sonlandır

> **Not:** Dosya betimleyicilerinin üst prosesten alt prosese miras kaldığını hatırlayınız.

Bu yöntemde aslında her bir istemci için bir iteratif sunucu oluşturmuş olduk. Farklı terminaller üzerinde istemciler çalıştırarak, beraber çalışabildiklerini gözleyebilirsiniz.

```
$ ./server
```

```
$ ./client 127.0.0.1
Mesaj: 1. istemci konusuyor
Yanıt: 1. istemci konusuyor
```

```
$ ./client 127.0.0.1
Mesaj: 2. istemci konuşuyor
Yanıt: 2. istemci konuşuyor
```

Son olarak benzer çalışma modelinin *select* ile nasıl yapılabildiğine bakalım.

select

select ile, yeni prosesler veya thread'ler oluşturmaksızın, tek bir proses üzerinden birçok istemciye hizmet vermek mümkündür. *select* ile dosya betimleyicilerin durumu izlenmektedir. *select* blokeli modda çalışmakta, izlenen dosya betimleyicilerin durumdalarında bir değişiklik olması durumunda bloke çözülmektedir. Ayrıca blokenin çözülmesi için bir zaman aşımı süresi de belirlenebilir. *select* fonksiyonunun prototipi ve aldığı parametreler aşağıdaki gibidir.

```
#include <sys/time.h>
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

Parametre	Görevi
nfds	İzlenecek maksimum dosya betimleyici sayısı (en yüksek betimleyici numarası + 1)
readfds	Okuma amaçlı izlenecek dosya betimleyici kümesi
writefds	Yazma amaçlı izlenecek dosya betimleyici kümesi
exceptfds	Hata durumları izlenecek dosya betimleyici kümesi
timeout	Blokenin çözüleceği zaman aşımı süresi

select aşağıdaki değerlerin biriyle dönmektedir.

Geri Dönüş Değeri	Anlamı
0 dışı değer	Hazır betimleyici sayısı
0	Zaman aşımı
-1	Hata durumu, *errno* değişkenine hatanın nedeni yazılır

select çağrısını, soketleri gerektiğinde bloke olmadan okuyabilmek için aşağıdaki biçimde kullanacak ve ilgilenmediğimiz parameterlere NULL değerini geçireceğiz.

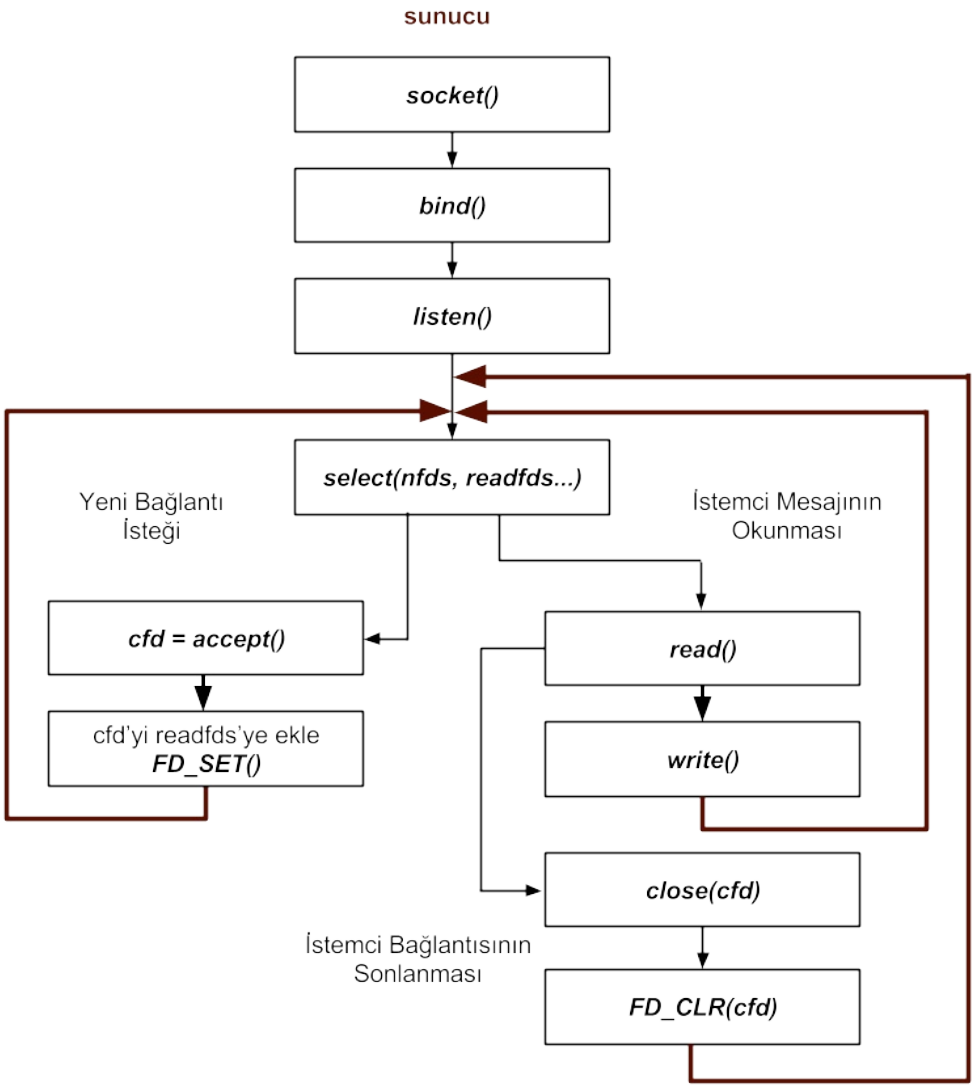
```
select (nfds, readfds, NULL, NULL, NULL)
```

readfds, *fd_set* türünde bir adres olup, okuma amaçlı izlenecek dosya betimleyici kümesini göstermektedir. *fd_set* türü, her bir biti bir dosya betimleyicisini temsil etmek üzere, 1024 (proses başına açılabilecek maksimum dosya sayısı) bitlik bir alanı göstermektedir. *readfds* üzerinden konuşacak olursak, ilk önce tüm bitleri sıfıra çekmeli, ardından izlemek istediğimiz dosya betimleyicilere karşılık gelen bitleri 1 değerine çekmeliyiz. *select*, *readfds* argümanını hem okuma hem de yazma amaçlı (value-result argument) olarak kullanmaktadır. *readfds*, fonksiyon çağrılmadan önce ilgilendiğimiz dosya betimleyicilerini göstermesine karşın, eğer hata durumu veya zaman aşımı oluşmamışsa, fonksiyon döndüğünde okumaya hazır betimleyicileri göstermektedir. *select* ile daha kolay çalışabilmek için aşağıdaki makrolar tanımlanmıştır.

> **Not:** *select*'e ilk argüman olarak geçirdiğimiz, *nfds* değeri *select* fonksiyonunun daha hızlı çalışması için kullanılmaktadır. Bu sayede *select* 1024 bitten oluşan kümelerin (örneğin *readfds*) tamamı üzerinde değil, ilk *nfds* biti üzerinde işlem yapacaktır.

Makro	Görevi
void FD_ZERO(fd_set *fdset)	fdset içindeki tüm bitleri sıfırlar
void FD_SET(int fd, fd_set *fdset)	fd'ye karşılık gelen bitin değerini 1 yapar
void FD_CLR(int fd, fd_set *fdset)	fd'ye karşılık gelen bitin değerini 0 yapar
int FD_ISSET(int fd, fd_set *fdset	fdset içinde fd'ye karşılık gelen bitin değerinin 1 olup olmadığını döner

Çoklu istemciye yanıt veren sunucuya ilişkin çalışma şeklini gösteren şekil ve örnek kod sırasıyla aşağıdaki gibidir.




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

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 8080
#define BUF_SIZE 100
#define BACKLOG 4

char buf[BUF_SIZE];
int numRead;

int main() {
    int sfd;
    int cfd;
    int b;
    struct sockaddr_in sa;
    FILE *client;
    fd_set readfds, masterfds;
    int nready;
    int i;
    int maxfd;

    if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return 1;
    }

    bzero(&sa, sizeof sa);

    sa.sin_family = AF_INET;
    sa.sin_port   = htons(PORT);

    if (INADDR_ANY)
        sa.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(sfd, (struct sockaddr *)&sa, sizeof sa) < 0) {
        perror("bind");
        return 2;
    }

    if (listen(sfd, BACKLOG) == -1)
        return 1;

    FD_ZERO(&masterfds);
    FD_ZERO(&readfds);
    FD_SET(sfd, &masterfds);
    maxfd = sfd;

    for (;;) {
        readfds = masterfds;
        nready = select(maxfd + 1, &readfds, NULL, NULL, NULL);
        for (i = 0; i <= maxfd && nready > 0; ++i) {
            if (!FD_ISSET(i, &readfds))
```

```
                continue;
            --nready;
            /*bağlantı isteği*/
            if (i == sfd) {
                cfd = accept(sfd, NULL, NULL);
                if (cfd == -1)
                    return 1;
                FD_SET(cfd, &masterfds);
                if (maxfd < cfd)
                    maxfd = cfd;
            }
            /*data*/
            else {
                numRead = read(i, buf, BUF_SIZE);
                if (numRead > 0) {
                    write(i, buf, numRead);
                }
                else {
                    close(i);
                    FD_CLR(i, &masterfds);
                }
            }
        }
    }
}
```

İlk bakışta genel akışı gösteren şekil ve *for* döngüsünü oluşturan kod karışık gelebilir, ilgili kod bölümlerini bloklar şeklinde inceleyelim.

Daha önce *select*'in okuduğu değerleri değiştirdiğinden bahsetmiştik, bu amaçla *fd_set* türünden iki adet değişken tutuyoruz.

- masterfds: İzlenecek betimleyici kümesi
- readfds: masterfds'in kopyası alınarak *select*'e geçirilen küme

readfds, *select* döndüğünde sonuç değeri gösterebildiğinden, izlenecek fd kümesini *masterfds* isimli ayrı bir değişkende saklıyoruz.

```
FD_ZERO(&masterfds);
FD_ZERO(&readfds);
FD_SET(sfd, &masterfds);
maxfd = sfd;
```

masterfds ve *readfds* değişkenlerini sıfırlıyor ve bağlantı isteklerinin dinlendiği *sfd*'yi izlenecek fd kümesine ekliyoruz. Son olarak, en yüksek fd değerini tutan *maxfd*'ye ilk değerini veriyoruz.

```
for (;;) {
    readfds = masterfds;
    nready = select(maxfd + 1, &readfds, NULL, NULL, NULL);
    for (i = 0; i <= maxfd && nready > 0; ++i) {
        if (!FD_ISSET(i, &readfds))
            continue;
        --nready;
        ...
    }
}
```

İlgilendiğimiz fd sayısını (maxfd+1) ve izlenecek fd seti *masterfds*'e eşitlediğimiz *readfds* adresini *select*'e geçiriyoruz. *select* döndüğünde *maxfd*'ye kadar olan aralıkta, *nready* tane bitin değeri değişmiş olacağından *for* döngüsünün şart ifadesini bu koşulu sağlayacak şekilde yazıyoruz. Bu aşamadan sonra *readfds* içinde değeri 1 olan bitleri bulmalıyız. FD_ISSET ile değeri 1 olan bir bite ulaşana kadar döngüyü devam ettiriyoruz. Değeri 1 olan bir bite ulaştığımızda bu bitin temsil ettiği sokete ulaştığımız için *nready* değerini 1 eksiltiyoruz.

Şimdi *for* içindeki sıradaki kod bloğuna bakalım.

```
if (i == sfd) {
    cfd = accept(sfd, NULL, NULL);
    if (cfd == -1)
        return 1;
    FD_SET(cfd, &masterfds);
    if (maxfd < cfd)
        maxfd = cfd;
}
```

Durumu değişen fd'nin yeni bağlantıları dinlediğimiz *sfd* olması durumunda, yeni bir bağlantı isteği geldiğini anlıyoruz. Sırasıyla, *accept* ile yeni bir haberleşme soketi oluşturuyor, bu sokete ilişkin fd'yi *select*'e geçirmek üzere *masterfds* değişkenine ekliyor ve *maxfd* değerini güncelliyoruz. Bu kod bloğu sayesine *select* yeni bağlanan istemciden gelen mesajlardan haberdar olabilecektir.

Şimdi *for* içindeki son kod bloğuna bakalım.

```
else {
    numRead = read(i, buf, BUF_SIZE);
    if (numRead > 0) {
        write(i, buf, numRead);
    }
    else {
        close(i);
        FD_CLR(i, &masterfds);
    }
}
```

Durumu değişen fd'nin istemci bağlantılarını dinlediğimiz sokete ilişkin olmaması durumunda, istemciden bir mesaj geldiğini anlıyoruz. fd okumaya hazır olduğundan bu aşamada *read* ile bloklanmadan okuma yapabiliriz. *read* fonksiyonunun sıfırdan büyük bir değer dönmesi durumunda istemciden gelen mesajı tekrar okuma yaptığımız soket üzerinden istemciye gönderiyoruz. İstemcinin bağlantıyı kapatması durumunda ise *select* bu durumdan haberdar olacak, bloke çözülecek ve *read* 0 değerine dönecektir. Bu durumda *close* ile bu haberleşme kanalını kapatıyor ve FD_CLR ile bu sokete ilişkin fd'yi izleme kümesinden çıkartıyoruz.

Özetleyecek olursak, şekilden de görüldüğü gibi, *select* döndüğünde akış ikiye ayrılmaktadır. Yeni bir istemci bağlanmak istediğinde, yeni bir bağlantı soketi oluşturulmakta ve sokete ilişkin fd izleme kümesine eklenerek akış yeniden *select*'e yönlendirilmektedir. Durumu değişen fd'nin bağlantıların dinlendiği sokete ilişkin olmaması durumunda ise istemcilerden mesaj geldiği anlaşılmakta ve bloklanmadan okuma işlemi yapılıp akış tekrar *select*'e yönlendirilmektedir. Bu sayede tek bir proses üzerinden birçok istemciyle çalışılabilmektedir. Farklı terminallerde birden çok istemci çalıştırarak bu durumu gözleyebilirsiniz.

Timer Kullanımı

Timer mekanizmaları, uygulamamızın önceden belirlediğimiz bir süre tamamlandığında işletim sistemi çekirdeği tarafından haberdar edilmesine imkan verir.

Dolayısıyla timer kullanımının ilk şartı, uygulamamızı haberdar etmesini istediğimiz zaman dilimini çekirdeğe bildirmek, zamanı geldiğinde de çekirdeğin uygulamamıza sinyaller yoluyla gönderdiği bilgiyi işleme alacak *callback* fonksiyonunu hazırlamaktır.

Unix tabanlı sistemlerde timer mekanizmaları geçmişten günümüze ihtiyaçlar doğrultusunda evrim geçirerek gelişmiştir. Bunula birlikte halen ilk versiyonu olan `alarm()` mekanizmasının da kullanılmaya devam etmekte olduğunu görmekteyiz.

Farklı problem tiplerine özgü olarak bir timer mekanizmasını bir diğerine tercih etme durumu söz konusu olmaktadır. Sırasıyla basitten karmaşığa doğru timer mekanizmalarını inceleyip, aralarındaki farkları ve kullanım senaryolarını örneklendirmeye çalışacağız.

Basit Timer Yapıları

Geleneksel Yaklaşım: `alarm()`

Timer kullanımının en basit yolu, prototipi aşağıda verilmiş olan `alarm` fonksiyonunu kullanmaktır:

```
unsigned int alarm (unsigned int seconds);
```

Bu yöntemle ancak saniye çözünürlüğünde zaman belirtmek mümkündür. Zaman sona erdiğinde işletim sistemi uygulamaya `SIGALRM` sinyalini gönderir. Zamanlayıcının dolduğunu uygulamada işleyebilmek için bu sinyali işleyecek callback fonksiyonu da tanımlanmış olmalıdır.

Aşağıda 1 saniyelik zamanlayıcı kurulması ve sinyal işleyici fonksiyonun tanımlanması örneklenmiştir:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>

void timer_callback (int signum)
{
    time_t now = time(NULL);
    printf ("Signal %d caught on: %li\n", signum, now);
}

int main ()
{
    signal(SIGALRM, timer_callback);
    alarm(1);
    sleep(3);
    return 0;
}
```

Bu şekilde kurulmuş olan bir timer her 1 saniyede uygulamaya sinyal gönderecektir. Eğer belirli bir eylem olduğunda timer aralığını örnek olarak 5 saniyeye çıkarmak isterseniz, `alarm(5)` şeklinde fonksiyonu yeniden kullanmanız yeterli olacaktır.

Timer'ı durdurmak isterseniz, `alarm(0)` şeklinde 0 parametresi ile kullanmanız yeterli olacaktır. Bu şekilde durdurulmuş bir timer, 0'dan büyük bir değer verilerek fonksiyon tekrar çağrılırsa yeniden başlatılacaktır.

Zaman dolduğunda kullandığınız timer periyodik olarak tekrar baştan başlamayacaktır. Örnek olarak her 1 saniyede callback fonksiyonunuzun çağrılmasını istiyorsanız, timer dolduğunda tekrar `alarm(1)` şeklinde mekanizmayı başlatmanız gerekmektedir.

Alarm fonksiyonu üzerinden örneklediğimiz bu yapı ancak çok basit senaryolarda tercih edilmelidir (bu yüzden sinyal yakalama kısmını da Sinyaller bölümünde işlediğimiz `sigaction()` yöntemiyle değil `signal()` fonksiyonuyla yaptık). Kullanım kolaylığına rağmen bu yöntemin başlıca dezavantajları:

- aynı anda sadece tek bir timer olması
- periyodik timer desteği olmaması
- timer çözünürlüğünün sadece saniyenin katları cinsinden verilebilmesi
- timer için kalan sürenin ne kadar olduğunu öğrenmenin bir yolu olmaması
- event loop mekanizması içerisinden kullanımına yönelik bir destek sunmaması

olarak sıralanabilir.

Şimdi tekrar yukarıda verdiğimiz örneğe geri dönelim. Uygulamayı `alarm.c` adıyla kaydedip, derleyip çalıştırdığımızda 1 saniye geçince *timer_callback* fonksiyonu çağrılacak, `sleep(3)` satırı nedeniyle kalan 2 saniye boyunca bekleyecek ve düzgün bir şekilde sonlanacak mıdır?

```
$ gcc -o alarm alarm.c
$ time ./alarm
Signal 14 caught on: 1427030338

real   0m1.001s
user   0m0.000s
sys    0m0.000s
```

Süreleri ölçebilmek için uygulamayı `time` komutunu vererek çalıştırdık. Fakat görüleceği üzere toplam çalışma süresi 3 saniye değil, toplamda 1 saniye şeklinde gerçekleşti. Neden?

Daha önceki Sistem Çağrıları ve Sinyaller konularında değindiğimiz maddeleri hatırlayınız. `sleep(3)` fonksiyonunun yol açtığı sistem çağrısı çalışıyorken 1. saniye dolduğunda `alarm(1)` 'den kaynaklanan `SIGALRM` sinyali geldiğinde, `sleep(3)` için başlatılan sistem çağrısı kesintiye uğradı.

`sleep()` fonksiyonu sistem çağrısı tarafından kesintiye uğradığında saniye cinsinden kalan zamanı geri döndürür. Zaman tamamen bitip normal şekilde geri döndüğünde ise 0 döndürür. Bu özelliği dikkate alarak, global `errno` değişkeni ile de `INTERRUPT` durumu olup

olmadığını test ederek uygulamamızı aşağıdaki gibi daha güvenli hale getirebiliriz. Ek olarak alarm sinyalini işlediğimiz *timer_callback* fonksiyonunda ilgili alarmı tekrar kurup periyodik çalışmasını sağladığımıza ve *while* döngüsüne girmeden önce yarım saniyelik ek bir bekleme koyduğumuza dikkat ediniz:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <errno.h>
#include <string.h>
#include "../common/debug.h"

void timer_callback (int signum)
{
    time_t now = time(NULL);
    printf ("Signal %d caught on: %li\n", signum, now);
    alarm(1);
}

int main ()
{
    unsigned int remaining = 3;
    signal(SIGALRM, timer_callback);
    alarm(1);
    usleep(500000);
    while ( (remaining = sleep(remaining)) != 0) {
        if (errno == EINTR) {
            debugf("sleep interrupted by signal");
        } else {
            errorf("sleep error: %s", strerror(errno));
        }
    }
    return 0;
}
```

Şimdi uygulama çıktısına tekrar göz atalım:

```
$ time ./alarm
Signal 14 caught on: 1427033478
debug: sleep interrupted by signal (main alarm.c:23)
Signal 14 caught on: 1427033479
debug: sleep interrupted by signal (main alarm.c:23)
Signal 14 caught on: 1427033480
debug: sleep interrupted by signal (main alarm.c:23)
Signal 14 caught on: 1427033481

real   0m4.001s
user   0m0.000s
sys    0m0.000s
```

Bu defa uygulamamız yaklaşık 4 saniye kadar çalışıp sonlandı. Oysa beklentimiz 3.5 saniye kadar sürmesi idi. Uygulama çıktısını daha dikkatli inceleyip nedenini yorumlamaya çalışınız.

Interval Timer Kullanımı

Interval timer mekanizması ilk olarak 4.2BSD versiyonunda görülmüş sonradan POSIX tarafından standardize edilmiştir.

Geleneksel `alarm()` tabanlı timer yöntemine oranla temel avantajları aşağıdaki şekilde sıralanabilir:

- mikrosaniye seviyesinde çözünürlük sağlar
- zaman ölçümünü 3 farklı mod üzerinden daha detaylı kontrol etme imkanı verir
- bir defa ayarlayıp, periyodik olarak çalışmasını sağlamak mümkündür
- herhangi bir anda ne kadar zaman kaldığı sorgulanabilir

Interval timer işlemleri için kullanılan fonksiyon prototipleri aşağıdaki gibidir:

```
#include <sys/time.h>

int setitimer (int which, const struct itimerval *new_value,
               struct itimerval *old_value)

int getitimer (int which, struct itimerval *value)

struct itimerval {
    struct timeval it_interval;   /* next value    */
    struct timeval it_value;      /* current value */
};

struct timeval {
    long tv_sec;
    long tv_usec;
};
```

Interval timer kurmak istendiğinde `setitimer` fonksiyonunun 2. parametresine, istemiş olduğumuz timer ile ilgili zaman bilgilerini, ilk çalıştığında istenen süre (veri yapısının `it_value` elemanı), sonraki çalışmalarında istenen süre (veri yapısının `it_interval` elemanı) biçiminde vermemiz gereklidir.

Örnek olarak, önce 1 saniye ardından 300 milisaniyede bir uygulamamızı haberdar edecek bir interval timer aşağıdaki şekilde kurulabilir:

```
struct itimerval new_timer;
struct itimerval old_timer;

new_timer.it_value.tv_sec     = 1;
new_timer.it_value.tv_usec    = 0;
new_timer.it_interval.tv_sec  = 0;
new_timer.it_interval.tv_usec = 300 * 1000;

setitimer(ITIMER_REAL, &new_timer, &old_timer);
```

Fonksiyonun 3. parametresine verilen `itimerval` türündeki değişken adresine, yeni değerler ayarlanmadan önce aktif durumda olan bir interval timer var ise onun değerleri aktarılır.

Interval timer mekanizmasıyla 3 farklı tipte timer kurulması mümkündür. Kullanılacak timer tipi `setitimer()` fonksiyonunun ilk parametresinde belirtilir.

Timer Tipi	Sinyal	Açıklama
ITIMER_REAL	`SIGALRM`	Uygulamanın harcadığı zamandan bağımsız, toplam geçen zaman üzerinden hesaplanır
ITIMER_VIRTUAL	`SIGVTALRM`	Uygulamanın sadece kullanıcı kipinde çalıştığı zaman üzerinden hesaplanır
ITIMER_PROF	`SIGPROF`	Uygulamanın hem kullanıcı kipinde harcadığı zaman, hem de yapmış olduğu sistem çağrıları içerisinde harcamış olduğu zamanın toplamı üzerinden hesaplanır. Pratikte uygulama içerisinde ITIMER_VIRTUAL ile birlike ayrı ayrı kurulup, uygulamanın kullanıcı kipi ve çekirdek kipinde ayrı ayrı ne kadar zaman harcadığını hesaplamakta kullanılır.

Yukarıdaki tabloyu incelediğimizde, ITIMER_REAL tipinde bir interval timer kurduğumuzda, tıpkı `alarm()` fonksiyonu ile kurmuş olduğumuz timer mekanizmasında olduğu gibi, **SIGALRM** sinyali ile uygulamamızı uyandıracağını görebiliriz. Dolayısıyla bu şekilde bir interval timer ile `alarm()` fonksiyonunun aynı uygulamada kullanımı, her ne kadar sinyalin işlendiği fonksiyonda `getitimer()` fonksiyonuyla kalan zaman üzerinde ikinci bir kontrol yapılmak suretiyle çözüm üretmek mümkün olsa da kafa karıştıracak ve yönetimi güçleştirecektir. Bu nedenle aynı anda kullanımı önerilmez.

Önceki örneğimizi interval timer kullanacak şekilde aşağıdaki gibi değiştirip çalıştıralım:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <errno.h>
#include <string.h>
#include "../common/debug.h"

void timer_callback (int signum)
{
    struct timeval now;
    gettimeofday(&now, NULL);
    printf ("Signal %d caught on: %li.%03li\n", signum, now.tv_sec, now.tv_usec / 1000
);
}

int main ()
{
    unsigned int remaining = 3;
    struct itimerval new_timer;
    struct itimerval old_timer;

    new_timer.it_value.tv_sec     = 1;
    new_timer.it_value.tv_usec    = 0;
    new_timer.it_interval.tv_sec  = 0;
    new_timer.it_interval.tv_usec = 300 * 1000;

    setitimer(ITIMER_REAL, &new_timer, &old_timer);

    signal(SIGALRM, timer_callback);

    while ( (remaining = sleep(remaining)) != 0) {
        if (errno == EINTR) {
            debugf("sleep interrupted by signal");
        } else {
            errorf("sleep error: %s", strerror(errno));
        }
    }

    return 0;
}
```

Yukarıdaki kodda öncelikle 1 saniye, sonra da 300 milisaniyede bir interval timer kurulmuş ve tıpkı ilk örneğimizdeki gibi, toplam çalışma süresi 3 saniye olacak şekilde `sleep()` fonksiyonunun sinyal tarafından kesintiye uğraması durumu ele alınmış. Örnek kodu `interval.c` isminde kaydedip derleyelim ve çalıştıralım:

```
$ gcc -o interval interval.c
$ time ./interval
Signal 14 caught on: 1427042689.280
debug: sleep interrupted by signal (main interval.c:34)
Signal 14 caught on: 1427042689.580
debug: sleep interrupted by signal (main interval.c:34)
Signal 14 caught on: 1427042689.880
debug: sleep interrupted by signal (main interval.c:34)
Signal 14 caught on: 1427042690.180
debug: sleep interrupted by signal (main interval.c:34)
Signal 14 caught on: 1427042690.480
debug: sleep interrupted by signal (main interval.c:34)
Signal 14 caught on: 1427042690.780
debug: sleep interrupted by signal (main interval.c:34)
Signal 14 caught on: 1427042691.080
debug: sleep interrupted by signal (main interval.c:34)
...
```

Uygulamamız timer ilk çalıştıktan sonra çıktıdan da anlaşılacağı üzere tam da beklediğimiz gibi 300 milisaniyede bir *callback* fonksiyonumuzu çağırdı. Ancak biraz daha beklenildiğinde uygulamanın sonlanmadığı ve 300 milisaniyede bir *callback* fonksiyonunu çalıştırmaya devam ettiği görülecektir. Interval değerini 600 milisaniye yaptığınızda ise uygulamanın sonlandığını görebilirsiniz.

> `sleep()` fonksiyonu etrafında düşünerek bu davranışın nedenlerini bulmaya çalışınız.

POSIX Timer API

Önceki bölümde değindiğimiz geleneksel timer yapılarının çok önemli bir handikapı vardı. En gelişmişi olan interval timer mekanizmasında bile aynı anda en fazla 3 farklı tipte timer işletmek mümkündü. Herhangi bir tipte yeni bir timer kurulduğunda, zaman olduğunda uygulamayı aynı yöntemle (aynı sinyalle) haberdar ettiğinden birbirinden bağımsız onlarca timer gerektiren senaryolar için verimli bir çözüm üretmek mümkün değildi.

Bu bölümde POSIX tarafından standardize edilen gelişmiş timer kullanımına değineceğiz. Bir POSIX timer'ın oluşturulması, süresinin ayarlanması ve artık ihtiyaç kalmadığı anda yok edilmesinden oluşan temelde 3 aşamalı bir yaşam döngüsü bulunur.

> Not: POSIX timer kullanan uygulamalar derlenirken `-lrt` ile **POSIX.1b** Realtime Extensions kütüphanesine de linklenmelidir.

Oluşturma: `timer_create`

Timer işlemlerinde `timer_t` tipindeki handle, aşağıda prototipi bulunan `timer_create()` fonksiyonu ile elde edilmelidir.

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

int timer_create (clockid_t clockid, struct sigevent *evp, timer_t *timerid);
```

İlk parametre ile kullanılacak *clock* tipi belirtilir.

Clock Tipi	Açıklama
CLOCK_REALTIME	Gerçek zamanda geçen süre cinsinden zaman hesaplanır. Bu değer sistem saatinin ileri veya geri alınmasından etkilenir.
CLOCK_MONOTONIC	Sistem açılışında sıfırlanan, saat ayarlamalarından etkilenmeyen güvenilir MONOTONIC zaman değeri üzerinden hesaplanır.
CLOCK_PROCESS_CPUTIME_ID	Uygulamanın tüm thread'lerinin kullanıcı kipi ve çekirdek kipinde harcadığı toplam süre üzerinden hesaplanır.
CLOCK_THREAD_CPUTIME_ID	Uygulama içerisinde sadece timer oluşturma isteğini gönderen thread için kullanıcı kipi ve çekirdek kipinde harcanan süre toplamı üzerinden hesaplanır.

Fonksiyonun 2. parametresi `struct sigevent` türünde olup prototipi aşağıdaki gibidir:

```
#include <signal.h>

struct sigevent {
    union sigval sigev_value;
    int sigev_signo;
    int sigev_notify;
    void (*sigev_notify_function)(union sigval);
    pthread_attr_t *sigev_notify_attributes;
};

union sigval {
    int sival_int;
    void *sival_ptr;
};
```

Yapı içerisinde yer alan `sigev_notify` elemanı, timer zamanı dolduğunda uygulamanın nasıl haberdar edileceğini kontrol etmemizi sağlar. Kullanılabilecek sabitler aşağıdaki gibidir:

sigev_notify	Açıklama
SIGEV_NONE	Timer sona erdiğinde uygulamaya herhangi bir bildirimde bulunulmaz
SIGEV_SIGNAL	Timer sona erdiğinde uygulamaya `sigev_signo` alanında girilen tipte sinyal gönderilir. Sinyal callback fonksiyonunun `si_value` değerine, bu yapı ile belirtilen *union* `sigev_value` alanıyla tanımlanan parametre geçirilir.
SIGEV_THREAD	Timer sona erdiğinde işletim sistemi yeni bir thread oluşturur. Thread içerisinde `sigev_notify_function` fonksiyonu çağrılır. Fonksiyona parametre olarak *union* `sigev_value` alanında belirtilen değer geçirilir. Ek olarak oluşturulacak threadin özellikleri belirlenmek istenirse `pthread_attr_t` tipindeki `sigev_notify_attributes` alanı kullanılabilir.

Oluşturulan POSIX timer için sinyalle uyandırılma seçildi ise `sigev_signo` alanında verilen sinyalin *callback* fonksiyonu standart sinyal işleme yöntemleriyle tanımlanmalıdır. Bu noktada dikkat edilecek ek husus, `struct sigaction` veri yapısı ile *callback* tanımlanırken `sa_flags` elemanının değerini **SA_SIGINFO** sabitine eşitlemektir.

Bu şekilde ilgili *callback* fonksiyonu sinyal nedeniyle çağrıldığında geleneksel sinyal işleme yöntemindeki gibi sadece sinyal numarasını parametre alan bir fonksiyon çağırmaz, bunun yerine aşağıdaki fonksiyon prototipinde bir çağrıda bulunur:

```
void callback(int sig, siginfo_t *si, void *context)
```

Bu sayede `siginfo_t` ile gelen elemanın içerisinden, timer oluşturulma sırasında ayarlanan `sigev_value` geri alınabilir. Asenkron çalışma sistematiği için bu şekildeki bir kullanım çoğu zaman zorunlu olacaktır.

> Üçüncü argüman olan genel amaçlı `context` pointer verisi timer mekanizmasında kullanılmaz. Kullanım senaryosunu incelemek için `getcontext()` fonksiyonuna bakabilirsiniz: http://manpages.org/getcontext/3

Ayarlama: `timer_settime`

Timer oluşturup `timer_t` türünde bir handle elde ettikten sonra, aşağıda prototipi yer alan `timer_settime()` fonksiyonuyla interval timer sürecine benzer şekilde değerler ayarlanır.

```
#include <time.h>

int timer_settime(timer_t timerid, int flags,
    const struct itimerspec * value, struct itimerspec * old_value)
int timer_gettime(timer_t timerid, struct itimerspec *curr_value)

struct itimerspec {
    struct timespec it_interval;  /* period of timer        */
    struct timespec it_value;     /* first expiration value */
};

struct timespec {
    time_t tv_sec;  /* second     */
    long tv_nsec;   /* nanosecond */
};
```

Interval timer kullanımından farkı `it_interval` ve `it_value` için `struct timeval` yerine, nanosaniye seviyesinde değer girilebilmesine imkan veren `struct timespec` tipi kullanılmaktadır. Benzer şekilde `it_value` elemanında ilgili timer'ın ilk çalıştığındaki kullanacağı süre değeri, `it_interval` elemanında ise sonraki seferlerde periyodik olarak kullanılacak değer yer alır.

İkinci parametre olan `flags` argümanı normalde 0 olarak kullanılır. Bu durumda öntanımlı davranış, kullanılan clock tipine göre göreceli şekilde `it_value` alanındaki timer'ın ilk çalışmasındaki süre değeri dikkate alınır. Örnek olarak `it_value` bölümünde 5 saniyelik bir zaman dilimi tanımlanmışsa, kullanılan clock tipine göre göreceli olarak timer ayarlama zamanından 5 saniye sonra ilk uyandırma gerçekleşir.

Bununla birlikte bazen ilk uyandırma işleminin göreceli olarak değil de belirli bir anda yaptırılması istenebilir. Örnek olarak tam olarak bir sonraki saat başında ilk olarak uyandırılmak, sonra da saniyede bir periyodik uyandırılmak istenebilir. Bu durumda `flags` bölümünde **TIMER_ABSTIME** sabiti kullanılabilir.

Verilen bir `timer_t` handle üzerinden bir sonraki uyandırma zamanına ne kadar kaldığı da `timer_gettime()` fonksiyonu alınabilmektedir.

Yok Etme: `timer_delete`

Kullanılmayan timer değerler, prototipi aşağıda belirtilen `timer_delete()` fonksiyonuyla yok edilir ve harcadığı sistem kaynakları geri verilir:

```
#include <time.h>

int timer_delete(timer_t timerid)
```

Örnek Uygulama

Aşağıdaki uygulamayı `posix.c` adıyla kaydedip ardından derleyip çalıştıralım.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <errno.h>
#include <string.h>
#include "../common/debug.h"

#define TIMER_SIGNAL SIGRTMIN + 1

struct person {
    int no;
    char name[32];
};

void timer_callback (int signum, siginfo_t *si, void *context)
{
    (void) context;
    struct timeval now;
    struct person *p;
    gettimeofday(&now, NULL);
    p = (struct person *) si->si_value.sival_ptr;
    printf ("Signal %d caught on: %li.%03li\t Name: %s\n",
        signum, now.tv_sec, now.tv_usec / 1000, p->name);
}

int main ()
{
    unsigned int remaining = 3;

    timer_t timer1;
    timer_t timer2;
    struct person *p1;
    struct person *p2;
    struct itimerspec new_value;
    struct itimerspec old_value;
    struct sigaction sa;
    struct sigevent sev;
    int ret;

    p1 = calloc(1, sizeof(struct person));
    p1->no = 1;
    strcpy(p1->name, "Name 1");

    p2 = calloc(1, sizeof(struct person));
```

```
    p2->no = 2;
    strcpy(p2->name, "Name 2");

    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = timer_callback;
    sigemptyset(&sa.sa_mask);

    if (sigaction(TIMER_SIGNAL, &sa, NULL) == -1) {
        errorf("sigaction error");
        exit(1);
    }

    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo  = TIMER_SIGNAL;

    sev.sigev_value.sival_ptr = p1;

    if ( (ret = timer_create(CLOCK_MONOTONIC, &sev, &timer1)) < 0) {
        errorf("timer1 create failed: %s", strerror(errno));
    }

    sev.sigev_value.sival_ptr = p2;

    if ( (ret = timer_create(CLOCK_MONOTONIC, &sev, &timer2)) < 0) {
        errorf("timer2 create failed: %s", strerror(errno));
    }

    new_value.it_value.tv_sec     = 1;
    new_value.it_value.tv_nsec    = 0;
    new_value.it_interval.tv_sec  = 0;
    new_value.it_interval.tv_nsec = 600 * 1000 * 1000;

    if (timer_settime(timer1, 0, &new_value, &old_value) < 0 ||
            timer_settime(timer2, 0, &new_value, &old_value) < 0) {
        errorf("timer settime error: %s", strerror(errno));
    }

    while ( (remaining = sleep(remaining)) != 0) {
        if (errno == EINTR) {
            debugf("sleep interrupted by signal");
        } else {
            errorf("sleep error: %s", strerror(errno));
        }
    }

    return 0;
}
```

Derleyip çalıştırdığımızda beklediğimiz gibi 2 farklı timer, kendileriyle ilgili özel veri yapısınının adresini de taşıyarak *callback* fonksiyonunu çağırmaktadır.

```
$ gcc -p posix posix.c -lrt
$ ./posix
Signal 35 caught on: 1427066563.117     Name: Name 1
Signal 35 caught on: 1427066563.117     Name: Name 2
debug: sleep interrupted by signal (main posix.c:86)
Signal 35 caught on: 1427066563.717     Name: Name 1
Signal 35 caught on: 1427066563.717     Name: Name 2
debug: sleep interrupted by signal (main posix.c:86)
Signal 35 caught on: 1427066564.317     Name: Name 1
Signal 35 caught on: 1427066564.317     Name: Name 2
```

Event Loop İçinde Kullanım

Sinyaller üzerinden çalışan asenkron *callback* fonksiyonları yönetmek bazen güçleşebilir. Özellikle uygulama kodu büyüdükçe, sinyal *callback* fonksiyonu içerisinde fazla işlem yapmak da riskli olduğu için çok fazla global değişken kullanımıyla karşı karşıya kalabiliriz. Ayrıca kullandığımız timer sayısı arttıkça (yüzlerce olduğunu düşünün) asıl uygulama kodunun normal akışının sürekli kesintiye uğramasının da getireceği kayıpları dikkate almalıyız.

Linux çekirdeğinin **2.6.25** versiyonuyla birlikte timer bildirimleri için sinyal ve thread kullanımına ek olarak, dosya betimleyicisi (file descriptor API) üzerinden kullanım desteği de eklenmiştir.

Bu model, her bir timer için ayrı bir dosya betimleyici (file descriptor) olması esasına dayanır. Elimizde *callback* fonksiyonları yerine dosya betimleyicileri olduğunda, `select()`, `poll()`, `epoll()` gibi event-loop mekanizmaları ile dosya betimleyicinin hazır olup olmadığını test edebilir, okumaya hazır durumda ise (timer zamanı gelmiş demektir) ilgili fonksiyonu çalıştırabiliriz.

Bunun en büyük avantajı, sadece timer için olanları değil, uygulamamızın kullandığı diğer tüm dosya betimleyicilerini aynı event-loop içerisinde dinlemenin (soketler, seri port, *inotify* ile izlediklerimiz vb.) asenkron çalışmanın getirdiği ek yükleri ortadan kaldırması ve işlemlerin kontrollü biçimde serileştirilerek yapılmasının sağlanmasıdır.

> Timer kullanımının bu yeni hali henüz POSIX tarafından standardize edilmemiş olduğundan, platformlar arası taşınabilir kod yazmak isteyenlerin bu durumda dikkat etmesi gereklidir.

Oluşturma: `timerfd_create`

İlk fonksiyonumuz timer oluşturup dosya betimleyici dönen `timerfd_create()` olup prototipi aşağıdaki gibidir:

```
#include <sys/timerfd.h>

int timerfd_create(int clockid , int flags)
```

Görüldüğü gibi `timer_create()` fonksiyonuna nazaran kullanımı daha basittir zira sinyal üzerinden geri bildirim nedeniyle yapılması gereken ek ayarlamalara ihtiyaç kalmamıştır.

Clock tipi olarak önceki yöntemlere benzer şekilde **CLOCK_REALTIME** ve **CLOCK_MONOTONIC** kullanılabilir.

Flags parametresi **0** olarak geçilebileceği gibi aşağıdaki değerler de kullanılabilir:

Flag	Açıklama
TFD_NONBLOCK	Açılacak dosya betimleyici üzerinde `O_NONBLOCK` bayrağını aktifleştir, non-blocking modda çalışmaya izin ver
TFD_CLOEXEC	Açılacak dosya betimleyici üzerinde `FD_CLOEXEC` (close-on-exec) bayrağını aktifleştir. Multi-threaded uygulamalarda `exec()` fonksiyon ailesinden bir çağrı yapıldığını otomatik ve atomik biçimde dosyayı kapatır

Başarılı bir çağrı sonrasında edindiğimiz dosya betimleyicinin geleneksel dosya betimleyicilerden farkı yoktur. Dolayısıyla timer ile işimiz bittiğinde yok etmek için özel ek bir fonksiyon bulunmaz. Elimizdeki dosya betimleyici üzerinden dosyaları kapatmakta kullandığımız `close()` fonksiyonunu çağırmak yeterli olacaktır.

Ayarlama: `timerfd_settime`

Elimizde bir `timerfd_create` sonrası elde edilmiş dosya betimleyici var ise, zamanlayıcı kurmak veya kalan zamanı öğrenmek için prototipleri aşağıda verilen `timerfd_settime` ve `timerfd_gettime` fonksiyonları kullanılır.

```
#include <sys/timerfd.h>

int timerfd_settime(int fd, int flags,
    const struct itimerspec * new_value, struct itimerspec * old_value)

int timerfd_gettime(int fd, struct itimerspec * curr_value);
```

FD_CLOEXEC Kullanımının Önemi

Uygulama kodunuz içerisinde `fork()` ile yeni bir process başlatıp ardından `exec()` ile yeni kodu yükleyip çalıştırdığınızı düşünelim. Linux process modelinde fork edilerek oluşturulan yeni çocuk process, kendisinin parent process'inden açık dosya betimleyicilerin de bir kopyasını almış olur.

Bu sebeple parent process içerisinde `timerfd_create` ile açılmış olan dosya betimleyicileri var ise, `exec` edilen yeni process içerisinde de zamanlayıcıda ayarlı süre dolduğunda bildirim mekanizmaları devreye girecektir.

Bu istenmeyen etkiden kurtulmanın yolu, yeni çalıştırılacak process'in öncelikle bu şekilde açık dosya betimleyicilerini kapatmasıdır. Ancak bu model, multi-threaded uygulamalarda çeşitli yarış durumu (race condition) problemlerine yol açabilmektedir.

Kesin çözümün sağlanması için, `fork` + `exec` modelinde `FD_CLOEXEC` bayrağını aktifleştirmek gereklidir. Bu bayrak ile açılmış olan dosyalar, ayrı bir process oluştuğunda kopyalanmak yerine yarış durumu problemi üretmeyecek şekilde otomatik ve atomik biçimde kapatılırlar.

Dosya Betimleyicinin Okunması

Oluşturulan ve zamanlayıcısı ayarlanan dosya betimleyiciler, süre dolumu gerçekleştiğinde okunabilir duruma geçerler (read-notification).

Bir event-loop kullanıyorsanız bir çok dosya betimleyicinin bu durumunu aynı anda kontrol edebilir ve hızlı haberdar olabilirsiniz.

Timer'ın ilgili olduğu dosya betimleyiciden okunabilir olduğu bilgisi geldiğinde, **8** byte uzunluğunda bir okuma yapılması gereklidir.

Okuma işleminin sonucunda `unsigned int64` tipinde bir veri gelecektir. Bu noktada okuma işlemi gerçekleştirilmez ise, event-loop kontrol fonksiyonuna dönüldüğünde uygulama anında tekrar uyandırılacak ve sonsuz döngüye girilecektir.

Örnek Uygulama

Önceki bölümde POSIX timer API ile gerçekleştirmiş olduğumuz uygulamayı bu defa event-loop ve `timerfd` kullanarak yapalım. Ek olarak 5 farklı timer kullanacağız. Herhangi bir timer bildirimi geldiğinde, dosya betimleyicisi üzerinden ilgili veri yapısına hızlıca ulaşabilmek için basit bir hash-table implementasyonu olan **uthash** da kullanacağız.

> uthash kaynak kodu ve dokümantasyonu için: https://github.com/troydhanson/uthash

Aşağıdaki örnek kodu `timerfd.c` adıyla kaydedip derleyiniz.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <sys/timerfd.h>
#include <inttypes.h>
#include <errno.h>
#include <string.h>
```

```
#include "../common/debug.h"
#include "../common/uthash.h"

struct person {
    int timerfd;
    int no;
    char name[32];
    UT_hash_handle hh;
};

int main ()
{
    unsigned int remaining = 3;

    struct person *p;
    struct person *p_tmp;
    struct itimerspec new_value;
    fd_set readfs;
    fd_set masterfs;
    int maxfd = 0;
    int ret;
    int i;
    struct person *persons = NULL;

    FD_ZERO(&masterfs);

    new_value.it_value.tv_sec     = 0;
    new_value.it_value.tv_nsec    = 500 * 1000 * 1000;
    new_value.it_interval.tv_sec  = 0;
    new_value.it_interval.tv_nsec = 900 * 1000 * 1000;

    for (i = 0; i < 5; i++) {
        p = calloc(1, sizeof(struct person));
        p->no = i;
        snprintf(p->name, sizeof(p->name), "Name : %d", i);
        p->timerfd = timerfd_create(CLOCK_MONOTONIC, TFD_CLOEXEC);
        if (timerfd_settime(p->timerfd, 0, &new_value, NULL) < 0) {
            errorf("timerfd_settime error: %s", strerror(errno));
            continue;
        }
        HASH_ADD_INT(persons, timerfd, p);
        FD_SET(p->timerfd, &masterfs);
    }

    for (i = 0; i < 1024; i++) if (FD_ISSET(i, &masterfs)) maxfd = i;

    while (remaining > 0) {
        struct timeval tout = { 1, 0};

select_again:

        readfs = masterfs;
```

```
        ret = select(maxfd + 1, &readfs, NULL, NULL, &tout);
        if (ret < 0) {
            if (errno == EINTR) {
                debugf("select interrupted by signal");
                goto select_again;
            } else {
                errorf("select error: %s", strerror(errno));
            }
        } else if (ret > 0) {
            for (i = 0; i <= maxfd && ret; i++) {
                if (FD_ISSET(i, &readfs)) {
                    uint64_t howmany;
                    read(i, &howmany, sizeof(uint64_t));
                    HASH_FIND_INT(persons, &i, p);
                    if (p == NULL) {
                        errorf("this shouldn't happen")
                    } else {
                        debugf("read fd: %d, val: %" PRIu64 ", name: %s", i, howmany,
p->name);
                    }
                    ret--;
                }
            }
            goto select_again;
        } else {
            // select timeout
            remaining--; // decrement 1 seconds
        }
    }

    HASH_ITER(hh, persons, p, p_tmp) {
        HASH_DEL(persons, p);
        free(p);
    }

    return 0;
}
```

Örnek uygulamamızda event-loop için `select()` yapısını kullandık. Eğer dinleyeceğimiz dosya betimleyici sayısı daha yüksek olsaydı `poll()` veya `epoll()` kullanmamız gerekecekti. Alıştırma yapmak için buradaki örneği `epoll()` ile çalışan bir event-loop'a dönüştürmeyi deneyebilirsiniz.

Önceki timer bölümlerinde sinyallerle bildirilen timer bildirimleri uygulamamızı asenkron kesintilere uğratıyordu. Aynı durumun bu örneğimiz için geçerli olup olmadığını, modelin avantajlı ve varsa dezavantajlı olduğu senaryoların neler olduğunu düşünmeye çalışınız.

Uygulamamız çalıştığında beklendiği gibi bir çıktı üretmektedir:

```
$ gcc -o timerfd timerfd.c -lrt
$ ./timerfd
debug: read fd: 3, val: 1, name: Name : 0 (main timerfd.c:80)
debug: read fd: 4, val: 1, name: Name : 1 (main timerfd.c:80)
debug: read fd: 5, val: 1, name: Name : 2 (main timerfd.c:80)
debug: read fd: 6, val: 1, name: Name : 3 (main timerfd.c:80)
debug: read fd: 7, val: 1, name: Name : 4 (main timerfd.c:80)
debug: read fd: 3, val: 1, name: Name : 0 (main timerfd.c:80)
debug: read fd: 4, val: 1, name: Name : 1 (main timerfd.c:80)
debug: read fd: 5, val: 1, name: Name : 2 (main timerfd.c:80)
debug: read fd: 6, val: 1, name: Name : 3 (main timerfd.c:80)
debug: read fd: 7, val: 1, name: Name : 4 (main timerfd.c:80)
debug: read fd: 3, val: 1, name: Name : 0 (main timerfd.c:80)
debug: read fd: 4, val: 1, name: Name : 1 (main timerfd.c:80)
debug: read fd: 5, val: 1, name: Name : 2 (main timerfd.c:80)
debug: read fd: 6, val: 1, name: Name : 3 (main timerfd.c:80)
debug: read fd: 7, val: 1, name: Name : 4 (main timerfd.c:80)
```

Daemon Oluşturma

Daemon prosesler, direkt olarak kullanıcının kontrolünde çalışmayan, arka planda (background) hizmet veren proseslerdir. Genellikle, sistem açılırken başlatılıp kapanana kadar çalışmaktadırlar. Sistemimizde çok sayıda daemon çalışmaktadır, birkaç bilindik deamon örneği aşağıdaki gibidir.

- **crond:** Komutların belirlenen zamanda çalışmasını sağlar.
- **sshd:** Uzak makinalardan sisteme oturum açılmasına olanak sağlar.
- **httpd:** Web sayfalarını sunar.
- **nfsd:** Ağ üzerinden dosya paylaşımını sağlar.

> **Not:** Daemon prosesler, bir zorunluluk olmamasına karşın, genellikle **d** harfi ile sonlanacak şekilde isimlendirilirler.

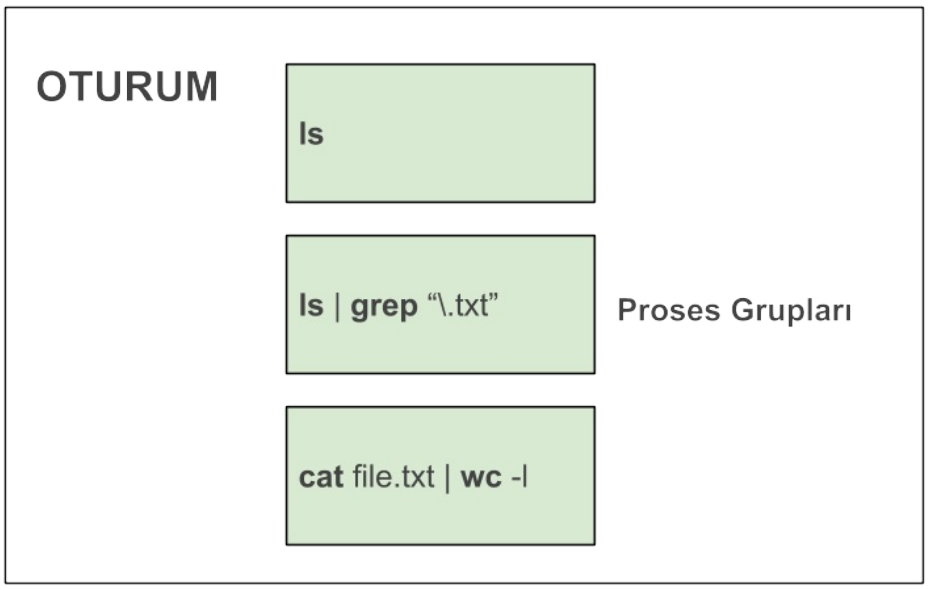
Bir prosesin daemon olarak çalışabilmesi için aşağıdaki gibi bir yol izlenebilir.

- Ayar dosyalarının okunması veya gerekli sistem kaynaklarının elde edilmesi gibi başlangıç işlemleri, proses gerçek anlamda daemon olmadan önce yapılmalıdır. Bu sayede, alınan hatalar kullanıcıya bildirilebilir ve proses uygun bir hata kodu ile sonlandırılabilir.
- Üst prosesi (parent process) *init* olan, arka planda çalışan bir proses oluşturulmalıdır. Bu amaçla, proses içinde önce *fork* işlemi yapılarak bir alt proses oluşturulmalı, sonrasında üst proses *exit* ile sonlandırılmalıdır.
- Yoluna devam eden proses içinde *setsid* fonksiyonu çağrılarak yeni bir oturum (session) açılmalı ve prosesin terminal (controlling terminal) ile ilişkisi kesilmelidir.
- Üst prosesten miras alınan tüm açık dosya betimleyicileri (file desriptor) kapatılmalı.
- Standart giriş, çıkış ve hata mesajları /dev/null aygıtına yönlendirilmeli.
- Prosesin çalışma dizini değiştirilmeli.

Daemon oluşturmanın detaylarına geçmeden önce, yukarıda da bahsettiğimiz oturum kavramına değinmek istiyoruz.

Oturum (Session)

Kullanıcılar, bir terminal üzerinden sisteme giriş yaptıktan sonra (login) kabuk programı (shell) üzerinden birçok uygulamayı çalıştırabilmektedirler. Kullanıcı sistemden çıktığında bu prosesler kapatılmalıdır. İşletim sistemi bu prosesleri, oturum (session) ve proses grupları şeklinde gruplandırır. Her bir oturum proses gruplarından oluşmaktadır. Bu durumu aşağıdaki gibi tasvir edebiliriz.



Oturumdaki proseslerin girdilerini aldıkları ve çıktılarını gönderdikleri terminal kontrol terminali (controlling terminal) olarak isimlendirilmektedir. Bir kontrol terminali aynı anda yalnız bir tane oturum ile ilişkili olabilir. Bir oturumun ve içindeki proses gruplarının kimlik (ID) numaraları bulunmaktadır, bu kimlik numaraları oturum ve proses grup liderlerinin proses kimlik numaralarıdır (PID). Bir alt proses üst prosesiyle aynı grubu paylaşmaktadır. Pipe mekanizmasıyla haberleştirilen proseslerde ilk proses, proses grup lideri olmaktadır. Basit bir örnek üzerinden bu duruma daha yakından bakalım.

test.c:

```
#include <stdio.h>

int main() {
    getchar();
    return 0;
}
```

```
$ gcc -otest test.c
```

Hangi terminal üzerinde olduğumuza baktıktan sonra, uygulamayı çalıştırabiliriz.

```
$ tty
/dev/pts/24

$ ./test
```

Uygulamamız önplanda çalıştığından, başka bir terminal üzerinden prosesimizle ilgili bilgilere aşağıdaki gibi ulaşabiliriz.

```
$ ps -C test -o "pid ppid pgid sid tty command"
  PID  PPID  PGID   SID TT       COMMAND
 8788  8703  8788  8703 pts/24   ./test
```

ps çıktısındaki kısaltmalar ve anlamları aşağıdaki gibidir.

Kısaltma	Anlamı
PID	Proses Kimliği (Process ID)
PPID	Üst Proses Kimliği (Parent Process ID)
PGID	Proses Grup Kimliği (Process Group ID)
SID	Oturum Kimliği (Session ID)
TT	Terminal

Prosesimizin, proses ve grup kimliklerinin aynı olduğunu görüyoruz, bu durumda proses kendi grubunun lideri durumundadır. Oturum kimlik değerinin ise 8703 olduğunu görmekteyiz. Prosesimizin oturum kimliği, oturum liderinin kimlik değeri (PID) olduğundan, bu kimliğin hangi prosese ait olduğunu aşağıdaki gibi bulabiliriz.

```
$ ps -jp 8703
  PID  PGID   SID TTY          TIME CMD
 8703  8703  8703 pts/24   00:00:00 bash
```

Terminale giriş yaptıktan sonra ilk çalışan kabuk prosesisin oturum lideri olduğunu görüyoruz. Kabuk prosesisin tüm kimlik değerlerinin aynı olduğuna dikkat ediniz.

Şimdi bir prosesi nasıl deamon yapabileceğimize daha yakından bakalım.

Daemon Proses Oluşturma

GNU C kütüphanesi, **daemon** isimli bir fonksiyon barındırmasına karşın bu fonksiyon POSIX standartlarında yer almamaktadır. daemon fonksiyon kodu, glibc ana dizininde *misc/daemon.c* dosyasında bulunmaktadır. Burada benzer bir fonksiyonu nasıl oluşturabileceğimizi inceleyeceğiz. Bu amaçla aynı arayüze sahip **_daemon** isimli bir fonksiyonu adım adım oluşturacağız. Daemon olarak çalışacak uygulama kodunu *test.c*, daemon fonksiyonunu adım adım oluşturacağımız kodu ise *daemon.c* olarak isimlendirelim.

```
#include <stdio.h>

int _daemon(int, int);

int main() {
    getchar();
    _daemon(0, 0);
    getchar();
    return 0;
}
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>
#include <linux/limits.h>

int _daemon(int nochdir, int noclose) {
    pid_t pid;
    int i;

    pid = fork ( );
    if (pid == -1)
        return -1;
    else if (pid != 0) {
        exit (EXIT_SUCCESS);
    }

    return 0;
}
```

Daemon oluşturmak için yapılması gerekenleri maddelerken, üst prosesi init olan arka planda çalışan bir prosese ihtiyacımız olduğundan bahsetmiştik. Şu haliyle `_daemon` kodumuz, bir alt proses oluşturmakta ve sonrasında üst prosesi öldürmektedir. Bu durumda yeni prosesimiz init prosesinin alt prosesi olacak ve arkaplanda çalışmaya devam edecektir. Uygulamanın hemen sonlanmasını engellemek için *test.c* içinde *getchar* fonksiyonlarını kullandık. Şimdi uygulamayı derleyip prosesin _deamon çağrılmadan önce ve sonrasındaki durumunu inceleyelim.

```
$ gcc -otest test.c daemon.c
```

Uygulamayı çalıştıralım herhangi bir tuşa basmaksızın başka bir terminale geçelim.

```
$ ./test
```

Prosesimizle ilgili değerlerin aşağıdaki gibi olduğunu görmekteyiz. Bu durumda henüz _daemon fonksiyonu çağrılmamış durumdadır.

```
$ ps -C test -o "pid ppid pgid sid tty stat command"
  PID  PPID  PGID   SID TT       STAT COMMAND
 8947  8703  8947  8703 pts/24   S+   ./test
```

STAT alanına baktığımızda prosesimizin çalışabilir durumda ama çizelge dışında bir olayın gerçekleşmesini bekleğini ve önplanda çalıştığını görüyoruz. STAT alanıyla ilgili bazı kısaltmalar ve anlamları aşağıdaki gibidir.

Kısaltma	**Anlamı**
S	Bir olayın gerçekleşmesi için uykuda bekleniyor
T	Uygulama durdurulmuş
s	Oturum lideri
+	Uygulama önplanda çalışıyor

Uygulamamızın üst prosesinin beklediğimiz üzere kabuk olduğunu görmekteyiz.

```
$ ps -jp 8703
  PID  PGID   SID TTY          TIME CMD
 8703  8703  8703 pts/24   00:00:00 bash
```

Şimdi uygulamamızı çalıştırdığımız terminale dönelim ve _daemon fonksiyonunun çağrılması için bir enter tuşuna basalım. Tekrar diğer terminal üzerinde proses bilgilerine bakalım.

```
$ ps -C test -o "pid ppid pgid sid tty stat command"
  PID  PPID  PGID   SID TT       STAT COMMAND
 9004  5842  9003  8703 pts/24   S    ./test
```

İlk olarak STAT alanında + karakterini görmediğimiz için yeni alt prosesin arkaplanda çalıştığını söyleyebiliriz. Şimdi prosesin üst prosesinin kim olduğunu bakalım.

```
$ ps -jp 5842
  PID  PGID   SID TTY          TIME CMD
 5842  5842  5842 ?        00:00:00 upstart
```

Artık prosesimizin üst prosesisin, geleneksel init yerine kullanılan, upstart prosesi olduğunu görmekteyiz (Ubuntu kullanıyorsanız).

Şimdi bir sonraki adıma geçebiliriz. Bir sonraki adımda yeni bir oturum açılması ve prosesin kontrol terminaliyle ilişkisinin kesilmesi gerektiğini söylemiştik. Bu amaçla setsid fonksiyonu kullanılmaktadır. _daemon fonksiyonumuza bu çağrıyı ekleyelim. Eklenecek kod parçası aşağıdaki gibidir.

```
if (setsid() == -1)
    return -1;
```

_daemon çağrılmadan önceki durumu incelediğimiz için artık *test.c* kodundaki ilk getchar fonksiyonunu kaldırabiliriz.

test.c:

```
#include <stdio.h>

int _daemon(int, int);

int main() {
    _daemon(0, 0);
    getchar();
    return 0;
}
```

Uyulamayı yeniden derleyelim çalıştırdıktan sonra incelemelerimizi yaptığımız terminale geçelim. Prosesimizin yeni durumu aşağıdaki gibidir.

```
$ ps -C test -o "pid ppid pgid sid tty stat command"
  PID  PPID  PGID   SID TT       STAT COMMAND
 9090  5842  9090  9090 ?        Ss   ./test
```

TT alanındaki `?` işareti prosesimizin artık bir terminale bağlı olmadığını göstermektedir. Bu durumda terminalin sonlanması durumunda prosesimiz yoluna devam edecek veya terminal üzerinden herhangi bir sinyal gönderilemeyecektir. Prosesimiz PID, PGID ve SID değerlerinin aynı olduğuna dikkat ediniz, prosesimiz artık oturum lideri durumundadır.

Sonrasında geçirdiğimiz argümanın değerine göre, çalışılan dizini root dizin olarak değiştiriyoruz. _daemon fonksiyonuna aşağıdaki kod parçasını ekleyebilirsiniz.

```
 if (!nochdir) {
    if (chdir("/") == -1)
        return -1;
}
```

Bir sonraki adımda ise, geçirilen argümana göre, tüm dosya betimleyicileri kapatılabilmektedir. _daemon fonksiyonuna aşağıdaki kodu ekliyoruz.

```
#define NR_OPEN 1024
if (!noclose) {
    for (i = 0; i < NR_OPEN; i++)
        close(i);

    open("/dev/null", O_RDWR);
    dup(0);
    dup(0);
}
```

Tüm dosya betimleyicileri kapatıldıktan sonra, daemon tarafında açılan yeni dosyalar sırasıyla `0`, `1` ve `2` dosya betimleyicileri ile gösterilecektir. Bu durumda örneğin kod içindeki printf komutları 2. açılan dosyaya yönlenmiş olacaktır. Bunu önlemek için ilk 3 betimleyicinin /dev/null aygıtını göstermesi sağlanmıştır. Yeni bir dosya açıldığında betimleyici olarak dosya betimleyici tablosundaki en küçük değerin verileceği garanti altına alınmıştır. Bu durumda *open* çağrısından sonra /dev/null aygıtı için 0 numaralı betimleyici tahsis edilecektir. dup fonksiyonlarıyla da sıradaki 1 ve 2 numaralı betimleyicilerin /dev/null aygıtını göstermesi sağlanmıştır. Bu durumda _daemon fonksiyonunun son hali aşağıdaki gibi olacaktır.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>
#include <linux/limits.h>

#define NR_OPEN 1024

int _daemon(int nochdir, int noclose) {
    pid_t pid;
    int i;

    pid = fork ( );
    if (pid == -1)
        return -1;
    else if (pid != 0) {
        exit (EXIT_SUCCESS);
    }

    if (setsid() == -1)
        return -1;

      if (!nochdir) {
        if (chdir("/") == -1)
            return -1;
    }

    if (!noclose) {
        for (i = 0; i < NR_OPEN; i++)
            close (i);
        open("/dev/null", O_RDWR);
        dup(0);
        dup(0);
    }

    return 0;
}
```

Örnek olarak, *sshd* uygulamasının daemon olarak çalışmaya başlatıldığı kod parçası aşağıdaki gibidir.

```
...
if (!(debug_flag || inetd_flag || no_daemon_flag)) {
        int fd;

        if (daemon(0, 0) < 0)
            fatal("daemon() failed: %.200s", strerror(errno));

        /* Disconnect from the controlling tty. */
        fd = open(_PATH_TTY, O_RDWR | O_NOCTTY);
        if (fd >= 0) {
            (void) ioctl(fd, TIOCNOTTY, NULL);
            close(fd);
        }
}
...
```

Capabilities API

Unix tabanlı sistemlerde yetki kontrolü temel olarak iki adımdan oluşur:

- Çalışan uygulamanın o anki etkin sahibi (effective user id, EUID) **0** ise yetki kontrolü yapılmaz
- EUID değeri sıfırdan farklı ise, ilgili uygulamanın efektif kullanıcı ve grubunun yetkileri doğrultusunda kontrol işlemi gerçekleştirilir

Bazı uygulamaların çalıştığı süre boyunca daha geniş yetkilerle donatılması ihtiyacına (SUID, SGIT bitleri) önceki konularda değinmiştik. En tipik örnek olarak **passwd** uygulamasını gösterebiliriz. Bu uygulama ile sistemdeki kullanıcılar kendi parolalarını değiştirebilmektedirler. Ancak parolaların şifrelenmiş hallerinin tutulduğu `/etc/shadow` dosyasına yazabilmek için **root** kullanıcı haklarıyla (yani user id = 0 olarak) çalışmak gereklidir.

Problemin çözümü için **passwd** uygulamasına SUID biti verilmiş ve bu uygulamayı hangi kullanıcı çalıştırırsa çalıştırsın, etkin sahibinin (EUID) **root** olması sağlanmıştır:

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 54192 Nov 21 00:03 /usr/bin/passwd
```

Geleneksel Unix yetki kontrolü modelindeki SUID uygulama çalıştırma imkanı problemi çözmüş görünmektedir. Ancak SUID biti verilmiş uygulamalardaki kritik hatalar, sistemde tam yetkiye sahip bir kullanıcı olarak istenmeyen kodların çalıştırılabilmesine kapı açmaktadır. İdeal bir uygulama, mümkünse **root** kullanıcısının haklarına ihtiyaç duymadan çalışabilmelidir. Böylelikle yıkıcı eylemlerin işletim sistemi çekirdeği tarafından engellenmesi mümkün olabilir.

Problem sadece SUID biti ile de bitmemektedir. Unix tabanlı sistemlerde 1024'den küçük olan ayrıcalıklı bir TCP veya UDP portunu dinlemek istediğinizde de **root** kullanıcı haklarına sahip olmanız gerekir. Örneğin bir e-posta sunucu servisi ya da web sunucusu çalıştırmak istiyorsanız, sırasıyla TCP port 25 ve TCP port 80'i dinleyebiliyor olmanız gerekir. Bu da ancak ilgili uygulamaların **root** kullanıcısı ile çalıştırılması ile mümkün olabilir. Yıllar içerisinde bu şekilde özellikle ağ ortamına servis sunan yazılımların tam yetkili kullanıcı hesabıyla çalıştırılmalarının ne kadar yıkıcı etkileri olduğu tecrübe edilince, bir ara çözüm olarak programın sadece belirli ve daha küçük bir kısmının ayrıcalıklı portu **root** olarak dinlemesi, sonrasındaki işlemler içinse etkin kullanıcı kimliğini bir başka kullanıcıya değiştirmesi yöntemi (örneğin kısıtlı haklara sahip nobody kullanıcısı gibi) benimsendi.

Yıllardır kullanılan bu sistem basitliğiyle iyi iş gördü ve halen verimli olarak kullanılmaktadır. Ancak günümüzde yukarıda anlatılan sistematiğin dışında **root** haklarına ihtiyaç duymaksızın, uygulama spesifik olarak bazı ek yeteneklere, **Linux Capabilities API** üzerinden kavuşmak mümkündür.

Standartlar

Capabilities API, ihtiyaçlardan doğan bir çözüm yöntemidir. **POSIX.1e** versiyon 17 taslak dokümanında 25. bölüm bu konuya ayrılmıştır.

> Belgeyi http://wt.tuxomania.net/publications/posix.1e/download.html adresinden indirebilirsiniz.

Capabilities API, bazı önemli yetkilerin ayrı bir başlıkta değerlendirilmesine ve bu şekilde **root** olarak çalışma ihtiyacı olmaksızın ilgili işlemlere izin verilmesine olanak sağlamaktadır.

POSIX.1e'de tanımlanan yetkiler ve daha fazlasının geliştirimi Linux çekirdeği versiyon **2.6.26** ile tamamlanmıştır.

Linux Capability Modeli

Capabilities API'nin en kapsamlı gerçekleştirimi Linux altında olmuştur. Modern Linux dağıtımları da bu yeni modeli sistem genelinde olabildiğince kullanmaya çalışmaktadır.

Örnek verecek olursak, eğer eski tarihli bir Linux dağıtımı kullanıyorsanız (örnek olarak 2007 çıkışlı Debian 4.0) şaşırtıcı biçimde **ping** uygulamasının SUID yetkileriyle donatılmış olduğunu görebilirsiniz:

```
$ ls -l /bin/ping
-rwsr-xr-x 1 root root 33064 2007-01-31 01:22 /bin/ping
```

Bunun sebebi, `ping` uygulamasının çalışabilmesi için normalde sadece **root** kullanıcısına has olan **RAW** soket açabilmesinin gerekliliğidir. Eski Linux dağıtımlarında sorun uygulamaya SUID biti verilerek, normal kullanıcılar tarafından da kullanılabilmesi sağlanmıştır. Bu versiyonlarda SUID bitini uygulamadan kaldırıp normal bir kullanıcı olarak uygulamayı çalıştırmayı denediğimizde aşağıdaki gibi bir hata alırız:

```
$ ping 8.8.8.8
ping: icmp open socket: Operation not permitted
```

Oysa şu an kullandığınız Linux dağıtımında `ping` uygulaması muhtemelen SUID bitine sahip değildir:

```
$ ls -l /bin/ping
-rwxr-xr-x 1 root root 44104 Nov  8 19:04 /bin/ping
```

Buna rağmen normal kullanıcı olarak başarılı biçimde uygulamayı çalıştırabilmektesiniz. İşte bunu mümkün kılan mekanizma, ping uygulamasının **CAP_NET_RAW** özel yeteneğine sahip olmasıdır.

Uygulamanın sahip olduğu ek yetenekleri **getcap** komutu ile aşağıdaki gibi öğrenebiliriz:

```
$ sudo getcap /bin/ping
/bin/ping = cap_net_raw+ep
```

Process Capability Modeli

Linux gerçekleştiriminde her bir process'in 3 başlık altında capability yetenekleri gruplanır:

Capability	Açıklama
permitted	Bu kümede ilgili process'in izin verilen ek capability listesi bulunur. İzin verilmesi o an aktif olarak kullanılabileceği anlamına gelmeyebilir, ek bir işlemle buradaki yetkilerin etkin (effective) capability kümesine dahil edilmesi mümkündür.
effective	İlgili process'in o anki etkin capability listesini gösterir. Capability sistemini düzenleyen yardımcı fonksiyonlarla bir capability'den vazgeçilebileceği gibi tekrar geri de alınabilir. Ancak her durumda bu işlem sadece *permitted* grubunda zaten izin verilmiş olanlar arasından yapılabilir.
inheritable	Process tarafından yeni bir process çalıştırıldığında, yeni çalıştırılan process'in *permitted* listesine miras yoluyla aktarılacak capability listesini gösterir.

Çalışan process'ler için herhangi bir andaki *permitted*, *effective* ve *inheritable* capability listesi, `/proc/<PID>/status` dosyasında **CapPrm**, **CapEff** ve **CapInh** satırlarında bitmask olarak gösterilir. Ayrıca **CapBnd** satırıyla da capability sınır kontrolü (boundary) işleminde kullanılan bitmask yer alır. Örnek olarak çalışan kabuk uygulamamıza ait değerleri `/proc/self/status` dosyasından okuyalım:

```
$ cat /proc/self/status | grep Cap
CapInh:    0000000000000000
CapPrm:    0000000000000000
CapEff:    0000000000000000
CapBnd:    0000003fffffffff
```

Dosya Capability Modeli

Dosyalar için capability sisteminin çalışması, **Virtual File System** katmanında bu özelliklerin saklanabilmesi önşartına bağlıdır. Process modeline benzer şekilde 3 başlık altında dosyalar için capability yetenekleri gruplanır:

Capability	Açıklama
permitted	Bu kümede ilgili *executable* dosya çalıştırıldığında, process'in izin verilen (permitted) capability kümesine dahil edilecek yetenekler belirtilir.
effective	Bu başlık Process Capability modelinden farklı olarak dosyalar için 1 bitlik sadece aktif ya da değil bilgisini saklamaktadır. Eğer bit aktif ise, dosyanın *permitted* listesindeki tanımlı capability'ler, bu dosya çalıştırılıp bir process oluşturulduğunda ilgili process'in *effective* capability listesine otomatik olarak aktarılır. Bit aktif değilse dosya üzerindeki *permitted* capability'lerin çalışan process'e otomatik aktarımı gerçekleştirilmez. Bununla birlikte ilgili uygulamanın kodu Capability sistemiyle entegre çalışıyorsa, dosyanın *permitted* kümesindeki izinleri sistem çağrıları ile etkin hale getirebilir. Bu davranış şeklinin temel amacı, yazılım kodu seviyesinde capability sistemine özgü kod geliştirmesi içermeyen eski uygulamaların herhangi bir kaynak kod değişikliğine ihtiyaç kalmaksızın, capability sistemiyle çalışabilmesini sağlamaktır. Daha iyi yazılmış uygulamaların capability'leri sadece gerektiğinde kullanması beklenir. Bit aktif ise, *permitted* listesindeki tüm capability'ler uygulama çalışmaya başladığında aktifleştirilmektedir.
inheritable	Process modeline benzer şekilde ilgili dosya çalıştırılıp bir process üretildikten sonra eğer process içerisinden bir başka uygulama daha çalıştırılacak olursa, yeni çalışacak olan process'in *permitted* listesine miras yoluyla aktarılacak capability listesini gösterir.

Limitleme: Capability Bounding

Paylaşımlı Kütüphaneler

Paylaşımlı kütüphaneler, çalışabilir dosya içeriğine kopyalanmaksızın, proses adres alanına sonradan dinamik olarak yüklenen kütüphanelerdir. Birden çok proses aynı paylaşımlı kütüphane kodunu kullanabilmektedir. Statik kütüphanelerin aksine, aynı kütüphaneyi kullanan uygulamalar kütüphane kodunu direkt olarak kopyalamadıklarından, disk ve RAM üzerinde kazanç elde edilmektedir.

Paylaşımlı kütüphaneler ayrıca, program tarafından ihtiyaç halinde eklenti (*plugin*) olarak yüklenebilmekte ve bu sayede program ek özellikler kazanabilmektedir.

Uygulamanın kütüphane koduna katı bir şekilde bağlanmaması sayesinde, bir çok durumda uygulama yeniden derlenmeksizin kütüphane üzerinde değişiklik yapılabilmektedir.

Paylaşımlı kütüphaneleri yukarıda saydıklarımıza ek bir çok özelliği daha bulunmaktadır, yeri geldikçe bunlara değinmeye çalışacağız.

Bu bölümde, temel olarak, Linux altında, x86 ve x64 hedefli paylaşımlı kütüphanelerin nasıl oluşturulduğunu ve kullanıldığını inceleyeceğiz.

Öncesinde, kütüphanelere neden ihtiyaç duyulduğuna ve statik kütüphanelere bakmak faydalı olacaktır.

> Paylaşımlı kütüphaneler için İngilizce *shared libraries*, *shared objects*, *dynamic shared objects* (*DSOs*) ve *dynamically linked libraries* (*DLLs*) ifadelerinin kullanıldığını görmekteyiz.

Kütüphane Gereksinimi

Uygulamalar temel bazı işlemlere sıklıkla ihtiyaç duymaktadır. Sık kullanılan bu rutinlerin, her seferinde uygulamaya özel yeniden yazılması yerine, bir defa yazılıp ayrı modüllerde makina kodu düzeyinde saklanması kullanım kolaylığı sağlayacaktır. Aynı kodun defalarca kullanılması hata riskini azaltmakta, kodun her seferinde derlenme zorunluluğunu ortadan kaldırmakta, ayrıca kaynak kodun gizlenerek taşınabilmesine olanak sağlamaktadır.

Örneğin, C standartları giriş/çıkış, yazı işleme ve matematiksel işlemlerle ilgili çok sayıda fonksiyon tanımlamıştır. Bu fonksiyonlar bir şekilde programcıya temin edilmelidir.

Standart fonksiyonların derleyici tarafından, içsel olarak, sağlanması bir yöntem olabilir. Örneğin derleyici *sin* fonksiyon çağrısıyla karşılaştığında, bu fonksiyonun standart bir fonksiyon olduğunu anlayıp, fonksiyona ilişkin makina kodunu direkt olarak kendisi yazabilir. Programcı açısından bu yöntem oldukça kullanışlı olmasına rağmen derleyiciyi yazanlar için bir takım zorluklar barındırmaktadır. Yeni bir fonksiyon eklendiğinde, çıkarıldığında veya değiştirildiğinde derleyicinin yeni bir versiyonu çıkarılmalıdır. Buna rağmen bazı standart ve standart olmayan fonksiyonlar derleyiciler tarafından eklenti olarak sağlanmaktadır. Örneğin GNU C derleyicisi çok sayıda eklenti sunmaktadır.

Kısıtlı bir kullanıma sahip bu yöntemde ihtiyaç duyulan tüm fonksiyonların derleyici tarafından sağlanması mümkün olmadığından, kütüphane kullanımı bir diğer alternatif olarak karşımıza çıkmaktadır.

Konumuzun temeli olan paylaşımlı kütüphanelerden önce, daha basit bir kullanıma sahip, statik kütüphaneleri kısaca inceleyelim.

Statik Kütüphaneler

Statik kütüphaneleri incelemeden önce bir C kodunun çalıştırılabilir hale gelene kadar geçtiği aşamalara kısaca bakalım.

test.c adıyla sakladığımız basit bir örneği, gcc uygulamasına **--save-temps** anahtarını geçirerek, aşağıdaki gibi derleyebiliriz.

```
int main() {
    return 0;
}
```

```
$ gcc -otest test.c --save-temps
```

--save-temps anahtarı ile derleyicinin ürettiği ara kodlar uygun ad ve uzantılarla dosya sistemine kaydedilmektedir. *test.c* için derleyici aşağıdaki dosyaları üretecektir.

Dosya Adı	İçerik
test.i	Önişlemcinin ürettiği kod
test.s	Derleyicinin ürettiği sembolik makina kodları
test.o	Gerçek makina kodlarını içeren ELF formatlı amaç kod
test	Çalıştırılabilir ELF formatlı kod

Bir C kodu çalıştırılabilir hale gelene kadar, temel olarak, aşağıdaki aşamalardan geçmektedir.

- Önişlem aşaması
- Derleyeci tarafından sembolik makina kodlarının üretilmesi
- Assembler tarafından gerçek makina kodlarının üretilmesi
- Bağlayıcı tarafından çalıştırılan dosyanın üretilmesi

> **Not:** Komut satırından kullandığımız **gcc** uygulaması aslında derleyici değil, derleme sürecinde gerekli olan uygulamaları uygun sıra ve parametrelerle çağıran bir sürücü (*driver*) programdır.
>
> Fakat çoğu zaman detaya girmeden bütün süreçten derleme işlemi olarak bahsedeceğiz.

Bağlayıcı *.o* uzantılı amaç dosyaları (*object file*) birleştirerek nihai çalıştırılabilir dosyayı oluşturmaktadır. Bu durumda ihtiyaç duyulacak tüm fonksiyonları tek bir amaç dosyada toplamak mümkündür. Bir önceki örnek kodumuza *foo* fonksiyonu çağrısını eklediğimizi ve *foo* ile beraber diğer tüm fonksiyon tanımlarının *liball.o* amaç dosyasında olduğunu varsayalım.

```
void foo();

int main() {
    foo();
    return 0;
}
```

Kodu aşağıdaki gibi derleyebiliriz.

```
$ gcc -otest test.c liball.o
```

Bu yöntemde bir problemle karşılaşmaktayız, bağlayıcı tarafından, *liball.o* içeriğinin tamamı çalışabilir dosyaya kopyalanacak ve *test* uygulaması gereksiz yere büyüyecektir. Diğer bir alternatif ise birbiriyle ilişkili olduğu düşünülen fonksiyonları aynı amaç dosyada toplamak olabilir. Tek bir *liball.o* yerine *lib1.o*, *lib2.o*,..., *libN.o* dosyalarına sahip olduğumuzu ve *foo* tanımının *lib1.o* içinde olduğunu kabul edelim. Bu durumda daha küçük bir çalışabilir kod elde etmek mümkündür.

```
$ gcc -otest test.c lib1.o
```

Fakat bu yöntemde de çok sayıda amaç dosyayı yönetmek zorlaşacaktır. Hangi fonksiyonun hangi amaç dosya içinde olduğu bilinmeli ve çoğu durumda komut satırına birden çok amaç dosya geçirilmelidir.

Statik kütüphaneler bu problemleri gidermek için geliştirilmiştir. Birbiriyle ilişkili olan fonksiyonları içeren amaç dosyalar tek bir dosya halinde arşivlenerek saklanmaktadır. Statik kütüphane içinden yalnız gerekli amaç dosyalar çalıştırılabilir koda kopyalanmaktadır. Statik kütüphane oluşturmak için Linux altında **ar** aracı kullanılmaktadır. Örnek bir statik kütüphane aşağıdaki gibi oluşturulabilir.

```
$ ar rcs liball.a lib1.o lib2.o ... libN.o
```

Kütüphane dosyalarına, bir zorunluluk olmamasına karşın geleneksel olarak, **.a** (*archive*) uzantısı verilmektedir. Şimdi statik kütüphane kullanımı göstermek için basit bir örnek yapalım.

Örnek Kütüphane Kullanımı

Aşağıdaki örnek kodları sırasıyla *test.c*, *foo.c* ve *bar.c* isimleriyle saklayalım.

```
void foo();

int main() {
    foo();
    return 0;
}
```

```
#include <stdio.h>

void foo() {
    puts(__func__);
}
```

```
#include <stdio.h>

void bar() {
    puts(__func__);
}
```

foo.c ve *bar.c* dosyalarını kullanarak statik bir kütüphaneyi aşağıdaki gibi oluşturabiliriz. gcc uygulamasına geçirilen **-c** (*compile*) anahtarı, amaç dosyalar oluşturulduktan sonra bağlayıcının çağrılmasını engelemektedir.

```
gcc -c foo.c
gcc -c bar.c
ar rcs liball.a foo.o bar.o
```

test.c adıyla sakladığımız uygulamamız, *foo* fonksiyonun tanımına gereksinim duymaktadır. Kütüphane dosyamızı gcc uygulamasına geçirerek uygulamamızı derleyebilir ve çalıştırabiliriz. Bu örnek için kütüphane dosyasının çalıştığımız dizinde olduğunu varsayıyoruz.

```
$ gcc -otest test.c liball.a

$ ./test
foo
```

Daha önce kütüphane içinden yalnız gerekli modüllerin çalışabilir dosyaya kopyalandığını söylemiştik. Bu durumda yalnız, *foo* fonksiyonunun tanımının bulunduğu, *foo.o* dosyasının *test* uygulamasına kopyalanmasını bekliyoruz. *test* uygulaması içindeki sembollere bakarak bu durumu gözleyebiliriz. Bu amaçla *nm* uygulamasını kullanabiliriz.

```
$ nm test | grep foo
000000000040054b T foo

$ nm test | grep bar
```

nm çıktısındaki *T* (*Text*) harfi, fonksiyon tanımının dosya içinde bulunduğunu göstermektedir. Uygulama içinde *bar* fonksiyonuna ilişkin herhangi bir sembol dolayısıyla taşınmış kod bulunmamaktadır.

Burada bir noktaya dikkatinizi çekmek istiyoruz. Statik kütüphanelerin komut satırındaki sıralaması önem taşımaktadır. Aynı örneği şimdi kütüphane dosyasını uygulama kodundan daha önce geçirerek derleyelim. Bu durumda aşağıdaki gibi bir hata ile karşılaşmaktayız.

```
$ gcc -otest liball.a test.c
/tmp/ccjg4071.o: In function `main':
test.c:(.text+0xa): undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

foo fonksiyonunun tanımı *liball.a* içinde bulunmasına karşın, bağlayıcı bize *foo* fonksiyonunun tanımını bulamadığından şikayet etmekte. Bağlayıcı komut satırında gösterilen dosyaları sırasıyla okumakta ve tanımını bulamadığı referansları, daha sonra çözümlemek üzere, kaydetmektedir. Bağlayıcı tanımsız bir referans gördüğünde bunu daha sonra gördüğü dosyalarda aramakta, geriye doğru bir arama yapmamaktadır. Hata aldığımız örnek için bağlayıcı *foo* çağrısıyla karşılaştığında, kütüphane dosyası üzerinden zaten geçtiği için *foo* fonksiyonunun tanımını bulamamaktadır. Böyle bir durumda bağlayıcı istenilen sembolleri aramaya zorlanabilir. *undefined* anahtarı ile tanımı aranacak referansı açık bir şekilde geçirebiliriz.

```
$ gcc liball.a -otest test.c -Wl,--undefined=foo

$ ./test
foo
```

> **Not:** gcc'ye geçirilen **Wl** anahtarı devam eden anahtarların bağlayıcıya geçirileceğini göstermektedir.

Kod Referanslarının Ele Alınması

Bu bölüm daha sonraki incelemelerimize bir alt yapı niteliği taşımaktadır. İlk olarak derleme zamanında tanımı bulunan içsel (internal) referansların nasıl ele alındığına bakalım.

Kaynak kod içindeki referanslar kod ve data belleğindeki alanlara karşılık gelmektedir. Fonksiyon ve değişken isimleri, en nihayetinde birer adrese dönüşmesi beklenen, kaynak kod düzeyinde referanslardır. Derleme aşamasında, çoğu durumda, kaynak kod içerisindeki referanslar nihai adreslere dönüştürülemezler.

Derleyici oluşturduğu sembolik makina kodlarına, herhangi bir adres işlemiyle ilgilenmeksizin, yalnız referans isimlerini, tiplerini ve bilinirlik düzeylerini not etmektedir. Sonrasında **assembler**, **.o** uzantılı, **ELF** formatındaki, amaç dosyayı (*relocatable object file*) oluşturmaktadır. assembler, makina kodlarına ilave olarak, ELF dosyasında, ilgilendiği referanslara ilişkin kayıtların tutulduğu bir **sembol tablosu** da oluşturur. assembler çoğu durumda sembollere, bağlayıcının nihai adreslerle değiştirilmesi beklenen, geçici özel adresler atar. Dışsal bağlanıma kapalı *statik* fonksiyonlara ise nihai adresleri assembler tarafından atanabilmektedir. Aslında atanan bu adres fonksiyonun gerçek adresi değil, yer değişimini gösteren görece bir adrestir. Bu konuya daha sonra değineceğiz. Basit bir örnek üzerinden incelememize devam edelim.

Örnek kodları sırasıyla *test.c*, *tar.c* olarak isimlendirip 32 bit hedefli olarak derleyelim.

```
#include <stdio.h>

void tar();

void foo() {
}

static bar() {
}

int main() {
    foo();
    bar();
    tar();
    return 0;
}
```

```
#include <stdio.h>

void tar() {
}
```

```
$ gcc -otest test.c tar.c -m32 --save-temps
```

test.c uygulamasının sırasıyla *foo*, *bar* ve *tar* fonksiyonlarını çağırdığını görmekteyiz. *tar* fonksiyonunun tanımının başka modülde (tar.c) olduğuna ve *bar* fonksiyonunun **static** olarak tanımlandığına dikkat ediniz. *test.c* için üretilen sembolik makina kodlarında bu fonksiyonların nasıl geçtiğine bakalım.

```
$ cat test.s | grep foo
    .globl  foo
    .type   foo, @function
foo:
    .size   foo, .-foo
    call    foo
```

```
$ cat test.s | grep bar
    .type   bar, @function
bar:
    .size   bar, .-bar
    call    bar
```

```
$ cat test.s | grep tar
    call    tar
```

Nokta ile başlayan komutlar, gerçek makina komutlarına karşılık gelmeyen, assembler'ı bilgilendirme amaçlı kullanılan direktiflerdir. Sembolik makina çıktısında **.type** ve **.globl** direktiflerini görmekteyiz. *type* ile referansın tipi, *globl* ile referansın bilinirlik alanı gösterilmektedir. Derleyici, *test.c* dosyasında *foo* ve *bar* tanımlarını gördüğü için ilgili referansların fonksiyon olduğunu not etmiş. Ayrıca *static* olarak tanımlanan *bar* fonksiyonunun dışsal bağlanıma kapalı olduğunu, *foo* fonksiyonunun ise *.globl* direktifi sayesinde global bilinirlik alanına sahip olduğunu görmekteyiz. Başka modüldeki *tar* fonksiyonuna ilişkin ise yalnız fonksiyon çağrısını görmekteyiz. Şimdi assembler'ın sembolik makina kodlarına ilişkin oluşturduğu gerçek makina kodlarına bakalım. Bu amaçla **objdump** aracını kullanacağız. *test.o* için makina kodları aşağıdaki gibidir.

> **Not:** Sadeleştirme amacıyla, sembolik makina kodlarını incelerken *.cfi* ile başlayan assembler direktiflerini göz ardı edeceğiz.

```
$ objdump -d test.o

test.o:     file format elf32-i386

Disassembly of section .text:

00000000 < foo >:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   5d                      pop    %ebp
   4:   c3                      ret

00000005 < bar >:
   5:   55                      push   %ebp
   6:   89 e5                   mov    %esp,%ebp
   8:   5d                      pop    %ebp
   9:   c3                      ret

0000000a < main >:
   a:   8d 4c 24 04             lea    0x4(%esp),%ecx
   e:   83 e4 f0                and    $0xfffffff0,%esp
  11:   ff 71 fc                pushl  -0x4(%ecx)
  14:   55                      push   %ebp
  15:   89 e5                   mov    %esp,%ebp
  17:   51                      push   %ecx
  18:   83 ec 04                sub    $0x4,%esp
  1b:   e8 fc ff ff ff          call   1c < main+0x12 >
  20:   e8 e0 ff ff ff          call   5 < bar >
  25:   e8 fc ff ff ff          call   26 < main+0x1c >
  2a:   b8 00 00 00 00          mov    $0x0,%eax
  2f:   83 c4 04                add    $0x4,%esp
  32:   59                      pop    %ecx
  33:   5d                      pop    %ebp
  34:   8d 61 fc                lea    -0x4(%ecx),%esp
  37:   c3                      ret
```

Sol tarafta gerçek sağ tarafta ise sembolik makina kodları listelenmektedir. Sırasıyla *foo*, *bar* ve *tar* fonksiyon çağrılarına ilişkin komutlar aşağıdaki gibidir.

```
  1b:   e8 fc ff ff ff          call   1c <main+0x12>
  20:   e8 e0 ff ff ff          call   5 <bar>
  25:   e8 fc ff ff ff          call   26 <main+0x1c>
  2a:
```

e8, sembolik *call* komutuna karşılık gelen gerçek makina kodu, sonraki 4 byte ise adres bilgisidir. e8 makina komutu, fonksiyonun mutlak adresini almak yerine, komut göstericisinin (Instruction Pointer, Program Counter) gösterdiği değere görece bir adres almaktadır (IP Relative Addressing, PC Relative Addressing). Komut göstericisi, bir komut işletilirken, bir sonraki komutun başlangıç adresini tutmaktadır.

Statik *bar* fonksiyonu için **e0**, diğer fonksiyonlar için ise **fc** ile başlayan ve **ff** ile devam sayılar görüyoruz. Bu gösterimde e8 komutunun sağındaki ilk byte en düşük anlamlı byte'ı göstermektedir (Little Endian). Bu sayılar işaretli olarak ele alınmaktadır, en yüksek anlamlı bit değerinin 1 olması sayının negatif olduğunu göstermektedir. Negatif sayılar bellekte ikiye tümleyen (Two's Complement) şeklinde tutulmaktadır. Bu durumda **fc ff ff ff** ve **e0 ff ff ff** sayıları sırasıyla **-0x4** ve **-0x20** sayılarına karşılık gelmektedir.

> **Not:** Bir sayının ikiye tümleyenini bulmak için, ikili sayı sisteminde temsil edilen sayının, 1 olan bitleri 0 ve 0 olan bitleri 1 yapılarak önce bire tümleyeni alınır. Sonrasında elde edilen sonuç 1 ile toplanarak ikiye tümleyenine ulaşılır.

Amaç kodlar içinde oldukça sık rastlanan **fc** sayısının rastgele bir sayı olmadığına dikkat ediniz. Bir sonraki makina komutunun adresinden 4 byte geri geldiğimizde, ilgili makina komutunun başlangıç adresinden bir sonraki adrese ulaşmaktayız. Böyle bir çağrı yapılması durumunda işlemci üzerinde *illegal instruction* hatası oluşacağı açıktır. **fc** ile başlayan bu değerler, daha sonra bağlayıcı tarafından değiştirilmesi beklenen geçici değerlerdir.

foo ve *tar* fonksiyonları için bu değerler *objdump* tarafından hesaplanarak, call komutunun operandı olarak, **1c** ve **26** şeklinde gösterilmektedir. Statik *bar* fonksiyonu için **5** değerine ulaşıldığına ve *bar* fonksiyonunun 5 numaralı adresten başladığına dikkat ediniz.

nm ve **readelf** araçları ile amaç dosya içindeki sembolleri aşağıdaki gibi listeleyebiliriz.

```
$ nm test.o
00000005 t bar
00000000 T foo
0000000a T main
         U tar
```

Tarif	Anlamı
t	Fonksiyon tanımının ilgili modül içinde olduğunu fakat dışsal bağlanıma kapalı olduğunu gösterir
T	Fonksiyon tanımının ilgili modül içinde olduğunu ve dışsal bağlanıma açık olduğunu gösterir
U	Fonksiyon tanımının ilgili modül içinde olmadığını gösterir

Benzer sonuçlara *readelf* ile aşağıdaki gibi ulaşabiliriz. Dışsal bağlanıma kapalı *bar* **LOCAL** olarak gösterilmektedir.

```
$ readelf -s test.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
    ...
     5: 00000005     5 FUNC    LOCAL  DEFAULT    1 bar
    ...
     9: 00000000     5 FUNC    GLOBAL DEFAULT    1 foo
    10: 0000000a    46 FUNC    GLOBAL DEFAULT    1 main
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND tar
```

Dana önce, ELF formatındaki amaç dosya içinde assembler tarafından oluşturulan, bir sembol tablosu tutulduğunu söylemiştik. ELF içinde ayrıca, assembler tarafından geçici adres verilen referanslara ilişkin bilgilerin tutulduğu yeniden konumlandırma (relocation) bölümü de bulunmaktadır.

Bağlayıcı nihai adreslerini atayacağı sembolleri *relocation* bölümüne bakarak bulmaktadır. *relocation* bölümünü aşağıdaki gibi listeleyebiliriz. *relocation* bölümünde, statik *bar* fonksiyonunun bulunmadığına dikkat ediniz.

```
$ readelf -r test.o

Relocation section '.rel.text' at offset 0x24c contains 2 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000001c  00000902 R_386_PC32        00000000   foo
00000026  00000b02 R_386_PC32        00000000   tar
```

Şimdi bağlanma aşamasından sonra uygulamanın makina kodlarına bakalım.

```
080483eb <foo>:
 80483eb:       55                      push   %ebp
 80483ec:       89 e5                   mov    %esp,%ebp
 80483ee:       5d                      pop    %ebp
 80483ef:       c3                      ret

080483f0 <bar>:
 80483f0:       55                      push   %ebp
 80483f1:       89 e5                   mov    %esp,%ebp
 80483f3:       5d                      pop    %ebp
 80483f4:       c3                      ret

080483f5 <main>:
 ...
 8048406:       e8 e0 ff ff ff          call   80483eb <foo>
 804840b:       e8 e0 ff ff ff          call   80483f0 <bar>
 8048410:       e8 0e 00 00 00          call   8048423 <tar>
 ...

 08048423 <tar>:
 8048423:       55                      push   %ebp
 8048424:       89 e5                   mov    %esp,%ebp
 8048426:       5d                      pop    %ebp
 8048427:       c3                      ret
```

Fonksiyonlara gerçek adresleri atanmasına karşın, *bar* fonksiyonu çağrısına ilişkin makina komutunun değişmediğini görüyoruz. İlgilendiğimiz fonksiyonların adresleri belirlendiğinden (*symbol resolution*) artık *relocation* bölümünde kayıtları bulunmamaktadır.

```
$ readelf -r test | grep foo
```

Bir uygulamanın paylaşımlı bir kütüphaneye bağımlı olması durumunda ise, uygulama içerisinde, tanımı kütüphanede olan, dışsal (external) referanslar olacaktır. Derleme zamanında bağlayıcı tarafıdan çözümlenemeyen bu referansların, yükleme zamanında dinamik bağlayıcı tarafından çözümlenmesi beklenmektedir.

Paylaşımlı Kütüphanelerin Oluşturulması

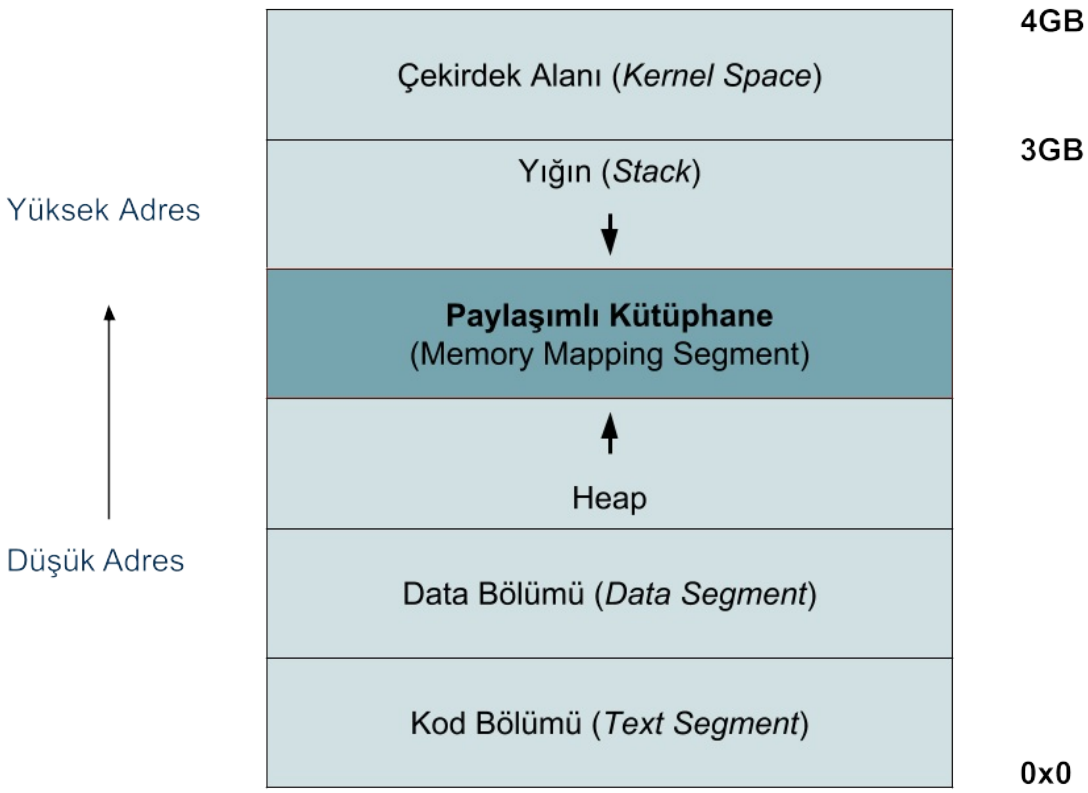
Bu bölümde 32 ve 64 bitlik sistemler için paylaşımlı kütüphaneleri nasıl oluşturabileceğimize ve arka planına bakacağız.

32 Bitlik Sistemlerde Paylaşımlı Kütüphanelerin Oluşturulması

Paylaşımlı kütüphane kodunun çalışabilir dosyaya kopyalanmadığını ve prosesin adres alanına sonradan yüklendiğini daha önce söylemiştik. Paylaşımlı kütüphanelerle ilgili temel zorluk derleme zamanında kütüphanenin nereye yükleneceğinin bilinememesidir.

32 bitlik bir sistem için bir prosesin bellekteki görüntü kabaca şekilde gösterildiği gibidir.

> **Not:** Burada prosesin fiziksel bellekteki gerçek görüntüsünü değil sanal bellek görüntüsünü kast ediyoruz.



Paylaşımlı kütüphanelerin *Memory Mapping* alanı içinde yüklenecekleri alan belli olmasına karşın, prosesten prosese farklılık gösterebilmektedir. Prosesler bu alana farklı sayı ve büyüklükte kütüphaneleri yükleyebilmektedirler. Dolayısıyla bir kütüphanenin tüm proseslerde aynı adrese yükleneceği, kolaylıkla, garanti altına alınamamaktadır. Hatta, adres alanı randomizasyonu (*Address Space Layout Randomization*) yapan modern sistemlerde, aynı uygulamaya ait prosesler bile paylaşımlı kütüphaneleri farklı adreslere yüklemektedir. İşetim sistemi çekirdeği, güvenlik amaçlı olarak, prosese ait yığın, heap ve kütüphane için ayrılmış alanları farklı adreslerden başlatmaktadır.

Bu noktada, bir kütüphanenin dinamik olarak yüklenebilmesinin onun paylaşımlı olduğu anlamına gelmediğini söyleyelim. Bir kütüphane dinamik olarak yüklenebilmesine karşın kütüphane kodu birden çok proses tarafından paylaşılamayabilir. Örneğin *Windows Vista* öncesi *dll* dosyaları paylaşıma izin vermemektedir.

Bir kütüphanenin bir proses tarafından dinamik olarak yüklenerek kullanılabilmesi için genel olarak aşağıdaki yöntemler kullanılabilir.

- Her kütüphane önceden belirlenmiş değişmez (*fixed*) bir adrese yüklenebilir
- Kütüphane yüklenirken kod bölümü üzerinde değişiklikler yapılabilir (*Load-time relocation*)
- Kütüphane kodu konumdan bağımsız olarak yazılabilir

Şimdi sırasıyla bu yöntemlere bakalım.

Kütüphanelerin Değişmez Adreslere Yüklenmesi

Daha önceleri *Windows* sistemlerinde kullanılmasına karşın, günümüzde Linux ve Windows sistemlerinde bu yöntem kullanılmamaktadır. Kütüphanelerin sabit adreslere sahip olması, adres alanı paylaşımında çakışmalara sebep olmaktadır. Bir kütüphane tarafından kullanılan aralık diğer kütüphaneler ve uygulamanının kendisi tarafından kullanılmamalıdır. Bu yöntem kütüphanelerin birbirine olan ve uygulamanın kütüphanelere olan bağımlılığını daha katı bir hale getirmekte ve yönetilmesi zor bir hal almaktadır.

Yükleme Zamanı Konumlandırma (*Load-time Relocation*)

Çalıştırılabilir dosya oluşturulurken, içsel sembollerin bağlayıcı tarafından yeniden nasıl konumlandırıldığını (*relocation*) daha önce incelemiştik. Assembler tarafından adresleri belirlenemeyen sembollere nihai adesleri bağlayıcı tarafından verilmekteydi. Çalışabilir dosyanın aksine kütüphanenin nereye yükleneceği derleme zamanında bilinemeyeceği için, bağlayıcı bu durumda tüm sembolleri çözümleyemeyecek ve bazı sembolleri daha sonra çözümlenmeleri için bırakacaktır. Derleme zamanında çözümlenemeyen bu semboller yükleme zamanında dinamik bağlayıcı tarafından çözümlenmektedir.

Bu yöntemde kütüphaneler dinamik olarak yüklenebilmesine karşın, birden çok proses tarafından ortak kullanılmalarında zorluklar barındırmaktadır. Kütüphaneler belleğe yüklendikten sonra kendilerini kullanacak olan prosese uygun olarak konumladırma işleminden geçmektedir.

Windows altındaki **dll** kütüphaneleri bu yöntemle gerçeklenmekte ve işletim sistemi desteğiyle birden çok proses tarafından paylaşılabilmektedir. Tercih edilmemesine karşın, Linux altında da 32 bitlik sistemler için bu yöntemle dinamik yüklenen kütüphaneler oluşturulabilmekte fakat birden çok proses tarafından paylaştırılamamaktadır.

Linux altında bu yöntemle dinamik kütüphanelerin nasıl oluşturulduğunda basit bir örnek üzerinden bakalım. Uygulama ve kütüphane kodlarına sırasıyla *driver.c*, *modul.c* adlarını verelim.

```
void foo();

int main() {
    foo();
    return 0;
}
```

```
#include <stdio.h>

void bar() {
    puts(__func__);
}

void foo() {
    bar();
}
```

Kütüphane dosyasını aşağıdaki gibi oluşturabiliriz.

```
$ gcc -shared -olibmodul.so modul.c -m32
```

Şimdi kütüphane bağımlı uygulamamızı derleyip çalıştırabiliriz.

```
$ gcc -odriver driver.c -L. -lmodul -m32
$ LD_LIBRARY_PATH=. ./driver
bar
```

shared anahtarı ile gcc'ye hedefin bir kütüphane dosyası olduğu bildirilmektedir. Ürettiğimiz kütüphane dosyası şu an için standart dışı bir dizinde bulunduğu dan, LD_LIBRARY_PATH çevre değişkeni ile dinamik bağlayıcıya kütüphane dosyamızın bulunduğu dizini

gösteriyoruz. Bu konuya daha sonra değineceğiz.

64 bitlik mimari için kütüphane dosyalarının bu şekilde oluşturulmasına izin verilmemektedir. 64 bitlik mimaride dinamik bağlayıcı kod üzerinde *relocation* işlemini desteklememektedir. Daha derleme aşamasında bağlayıcı tarafından aşağıdaki gibi bir hata mesajı verilmektedir.

```
$ gcc -shared -olibmodul.so modul.c
/usr/bin/ld: /tmp/cco6wTkV.o: relocation R_X86_64_32 against `.rodata' can not be used
 when making a shared object; recompile with -fPIC
/tmp/cco6wTkV.o: error adding symbols: Bad value
collect2: error: ld returned 1 exit status
```

Kütüphane Kodunun Pozisyon Bağımsız Yazılması (PIC, Position Independent Code)

Linux altında, 32 ve 64 bitlik sistemler için, birden çok proses tarafından paylaşılabilen kütüphaneler bu yöntemle oluşturulmaktadır. Çalışabilir dosyaların aksine paylaşımlı kütüphanelerin nereye yükleneceğinin derleme zamanında bilinemediğinden daha önce söz etmiştik.

Her proses gerçekte fiziksel belleğin farklı alanlarını kullanıyor olmasına karşın, sanal bellek kullanımı sayesinde, tüm bellek kendisine aitmiş gibi görmektedir. Program kodu içerisinde gerçek adresler yerine mantıksal adresler yer almakta, bu sayede bağlayıcı tarafından tüm uygulamalara aynı adresler atanabilmekte ve çalışma zamanında herhangi bir yeniden konumlandırma işlemine ihtiyaç duyulmamaktadır. Paylaşımlı kütüphanelerin ise farklı proseslerin farklı adres alanlarına yüklenmesi ve bellekte bir adet kopyası olması istenmektedir. Kütüphane kodunun pozisyondan bağımsız yazılması durumunda bu özellik sağlanabilmektedir.

Aslında kütüphanenin paylaşımlı olması kod bölümüne ilişkindir. Kütüphanenin kod bölümünün, fiziksel olarak, bir örneği olmasına karşın *data* alanının her proses için bir kopyası çıkarılmaktadır. Kütüphanenin data alanı prosesler arasında paylaşılmaz, bu sebeple Linux altında paylaşımlı kütüphaneler ortak bir data alanı üzerinden haberleşemezler. Başlangıçta tüm prosesler aynı data alanını görmelerine karşın, bu alan üzerine bir yazma işlemi yapmak istediklerinde data alanının fiziksel olarak bir kopyası çıkarılmaktadır. Bu işlem *Copy-on-write* olarak isimlendirilmekte ve sanal bellek kullanımıyla mümkün olmaktadır. Proses fiziksel bellekteki başka bir alana aynı sanal adreslerle ulaşmaya devam etmektedir. Bu sayede data alanından yalnız okuma yapan prosesler aynı alanı kullanabilmekte gereksiz yere aynı bilgiler kopyalanmamaktadır.

> **Not:** Windows altında prosesler dll'ler üzerinde paylaşımlı bölümler oluşturularak haberleşebilmektedir.

Bir örnek üzerinden kodun pozisyon bağımlılığını inceleyelim. Aşağıdaki örneği *main.c* adıyla saklayıp derleyebilirsiniz.

```
int glob;

void foo() {
}

int main() {
    foo();
    int local = 1;
    global = 1;
    return 0;
}
```

```
$ gcc -omain main.c -m32 --save-temps
```

main ve *foo* fonksiyonları için üretilen gerçek ve sembolik makina komutlarının aşağıdaki gibi olduğunu görmekteyiz.

```
$ objdump -d main
```

```
080483eb <foo>:
 80483eb:       55                      push   %ebp
 80483ec:       89 e5                   mov    %esp,%ebp
 80483ee:       5d                      pop    %ebp
 80483ef:       c3                      ret

080483f0 <main>:
1.  80483f0:       55                      push   %ebp
2.  80483f1:       89 e5                   mov    %esp,%ebp
3.  80483f3:       83 ec 10                sub    $0x10,%esp
4.  80483f6:       e8 f0 ff ff ff          call   80483eb <foo>
5.  80483fb:       c7 45 fc 01 00 00 00    movl   $0x1,-0x4(%ebp)
6.  8048402:       c7 05 20 a0 04 08 01    movl   $0x1,0x804a020
7.  8048409:       00 00 00
8.  804840c:       b8 00 00 00 00          mov    $0x0,%eax
9.  8048411:       c9                      leave
10. 8048412:       c3                      ret
```

Komut adreslerinin başına, incelememizi kolaylaştırmak için, numaralar verdik. *main* fonksiyonunu 4. satırda *foo* fonksiyonunu çağırdığını görüyoruz.

```
e8 f0 ff ff ff
```

Daha önce, referansların çözümlenmesinden bahsettiğimiz bölümde, **e8** makina komutunun mutlak adres yerine göreli adres aldığını (*Relative Call*) söylemiştik. Kod içinde *foo* fonksiyonunun adresi, 80483eb olarak açık bir şekilde yazılmak yerine, bir sonraki komut adresi yani IP (Instruction Pointer) göreli adres yazılmış. Sağ taraftaki sembolik makina kodundaki değer *objdump* tarafından hesaplanmaktadır, işlemci de benzer bir işlem yaparak gerçek adrese ulaşmaktadır.

> **Not:** Daha önce negatif sayıların bellekte ikiye tümleyen şeklinde tutulduğundan bahsetmiştik. İlk önce 1'e tümleyenini alıyor sonrasında 1 ile topluyorduk. Aslında yapılmak istenen sayının alabileceği maksimumum değerin 1 fazlasıyla olan farkını bulmakdır. Bu örnek için call f0 ff ff ff komutu, temsili olarak, aşağıdaki gibi ele alınmaktadır.
>
> 0xffffffff - 0xfffffff0 = 0xf
>
> 0xf + 0x1 = 0x10
>
> 0x80483fb - 0x10 = 0x80483eb (*IP değeri - 0x10*)
>
> call 0x80483eb

`5.` satırda yerel *local* değişkenine 1 sayısının atandığını görüyoruz. Yerel değişkenler yığın üzerinde gerektiğinde oluşturulmakta ve yazmaç göreli olarak konumlandırılmaktadır. Değişmez (*hardcoded*) adresleri bulunmamaktadır.

`6.` satırda ise *glob* isimli global değişkene 1 değerinin atandığını görüyoruz.

```
c7 05 20 a0 04 08 01    movl   $0x1,0x804a020
```

Sembolik makina eşdeğerinde *glob* değişkenini gösteren adresin gerçek makina kodunda *Little Endian* olarak kodlandığını görüyoruz.

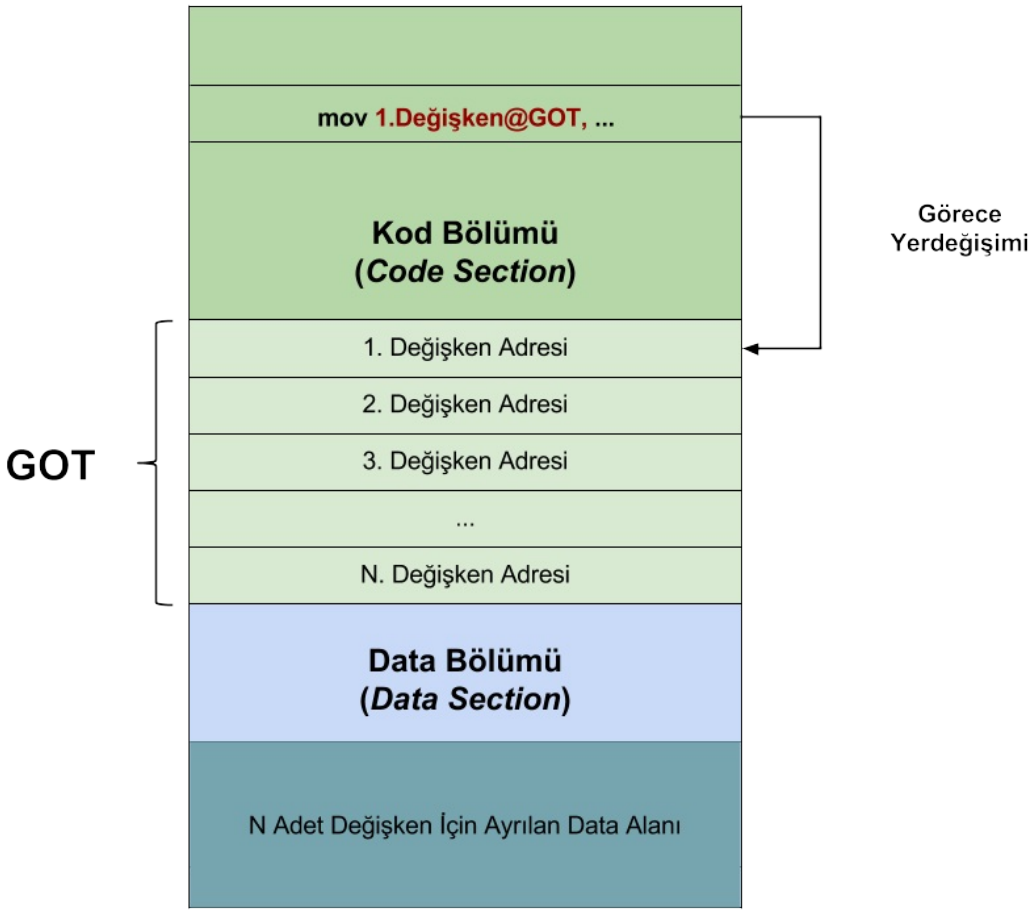
x86 mimarisinde, kod referanslarının aksine, data referanslarında göreli adresler kullanılamamakta, adresler mutlak olmak zorundadır. Kod içindeki bu değişmez adres kodu pozisyon bağımlı hale getirmektedir. *main* ve *foo* fonksiyonlarının yüklendikleri adresler değişse bile aralarındaki uzaklık değişmeyeceğinden bir problem olmayacak, *main* içinde *foo* problemsiz bir şekilde çağrılmaya devam edecektir. Benzer şekilde dallanma komutları (*jmp,...*) ve yerel değişkenler için de bir problem çıkmayacaktır. Dallanma komutları hem göreli hem de mutlak adresleri desteklemektedir.

Özetleyecek olursak, kod içinde data referansları bulunması durumunda bu referanslar *.data* alanının yükleneceği adrese bağımlı olacaklar ve uygulama istenilen bir bellek alanına yüklenemeyecektir.

Şimdi, paylaşımlı kütüphaneler için, pozisyon bağımsız bir kodun nasıl yazıldığına bakalım. İlk olarak data sonrasında kod referanslarının nasıl ele alındığına bakacağız.

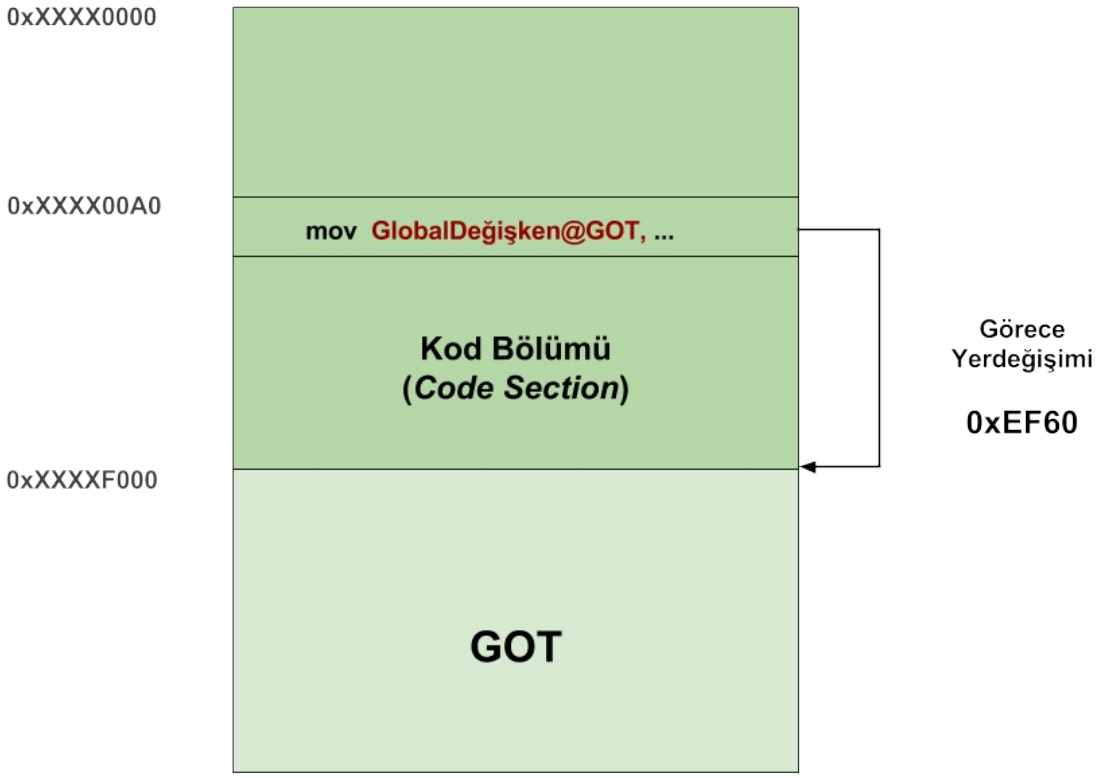
Data Referanslarının Ele Alınması

Global data erişimi, değişmez adresler kullanılarak yapılmak yerine, **GOT** (*Global Offset Table*) adı verilen bir tablo üzerinden yapılmaktadır. Bu durumu görsel olarak aşağıdaki gibi gösterebiliriz.



GOT tablosu data alanın başında bulunmaktadır. Şekilde kod, GOT ve data alanı içinde yer alan global değişkenler için ayrılan alanlar gösterilmiştir. Kod içinde global değişkenlere direkt adresleriyle ulaşıldığında, kodun data alanının pozisyonuna bağlı olduğunu hatırlayınız. Burada ise GOT tablosu üzerinden değişken adreslerine ulaşılmaktadır. GOT tablosu kütüphane yüklenirken dinamik bağlayıcı tarafından doldurulmaktadır. Data alanının başlangıç adresine göre, GOT tablosunun girişleri global değişkenlerin adreslerini gösterecek şekilde doldurulmaktadır. Bu soyutlama sayesinde kodun data alanının yerleşimine bağımlılığı ortadan kaldırılmaktadır.

Bu aşamada, GOT tablosuna nasıl ulaşılacağı ayrı bir problem olarak ortaya çıkmaktadır. Data ve kod alanları, dinamik bağlayıcı tarafından, istenilen bir adresten başlamak üzere yüklenecekse GOT tablosunun da değişmez bir adresi olamaz. GOT tablosunun adresine derleyicinin ekstradan yazdığı kodlar sayesinde ulaşılmaktadır. Temelde, kod ve data alanlarının boyutlarının ve birbirlerine göre konumlarının biliniyor olmasından faydalanılmaktadır. Bu durumu örnek bir şekil üzerinden inceleyelim.



GOT tablosu derleme zamanında bağlayıcı tarafından oluşturulmaktadır. Bağlayıcı derleme zamanında kod ve GOT bölümlerinin boyutlarını ve birbirlerine göre olan göreli uzaklıklarını bilmektedir. Yükleme zamanında başlangıç adresleri değişmesine karşın, bu bölümler arasındaki mesafe sabit kalmaktadır. Şekilde dinamik yükleyici tarafından belirlenecek *0xXXXX* ile başlayan adresler ne olursa olsun, örnek komut ile GOT başlangıcı arasındaki uzaklık sabit (*0xEF60*) kalacaktır. Bu bilgi bağlayıcı tarafından kullanılacaktır. Örnek komutun kendi adresinin bilinmesi durumunda GOT adresine ulaşılabilir. Bu aşamada ilk olarak derleyicinin pozisyon bağımsız bir kodu nasıl yazdığına bir örnek üzerinden bakalım. Örnek kodu *test.c* adıyla saklayıp aşağıdaki gibi derleyebilirsiniz.

```
int global;

void foo() {
    global = 111;
}
```

```
$ gcc -c -fPIC test.c -m32 --save-temps
```

Pozisyon bağımsız kod üretebilmek için derleyiciye **fPIC** anahtarını geçirmelisiniz. Sembolik makina kodlarına baktığımızda derleyicinin *foo* için aşağıdaki gibi bir kod yazdığını görmekteyiz. Ayrıca derleyici tarafından, dışsal bağlanıma kapalı, **__x86.get_pc_thunk.cx** isimli bir fonksiyon yazılmış. İncelememizi kolaylaştırmak için *foo* içindeki komutları numaralandırdık.

```
foo:
.LFB0:
1.    pushl   %ebp
2.    movl    %esp, %ebp
3.    call    __x86.get_pc_thunk.cx
4.    addl    $_GLOBAL_OFFSET_TABLE_, %ecx
5.    movl    global@GOT(%ecx), %eax
6.    movl    $111, (%eax)
7.    popl    %ebp
8.    ret

__x86.get_pc_thunk.cx:
.LFB1:
    movl    (%esp), %ecx
    ret
```

Kodumuzu *fPIC* anahtarı geçirmeden derleseydik, derleyici tarafından, aşağıdaki gibi bir sembolik makina kodu üretilecekti. *global* sembolüne, assembler tarafından geçici bir adres atanacak, sonrasında bağlayıcı tarafından nihai adresi atanacak ve kod data alanının adresine bağımlı olacaktı.

```
foo:
.LFB0:
    pushl   %ebp
    movl    %esp, %ebp
    movl    $111, global
    popl    %ebp
    ret
```

Pozisyon bağımsız kodumuzu incelemeye devam edelim. İlk olarak *__x86.get_pc_thunk.cx* fonksiyonuna bakalım. *__x86.get_pc_thunk.cx* fonksiyonu, komut göstericisinin değerini yani bir sonraki komutun başlangıç adresini elde etmek için kullanılmaktadır. Komut göstericisinin (*eip*) değerini direkt olarak öğrenmenin bir yolu bulunmamaktadır. Fonksiyon çağrıldıktan sonra yığının tepesindeki geri dönüş adresini *ecx* yazmacına yazmış ve sonrasında *ret* komutu ile bu değeri yığından çekerek geri dönmüştür. *call* komutuyla bir fonksiyon

çağrıldığında, bir sonraki komutun başlangıç adresi gizli bir şekilde yığına atılmaktadır. Bu durumda *__x86.get_pc_thunk.cx* fonksiyonu döndüğünde *ecx* yazmacında `4` numaralı satırdaki makina kodunun adresi bulunacaktır.

`4` numaralı komutta *ecx* değeriyle yani komutun kendi adresiyle öntanımlı *GLOBAL_OFFSET_TABLE* sembolünün değerinin toplandığını görüyoruz. Kod bölümündeki herhangi bir komut ile GOT arasındaki mesafenin bağlayıcı tarafından bilindiğini daha önce söylemiştik. Derleyici bu sembole, sembolün kullanıldığı komut ile GOT arasındaki mesafeyi atayacaktır. Bu mesafenin çalışma zamanında da değişmediğini hatırlayınız. Bu durumda *ecx* yazmacında GOT başlangıç adresi bulunacaktır.

`5` numaralı komutta GOT tablosundan *global* değişkenine ilişkin girişin değerine yani *global* değişkeninin adresine ulaşılmakta ve bu değer *eax* yazmacında saklanmaktadır.

`6` numaralı komut ile *dereference* işlemi yapılarak *global* değişkenine 111 değeri atanmaktadır.

Elde ettiğimiz pozisyon bağımsız amaç kod ile bir paylaşımlı kütüphaneyi aşağıdaki gibi oluşturabiliriz.

```
$ gcc -shared -olibtest.so test.o -m32
```

Pozisyon bağımsız derleme ve bağlama işlemlerini tek seferde aşağıdaki gibi de yapmak mümkündür.

```
gcc -fPIC -shared -olibtest.so test.c -m32
```

gcc tarafından, *fPIC* derleyeciye *shared* ise bağlayıcıya geçirilmektedir. Son olarak kütüphane içindeki gerçek makina kodlarına bakalım.

```
$ objdump -d libtest.so
```

```
00000515 <foo>:
 515:   55                      push   %ebp
 516:   89 e5                   mov    %esp,%ebp
 518:   e8 14 00 00 00          call   531 <__x86.get_pc_thunk.cx>
 51d:   81 c1 e3 1a 00 00       add    $0x1ae3,%ecx
 523:   8b 81 ec ff ff ff       mov    -0x14(%ecx),%eax
 529:   c7 00 6f 00 00 00       movl   $0x6f,(%eax)
 52f:   5d                      pop    %ebp
 530:   c3                      ret
```

Bağlayıcının, komutla GOT arasındaki göreli yer değişimini gösteren, *GLOBAL_OFFSET_TABLE* sembolüne *0x1ae3* değerini atadığını, *global* değişkeni için ise GOT içinde 14 numaralı girişi ayırdığını görüyoruz. Dinamik bağlayıcı yükleme zamanında GOT tablosunun 14. elemanına *global* değişkeninin adresini yazmalıdır, bunun için derleme zamanında ELF içindeki *relocation* bölümüne aşağıdaki gibi bir kayıt eklenmiştir.

```
$ readelf -r libtest.so
 Offset     Info    Type            Sym.Value  Sym. Name
00001fec  00000906 R_386_GLOB_DAT    0000201c   global
```

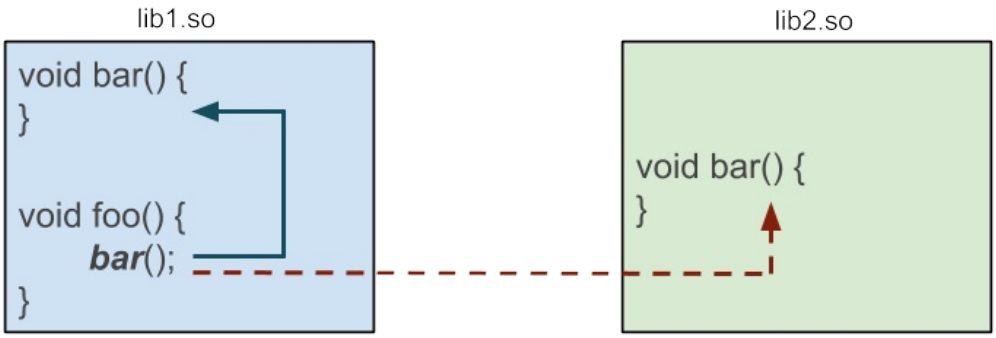
global değişkenine ilişkin kaydın ELF içindeki ofset değerine, ilk olarak ELF içinde GOT tablosunun adresini, sonrasında *global* için ayrılan girişi bularak ulaşabiliriz.

```
0x51d + 0x1ae3 = 0x2000
0x2000 - 0x14 = 0x1fec
```

Fonksiyon Çağrılarının Ele Alınması

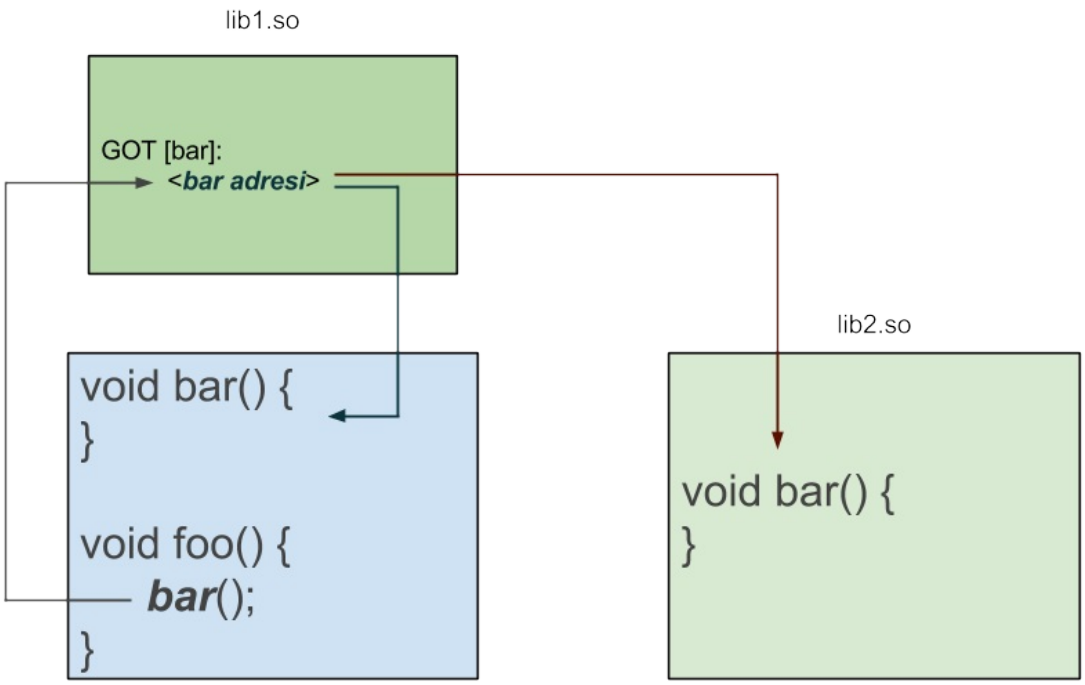
Data referanslarının aksine, fonksiyon çağrılarının değişmez mutlak adresler yerine göreli yer değişimi kullanılarak yapıldığını daha önce incelemiştik. Fonksiyonların bellekte nereye yüklendiklerinden bağımsız olarak aralarındaki mesafe korunmakta ve fonksiyon çağrıları sorunsuz olarak yapılabilmektedir. Fonksiyon çağrıları zaten pozisyon bağımsız olduğu için bu durumda birşey yapılmasına gerek olmadığını düşünebilirsiniz. Gerçekte ise ilk bakışta karmaşık görünen bir yöntem kullanılmaktadır.

Gerçekte kullanılan yönteme ve gerekçesine adım adım gidelim. İlk olarak, paylaşımlı kütüphane içindeki fonksiyonların yönlendirilebilmesi istenmektedir. Aşağıdaki şekli inceleyiniz.



Bazı durumlarda *lib1.so* içindeki *foo* fonksiyonunun kendi *bar* fonksiyonu yerine *lib2.so* içindeki *bar* fonksiyonunu çağırması istenmektedir. Bu özelliğin öneminden daha sonra bahsedeceğiz. *foo* kodunda *bar* fonksiyonunun direkt olarak, aralarındaki yer değişimi ile çağrılması durumunda bu mekanizma sağlanamayacaktır.

Bu durumda data referanslarına benzer şekilde GOT tablosunda fonksiyonlar için girişler ayrılabilir ve yükleme zamanında dinamik bağlayıcı tarafından bu girişlere uygun değerler verilerek fonksiyon yönlendirmesi sağlanabilirdi. Çağrılar direkt yapılmak yerine, GOT üzerinden indirekt yapılarak istenilen soyutlama sağlanmış olurdu. Bu duruma ilişkin aşağıdaki şekli inceleyiniz.



Kütüphane yüklenirken, kullanılacak olan *bar* fonksiyonunun adresi ilgili GOT girişine yazılabilir ve *lib1.so* için, sonraki *bar* fonksiyonu çağrıları bu giriş üzerinden dolaylı olarak yapılabilirdi. Hedeflenen çalışma şekline ulaşmamıza karşın bu yöntemin bir dezavantajı bulunmaktadır. Kütüphane yüklenirken kütüphane içindeki tüm fonksiyon çağrıları için GOT girişleri doldurulmalıdır. Bunun için dinamik bağlayıcı yükleme zamanında ELF tablolarını okumalı, ilgili sembollerin adreslerini çözümlemeli ve girişlere yazmalıdır. Kütüphane içinde özellikle hata işleyen fonksiyonlar olmak üzere birçok fonksiyon çok az veya hiç çağırılmamaktadır. Bu durumda en baştan tüm fonksiyonlarla ilgili işlem yapmak kütüphanenin yüklenme sürecini dolayısıyla uygulamanın açılma süresini arttıracaktır. Bu sebeple fonksiyonlara ilişkin adres çözümlemesinin gerekli olduğu durumda yapılabilmesi istenmiştir. Bu çözümleme işlemi *lazy binding* olarak isimlendirilmektedir.

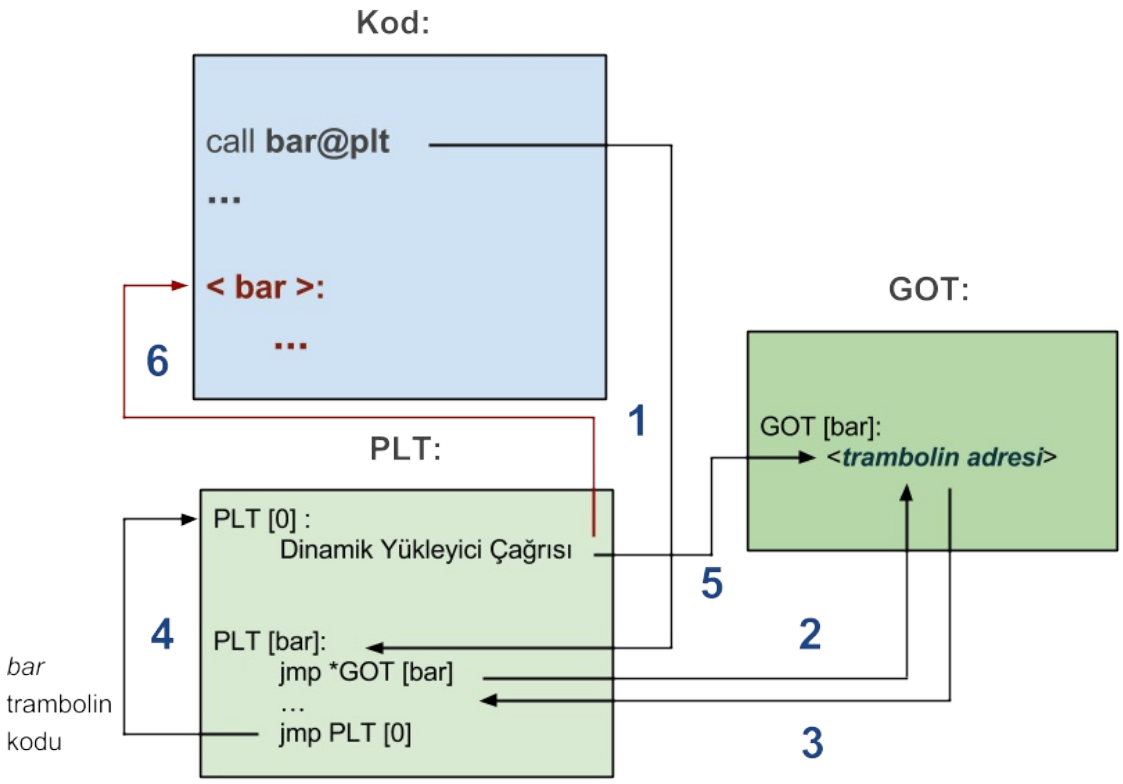
Bu amaçla bir soyutlama katmanı daha getirilmiştir. Fonksiyon çağrıları GOT içindeki adresler üzerinden dolaylı yapılmak yerine öncesinde, gerçek fonksiyonu çağırmaktan sorumlu, küçük bir kod çağırılmaktadır. Bu amaçla bağlayıcı tarafından **PLT** (*Procedure Linkage Table*) adı verilen bir kod alanı daha oluşturmaktadır. PLT girişlerindeki kodlar trambolin kodu olarak isimlendirilmektedir ve her bir fonksiyon çağırısı için bir giriş tutulmaktadır.

Bir fonksiyon ilk kez çağrıldığında, fonksiyonun gerçek kodu çağrılmak yerine, trambolin kodu üzerinden dinamik bağlayıcı çağrılmaktadır. Trambolin kodu yığına, adres çözümlemesi yapılacak fonksiyon ile ilgili bir kimlik bilgisi geçirmekte ve dinamik bağlayıcının adres çözümleme rutinini çağırmaktadır.

> **Not:** Dinamik bağlayıcının kendisi de bir paylaşımlı kütüphanedir, bu sayede dinamik bağlayıcı rutinlerini çağırmak mümkün olmaktadır.

Dinamik bağlayıcı, ilgili fonksiyona ilişkin adres çözümlemesini yaptıktan sonra fonksiyonun adresinin tutulduğu GOT girişine fonksiyonun adresini yazar ve fonksiyon koduna dallanır. Bir sonraki çağrı ise GOT üzerindeki adres üzerinden, dinamik bağlayıcıya ihtiyaç duyulmaksızın, yapılmaktadır.

Bir fonksiyon ilk kez çağrıldığında gerçekleşecek akış temel olarak şekilde gösterildiği gibidir.



Neler olduğunu adım adım inceleyelim.

1. İlk olarak *bar* fonksiyonuna ilişkin trambolin kodu çağrılacaktır.

`2.` Trambolin kodu, *bar* fonksiyonunun adresini tutmak için ayrılmış GOT girişindeki adrese dallanacaktır. **jmp *GOT [bar]** sembolik gösterimindeki `*`, GOT girişinin adresi üzerinden dolaylı dallanma (*indirect jump*) yapıldığını göstermektedir.

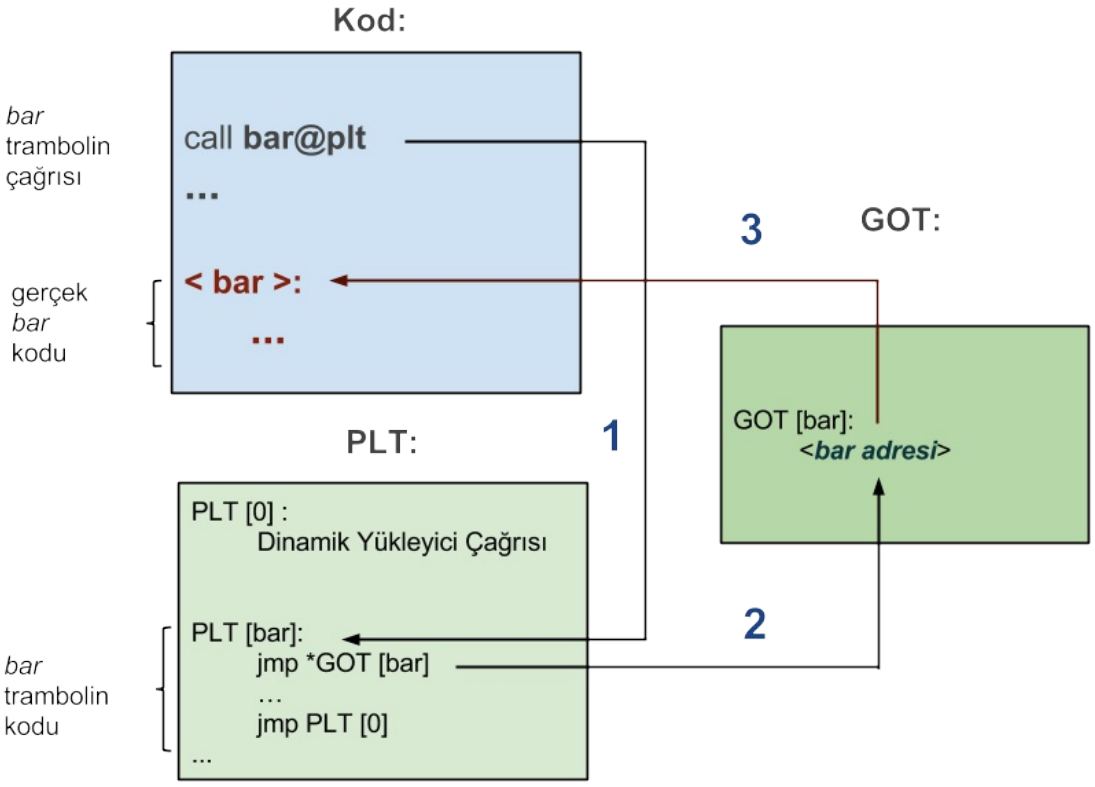
`3.` Akış tekrak trambolin koduna dönecektir. Bu durum ilk bakışta tuhaf görünmektedir. Dinamik bağlayıcı kütüphane kodunu yüklerken, sembol çözümlemesi yapmaksızın, GOT girişlerine trambolin kodu adreslerini yazmaktadır. Bu sayede, fonksiyonun çağırılıp çağrılmayacağı bilinmeyen bir aşamada, sembol çözümlemesi yapılmamakta ve fonksiyon çağrıldığında trambolin kodu üzerinden bu işlem yapılmaktadır.

`4.` Üç noktayla gösterdiğimiz bölümde, *bar* fonksiyonuna ilişkin bir kimlik değeri yığına geçirilip sonrasında PLT'nin ilk girişine dallanılmaktadır. Dinamik bağlayıcı bu bilgi sayesinde hangi fonksiyonu çözümleyeceğini bilmektedir. Bu aşamada, kimlik bilgisiyle ilgili detay kısmını atlıyoruz. PLT'nin ilk girişi, fonksiyon trambolin kodlarını tutan, diğer girişlerden farklı olarak özel bir giriştir ve dinamik bağlayıcının sembol çözümleme fonksiyonunu çağırmaktan sorumludur.

`5.` Dinamik bağlayıcı *bar* için sembol çözümlemesini yaptıktan sonra *bar* fonksiyonunun adresini şekilde GOT [bar] ile gösterilen girişe yazmaktadır.

`6.` Dinamik bağlayıcı GOT girişini düzenledikten sonra *bar* fonksiyonuna dallanmakta ve nihayetinde gerçek fonksiyon çağrılmaktadır.

Fonksiyon ikinci kez çağrıldığında ise bu sefer akış aşağıdaki gibi olacaktır.



Bu kez ilgili GOT girişi fonksiyonun adresini tuttuğundan, herhangi bir sembol çözümlemesi yapılmaksızın *bar* fonksiyonu çağrılmaktadır.

Şimdi basit bir örneği *gdb* ile çalıştırarak gerçekte neler olduğuna daha yakından bakalım.

Kütüphane koduna ve onu kullanacak koda sırasıyla *test.c* ve *driver.c* adlarını verip aşağıdaki gibi derleyebilirsiniz. Kütüphaneyi derlerken debug sembollerini eklemek için **-g** anahtarını geçiriyoruz.

```
void bar() {
    puts(__func__);
}

void foo() {
    bar();
}
```

```
void foo();

int main() {
    foo();
    return 0;
}
```

```
$ gcc -fPIC -shared -olibtest.so test.c -m32 -g
$ gcc -odriver driver.c -L. -ltest -m32
$ LD_LIBRARY_PATH=. ./driver
bar
```

driver uygulamasını gdb ile çalıştırıp, dinamik bağlayıcının kütüphaneyi bulabilmesi için LD_LIBRARY_PATH çevre değişkenini set ediyoruz.

```
$ gdb -q driver
Reading symbols from driver...(no debugging symbols found)...done.
(gdb) set environment LD_LIBRARY_PATH=.
```

main fonksiyonunu kesme noktası olarak işaretliyor ve uygulamayı çalıştırıyoruz.

```
(gdb) break main
Breakpoint 1 at 0x8048549
(gdb) run
Starting program: /home/serkan/embedded/so/test/29/driver

Breakpoint 1, 0x08048549 in main ()
```

Bu aşamadan sonra kütüphane dosyası yüklenmiş olacağından, sırasıyla *foo* ve *bar* fonksiyonlarını kesme noktaları olarak belirliyoruz.

```
(gdb) break test.c:foo
Breakpoint 2 at 0xf7fd45b0: file test.c, line 6.
(gdb) break test.c:bar
Breakpoint 3 at 0xf7fd4587: file test.c, line 2.
```

Uygulamayı devam ettirdiğimizde, kodun *bar* fonksiyonuna ilişkin trambolin kodu çağrısında durduğunu görüyoruz. *foo* fonksiyonunun başlangıç kodlarının (*prologue*) çalıştırıldığına dikkat ediniz. Bu aşamada *ebx* yazmacında GOT başlangıç adresi bulunmaktadır.

```
(gdb) continue
Continuing.

Breakpoint 2, foo () at test.c:6
6               bar();
(gdb) disas
Dump of assembler code for function foo:
   0xf7fd459e <+0>:     push   %ebp
   0xf7fd459f <+1>:     mov    %esp,%ebp
   0xf7fd45a1 <+3>:     push   %ebx
   0xf7fd45a2 <+4>:     sub    $0x4,%esp
   0xf7fd45a5 <+7>:     call   0xf7fd4440 <__x86.get_pc_thunk.bx>
   0xf7fd45aa <+12>:    add    $0x1a56,%ebx
=> 0xf7fd45b0 <+18>:    call   0xf7fd4400 <bar@plt>
   0xf7fd45b5 <+23>:    add    $0x4,%esp
   0xf7fd45b8 <+26>:    pop    %ebx
   0xf7fd45b9 <+27>:    pop    %ebp
   0xf7fd45ba <+28>:    ret
End of assembler dump.
```

Sonrasında makina komutlarını adım adım çalıştırarak trambolin koduna bakalım.

```
(gdb) stepi
0xf7fd4400 in bar@plt () from ./libtest.so
(gdb) disas
Dump of assembler code for function bar@plt:
=> 0xf7fd4400 <+0>:     jmp    *0xc(%ebx)
   0xf7fd4406 <+6>:     push   $0x0
   0xf7fd440b <+11>:    jmp    0xf7fd43f0
End of assembler dump.
```

Trambolin kodundaki ilk komut, daha önce söylediğimiz gibi, *bar* fonksiyonu için ayrılmış alandaki adrese dallanmaktadır. *bar* fonksiyonuna ilişkin GOT girişinde tutulan değere aşağıdaki gibi ulaşabiliriz. *ebx* yazmacında hala GOT başlangıç adresi tutulmaktadır.

```
(gdb) print /x *(0xc + $ebx)
$2 = 0xf7fd4406
```

Elde edilen adresin, 0xf7fd4406, trambolin kodunun 2. komutunu gösterdiğine dikkat ediniz. Bu aşamadan sonra, daha önce de söylediğimiz gibi, trambolin kodu dinamik bağlayıcıyı çağıracak, dinamik bağlayıcı da ilgili GOT girişini güncelleyip *bar* fonksiyonunu çağıracaktır. Kodu kaldığı yerden çalıştırmaya devam ettiğimizde *bar* fonksiyonun çağrıldığını görüyoruz.

```
(gdb) continue
Continuing.

Breakpoint 3, bar () at test.c:2
2               puts(__func__);
(gdb) disas
Dump of assembler code for function bar:
   0xf7fd4575 <+0>:     push   %ebp
   0xf7fd4576 <+1>:     mov    %esp,%ebp
   0xf7fd4578 <+3>:     push   %ebx
   0xf7fd4579 <+4>:     sub    $0x4,%esp
   0xf7fd457c <+7>:     call   0xf7fd4440 <__x86.get_pc_thunk.bx>
   0xf7fd4581 <+12>:    add    $0x1a7f,%ebx
=> 0xf7fd4587 <+18>:    sub    $0xc,%esp
   0xf7fd458a <+21>:    lea    -0x1a30(%ebx),%eax
   0xf7fd4590 <+27>:    push   %eax
   0xf7fd4591 <+28>:    call   0xf7fd4420 <puts@plt>
   0xf7fd4596 <+33>:    add    $0x10,%esp
   0xf7fd4599 <+36>:    mov    -0x4(%ebp),%ebx
   0xf7fd459c <+39>:    leave
   0xf7fd459d <+40>:    ret
End of assembler dump.
```

Bu aşamada, GOT girişinde *bar* fonksiyonunun adresi bulunuyor olmalı. Bu durumu aşağıdaki gibi doğrulayabiliriz. Daha önce ilgili girişte trambolin kodunun adresi olduğunu hatırlayınız.

```
(gdb) print /x *(0xc + $ebx)
$3 = 0xf7fd4575

(gdb) print bar
$4 = {void ()} 0xf7fd4575 <bar>
```

bar fonksiyonun sonraki çağrısında bu süreç tekrarlanmayacak ve GOT üzerinden fonksiyonun dolaylı olarak çağrılabilecektir.

64 Bitlik Sistemlerde Paylaşımlı Kütüphanelerin Oluşturulması

64 bitlik mimari için de pozisyon bağımsız kod 32 bitlik mimariye benzer şekilde oluşturulmakta, yine GOT ve PLT tabloları kullanılmaktadır. Temel farklılık GOT girişlerine erişim şeklindedir.

İki mimari arasında data referanslarını ele alış biçiminde farklılık bulunmaktadır. 32 bitlik mimarinin aksine 64 bitlik mimaride data adresleri de göreli olabilir. 64 bitlik mimaride bu adresleme şekli **RIP-relative** olarak isimlendirilmektedir. Bu sayede 64 bitlik mimaride paylaşımlı kütüphane kodu yazımı daha kolay bir hal almaktadır.

Daha önce incelediğimiz basit bir örnek üzerinden bu duruma bakalım. Kodu *test.c* adıyla saklayıp aşağıdaki gibi derleyebilirsiniz.

```
int global;

void foo() {
    global = 111;
}
```

```
$ gcc -fPIC -shared -olibtest.so test.c --save-temps
```

test.s içindeki sembolik makina kodlarına baktığımızda, derleyicinin aşağıdaki gibi bir kod yazdığını görmekteyiz.

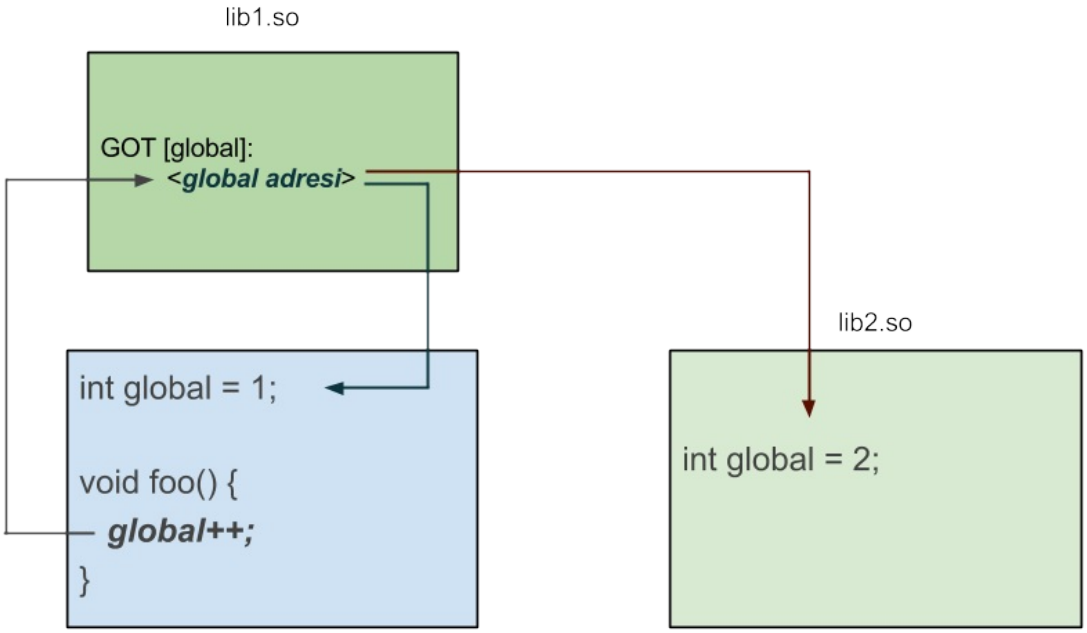
```
foo:
.LFB0:
 1.   pushq   %rbp
 2.   movq    %rsp, %rbp
 3.   movq    global@GOTPCREL(%rip), %rax
 4.   movl    $111, (%rax)
 5.   popq    %rbp
 6.   ret
```

Aynı kodun 32 bitlik mimari için derlenmesi durumunda aşağıdaki gibi bir kod üretildiğini daha önce görmüştük. Her iki komut listesini de incelemeyi kolaylaştırmak için numaralandırdık.

```
foo:
.LFB0:
 1.   pushl   %ebp
 2.   movl    %esp, %ebp
 3.   call    __x86.get_pc_thunk.cx
 4.   addl    $_GLOBAL_OFFSET_TABLE_, %ecx
 5.   movl    global@GOT(%ecx), %eax
 6.   movl    $111, (%eax)
 7.   popl    %ebp
 8.   ret
```

32 bit için, `3`, `4` ve `5` numaralı komutlarla yapılan iş 64 bitlik mimaride `3` numaralı komut ile yapılmaktadır. Tek bir komut ile GOT girişinin yani global değişkenin adresine ulaşılabilmektedir. Bu durumda, 64 bitlik mimari için derleyici fazladan __x86.get_pc_thunk.cx gibi bir fonksiyon yazmak zorunda kalmamakta, ayrıca bir yazmacı GOT başlangıç adresini tutması için tahsis etmemektedir.

Bu noktada, data referanslarına neden göreli adres kullanılarak erişilmeyipte GOT üzerinden erişildiğini merak edebilirsiniz. Kod bölümündeki bir kod ile GOT arasındaki mesafe derleme zamanında bilindiği gibi data alanı ile olan mesafe de bilinmektedir. Data referanslarına bu şekilde erişilmemesinin nedeni fonksiyonlardan beklenen özellikle aynıdır. Daha önce paylaşımlı kütüphanelerde fonksiyonların nasıl ele alındığını incelediğimiz bölümde, bir fonksiyonun aynı kütüphane içindeki bir fonksiyon yerine başka bir fonksiyona bağlanabileceğinden bahsetmiştik, aynı durum data referansları için de geçerlidir. Bu mekanizmanın sağlanabilmesi için data alanlarına direkt erişilmek yerine erişim GOT üzerinden dolaylı olarak yapılmaktadır. Aşağıdaki şekli inceleyiniz.



Kod ve data referanslarının öncelik sırasının neye göre belirlendiğine daha sonra bakacağız.

Dinamik Yükleme

Dinamik bağımlılığı olan uygulamalar ihtiyaç duydukları tüm kodu bünyelerinde bulundurmamaktadırlar. Bu tür uygulamalar yüklenirken, akış program koduna geçmeden önce, ihtiyaç duydukları kodları barındıran paylaşımlı kütüphaneler yüklenmiş olmalıdır. Gerekli kütüphanelerin bulunması ve yüklenmesi dinamik bağlayıcının görevidir. Derleme aşamasında bağlayıcı, uygulamanın ihtiyaç duyduğu paylaşımlı kütüphanelerin isimlerini çalışabilir dosya içine not etmektedir. Bu sayede dinamik bağlayıcı gerekli kütüphane dosyalarını tespit edebilmektedir.

> **Not:** Kütüphanelerin, uygulama kodu tarafından yüklendiği çalışma şekli bu durumun dışındadır. Bu durumu daha sonra inceleyeceğiz.

Basit bir örnek üzerinden bu durumu inceleyelim. Uygulama ve kütüphane koduna *driver.c* ve *test.c* adını verelim.

```
void foo();

int main() {
    foo();
    return 0;
}
```

```
void foo() {

}
```

Kütüphane ve çalışabilir dosyayı aşağıdaki gibi oluşturabiliriz.

```
$ gcc -fPIC -shared -olibtest.so test.c
$ gcc -odriver driver.c libtest.so
```

Uygulama oluşturulurken bağlayıcı gerekli kütüphane isimlerini ELF formatında DT_NEEDED etiketine yazmaktadır. *readelf* aracı ile uygulamanın bağımlı olduğu kütüphane listesine ulaşılabiliriz. Örneğimiz için bağımlılık listesi aşağıdaki gibidir.

```
readelf --dynamic driver | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libtest.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Bağımlılık listesinde *libtest.so* ile beraber, gcc tarafından bağlayıcıya gizli bir şekilde geçirilen, stadart kütüphane dosyasını da (*libc.so.6*) görmekteyiz.

Bağlayıcı yalnız gerekli dosyaları uygulamanın bağımlılık listesine eklemektedir. Daha önce statik kütüphaneleri incelediğimiz bölümde de söylediğimiz gibi, bağlayıcı bir dosya içinde tanımı bulunmayan bir sembolle karşılaştığında içsel bir tabloya bu sembolü kaydetmekte ve sonraki dosyalarda bu sembolün tanımını aramaktadır. Örneğimiz için bağlayıcı *driver.c* dosyasından üretilen amaç kod içinde *foo* fonksiyonunun tanımını bulamamış, daha sonra bu tanımı *libtest.so* dosyasında bularak kütüphane dosyasının ismini çalışabilir dosyanın bağımlılık listesine eklemiştir.

Paylaşımlı kütüphanelerden uygulamaya kod kopyalanmadığından, ilk bakışta paylaşımlı kütüphanelere yalnız çalışma zamananında ihtiyaç olduğu düşünülebilir. Fakat bahsettiğimiz nedenden dolayı bağlanma aşamasında da paylaşımlı kütüphane dosyalarına ihtiyaç duyulmaktadır, kütüphane dosyalarının sembol tabloları bağlayıcı tarafından kullanılmaktadır.

> **Not:** Windows altında ise, dll dosyaları içindeki sembol bilgilerini barındıran ayrı **.lib** uzantılı dosyalar oluşturulmakta ve bağlanma işleminde bu dosyalar kullanılmaktadır.

Şimdi bağlayıcının gerekli kütüphaneleri belirlerken izlediği yola bir önceki örneğimizi genişleterek tekrar bakalım.

```
void foo();

void foo() {
    puts("FOO");
}

int main() {
    foo();
    return 0;
}
```

```
void foo() {
    puts(__func__);
}

void bar() {
    puts(__func__);
}
```

Gerekli dosyaları bir önceki örnekteki gibi yeniden oluşturalım.

```
$ gcc -fPIC -shared -olibtest.so test.c
$ gcc -odriver driver.c libtest.so
```

Uygulamanın bağımlılık listesinde bu sefer *libtest.so* dosyası bulunmamaktadır.

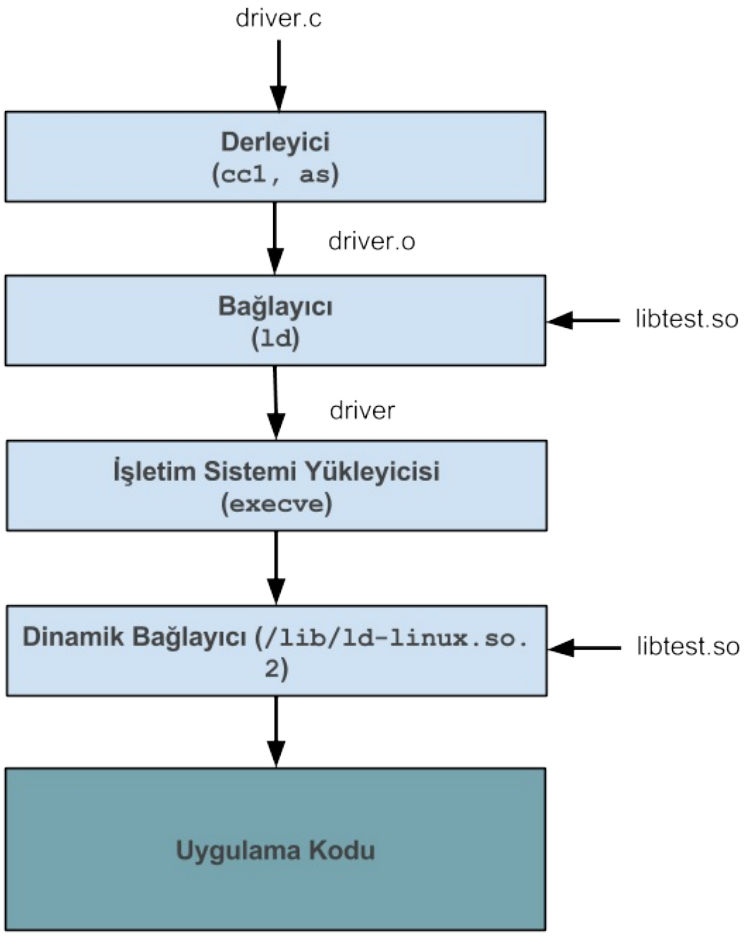
```
$ readelf --dynamic driver | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Bağlayıcı *foo* tanımını bulduğu için *libtest.so* dosyasının gerekli olmadığı sonucuna varmıştır. Böyle bir durumda bağlayıcıya **--no-as-needed** seçeneği geçirilerek, komut satırındaki tüm kütüphanelerin uygulamanın bağımlılık listesine eklenmesi sağlanabilir.

```
$ gcc -odriver driver.c -Wl,--no-as-needed libtest.so

$ readelf --dynamic driver | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libtest.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Bu durumda hangi *foo* fonksiyonunun çağrılacağını merak edebilirsiniz, bu konuya daha sonra değineceğiz fakat bu noktada uygulama içindeki kodun çağrılacağını söyleyelim. Kullanılmıyor gibi gözüken bir kütüphanenin neden yüklenmek istenebileceğine de yine daha sonra değineceğiz.

Gerekli kütüphaneler dinamik bağlayıcı tarafından yüklendikten sonra uygulama başlatılmaktadır. Uygulamayı başlatan işletim sistemi yükleyicisi değil, dinamik bağlayıcının kendisidir. Dinamik bağımlılığı olan bir uygulamanın derlemesinden çalıştırılmasına kadar geçen süreci ve görev alan araçları kabaca aşağıdaki gibi gösterebiliriz.



Örneğimiz üzerinden gidecek olursak *driver* uygulamasını çalıştırmak istediğimizde ilk olarak kabuk (*shell*) tarafından işletim sistemi yükleyicisi çağrılacaktır. Uygulamanın dinamik bağımlılıkları olması durumunda yükleyici dinamik bağlayıcıyı çağıracak, dinamik bağlayıcı da gerekli kütüphaneleri yükledikten sonra uygulamayı çalıştıracaktır. İncelemelerimizin başından beri sürekli bağlayıcı ve dinamik bağlayıcıdan bahsettik. Şimdi bu araçlara daha yakından bakalım.

Statik ve Dinamik Bağlayıcı

İlk olarak bu bağlayıcıların isimlendirilmesinden bahsedelim. Bağlayıcılardan bir tanesi derleme diğeri ise çalışma zamanında görev almaktadır.

Derleme ya da statik zaman (*compile time, static time*) olarak adlandırılan süreçte kullanılan bağlayıcı, kısaca bağlayıcı ya da statik bağlayıcı olarak isimlendirilebilir. İngilizce *linker*, *static linker* ve *link editor* isimlerinin kullanıldığını görmekteyiz.

Çalışma zamanında görev alan bağlayıcıyı ise dinamik bağlayıcı olarak kullandık. İngilizce *dynamic linker*, *run-time linker* ve *dynamic linking loader* terimlerinin kullanıldığını görmekteyiz.

Statik bağlama işlemi, **binutils** paketinden çıkan **ld** uygulaması tarafından sağlanmaktadır. gcc, *ld* uygulamasını uygun parametre ve seçeneklerle gizli olarak çağırmaktadır. Dinamik bağlama işlemi ise kendisi de paylaşımlı kütüphane olan çalıştırılabilir bir dosya tarafından sağlanmaktadır. Bir paylaşımlı kütüphanenin bir uygulamaya bağlanmaksızın kendi başına çalıştırılabilir olması ilk bakışta tuhaf gelebilir. Bu durumu ilerleyen kısımda inceleyeceğiz. Dinamik bağlayıcı **libc** paketinden çıkmaktadır. Kullandığımız sistemde, 32 ve 64 bitlik dinamik yükleyiciler ve paketleri aşağıdaki gibidir.

Paket	Dinamik Yükleyici
libc6:i386	/lib/ld-linux.so.2
libc6:amd64	/lib64/ld-linux-x86-64.so.2

Dinamik bağlayıcının dışsal bir bağımlılığı bulunmamaktadır ve işletim sistemi yükleyicisi tarafından yüklenip çalıştırılabilmektedir. İşletim sistemi, yükleyeceği dinamik bağlayıcıyı çalışabilir dosya içeriğinden öğrenmektedir. Derleme zamanında statik bağlayıcı ELF içinde **.interp** (interpreter) alanına kullanılması istenen dinamik bağlayıcının yol ifadesini yazmaktadır. *readelf* ile bu değere aşağıdaki gibi ulaşabiliriz.

```
$ readelf -l driver | grep interpreter
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

Dinamik bağlayıcıyı çalıştırdığımızda aldığı seçenekleri gösteren bir yardım mesajıyla karşılaşmaktayız. Bu zamana kadar bir uygulamanın dinamik bağımlılıklarını ELF içindeki ilgili alana bakarak listeledik. ELF içindeki dinamik bağımlılık listesi, uygulamanın direkt bağımlı olduğu kütüphaneleri listelemektedir. Buradaki kütüphanelerin de başkaca bağımlılıkları olabilir. Dinamik bağlayıcı bu bağımlılıkları da çözmektedir. Aşağıdaki çıktıları inceleyiniz.

```
$ readelf --dynamic driver | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libtest.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

```
$ LD_TRACE_LOADED_OBJECTS=1 /lib64/ld-linux-x86-64.so.2 ./driver
    linux-vdso.so.1 =>  (0x00007fff96d5b000)
    libtest.so => not found
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f0b5c8d0000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f0b5ccba000)
```

Dinamik bağlayıcınının davranışını değiştiren çevre değişkenlerinden yeri geldiğince bahsetmeye çalışacağız. Dinamik bağlayıcının başındaki çevre değişkeninin kullanılmaması durumunda bağlayıcı *libtest.so* dosyasını bulamadığı için hata vererek sonlanacaktır.

Dinamik bağlayıcının kütüphaneleri arama kurallarına daha sonra bakacağız. Uygulamaların dinamik bağımlılıklarını izlemek için Linux sistemlerinde ayrıca **ldd** isimli, *libc-bin* paketinden çıkan bir betik (script) dosyası da mevcuttur. Dinamik bağlayıcıyı açık bir şekilde kullanmaksızın bir uygulamanın tüm dinamik bağımlılıklarını aşağıdaki gibi de daha kolay bir şekilde listeleyebilirsiniz.

```
$ ldd driver
    linux-vdso.so.1 =>  (0x00007fffee5fc000)
    libtest.so => not found
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2e5dcd5000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f2e5e0bf000)
```

Bu bölümde son olarak paylaşımlı bir kütüphanenin nasıl çalıştırılabileceğinden kısaca söz edelim.

Çalıştırılabilir Paylaşımlı Kütüphaneler

Bir kütüphane dosyasının kendi başına çalıştırılabilir olması, dinamik bağlayıcı dışında çok aranan bir özellik değildir. Buna karşın, kütüphanenin kullanımı, versiyonu ve sahipliği hakkında bilgi vermek için kullanılabilir. Örneğin standart C kütüphanesi böyle bir kullanıma sahiptir.

```
$ /lib/x86_64-linux-gnu/libc.so.6

GNU C Library (Ubuntu GLIBC 2.19-10ubuntu2.1) stable release version 2.19, by Roland M
cGrath et al.
Copyright (C) 2014 Free Software Foundation, Inc.
...
```

Bir uygulamanın hangi noktadan itibaren çalışmaya başlayacağı (*entry point*) bağlayıcı tarafından çalıştırılabilir dosya içine yazılmaktadır. İşletim sistemi yükleyicisi veya dinamik bağlayıcı, ELF içindeki bu adrese dallanarak kontrolü uygulama koduna bırakmaktadır. Bağlayıcı çalıştırılabilir dosyalar için öngörülen giriş noktası olarak, tanımı derleyicinin başlangıç (*startup*) kodlarında bulunan ve nihayetinde **main** fonksiyonunu çağıracak olan, **_start** sembolünü seçmektedir. *readelf --file-header* şeklinde bu adrese ulaşabilirsiniz. Bağlayıcı ayrıca giriş noktasının kullanıcı tarafında belirlenebilmesine izin vermektedir. Aşağıdaki örneği inceleyiniz.

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

void foo() {
    puts(__func__);
    exit(0);
}

int main() {
    puts(__func__);
    return 0;
}
```

```
$ gcc -otest test.c -Wl,--entry=foo
```

test uygulamasını çalıştırdığımızda *main* yerine *foo* fonksiyonunun çalıştığını görmekteyiz.

```
$ ./test
foo
```

Paylaşımlı dosyaların dinamik bağlayıcı tarafından yüklendiğini ve dinamik bağımlılığı olan uygulamaların yine dinamik bağlayıcı tarafından çalıştırılığını söylemiştik. Bu durumda paylaşımlı dosyamızın *.interp* alanına dinamik yükleyiciyi yazmalı ve istediğimiz giriş noktasını belirtmeliyiz. *.interp* alanı çalıştırılabilir dosyalar için statik bağlayıcı tarafından doldurulmaktadır, paylaşımlı kütüphaneler için ise bu değeri biz yazmalıyız. Aşağıdaki örneği inceleyiniz.

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

#ifdef __x86_64__
const char interp[] __attribute__((section(".interp"))) = "/lib64/ld-linux-x86-64.so.2"
;
#else
const char interp[] __attribute__((section(".interp"))) = "/lib/ld-linux.so.2";
#endif

void foo() {

}

void version() {
    puts("Version 1.9.7.8");
    exit(0);
}
```

Uygulamanın 32 veya 64 bit hedefli derlenmesi durumunda uygun dinamik bağlayıcı ifadesi öntanımlı `__x86_64__` makrosu ile seçilmekte ve GNU C eklentisiyle **.interp** alanına yazılmaktadır. Kütüphanemizi aşağıdaki gibi oluşturup çalıştırabiliriz.

```
$ gcc -fPIC -shared -olibtest.so test.c -Wl,--entry=version
$ ./libtest.so
Version 1.9.7.8
```

Derleme Zamanında Kütüphanelerin Aranma Sırası

Statik bağlayıcıya gerekli kütüphaneler yol ifadeleriyle beraber geçirebildiği gibi yalnız isimleri de kullanılabilir. Yalnız isimlerinin geçirildiği durumda kütüphane dosyalarının nerede olduğu bilinemediğinden aranması gerekmektedir. Bağlayıcı öntanımlı olarak bazı dizinlere bakmaktadır. Ayrıca kullanıcının yeni arama dizinleri eklemesine izin verilmiştir. Yeni bir arama dizini eklemek için **-L** seçeneği kullanılmaktadır. Bu şekilde belirtilen dizinlere öngörülenlerden daha önce bakılmaktadır. *-L* seçeneği komut satırında istenilen sayıda kullanılabilmektedir. Kütüphane dosyalarının yalnız isimlerinin kullanılabilmesi için **lib*XXX*.so** şeklinde isimlendirilmeleri gerekmektedir.

Basit bir örnek üzerinde bu durumu inceleyelim. Örnekleri sırasıyla *driver.c* ve *test.c* adlarıyla saklayabilirsiniz.

```
void foo();

int main() {
    foo();
    return 0;
}
```

```
void foo() {
    puts(__func__);
}
```

Kütüphane dosyamızı oluşturalım.

```
$ gcc -fPIC -shared -olibtest.so test.c
```

Uygulamamızı ilk olarak kütüphane dosyasını göstermeden derlemeyi deneyelim.

```
$ gcc -odriver driver.c
/tmp/ccUapyFF.o: In function `main':
driver.c:(.text+0xa): undefined reference to `foo'
collect2: error: ld returned 1 exit status
```

Bağlayıcı, *foo* fonksiyonunu bulamadığından şikayet etmekte. Bağlayıcı öntanımlı olarak baktığı dizinlerde *foo* tanımını içeren herhangi bir kütüphane (*libtest.so*) bulamadığından bu hata mesajını vermektedir. Bu durumda aşağıdaki yöntemleri kullanabiliriz. Öncesinde incelememizi kolaylaştırmak için bulunduğumuz dizinde *lib* isminde bir dizin oluşturup *libtest.so* dosyasını bu dizine taşıyalım.

```
$ mkdir lib
$ mv libtest.so lib
```

1. Kütüphane yol ifadesiyle beraber geçirilebilir.

```
    $ gcc -odriver driver.c lib/libtest.so
```

2. Kütüphane isminden önce aranacak dizinlere kütüphanenin bulunduğu dizin yolu eklenebilir. Kütüphane **libXXX.so** şeklinde isimlendirilmesi durumunda bağlayıcıya **-lXXX** şeklinde geçirilmelidir.

```
    $ gcc -odriver driver.c -Llib -ltest
```

Bağlayıcı tarafından bakılan tüm dizin kümesini merak edebilirsiniz. Bağlayıcının öntamımlı olarak baktığı arama dizinleri, içsel olarak kullanıdığı bağlayıcı betiğinde (*linker script*) tanımlanmıştır. Bu dizinleri aşağıdaki gibi öğrenebiliriz.

```
$ ld --verbose | grep SEARCH_DIR

SEARCH_DIR("=/usr/x86_64-linux-gnu/lib64"); SEARCH_DIR("=/usr/local/lib/x86_64-linux-g
nu"); SEARCH_DIR("=/usr/local/lib64"); SEARCH_DIR("=/lib/x86_64-linux-gnu"); SEARCH_DI
R("=/lib64"); SEARCH_DIR("=/usr/lib/x86_64-linux-gnu"); SEARCH_DIR("=/usr/lib64"); SEA
RCH_DIR("=/usr/x86_64-linux-gnu/lib"); SEARCH_DIR("=/usr/local/lib"); SEARCH_DIR("=/li
b"); SEARCH_DIR("=/usr/lib");
```

gcc ile derleme işlemi yaparken **--verbose** seçeneğinin geçirilmesi durumunda, bağlayıcıya çok sayıda *-L* anahtarıyla dizin yolunun geçirildiğini görmekteyiz. gcc tarafından ilave olarak bağlayıcıya geçirilen arama dizin listesine de aşağıdaki gibi ulaşabiliriz.

```
$ gcc -print-search-dirs | grep libraries

libraries: =/usr/lib/gcc/x86_64-linux-gnu/4.9/:/usr/lib/gcc/x86_64-linux-gnu/4.9/../..
/../../x86_64-linux-gnu/lib/x86_64-linux-gnu/4.9/:/usr/lib/gcc/x86_64-linux-gnu/4.9/..
/../../../x86_64-linux-gnu/lib/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-gnu/4.9/../
../../../x86_64-linux-gnu/lib/../lib/:/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_6
4-linux-gnu/4.9/:/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/:/usr/lib
/gcc/x86_64-linux-gnu/4.9/../../../../lib/:/lib/x86_64-linux-gnu/4.9/:/lib/x86_64-linu
x-gnu/:/lib/../lib/:/usr/lib/x86_64-linux-gnu/4.9/:/usr/lib/x86_64-linux-gnu/:/usr/lib
/../lib/:/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../x86_64-linux-gnu/lib/:/usr/lib/
gcc/x86_64-linux-gnu/4.9/../../../:/lib/:/usr/lib/
```

Ayrıca tüm bu dizinlerden önce siz de komut satırında *-L* ile istediğiniz dizinlerin aranmasını sağlayabilirsiniz.

Çalışma Zamanında Kütüphanelerin Aranma Sırası

Çalışma zamanında kütüphanelerin aranmasında, derleme zamanına görece, daha karmaşık bir yol izlenmektedir. İlk olarak kütüphanelerin aranmasıyla ilgili, derleme sürecini de ilgilendiren, bir özellikten bahsedelim.

Arama Dizinlerinin ELF Dosya İçeriğine Yazılması

Çalışma zamanında kütüphanelerin, standart arama dizinlerine ek olarak, nerelerde aranacağı ELF dosya içeriğine yazılabilmektedir. Çalışabilir ve paylaşımlı kütüphane dosyalarının her ikisinde de bu özellik kullanılabilmektedir.

Gerekli kütüphanelerin standart aranacak dizinlerde olmadığı ve konumlarının bilindiği durumlarda bu özellik kullanışlı olabilmektedir. Bağlayıcı bu amaçla **rpath** seçeneğini barındırmaktadır. Basit bir örnek üzerinden bu özelliği inceleyelim. Uygulama ve kütüphane kodlarını sırasıyla *driver.c* ve *test.c* olarak isimlendirelim.

```
void foo();

int main() {
    foo();
    return 0;
}
```

```
void foo() {
    puts(__func__);
}
```

İlk olarak, bu zamana kadar yaptığımız gibi kütüphaneye ilişkin herhangi bir arama bilgisi kullanmaksızın dosyaları derleyelim.

```
$ gcc -fPIC -shared -olibtest.so test.c
$ gcc -odriver driver.c -L. -ltest
```

Uygulamayı çalıştırdığımızda, dinamik bağlayıcı tarafından, aşağıdaki gibi bir hata almaktayız.

```
$./driver

./driver: error while loading shared libraries: libtest.so: cannot open shared object
file: No such file or directory
```

Dinamik bağlayıcı, *driver* uygulamasının bağımlılık listesinde bulunan *libtest.so* dosyasını, öntanımlı baktığı dizinlerde, bulamadığından uygulamayı çalıştıramamaktadır. Dinamik bağlayıcı tarafından uygulamanın çalıştığı dizine bakılmamaktadır. Standart dışı dizinlerde bulunan kütüphanelerin dinamik bağlayıcıya, daha pratik olarak, nasıl gösterilebileceğine bakacağız, fakat bu noktada bu işlemin *rpath* ile nasıl yapılabildiğine bakalım.

Test amaçlı olarak çalışma dizinimizde **lib** adıyla bir dizin oluşturup, *libtest.so* dosyasını bu dizine kopyalayalım.

```
$ mkdir lib
$ cp libtest.so lib
```

Uygulamamızı bu sefer kütüphaneye ilişkin arama dizin bilgisiyle beraber derleyelim.

```
$ gcc -odriver driver.c -Llib -ltest -Wl,-rpath,$PWD/lib
```

> **Not:** PWD çevre değişkeni çalışılan dizini göstermektedir.

Bu sefer dinamik bağlayıcı *libtest.so* dosyasını bulmakta ve uygulamayı çalıştırabilmektedir.

```
$ ./driver
foo
```

Bu noktada derleme işleminde hem **L** hem de **rpath** seçeneklerine *lib* değerini geçirdik, fakat bu değerler birbirinden farklı olabilirdi. *L* seçeneği geliştirme ortamına *rpath* ise çalışma ortamına ilişkindir.

Uygulama veya kütüphane dosyalarının *rpath* listeleri ELF içinde **DT_RPATH** isimli bir alanda tutulmaktadır, bu bilgiye aşağıdaki gibi ulaşmak mümkündür.

```
$ readelf -d driver | grep PATH
 0x000000000000000f (RPATH)              Library rpath: [/home/serkan/embedded/so/test
/29/lib]
```

Dinamik bağlayıcı kütüphaneleri ararken, ilk olarak *rpath* listesindeki dizinlere bakmaktadır. Bağlayıcının bu davranışı değiştirilebilmesine karşın, bu özellik her zaman kullanışlı olmamaktadır. Test ortamlarında standart dışı dizinlerde bulunan kütüphaneler dinamik

bağlayıcıya gösterilmek istenmektedir, *rpath* bu yöntemin kullanılmasını engellemektedir. Bu yüzden ELF formatına önceliği daha düşük **DT_RUNPATH** isimli bir alan eklenmiştir. *rpath* olarak bu alanın kullanılabilmesi için bağlayıcıya **--enable-new-dtags** seçeneği geçirilmelidir.

> **Not:** Bağlayıcıya *--inhibit-rpath* seçeneği geçirilerek, *rpath* listesine bakması engellenebilir.

Aynı uygulamayı bu şekilde tekrar derleyelim.

```
$ gcc -odriver driver.c -Llib -ltest -Wl,-rpath,$PWD/lib -Wl,--enable-new-dtags
```

Bu durumda ELF içinde DT_RUNPATH alanının kullanıldığını görmekteyiz.

```
$ readelf -d driver | grep PATH
 0x000000000000001d (RUNPATH)            Library runpath: [/home/serkan/embedded/so/te
st/29/lib]
```

rpath dizinleri mutlak olmak zorunda değildir, *driver* uygulamasını bu kez yalnız *lib* dizinini göstererek derleyelim.

```
$ gcc -odriver driver.c -Llib -ltest -Wl,-rpath,lib
$ readelf -d driver | grep PATH
 0x000000000000000f (RPATH)              Library rpath: [lib]

$ ./driver
foo
```

Uygulamanın çalıştığı dizinde *lib* dizini bulunduğu için bir problem çıkmayacaktır. Şimdi çalışma dizinimizde *test* isimli bir dizin oluşturalım ve uygulamayı bu dizindeyken çalıştırmayı deneyelim.

```
$ mkdir test
$ cd test
$ ../driver
../driver: error while loading shared libraries: libtest.so: cannot open shared object
 file: No such file or directory
```

Dinamik bağlayıcı çalışma dizininde *lib* dizinini bulamadığı için beklediğimiz üzere hata vermektedir. Dinamik bağlayıcının, göreli yol ifadelerini çözümlerken, çalışma dizini yerine uygulamanın bulunduğu dizine bakması sağlanabilir. Bunun için **$ORIGIN** özel değeri kullanılmaktadır. *$ORIGIN* değeri, dinamik bağlayıcı tarafından, uygulamanın veya

paylaşımlı kütüphanenin bulunduğu dizin ifadesine genişletilir. Tekrar çalışma dizinine geçerek örneğimizi aşağıdaki gibi tekrar derleyelim. Sonrasında *test* dizinine geçip tekrar çalıştıralım.

```
$ cd ..
$ gcc -odriver driver.c -Llib -ltest -Wl,-rpath,'$ORIGIN'/lib
$ cd test
$ ../driver
foo
```

Uygulamanın bu kez sorunsuz çalıştığını görmekteyiz. Bu özellik uygulamanın gerekli kütüphanelerle beraber dağıtıldığı durumlarda oldukça işe yaramaktadır. Örneğin, uygulama ve gerekli kütüphaneler arşivlendikten sonra hedef sistemde istenilen bir dizine açılarak çalıştırılabilir. Hedef sistemde kütüphaneler herhangi başka bir yere kopyalanmak zorunda değildir, ayrıca uygulamanın bulunduğu dizin *PATH* çevre değişkenine eklenerek, uygulama herhangi bir dizinden çalıştırılabilir.

> **Not:** *rpath* seçeneğine alternatif olarak, önceliği daha düşük olan, LD_RUN_PATH çevre değişkeni kullanılabilir.

Standart Dışı Dizinlerde Arama

Standart dışı dizinlerin dinamik bağlayıcıya gösterilmesi gerekmektedir. Bu amaçla **LD_LIBRARY_PATH** çevre değişkeni kullanılmakta veya dinamik bağlayıcıya, **--library-path** seçeneği ile, arama listesi açık bir şekilde geçirilebilmektedir.

Örnek uygulamamızı *rpath* kullanmaksızın yeniden derleyip çalıştıralım.

```
gcc -odriver driver.c -Llib -ltest
$ ./driver
./driver: error while loading shared libraries: libtest.so: cannot open shared object
file: No such file or directory
```

Beklediğimiz üzere, dinamik bağlayıcı kütüphane dosyasın bulamamakta. Bu durumda *LD_LIBRARY_PATH* çevre değişkenini kullanabiliriz.

```
$ LD_LIBRARY_PATH=. ./driver
foo
```

Diğer bir alternatif ise dinamik bağlayıcı uygun şekilde direkt olarak çağırmak şeklinde olabilir.

```
$ /lib64/ld-linux-x86-64.so.2 --library-path . ./driver
foo
```

/etc/ld.so.conf ve /etc/ld.so.cache Dosyaları

Paylaşımlı kütüphaneler bir çok farklı dizinde bulunabilmektedir, tüm bu dizinlerin yükleme zamanında dinamik bağlayıcı tarafından aranması oldukça maliyetli olacaktır. Bu yüzden */etc/ld.so.conf* ve */etc/ld.so.cache* dosyası kullanılmaktadır.

/etc/ld.so.conf dosyasının içeriği aşağıdaki gibidir.

```
$ cat /etc/ld.so.conf
include /etc/ld.so.conf.d/*.conf
```

/etc/ld.so.conf.d/ dizininde ise sonu *.conf* ile biten, dizin yolu gösteren, dosyalar bulunmaktadır. Bu dizindeki dosyalardan birinin içeriği aşağıdaki gibidir.

```
cat x86_64-linux-gnu.conf
# Multiarch support
/lib/x86_64-linux-gnu
/usr/lib/x86_64-linux-gnu
```

/etc/ld.so.conf dosyası üzerinden bir çok *.conf* dosyasına ulaşılmaktadır. Kendimiz de buraya *.conf* uzantılı başka dosya isimleri ekleyebiliriz. **ldconfig** uygulaması ile tüm bu *.conf* dosyalarının içerdiği dizinler, ardından */lib* ve */usr/lib* dizinleri dolaşılarak */etc/ld.so.cache* dosyası oluşturulmakta veya güncellenmektedir. */etc/ld.so.cache* içinde kütüphane isimleri ve yol ifadeleri bulunmaktadır. Bu dosya bir yazı dosyası olmadığından içeriğine *ldconfig -p* şeklinde bakabiliriz. Sistemimizde, *cache* dosyasındaki ilk kayıtlar aşağıdaki gibidir.

```
$ ldconfig -p /etc/ld.so.cache | head
1520 libs found in cache `/etc/ld.so.cache'
        libzvbi.so.0 (libc6,x86-64) => /usr/lib/x86_64-linux-gnu/libzvbi.so.0
        libzvbi-chains.so.0 (libc6,x86-64) => /usr/lib/x86_64-linux-gnu/libzvbi-chains
.so.0
        libzephyr.so.4 (libc6,x86-64) => /usr/lib/x86_64-linux-gnu/libzephyr.so.4
        libzeitgeist-2.0.so.0 (libc6,x86-64) => /usr/lib/x86_64-linux-gnu/libzeitgeist
-2.0.so.0
        libzeitgeist-1.0.so.1 (libc6,x86-64) => /usr/lib/libzeitgeist-1.0.so.1
        libzbar.so.0 (libc6,x86-64) => /usr/lib/libzbar.so.0
        libz.so.1 (libc6,x86-64) => /lib/x86_64-linux-gnu/libz.so.1
        libz.so.1 (libc6) => /lib/i386-linux-gnu/libz.so.1
        libz.so (libc6,x86-64) => /usr/lib/x86_64-linux-gnu/libz.so
```

Dinamik bağlayıcı bu *cache* dosyasına bakarak aradığı kütüphanelere hızlı bir şekilde ulaşabilmektedir.

Kütüphanelerin Aranma Sırası

Dinamik bağlayıcı kütüphaneleri belli bir sıraya göre aramaktadır. Öncelik sıralaması aşağıdaki gibidir.

1. Bağımlılık listesinde "/" karakteri varsa kütüphane ismiyle beraber yol ifadesinin de geçirildiği kabul edilir ve arama bu dizinde yapılır. Yol ifadesi mutlak veya göreli olabilir.

2. Dosya DT_RUNPATH listesi içermiyorsa DT_RPATH alanındaki liste aranır. Her iki alanında arama listesi içermesi ilk bakışta anlamsız gelebilir fakat DT_RUNPATH öncesi bağlayıcıların da çalışabilmesi için statik bağlayıcılar DT_RPATH alanına da DT_RUNPATH bilgilerini kopyalabilmektedirler. Bu durumda eski bağlayıcılar yalnız DT_RPATH içeriğine bakarken, yeni bağlayıcılar DT_RUNPATH bir liste içeriyorsa bu aşamada ELF dosya içindeki arama dizinlerini gözardı eder.

3. LD_LIBRARY_PATH çevre değikenin değerine bakılır.

4. DT_RUNPATH listesine bakılır. DT_RUNPATH alanının eklenme nedeni, ELF içindeki arama dizinlerine LD_LIBRARY_PATH değişkeninden sonra bakılmak istenmesidir.

5. /etc/ld.so.cache dosyasına bakılır.

6. /lib ve /usr/lib dizinlerine bakılır.

Statik ve Dinamik Kütüphanelerin Beraber Kullanımı

Bazı kütüphanelerin hem statik hem de dinamik halleri bulunabilmektedir. Bağlayıcı seçim olanağı varsa dinamik kütüphaneleri seçmektedir. Örneğin, kütüphanenin *-lXXX* şeklinde gösterilmesi durumunda, bağlayıcı *libXXX.a* dosyasını değil *libXXX.so* dosyasını seçmektedir.

Bazı durumlarda dinamik kütüphaneler yerine statik hallerini kullanmayı tercih edebiliriz. Bir örnek üzerinden bu durumu inceleyelim.

driver.c:

```
void foo();
void bar();

int main() {
    foo();
    bar();
    return 0;
}
```

foo.c:

```
void foo() {
    puts(__func__);
}
```

bar.c:

```
void bar() {
    puts(__func__);
}
```

foo ve *bar* için sırasıyla statik ve dinamik kütüphaneler oluşturalım.

```
$ gcc -c foo.c
$ gcc -c bar.c
$ ar rcs libfoo.a foo.o
$ ar rcs libbar.a bar.o
```

```
$ gcc -fPIC -shared -olibfoo.so foo.c
$ gcc -fPIC -shared -olibbar.so bar.c
```

Uygulamayı aşağıdaki gibi derlediğimizde kütüphanelerin dinamik hallerinin seçildiğini görmekteyiz.

```
$ gcc -odriver driver.c -L. -lfoo -lbar
$ readelf -d driver | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libfoo.so]
 0x0000000000000001 (NEEDED)             Shared library: [libbar.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Statik kütüphanelerin seçilmesi için aşağıdaki yöntemler uygulanabilir.

1. Kütüphane ismi tam adıyla kullanılabilir.

```
$ gcc -odriver driver.c -L. libfoo.a -lbar
$ readelf -d driver | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libbar.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

2. gcc, **-static** seçeneği kullanılabilir.

```
$ gcc -odriver driver.c -L. -lfoo -lbar -static
$ readelf -d driver | grep NEEDED
```

Bu durumda uygulamanın dinamik bir bağımlılığı olmadığını görüyoruz. Bu seçeneğin kullanılabilmesi için tüm kütüphanelerin statik hallerinin bulunması gerekmektedir. Bu durumu test etmek için *bar* kütüphanesinin statik halinin olmadığını varsayalım, *libbar.a* dosyasını silip uygulamayı yeniden derlemeye çalışalım.

```
$ rm libbar.a
$ gcc -odriver driver.c -L. -lfoo -lbar -static
/usr/bin/ld: cannot find -lbar
collect2: error: ld returned 1 exit status
```

Bağlayıcının yaptığı işlemlere daha yakından bakmak için **--verbose** seçeneğini kullanabilirsiniz.

```
$ gcc -odriver driver.c -L. -lfoo -lbar -static -Wl,--verbose | grep libfoo.a
attempt to open ./libfoo.a failed
attempt to open /usr/lib/gcc/x86_64-linux-gnu/4.9/libfoo.a failed
attempt to open /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../x86_64-linux-gnu/libfoo.a f
ailed
attempt to open /usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../lib/libfoo.a failed
...
```

Bağlayıcının *libfoo.a* dosyasını bulamadığını ve nerelerde arama yaptığını görmekteyiz.

`3.` Tüm uygulamayı statik olarak derlemek yerine, bağlayıcının istenilen kütüphanelerin statik hallerini kullanması sağlanabilir. Statik halleri kullanılmak istenen kütüphaneler **-Wl,-Bstatic** *-lXXX -lYYY* **-Wl,-Bdynamic** şeklinde gösterilebilir. Bu durumda *foo* kütüphanesinin statik, *bar* kütüphanesinin ise dinamik halini kullanabiliriz.

```
$ gcc -odriver driver.c -L. -Wl,-Bstatic -lfoo -Wl,-Bdynamic -lbar
$ readelf -d driver | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libbar.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Bağlayıcı *Bstatic* seçeneğini gördüğünde *Bdynamic* seçeneğini görene kadar yalnız statik kütüphaneleri kullanmaktadır, statik kütüphane bulunamaması durumunda dinamik hali kullanılmamaktadır.

Versiyon Yönetimi

Paylaşımlı kütüphanelerin kodu zaman içerisinde değişebilmekte ve neticesinde yeni versiyonları çıkmaktadır. Yapılan değişikliğin derecesine göre versiyonları iki gruba ayırabiliriz.

- *Major*, önceki versiyonlarla uyumluluğun (compatibility) korunmadığı, genel olarak kapsamlı değişikliklerin yapıldığı versiyonlardır.
- *Minör*, önceki versiyonlarla uyumluluğun korunduğu versiyonlardır.

Örneğin, kütüphanenin arayüzünden bir fonksiyonun çıkarılması durumunda, kütüphanenin eski versiyonlarında bu fonksiyonu kullanan uygulamalar artık yeni halini kullanamayacak ve ABI (Application Binary Interface) uyumluluğu kırılacaktır. Benzer şekilde var olan fonksiyonların ürettikleri sonuçların veya parametrik yapılarının değiştirilmesi durumunda da geçmişe dönük uyumluluk kırılacaktır. Buna karşın, çoğu durumda bu tip değişikliklere ihtiyaç duyulmamaktadır. Kütüphaneye yeni fonksiyonlar eklenmesi veya var olanların parametrik yapıları korunarak iyileştirilmeleri durumunda geçmişe uyumluluğun korunduğu yeni versiyonları çıkarılmaktadır.

Paylaşımlı kütüphanelerin versiyonlarının kontrolünde iki yöntem kullanılabilmektedir. Sırasıyla bu yöntemlere bakalım.

İsimlendirme Geleneği (Naming Conventions)

Kütüphaneler, versiyonlarını da gösterecek biçimde *libisim.so.*[**major**].[**minör**].[**revizyon**] şeklinde isimlendirilmektedir. Revizyon numarası her durumda kullanılmayabilir. Genel yaklaşım versiyon arttıkça versiyon numaralarının değerini 1 arttırmak şeklindedir. Örneğin, sistemimizdeki Qt kütüphanesinin aşağıdaki gibi isimlendirildiğini görmekteyiz.

```
libQtCore.so.4.8.6
```

Bir uygulamanın, bağımlı olduğu kütüphanenin tüm *minör* versiyonlarıyla çalışması istenmektedir. Uygulamanın bağımlılık listesine kütüphanenin gerçek isminin yazılması durumunda, uygulama kütüphanenin o versiyonuna katı bir şekilde bağlanacak ve başka uyumlu versiyonlarıyla çalışamayacaktır. Bir örnek üzerinden bu durumu inceleyelim.

driver.c:

```
#include <stdio.h>

void foo(int);

int main() {
    int ret = foo(16);
    printf("%d\n", ret);
    return 0;
}
```

test.c:

```
void foo(int val) {
    return val / 2;
}
```

Kütüphanemizi isimlendirme kurallarına uygun isimlendirip, ardından uygulamamızı derleyip çalıştırabiliriz.

```
$ gcc -fPIC -shared -olibtest.so.1.0.0 test.c
$ gcc -odriver driver.c libtest.so.1.0.0
$ LD_LIBRARY_PATH=. ./driver
8
```

Uygulamımızın bağımlılık listesinde beklediğimiz üzere *libtest.so.1.0.0* kütüphanesini görmekteyiz.

```
$ readelf -d driver | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libtest.so.1.0.0]
```

Kütüphane kodunda, uyumluluğu gözeterek, bir iyileştirme yaptığımızı ve kütüphanenin revizyon numarasını 1 arttırarak, *libtest.so.1.0.1* adıyla kütüphaneyi yeniden derlediğimizi düşünelim

```
gcc -fPIC -shared -olibtest.so.1.0.1 test.c
```

Bu durumda *driver* uygulamasının yeni kütüphaneyi kullanamayacağı açıktır. Uygulamanın yeni kütüphaneye statik olarak yeniden bağlanması gerekmektedir. Bu problemi gidermek için kütüphanelere **soname** denilen mantıksal isimler verilmektedir. Mantıksal isimler kütüphane ismiyle beraber yalnız major versiyon numarasını içermektedir. Bu sayede birbiriyle uyumlu olan tüm versiyonlar aynı mantıksal ismi paylaşmakta ve bu şekilde ortak bir isim alanı oluşturulmaktadır. Bu mantıksal isimler kütüphaneler oluşturulurken, bağlayıcı

tarafından kütüphaneye dosyasına yerleştirilmektedir. Bu amaçla, ELF formatında DT_SONAME alanı bulunmaktadır. Bir uygulamanın kütüphaneye bağlanması aşamasında, uygulamanın bağımlılık listesine, kütüphanenin adı değil mantıksal ismi (soname) yazılmaktadır. Bu kez kütüphanemize mantıksal bir isim atayarak yeniden derleyelim.

```
$ gcc -fPIC -shared -Wl,-soname,libtest.so.1 -olibtest.so.1.0.0 test.c
$ gcc -odriver driver.c libtest.so.1.0.0
```

Kütüphanenin mantıksal ismini aşağıdaki gibi öğrenebiliriz.

```
$ readelf -d libtest.so.1.0.0 | grep SONAME
 0x000000000000000e (SONAME)             Library soname: [libtest.so.1]
```

Uygulamayı çalıştırmayı denediğimizde, bağlayıcının *libtest.so.1.0.0* dosyasını değil *libtest.so.1* dosyasını aradığını görmekteyiz.

```
$ ./driver
$ LD_LIBRARY_PATH=. ./driver: error while loading shared libraries: libtest.so.1: cann
ot open shared object file: No such file or directory
```

Bu durumda gerçek kütüpnane dosyasına, kütüphanenin mantıksal isminde bir sembolink link oluşturabiliriz.

```
$ ln -s libtest.so.1.0.0 libtest.so.1
$ LD_LIBRARY_PATH=. ./driver
8
```

Bu kez uygulamanın çalıştığını görmekteyiz. Şimdi daha önce hedeflediğimiz gibi, uyumluluğu koruyarak, kütüphanede bir değişiklik yapıp revizyon numarasını 1 arttıralım.

test.c:

```
int foo(int val) {
    puts("İyileştirilmiş versiyon");
    return val >> 1;
}
```

```
gcc -fPIC -shared -Wl,-soname,libtest.so.1 -olibtest.so.1.0.1 test.c
```

Uygulama üzerinde herhangi bir işlem yapmaksızın yalnız sembolik bağlantıyı yeni kütüphaneyi gösterecek şekilde değiştirelim.

```
$ rm libtest.so.1
$ ln -s libtest.so.1.0.1 libtest.so.1
```

Uygulamayı yeniden çalıştırdığımızda kütüphanenin yeni versiyonunu kullandığını görmekteyiz.

```
LD_LIBRARY_PATH=. ./driver
İyileştirilmiş versiyon
8
```

Ayrıca derleme zamanında kütüphanenin tam ismini yazmak yerine versiyon bilgisi içermeyen bir bağlayıcı adı (linker name) kullanılmaktadır. Bu sebeple kütüphanenin yalnız adını içeren bir sembolik link daha oluşturulmaktadır. Bu link gerçek kütüphaneyi gösterebilmesine karşın, mantıksal isim linkini göstermesi daha kullanışlıdır. Gerekli sembolik bağlantıyı aşağıdaki gibi oluşturup, derleme zamanında kütüphanenin gerçek adı yerine kullanabiliriz.

```
$  ln -s libtest.so.1 libtest.so
$  gcc -odriver driver.c libtest.so.1.0.0
$  gcc -odriver driver.c -L. -ltest
$  LD_LIBRARY_PATH=. ./driver
İyileştirilmiş versiyon
8
```

Örnek kütüphanemiz için isimlendirmeye ilişkin durumu özetleyecek olursak:

İsimlendir	Dosya Adı
Gerçek İsim (Real Name)	libtest.so.1.0.0
Mantıksal İsim (Soname)	libtest.so.1
Bağlayıcı İsmi (Linker Name)	libtest.so

İlgili dosyalar ise aşağıdaki gibidir.

```
$ ls -l libtest.so* | awk '{print $9, $10, $11}'
libtest.so -> libtest.so.1
libtest.so.1 -> libtest.so.1.0.1
libtest.so.1.0.0
libtest.so.1.0.1
```

Bu yöntemde, uygulamayı yeniden oluşturmak zorunda kalmamamıza karşın kütüphaneye olan sembolik bağlantıyı yönetmek zorundayız. Yeni bir uyumlu versiyon yüklendiğinde soname bağlantısı yeni kütüphaneyi göstermeli, yeni bir major versiyon yüklendiğinde ise

gerekli soname bağlantısı oluşturulmalı. Bu işlemler için daha önce de bahsettiğimiz **ldconfig** aracını kullanabiliriz. Örneğimiz üzerinden bu durumu inceleyelim.

Kütüphanemizin ilk versiyonunu, *libtest.so.1.0.0*, /usr/lib altına kopyalayım ve ardından ldconfig aracını çalıştırıp neler olduğuna bakalım.

```
# cp libtest.so.1.0.0 /usr/lib
# ldconfig -v | grep libtest
...
    libtest.so.1 -> libtest.so.1.0.0 (changed)
```

ldconfig tarafından /usr/lib altında soname bağlantısının oluşturulduğunu ve /etc/ld.so.cache dosyasına kütüphanemizle ilgili bir girişin eklendiğini görüyoruz.

```
$ ls -l /usr/lib/libtest.so*
lrwxrwxrwx 1 root root   16 Mar 15 18:42 /usr/lib/libtest.so.1 -> libtest.so.1.0.0
-rwxr-xr-x 1 root root 7848 Mar 15 18:42 /usr/lib/libtest.so.1.0.0
```

```
$ ldconfig -p /etc/ld.so.cache | grep libtest
    libtest.so.1 (libc6,x86-64) => /usr/lib/libtest.so.1
```

Bu durumda uygulamamızı, LD_LIBRARY_PATH değişkenini kullanmadan, çalıştırıdığımızda kütüphanenin eski versiyonuyla çalışacaktır.

```
$ ./driver
8
```

Şimdi kütüphanenin yeni versiyonunun yine /usr/lib altına atalım ve ldconfig uygulamasını çalıştıralım.

```
# cp libtest.so.1.0.1 /usr/lib
# ldconfig -v | grep libtest

    libtest.so.1 -> libtest.so.1.0.1 (changed)
```

soname bağlantısının bu kez yeni versiyonu gösterdiğini görüyoruz.

```
$ ls -l /usr/lib/libtest.so*
lrwxrwxrwx 1 root root   16 Mar 15 18:47 /usr/lib/libtest.so.1 -> libtest.so.1.0.1
-rwxr-xr-x 1 root root 7848 Mar 15 18:42 /usr/lib/libtest.so.1.0.0
-rwxr-xr-x 1 root root 7992 Mar 15 18:47 /usr/lib/libtest.so.1.0.1
```

Uygulamayı çalıştırdığımızda kütüphanenin yeni versiyonunun kullanıldığını görmekteyiz.

```
$ ./driver
İyileştirilmiş versiyon
8
```

Sembollere Versiyon Atanması (Symbol Versioning)

Sembollere versiyon atayarak, aynı kütüphane içinde bir fonksiyonun birden çok versiyonunun barındırılması hedeflenmektedir. Kütüphane içinde uyumluluğu bozacak değişiklikler yapılmasına karşın, uygulamalar kütüphane içinde bağlandıkları fonksiyonları kullanmaya devam edebilmektedir. Bu sayede bir önceki yöntemde gördüğümüz gibi kütüphanenin yeni versiyonlarını çıkarmaya gerek kalmamaktadır.

Bu amaçla, bağlayıcı versiyon betikleri (Linker Version Scripts) kullanılmaktadır. Versiyon betikleri statik bağlayıcı, ld, tarafından kullanılan, genellikle *.map* uzantılı, yazı dosyalarıdır. Temel olarak süslü parantezlerle gruplanmış ve başında versiyon etiketi olan düğümlerden oluşmaktadır. Basit bir örneği aşağıdaki gibidir.

```
VER_1 {
    global: foo;
    local: *; # Diğer semboller gizlenmekte
};
```

global ve *local* anahtar kelimeleri kütüphane içinde sembollerin görünürlüğünü belirlemektedir. *local* olarak belirtilen semboller kütüphane dışından kullanılamamaktadır. `*`, açık bir şekilde global olarak belirtilen sembollerin dışındaki tüm sembolleri göstermektedir.

Bir önceki örneğimizi bu kez versiyon betiği kullanarak yapalım. İlk olarak, kütüphanemizin ilk halini yukarıda gösterdiğimiz örnek versiyon betiğini kullanarak oluşturalım. Versiyon betiğini *ver1.map* olarak isimlendirebiliriz.

```
$ gcc -fPIC -shared -olibtest.so test.c -Wl,--version-script,ver1.map
```

Kütüphane içindeki *foo* ile ilgili semboller aşağıdaki gibidir.

```
$ nm libtest.so | grep foo
0000000000000630 T foo

$ readelf -s libtest.so | grep foo
     8: 0000000000000630    21 FUNC    GLOBAL DEFAULT   12 foo@@VER_1
    50: 0000000000000630    21 FUNC    GLOBAL DEFAULT   12 foo
```

nm çıktısında yalnız *foo* sembolünü görmemize karşın, readelf ile bir de *foo@@VER_1* sembolünün bulunduğunu görmekteyiz. Paylaşımlı kütüphaneler ve bu kütüphanelere bağımlılığı olan çalışabilir dosyalar içinde *.symtab* ve *.dynsym* olmak üzere 2 tane sembol tablosu tutulmaktadır. *.symtab* kütüphane içinde tüm sembolleri içermekte ve statik bağlanım sırasında kullanılmaktadır. *.dynsym* ise yalnız global sembolleri içeren daha küçük bir tablodur ve dinamik bağlayıcı tarafından kullanılmaktadır. readelf çıktısındaki *foo@@VER_1* sembolü *.dynsym* tablosunda bulunmaktadır. Normalde **@** karakterinin sembol isimlerinde kullanılmasına izin verilmez. Bu sayede bu sembolün versiyonu olduğunu anlaşılmaktadır.

Şimdi uygulamamızı *libtest.so* ile, ara dosyaları da saklayacak şekilde, derleyelim.

```
$ gcc -odriver driver.c -L. -ltest --save-temps
```

Amaç koda baktığımızda *foo* sembolünün beklediğimiz isimde saklandığını görmekteyiz.

```
$ readelf -s driver.o | grep foo
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND foo
```

Uygulama içinde ise *foo* sembolünün sırasıyla .dynsym ve .symtab bölümlerinde versiyon bilgisiyle beraber oluşturulduğunu görüyoruz. Bağlayıcı uygulama içinde çağrılan *foo* sembolünün tanımını ararken kütüphane içinde *foo@@VER_1* sembolüne ulaşmış ve amaç kod içindeki .dynsym tablosundaki *foo* sembolünü *foo@VER_1* olarak değiştirmiş. Sembollerdeki @@ sembolün versiyonunun diğerlerine tercih edilmesini sağlamaktadır. Neden hem @@ hem de @ karakterlerinin kullanıldığını birden çok versiyon olduğunda daha iyi anlayacağız.

```
$ readelf -s driver | grep foo
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND foo@VER_1 (3)
    61: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND foo@@VER_1
```

Uygulamanın beklediğimiz gibi çalıştığını görmekteyiz.

```
$ LD_LIBRARY_PATH=. ./driver
8
```

Şimdi *foo* fonksiyonunun eski halini koruyarak yeni bir versiyonunu nasıl ekleyebileceğimize örnek üzerinden bakalım. Ayrıca kütüphaneye *bar* isimli yeni bir fonksiyon da ekledik. Kütüphane kodunun yeni hali aşağıdaki gibidir.

```
#include <stdio.h>

__asm__(".symver foo_old,foo@VER_1");
__asm__(".symver foo_new,foo@@VER_2");

int foo_old(int val) {
    return val / 2;
}

int foo_new(int val) {
    puts("İyileştirilmiş versiyon");
    return val >> 1;
}

void bar() {
    printf("v2 bar\n");
}
```

Kütüphane içinde sembolik makina direktifleri görmekteyiz. **.symver** ile bir sembolün başka isimle bir kopyası (alias) oluşturulur. Bu takma isim normalde izin verilmeyen @ karakterini içerebilmektedir. Genel şekli aşağıdaki gibidir.

```
.symver name, name2@nodename
```

name2 sembolün gerçek adını, *nodename* ise versiyon bilgisini göstermektedir. Kütüphanenin bir önceki versiyonunda uygulamamanın *foo@VER_1* sembolüne bağlandığını hatırlayınız. Bu sembol şimdi *foo_old* fonksiyonunu göstermektedir. Kütüphanenin yeni halini kullanacak uygulamalar ise *foo@@VER_2* sembolünü dolayısıyla *foo_new* fonksiyonunu kullanacaklardır. @@ karakterleri versiyonun öncelikli olduğunu göstermektedir. Bu aşamada versiyon betiğine *VER_2* düğümünü eklemeliyiz. Aşağıdaki versiyon betiğini ver2.map adıyla saklayıp kullanabiliriz. *VER_2* düğümünün sonundaki *VER_1*, iki versiyon arasında ilişki kurmakta ve ilk versiyondaki *global* ve *local* bildirimlerinin ikinci versiyonda da geçerli olmasını sağlamaktadır.

```
VER_1 {
    global: foo;
    local: *;
};

VER_2 {
    global: bar;
} VER_1;
```

Kütüphaneyi yeniden derleyelim ve *foo*'ya ilişkin sembollere bakalım.

```
$ gcc -fPIC -shared -olibtest.so test.c -Wl,--version-script,ver2.map
```

```
$ readelf -s libtest.so | grep foo
     9: 0000000000000710    21 FUNC    GLOBAL DEFAULT   12 foo@VER_1
    10: 0000000000000725    30 FUNC    GLOBAL DEFAULT   12 foo@@VER_2
    40: 0000000000000725    30 FUNC    LOCAL  DEFAULT   12 foo_new
    44: 0000000000000710    21 FUNC    LOCAL  DEFAULT   12 foo_old
    ...
```

foo_old ve *foo_new* sembollerinin LOCAL yani dışsal bağlanıma kapalı olduğunu, bunun yerine GLOBAL düzeydeki takma isimlerinin kullanıldığını görüyoruz.

Uygulamamıza bu kez *driver2* ismini vererek, kütüphanenin bu haliyle derleyelim ve *foo* çağrısının hangi sembol ile temsil edildiğine bakalım.

```
gcc -odriver2 driver.c -L. -ltest --save-temps
```

```
$ readelf -s driver | grep foo
    3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND foo@VER_2 (3)
    ...
```

Uygulamamızı çalıştırdığımızda *foo* fonksiyonunun yeni versiyonunun kullanıldığını görmekteyiz.

```
$ LD_LIBRARY_PATH=. ./driver2
İyileştirilmiş versiyon
8
```

Kütüphanenin eski versiyonuna bağımlı uygulamamız da beklediğimiz gibi çalışmakta.

```
$ LD_LIBRARY_PATH=. ./driver
8
```

Sembol versiyonları kullanılarak, kütüphane kodunda kapsamlı değişlikler yapılmasına karşın, tek bir kütüphane dosyası ile tüm versiyonlar kullanılabilmektedir. glibc 2.1 versiyonundan beri kütüphane içinde versiyon sembolleri kullanılmakta ve tek bir major kütüphane dosyası, libc.so.6, bulunmaktadır.

Process'ler Arası Haberleşme

Memory Allocation

Türkçe karşılığında en çok zorlandığımız terminolojilerin biri de bu meşhur Memory Allocation.

Bellek tahsisatı diyenler var, bellek edinimi şeklinde kullanımlarını da görüyoruz. Tahsisat fena durmuyor aslında ama tam da içimize sinmiyor. Allokasyon desek olmayacaktı, o yüzden şimdilik olduğu gibi bırakıyorum.

Konumuza geçelim.

Allocation gerçekte nasıl yapılır?

Bu soru şimdiye kadar hiç aklınıza takıldı mı? Yazılım mühendislerinin ekseri çoğunluğu bu sürecin detaylarını pek bilmez. Ama sistem programcısı adayı olarak bu satırları okuyorsanız, bu süreç hakkında daha fazla bilgi sahibi olmanız gerekir.

Allocation sürecine Linux ve *glibc* kütüphanesi özelinde biraz daha detaylı bakmaya çalışalım.

Uygulamalarda bellek ihtiyacımız olduğunda, işletim sisteminden bu alanı talep etmemiz gerekir. Çekirdekten yapılacak bu talep doğal olarak sistem çağrısı gerektirecektir; kullanıcı kipinde kendi kendimize bellek tahsisatı yapamayız.

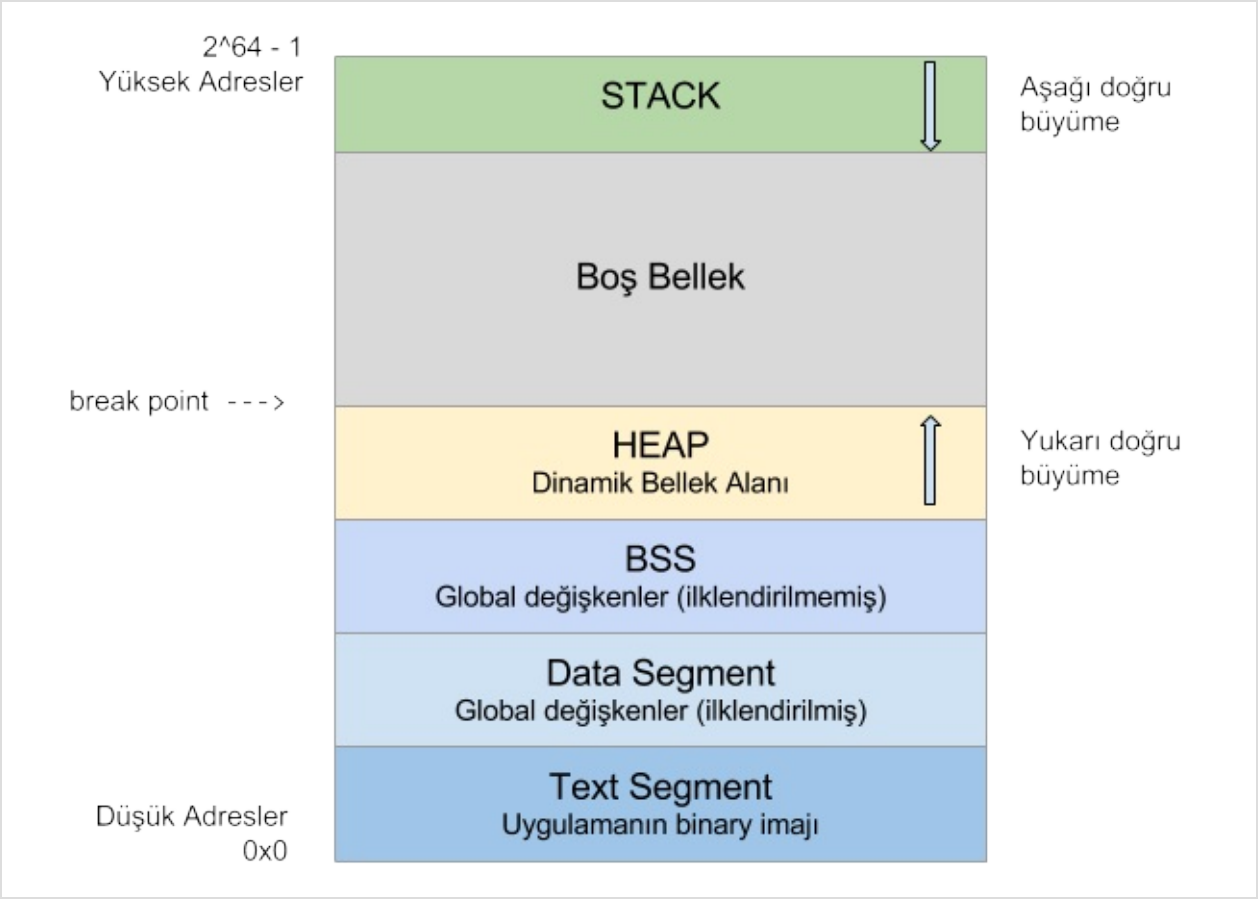
> Bu noktada henüz okumadıysanız Sistem Çağrıları bölümüne hızlıca göz atmanız önerilir.

C dilinde allocation için temel olarak `malloc()` fonksiyon ailesi kullanılır. Peki bir *glibc* fonksiyonu olarak `malloc()` doğrudan bir sistem çağrısında mı bulunuyordur, yani `open()` fonksiyonu ve sistem çağrısı örneğindeki gibi birebir bir karşılığı mevcut mudur?

Linux çekirdeğinde `malloc` adında bir sistem çağrısı bulunmamaktadır.

Uygulamaların bellek talepleri için 2 adet sistem çağrısı mevcut olup bunlar sırasıyla `brk` ve `mmap` çağrılarıdır.

Bizler uygulamamızda bellek talebini *glibc* fonksiyonları aracılığıyla yaptığımızdan bu noktada aklınıza *glibc*'nin bu sistem çağrılarından hangisini kullanıyor olduğu sorusu gelebilir. Cevabımız, her ikisini de kullandığı yönünde olacaktır. Peki o zaman bu iki sistem çağrısı arasındaki fark nedir ve hangi durumlarda kullanılmaktadır?



Process Bellek Yerleşimi (sadeleştirilmiş hali)

`brk`

Her process, ardışıl bir (contiguous) *data* alanına sahiptir. `brk` sistem çağrısı ile bu alanın sınırını belirleyen **program break** değeri artırılarak *allocation* işlemi de gerçekleştirilmiş olur.

> **NOT:** *data* alanını büyütmekten kastımız HEAP (Program Break) sınırını yukarı yönlü kaydırmaktır. Data Segment ve BSS'in sınırları uygulama için sabittir ve çoğu dokümanda HEAP alanını büyütmek yerine *data* alanını büyütmek tabiri kullanılmakta, bu da Data Segment ile karıştırılabilmektedir. Data alanını birbirine komşu olan (Data Segment + BSS + HEAP) şeklinde düşünecek olursanız anlaşılması daha kolay olacaktır.

Bu yöntemle yapılan bellek tahsisatları çok hızlı olmasına rağmen, önemli bir dezavantaja sahiptir. Ardışıl yapısı nedeniyle, artık kullanılmayan bir alanı sisteme geri vermek her zaman mümkün olmaz.

Örnek olarak her biri 16 KB büyüklüğünde 5 adet alanı sırayla `malloc()` fonksiyonu üzerinden `brk` sistem çağrısı ile tahsis ettiğimizi düşünelim. Bu alanlardan 2 nolu olanla işimiz bittiğinde, ilgili kaynağı sistemin kullanabilmesi için geri vermemiz (*deallocation*)

mümkün değildir zira `brk` çağrısıyla adres değerini 2 nolu alanımızın başladığı yeri gösterecek şekilde azaltacak olursak, 3, 4 ve 5 nolu alanlar için de *deallocation* işlemi yapmış oluruz.

Glibc içerisindeki `malloc` implementasyonu bu senaryodaki bellek kaybını önleyebilmek adına, process *data* alanında bu şekilde tahsis edilmiş ve sonrasında `free()` fonksiyonu ile artık sisteme geri verilebileceği belirtilmiş olan yerleri, sonraki bellek tahsisatlarında kullanmak üzere takip eder.

Yani 16 KB'lık 5 adet alan tahsis edildikten sonra 2 nolu alan `free()` fonksiyonuyla geri verilmek istenip bir süre sonra tekrar bir 16 KB'lık alan daha istenecek olursa, `brk` sistem çağrısı aracılığıyla *data* alanını büyütmek yerine, hazırdaki 2 nolu alanın adresi geri dönülür.

Ancak eğer yeni talep edilen alan bu örneğimiz için 16 KB'dan büyük ise, bu durumda 2 nolu alan kullanılamayacağından, `brk` sistem çağrısı ile yeni bir alan daha ayrılarak *data* alanı büyütülmüş olur, 2 nolu alan ise kullanımda olmamasına rağmen sisteme geri de verilemez ve yer harcamaya devam eder. İşte bunun gibi senaryolar nedeniyle **internal fragmentation** diye adlandırılan durum oluşur ve aslında belleğin her tarafını sonuna kadar hemen hiç bir zaman kullanamayız. Meşhur Türk özdeyişi, *restart kan yapar*'ın kökeni esasen buralara kadar dayanmaktadır.

Aşağıdaki örnek uygulamayı derleyip çalıştırmayı deneyiniz.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
    char *ptr[7];
    int i;

    printf("Pid of %s: %d\n", argv[0], getpid());
    printf("Initial program break        : %p\n", sbrk(0));
    for (i = 0; i < 5; i++) ptr[i] = malloc(16 * 1024);
    printf("After 5 x 16KB malloc        : %p\n", sbrk(0));
    free(ptr[1]);
    printf("After free of second 16KB    : %p\n", sbrk(0));
    ptr[5] = malloc(16 * 1024);
    printf("After allocating 6th of 16KB : %p\n", sbrk(0));
    free(ptr[5]);
    printf("After freeing last block     : %p\n", sbrk(0));
    ptr[6] = malloc(18 * 1024);
    printf("After allocating a new 18KB  : %p\n", sbrk(0));
    getchar();
    return 0;
}
```

Uygulama çıktısı:

```
Pid of ./m1: 5877
Initial program break        : 0x9f6000
After 5 x 16KB malloc        : 0xa1b000
After free of second 16KB    : 0xa1b000
After allocating 6th of 16KB : 0xa1b000
After freeing last block     : 0xa1b000
After allocating a new 18KB  : 0xa1b000
```

`strace` ile `brk` için çıktısı:

```
$ strace -e trace=brk ./m1 > /dev/null
...
brk(0)                                  = 0x9f6000
brk(0xa1b000)                           = 0xa1b000
```

Yukarıdaki çıktıda `brk(0)` şeklindeki özel çağrı yöntemiyle (parametre olarak 0 geçirilmesi) *data* alanının mevcut bitiş adresinin öğrenildiğini görüyoruz (`0x9f6000`). Ardından üzerine `0x25000` eklenerek *data* alanının bitiş adresi `0xa1b000` değerine kaydırılmak istenmiş ve herhangi bir hata alınmamış. Dolayısıyla tam bu noktada `0x25000` yani yaklaşık 148 Kb boyutunda bellek tahsisatı yapıldığını görmekteyiz. Şu soruları kendimize soralım:

- Peki neden tam olarak örnek kodumuzdaki miktar kadar değil de ondan biraz daha fazla allocation gerçekleşti?
- Allocation'ı sağlayan `brk` çağrısı kaynak kodumuzda tam olarak hangi satırdan kaynaklandı? İlk `malloc()` çağrısında mı gerçekleşti yoksa derleyici bir optimizasyon yapıp toplamdaki talebimizden biraz fazlasını talep ederek fazladan sistem çağrısı yapmamızı mı önledi?

Bu soruların yanıtlarını kendiniz bulmaya çalışın, öğretici bir süreç olacaktır.

Address Space Layout Randomization: ASLR

Yukarıdaki örnek uygulamamızı arka arkaya çalıştırdığınızda, her defasında farklı adres değerleri görülecektir. Bu konuyu **Address Space Layout Randomization** (ASLR) başlığı altında ayrıca detaylandırmamız gerekiyor. Şimdilik şu kadarını söylemekle yetinelim, adres alanını bu şekilde rastgele değişecek hale getirmek, yazılımlara yönelik güvenlik ataklarının işini önemli ölçüde zorlaştırmakta ve yazılım güvenliğini artırmaktadır. Bununla birlikte 32 bitlik mimarilerde adres alanı rastgele hale getirmek için genelde 8 bit kullanılır. Bit sayısını artırmak, geriye kalan bit'ler üzerinden adreslenebilecek alan çok düşeceğinden uygun olmamakta, bununla birlikte sadece 8 bitlik kombinasyonların kullanımı da saldırgan açısından işleri yeterince zorlaştırmamaktadır. 64 bitlik mimarilerde ise ASLR işlemi için ayrılabilecek fazla fazla bit olduğundan çok daha geniş bir rastgelelik sağlanmakta ve

güvenlik derecesi artmaktadır. Android tabanlı sistemlerde de Linux çekirdeği kullanılmaktadır ve ASLR özelliği Android 4.0.3 ve sonrasında tam olarak aktifleştirilmiştir. Sadece bu sebeple bile olsa, 64 bitlik bir akıllı telefonun 32 bitlik versiyonlarına göre önemli bir güvenlik avantajı sağladığını söylememiz yanlış olmayacaktır.

ASLR özelliğini aşağıdaki komutla geçici olarak devre dışı bıraktığınızda, bir önceki test uygulamasının her çalıştırıldığında aynı adres değerlerini verdiği görülecektir:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Önceki haline geri döndürmek için aynı dosyaya **0** yerine **2** yazmanız yeterli olacaktır.

`mmap`

Bellek tahsisatı için kullanılan ikinci sistem çağrısıdır. `brk` sistem çağrılarıyla *data* alanını sadece tek yönlü kaydırabildiğimiz ve doğası itibariyle **internal fragmentation** ürettiği için, farklı bir yönteme de ihtiyaç duyulmuştur.

`mmap` çağrısı ile belleğin herhangi bir alanıdaki boş yer, çağrıyı yapan process'in adres uzayına haritalanır (*mapping*).

Bu şekilde yapılan bir bellek tahsisatında, bir önceki `brk` örneğimizdeki 5 adet 16 Kb'lık bölümden ikincisini `free()` fonksiyonu ile geri vermek istediğimizde, bu işlemi engelleyecek bir mekanizma bulunmaz ve ilgili bellek bölümü process'in adres uzayından çıkartılıp artık kullanmıyor şeklinde işaretlenerek sisteme iade edilir.

Peki madem `mmap` bu kadar iyi ve **internal fragmentation** oluşturmayacak bir model ile çalışıyor, neden sadece bunu kullanmıyoruz?

Çünkü `mmap` ile yapılan bellek tahsisatları, `brk` ile yapılanlara oranla inanılmaz yavaştır.

`mmap` ile belleğin herhangi boş alanındaki bir bölüm process'in adres uzayına haritalandığından, bu işlem tamamlanmadan önce tahsis edilen alanın içeriği sıfırlanır. Eğer bu şekilde sıfırlama yapılmıyor olsaydı, ilgili bellek alanını daha önce kullanan process'e ait verilere konuyla hiç alakası olmayan sonraki tahsisatı yapan process de ulaşabilir olurdu ki bu da sistemlerde güvenlik diye bir konudan bahsetmeyi olanaksız hale getirirdi.

Memory Barriers

Hata Ayıklama Yöntemleri

GNU Debugger

Bu bölümde hata ayıklayıcı olarak, GNU sistemlerinde standart olan, **GDB** (GNU Debugger) uygulamasını inceleyeceğiz. `gdb` bir komut satırı uygulaması olarak çalışmakta ve C, C++, Objective-C, assembly ve Java olmak üzere bir çok dili desteklemektedir. gdb ayrıca, Eclipse, Qt Creator, NetBeans gibi bir çok IDE ve GNU DDD (Data Display Debugger) grafik önyüz uygulaması üzerinden de kullanılabilir.

gdb ile bir programın içsel durumunu inceleyebilir, işleyişine müdahale edebiliriz. Tipik olarak, kodu adım adım işletebilir, değişken ve yazmaçların değerlerini gözleyip değiştirebilir, kod üzerinde kesme noktaları belirleyebilir, akışa müdahale edebilir ve fonksiyonların çağrılma sırasını takip edebiliriz. gdb çok sayıda komut ve seçeneği barındırmasına karşın çoğu durumda göreli olarak daha az bir komut setiyle bir çok işi yapabilmekteyiz. Bu bölümde temel kullanım senaryolarına değinmeye çalışacağız.

GDB Kullanımı

İlk olarak, programların debug amaçlı olarak nasıl derlendiğine ve ardından gdb ile nasıl çalıştırıldığına bakalım.

Debug Amaçlı Derleme

Bir programın içsel durumunu etkin bir şekilde inceleyebilmek için çalışabilir dosya içinde debug sembolleri bulunmalıdır. Bu sembollerin bulunmaması durumunda gdb aşağıdaki gibi bir uyarı verecektir.

```
Reading symbols from debug...(no debugging symbols found)...done.
```

Uygulama içerisine debug sembollerini eklemek için, derleme sürecinde tipik olarak `-g` ve `-ggdb` anahtarları kullanılmaktadır. `-ggdb` anahtarı ile, `-g` anahtarından farklı olarak, yalnız gdb'nin kullanabileceği özel bazı semboller de üretilmektedir. Bu anahtarların ayrıca, fazladan bilgi üreten, seviye gösteren kullanımları da mevcuttur. Örneğin önişlemci makro tanımları için `-g3` veya `-ggdb3` anahtarları kullanılmalıdır. Makro kullanımına ilerideki bölümlerde değineceğiz.

Uygulamanın debug modda derlenmesi, çalıştırılabilir dosya formatına yeni alanları eklediğinden, kodun bir miktar büyümesine neden olacaktır. Aşağıdaki gibi basit bir kodu önce normal, sonrasında debug modda derleyerek karşılaştıralım.

```c
int main() {
    int i = 111;
    return 0;
}
```

Kodu debug.c adıyla saklayıp sırasıyla aşağıdaki gibi derleyip, gcc tarafından üretilen dosyaları inceleyebilirsiniz.

```
$ gcc -odebug debug.c --save-temps
$ ls -lh debug.s
  -rw-r--r-- 1 root root 384 Şub  9 11:18 debug.s

$ readelf -S debug | wc -l
69

$ gcc -odebug debug.c --save-temps -g
$ ls -lh debug.s
  -rw-r--r-- 1 root root 2,5K Şub  9 11:18 debug.s

$ readelf -S debug | grep -i debug
  [27] .debug_aranges    PROGBITS         0000000000000000  00001080
  [28] .debug_info       PROGBITS         0000000000000000  000010b0
  [29] .debug_abbrev     PROGBITS         0000000000000000  00001113
  [30] .debug_line       PROGBITS         0000000000000000  0000115b
  [31] .debug_str        PROGBITS         0000000000000000  00001197
```

Bu örnek için üretilen sembolik makina kodunun 384B'tan 2.5K'ya çıktığını ve çalışabilir dosya formatına yeni bölümlerin eklendiğini görmekteyiz.

> **Not:** Linux altında, derleyicinin ürettiği amaç dosyalar, paylaşımlı kütüphaneler ve çalışabilir dosyalar ELF (Executable and Linkable Format) formatında saklanmaktadır. ELF formatı çok sayıda bölümden oluşmaktadır. readelf aracı ile ELF dosya formatını inceleyebilir, S anahtarı ile bölüm başlıklarını (section headers) listeleyebilirsiniz.

Programların GDB İle Çalıştırılması

İncelenecek olan programlar gdb ile çalıştırılabildiği gibi çalışan uygulamalar da, proses kimlikleri kullanılarak, gdb üzerinden incelenebilir. Uygulama isimlerini ve proses kimliklerini gdb'ye argüman olarak geçirebildiğimiz gibi aynı işlemleri gdb komut satırından da yapabiliriz. Tipik kullanımlar aşağıdaki gibidir.

Kullanım
$ gdb programismi
(gdb) file programismi
$ gdb -p proseskimliği
(gdb) attach proseskimliği

Program isminin belirtildiği, tablodaki ilk iki kullanım şeklinde, uygulama öncelikle gdb tarafından yüklenmektedir. Uygulamayı çalıştırmak için sonrasında **run** komutunu kullanabilirsiniz.

> **Not:** gdb açılışta sahiplik bilgilerini de içeren bir karşılama mesajı basmaktadır. Komut satırından `-q` anahtarını kullanarak bu mesajın görüntülenmesini engelleyebilirsiniz.

Programlara Komut Satırı Argümanlarının Geçirilmesi

Komut satırı argümanlarını aşağıda gösterilen 3 farklı şekilde de geçirmek mümkündür.

```
$ gdb -q --args debug ARGUMENT
(gdb) set args ARGUMENT
(gdb) run ARGUMENT
```

Uygulamaya geçirilen argümanlar aşağıdaki gibi listelenebilir.

```
(gdb) show args
Argument list to give program being debugged when it is started is "ARGUMENT".
```

Uygulamanın Çalışmasının Sonlandırılması ve Askıya Alınması

kill komutu ile uygulama sonlandırabilir, **Ctrl-C** tuşlarıyla uygulamanın çalışmasını geçici olarak durdurabilirsiniz.

GDB'nin Sonlandırılması

gdb uygulamasından **quit** komutu veya **Ctrl-D** seçenekleriyle çıkabilirsiniz.

Yardım

Komut satırında **help** yazarak değişik seviyelerde yardım alabilirsiniz. Yalnız help yazarak genel yardım listesini alabilir, sonrasında ilgilendiğiniz komuta ulaşarak daha detaylı bilgi alabilirsiniz. Örnek bir kullanım aşağıdaki gibi olabilir.

```
(gdb) help
  List of classes of commands:

  aliases -- Aliases of other commands
  breakpoints -- Making program stop at certain points
  data -- Examining data
  ...
```

breakpoints kullanımı ile ilgili olduğumuzu var sayalım.

```
(gdb) help breakpoints
  Making program stop at certain points.

  List of commands:

  awatch -- Set a watchpoint for an expression
  break -- Set breakpoint at specified line or function
  break-range -- Set a breakpoint for an address range
  ...
```

Nihayetinde gerçek break komut ile ilgili yardım alabiliriz.

```
(gdb) help break
  Set breakpoint at specified line or function.
  ...
```

Çoğu durumda, bir gdb komutunun tamamını yazmaksızın, sadece diğer komutlardan ayrım yapılabilecek kadar olan kısmını yazarak, komutu kullanabilirsiniz. Ayrıca tab tuşana basarak gdb'nin kodu tamamlamasını veya adayları listelemesini sağlayabilirsiniz. Örneğin aşağıdaki komutların hepsi aynı sonucu üretecektir.

```
(gdb) disas main
(gdb) disass main
(gdb) disasse main
(gdb) disassem main
(gdb) disassemb main
(gdb) disassembl main
(gdb) disassemble main
```

GDB Temel Özellikleri

Bu bölümde, gdb kullanarak, bir uygulama hakkında nasıl bilgi alabileceğimize ve işleyişine nasıl müdahale edebileceğimize bakacağız.

Kaynak Kodun Listelenmesi

Kaynak kod **list** komutu ile listelenebilir. Aldığı argümanlar ve temel kullanım şekilleri aşağıdaki gibidir.

Komut	Açıklama
list	Son listelenen noktadan ya da kodun başından itibaren 10 satırı görüntüler
list SATIRNUMARASI	İstenilen noktayı çevreleyen 10 satırı listeler
list BAŞLANGIÇSATIRI,BİTİŞSATIRI	Belirtilen başlangıç ve bitiş noktalarının arasını listeler
list FONKSİYONADI	Belirtilen fonksiyonu görüntüler
list DOSYAADI:SATIRNUMARASI	Projenin birden çok dosyadan oluşması durumunda satır numarasından önce dosya ismi belirtilebilir
list DOSYAADI:FONKSİYONADI	Projenin birden çok dosyadan oluşması durumunda fonksiyon adından önce dosya ismi belirtilebilir

Makina Kodlarının Listelenmesi

Sembolik ve karşılık geldikleri gerçek makina kodları **disassemble** komutu ile listelenebilir. disassemble argüman olarak bellek adresi almaktadır, örnek kullanımları aşağıdaki gibidir.

Komut	Açıklama
disas FONKSİYONADI	Belirtilen fonksiyona ait sembolik makina kodlarını listeler
disas /m FONKSİYONADI	Sembolik makina kodlarıyla beraber karşılık gelen kaynak kod satırları da listelenir
disas /r FONKSİYONADI	Sembolik makina kodlarıyla beraber gerçek makina kodları da listelenir

Program Akışının İzlenmesi

Programın akışı, kullanıcı tarafından **Ctrl-C** tuşlarına basılarak veya önceden belirlenen kesme noktalarıyla durdurulabilir. Kesme noktalarının kullanımına ilerleyen bölümlerde değineceğiz. Durdurulan program sonrasında kaldığı yerden, kullanıcı müdahalesi olmaksızın, yoluna devam edebileceği gibi kontrollü bir şekilde adım adım da çalıştırılabilir.

Akış komutları ve kullanım şekilleri aşağıdaki gibidir.

Komut	Açıklama
`continue`	Akış kaldığı yerden devam ettirilir
`next [N]`	1Aldığı argüman sayısınca, fonksiyon çağrılarını tek satır olarak ele alarak, kodu kaynak kod düzeyinde satır satır çalıştırır
`step [N]`	Aldığı argüman sayısınca, çağrılan fonksiyonların içine girerek, kodu kaynak kod düzeyinde satır satır çalıştırır
`nexti [N]`	Aldığı argüman sayısınca, fonksiyon çağrılarını tek satır olarak ele alarak, makina kodlarını adım adım çalıştırır
`stepi [N]`	Aldığı argüman sayısınca, çağrılan fonksiyonların içine girerek, makina kodlarını adım adım çalıştırır

next ve **step** komutları arasındaki, ister programın yazıldığı kaynak kod düzeyinde ister sembolik makina komutları düzeyinde olsun, fonksiyon çağrılarını ele alış biçimlerindeki farklılığa dikkat ediniz. **next** komutlarında fonksiyonlar tek hamlede işletilmekte buna karşın **step** komutlarında akış çağrı yapılan fonksiyon kodundan devam etmektedir. Aradaki farkı görmek için basit bir örnek yapalım. Aşağıdaki kodu debug.c adıyla saklayıp derleyebilirsiniz.

```
#include <stdio.h>

void foo() {
    int i;
    for (i = 0; i < 5; ++i) {
        puts(__func__);
    }
}

int main(int argc, char **argv) {
    foo();
    return 0;
}
```

```
$ gcc -odebug debug.c -m32 -g
```

Şimdi gdb'yi uygulamamız için çalıştıralım.

```
$ gdb -q debug
Reading symbols from debug...done.
(gdb)
```

Program akışı main fonksiyonunda kesilecek şekilde, **break** komutu ile, bir kesme noktası tanımlayalım ve uygulamayı çalıştıralım. **break** komutunun detaylarına ilerleyen bölümlerde bakacağız.

```
(gdb) break main
Breakpoint 1 at 0x8048457: file debug.c, line 11.
(gdb) run
Starting program: /home/serkan/embedded/gdb/debug

Breakpoint 1, main (argc=1, argv=0xffffd574) at debug.c:11
11              foo();
```

Akışın 11. satırda yani *foo* fonksiyonu çağrısında durduğunu görüyoruz. Daha detaylı inceleme yapabilmek için, disassemble komutuyla, sembolik makina kodlarına bakalım.

```
(gdb) disas
    Dump of assembler code for function main:
      0x08048446 <+0>:     lea    0x4(%esp),%ecx
      0x0804844a <+4>:     and    $0xfffffff0,%esp
      0x0804844d <+7>:     pushl  -0x4(%ecx)
      0x08048450 <+10>:    push   %ebp
      0x08048451 <+11>:    mov    %esp,%ebp
      0x08048453 <+13>:    push   %ecx
      0x08048454 <+14>:    sub    $0x4,%esp
   => 0x08048457 <+17>:    call   0x804841b <foo>
      0x0804845c <+22>:    mov    $0x0,%eax
      0x08048461 <+27>:    add    $0x4,%esp
      0x08048464 <+30>:    pop    %ecx
      0x08048465 <+31>:    pop    %ebp
      0x08048466 <+32>:    lea    -0x4(%ecx),%esp
      0x08048469 <+35>:    ret
End of assembler dump.
```

Sembolik makina kodlarındaki ok işareti, henüz işletilmemiş, sıradaki ilk komutu göstermektedir. **nexti** ile kodun işleyişini bir adım ilerletelim.

> **Not:** Kesme noktasının gösterdiği makina komutunun fonksiyonun ilk makina komutu değil, derleyici tarafından yazılan başlangıç kodlarını (prologue) takip eden, foo fonksiyonu çağrı komutu olduğuna dikkat ediniz.

```
(gdb) nexti
foo
foo
foo
foo
foo
12              return 0;
```

Tekrar sembolik makina kodlarına bakalım.

```
(gdb) disas
    Dump of assembler code for function main:
      0x08048446 <+0>:     lea    0x4(%esp),%ecx
      0x0804844a <+4>:     and    $0xfffffff0,%esp
      0x0804844d <+7>:     pushl  -0x4(%ecx)
      0x08048450 <+10>:    push   %ebp
      0x08048451 <+11>:    mov    %esp,%ebp
      0x08048453 <+13>:    push   %ecx
      0x08048454 <+14>:    sub    $0x4,%esp
      0x08048457 <+17>:    call   0x804841b <foo>
   => 0x0804845c <+22>:    mov    $0x0,%eax
      0x08048461 <+27>:    add    $0x4,%esp
      0x08048464 <+30>:    pop    %ecx
      0x08048465 <+31>:    pop    %ebp
      0x08048466 <+32>:    lea    -0x4(%ecx),%esp
      0x08048469 <+35>:    ret
    End of assembler dump.
```

nexti komutuyla *foo* fonksiyonunun işletildiğini ve akışın main fonksiyonun bir sonraki komutundan devam ettirildiğini görüyoruz. Şimdi benzer işlemi stepi komutuyla yapalım. Öncesinde kill komutuyla uygulamayı sonlandırıp uygulamayı yeniden çalıştıralım.

```
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) run
Starting program: /home/serkan/embedded/gdb/debug

Breakpoint 1, main (argc=1, argv=0xffffd574) at debug.c:11
11              foo();
```

Bu sefer **stepi** ile kodun işleyişini bir adım ilerletelim.

```
(gdb) stepi
foo () at debug.c:3
3       void foo() {
```

stepi komutu ile *foo* fonksiyonu tek hamlede çalıştırılmak yerine akış *foo* fonksiyonuna dallanmaktadır. Sembolik makina kodlarına baktığımızda bu durum daha açık şekilde gözükmektedir.

```
(gdb) disas
    Dump of assembler code for function foo:
   => 0x0804841b <+0>:     push   %ebp
      0x0804841c <+1>:     mov    %esp,%ebp
      0x0804841e <+3>:     sub    $0x18,%esp
      0x08048421 <+6>:     movl   $0x0,-0xc(%ebp)
      0x08048428 <+13>:    jmp    0x804843e <foo+35>
      0x0804842a <+15>:    sub    $0xc,%esp
      0x0804842d <+18>:    push   $0x8048500
      0x08048432 <+23>:    call   0x80482f0 <puts@plt>
      0x08048437 <+28>:    add    $0x10,%esp
      0x0804843a <+31>:    addl   $0x1,-0xc(%ebp)
      0x0804843e <+35>:    cmpl   $0x4,-0xc(%ebp)
      0x08048442 <+39>:    jle    0x804842a <foo+15>
      0x08048444 <+41>:    leave
      0x08048445 <+42>:    ret
    End of assembler dump.
```

Fonksiyonların Çağrılması

Program kodundaki veya programa linklenmiş dinamik kütüphane içeriğindeki fonksiyonları **call** komutuyla çağırabilirsiniz. **finish** komutuyla da bir fonksiyonu sonlandırarak çağıran fonksiyona geri dönülebilir.

Program Verisinin İncelenmesi

Değişkenlerle temsil edilen veya direkt adres kullanılarak gösterilen bellek alanlarını, **print** ve **x** komutlarıyla inceleyebiliriz, ayrıca bu bölümde yazmaç değerlerini nasıl elde edebileceğimize de bakacağız. Tam olarak aynı işi yapmamalarına karşın, birbirinin yerine geçebilen kullanımları olan bu komutlara daha yakından bakalım.

Komut	Açıklama
print /FORMAT İFADE	Programın yazıldığı kaynak kod düzeyindeki anlamlı bir ifadeyi belirtilen formatta gösterir
print /FORMAT ADRESGÖSTERENDEĞER@N	İfadenin bir adres göstermesi durumunda, @ operatoru ile devam eden N-1 adet bellek bölgesi de, tür bilgisi gözetilerek, gösterilir

Bu noktada bir ifadeye ait tür bilgisinin whatis komutuyla öğrenebildiğini söyleyelim.

Şimdi **print** komutunun kullanımına bakalım, aşağıdaki örnek kodu debug.c adıyla saklayıp derleyebilirsiniz.

```
#include <stdio.h>

int main(int argc, char **argv) {
    const char *str = "zeytin";
    int i = 111;
    int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    return 0;
}
```

```
$ gcc -odebug debug.c -g
```

gdb'yi çalıştırdıktan sonra, ilgilendiğimiz yerel değişkenlerin sonrasına bir kesme noktası koyalım ve programı çalıştıralım. Bu sayede bu program sonlanmayacak ve incelemelerimize devam edebileceğiz.

```
$ gdb -q debug
Reading symbols from debug...done.
(gdb) list
1       #include <stdio.h>
2
3       int main(int argc, char **argv) {
4               const char *str = "zeytin";
5               int i = 111;
6               int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
7               return 0;
8       }
(gdb) br 7
Breakpoint 1 at 0x80484b3: file debug.c, line 7.
(gdb) run
Starting program: /home/serkan/embedded/gdb/debug

Breakpoint 1, main (argc=1, argv=0xffffd574) at debug.c:7
7               return 0;
```

Sırasıyla *s*, *i* ve *arr* yerel değişkenleri için bazı örnekler yapalım.

> **Not:** Normal bir derleme sürecinde, değişmez adresleri olan statik ömürlü değişkenlerin aksine, konumları yazmaç göreli olarak değişen yerel değişken isimleri derleyici tarafından üretilen amaç kod (object code) içinde saklanmazlar. Debug hedefli derleme yaparak yerel değişken isimlerini kullanabildiğimize dikkat ediniz.

```
(gdb) print str
$2 = 0x8048570 "zeytin"
(gdb) print i
$3 = 111
(gdb) print &i
$4 = (int *) 0xffffd490
(gdb) print *0xffffd490
$5 = 111
(gdb) print /x i
$6 = 0x6f

(gdb) print arr
$7 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
(gdb) print arr[0]
$8 = 0
(gdb) print arr[0]@5
$9 = {0, 1, 2, 3, 4}
(gdb) print i@5
$10 = {111, 0, 1, 2, 3}
(gdb) print /x i@5
$11 = {0x6f, 0x0, 0x1, 0x2, 0x3}
```

Şimdi belleği incelemek için kullanabileceğimiz bir diğer komut olan *x* (Examine memory) komutuna bakalım.

Komut	Açıklama
x /FORMAT ADRESİFADESİ	Belirtilen adresteki bellek bölgesinin değeri formatlı bir şekilde gösterilir

Şimdi x komutuyla bazı basit örnekler yapalım.

```
(gdb) x /c str
0x8048570:      122 'z'
(gdb) x /s str
0x8048570:      "zeytin"
(gdb) x &i
0xffffd490:     111
```

Yazmaç Değerlerinin Elde Edilmesi

Yazmaç değerlerini **print** ve **info** komutlarıyla almak mümkündür. Genel şekilleri ve örnekler aşağıdaki gibidir.

Komut
print /FORMAT $YAZMAÇ
info registers

```
(gdb) info registers
 eax            0x0      0
 ecx            0xffffd4e0       -11040
 edx            0xffffd504       -11004
 ebx            0xf7fb1000       -134541312
 esp            0xffffd470       0xffffd470
 ebp            0xffffd4c8       0xffffd4c8
 esi            0x0      0
 edi            0x0      0
 eip            0x80484b3        0x80484b3 <main+120>
 eflags         0x246    [ PF ZF IF ]
 cs             0x23     35
 ss             0x2b     43
 ds             0x2b     43
 es             0x2b     43
 fs             0x0      0
 gs             0x63     99
 (gdb) info registers eip
 eip            0x80484b3        0x80484b3 <main+120>
 (gdb) info registers eax
 eax            0x0      0
 (gdb) print /x $eax
 $29 = 0x0
 (gdb) print /x $eip
 $30 = 0x80484b3
```

> **Not:** Format karakterlerinin çoğu, C dilinden aşina olduğumuz karakterlerden oluşmaktadır, tam liste için help komutundan faydalanabilirsiniz.

Program Verisinin Değiştirilmesi

Bellek alanlarının ve yazmaçların değerlerini **set** komutuyla değiştirebilirsiniz. **set** komutu çok sayıda alt komuta (subcommand) sahiptir, tam liste için **help set** şeklinde yardım alabilirsiniz. Genel kullanım şekli ve bir önceki kod için örnekler aşağıdaki gibidir.

Komut
set var GÜNCELLENECEKALAN=İFADE

```
(gdb) print i
$1 = 111
(gdb) print &i
$2 = (int *) 0xffffd490
(gdb) set var i = 33
(gdb) print i
$3 = 33
(gdb) set var *(unsigned int*)0xffffd490 = 44
(gdb) print i
$4 = 44
(gdb) print /x $eax
$5 = 0x0
(gdb) set var $eax = 55
(gdb) print $eax
$6 = 55
(gdb) print /s str
$7 = 0x8048570 "zeytin"
(gdb) set var *(char *)0x8048570 = 'm'
(gdb) print /s str
$8 = 0x8048570 "meytin"
```

Geçmiş Değerlerin Kullanımı

gdb, **print** komutunun ürettiği sonuçları, geriye dönük takip edebilmek ve yeniden kullanabilmek için, $NUM, şeklinde numaralandırdığı değişkenlerde saklamaktadır. **show values** komutuyla geçmiş değerleri listeleyebilirsiniz.

```
(gdb) print i
$15 = 555
(gdb) print $15
$16 = 555
(gdb) print $15
$16 = 555
(gdb) show values
$7 = 0x8048570 "meytin"
$8 = 0x8048570 "meytin"
$9 = 44
$10 = 44
$11 = 111
$12 = (int *) 0xffffd490
$13 = 33
$14 = 44
$15 = 555
$16 = 555
```

Macro İşlemleri

Önişlemci makrolarına ilişkin bilgileri de debug bilgisi olarak saklayabilmek için gcc'ye `g3` veya `ggdb3` anahtarlarından birini geçirmeliyiz. Aşağıdaki örnek üzerinden macro işlemlerini nasıl yapabileceğimize bakalım. Kodu debug.c olarak saklayıp derleyebilirsiniz.

```
#define NUMBER 111

int main() {
    return 0;
}
```

```
$ gcc -odebug debug.c -g3
```

Program çalışmıyor iken bile list komutuyla makro tanımlarının üzerinden geçip sonrasında makro değerlerlerine ulaşabiliriz.

Makroların karşılık geldiklerini değerleri öğrenmek için macro expand veya info macro komutlarını kullanabilirsiniz. Ayrıca **macro define** ve **macro undef** ile makro tanımlayabilir veya geçersiz kılabilirsiniz. **help macro** komutuyla makro işlemleriyle ilgili yardım alınabilir.

Tanımladığımız NUMBER makrosunun değerine iki farklı şekilde ulaşabiliriz.

```
$ gdb -q debug
Reading symbols from debug...done.
(gdb) list 1,5
1       #define NUMBER 111
2
3       int main() {
4               return 0;
5       }
(gdb) macro expand NUMBER
expands to: 111
(gdb) info macro NUMBER
Defined at /home/serkan/embedded/gdb/debug.c:1
#define NUMBER 111
```

Yığınının (Call Stack) İncelenmesi

Bir fonksiyon çağrıldığında, yığında fonksiyon için, yığın çerçevesi (call frame) olarak isimlendirilen ve fonksiyon sonlandığında geri verilen, yeni bir alan ayrılır. Fonksiyona geçirilen argümanlar, yerel değişkenler ve fonksiyonun geri dönüş adresi yığın çerçevesinde bulunmaktadır. Yığının, son girenin ilk çıktığı (LIFO) bir veri alanı olduğunu hatırlayınız. gdb yığın üzerinde işlem yapabilmek için gerekli komutları barındırmaktadır. Şimdi bu komutların genel kullanımına bakalım.

Komut	Açıklama
`backtrace [N] [full]`	Argümansız kullanılması durumunda tüm yığın çerçevelerini görüntüler. N değerinin pozitif olması durumunda en içteki N adet, negatif olması durumunda ise en dıştaki N adet çerçeve listelenir. full niteleyicisi geçirilmesi durumunda ise yerel değişken değerleri de listelenir
`frame [N]`	Argümansız kullanıması durumunda gündemdeki (current) yığın çerçevesine ait bilgileri görüntüler. Argümanlı kullanımda belirtilen çerçeveyi seçer
`info frame [N]`	Gündemdeki veya N argümanıyla belirtilen yığın çerçevesine ait bilgileri görüntüler
`info locals`	Gündemdeki yığın çerçevesine ait yerel değişken değerlerini görüntüler

İncelememize bir örnek üzerinden devam edelim, aşağıdaki örneği debug.c ismiyle saklayıp derleyebilirsiniz.

```
void bar() {
    int k = 111;
}

void foo() {
    int j = 111;
    bar();
}

int main() {
    int i = 111;
    foo();
    return 0;
}
```

```
$ gcc -odebug debug.c -g
```

bar fonksiyonuna bir kesme noktası koyarak uygulamayı çalıştıralım ve yığın çerçevelerine bakalım.

```
$ gdb -q debug
Reading symbols from debug...done.
(gdb) break bar
Breakpoint 1 at 0x4004fa: file debug.c, line 2.
(gdb) run
Starting program: /home/serkan/embedded/gdb/debug

Breakpoint 1, bar () at debug.c:2
2               int k = 111;
(gdb) backtrace
#0  bar () at debug.c:2
#1  0x000000000040051c in foo () at debug.c:7
#2  0x0000000000400537 in main () at debug.c:12
```

backtrace komutunun, içten dışa doğru, yığın çerçevelerini numaralandırarak listelediğini görmekteyiz. **#** karakteriyle başlayan bu numaralar yığın çerçevelerini tanımlamak için kullanılmaktadır, daha sonra **frame** komutunda da bu numaraları kullanacağız. backtrace komutuna, ilk veya son yığın çerçevesinden başlayarak, listelemesini istediğimiz çerçeve sayısını argüman geçirebiliriz, aşağıdaki kullanımları inceleyiniz.

```
(gdb) bt 1
#0  bar () at debug.c:2
(More stack frames follow...)
(gdb) bt -1
#2  0x0000000000400537 in main () at debug.c:12
(gdb)
(gdb) bt 2
#0  bar () at debug.c:2
#1  0x000000000040051c in foo () at debug.c:7
(More stack frames follow...)
(gdb) bt -2
#1  0x000000000040051c in foo () at debug.c:7
#2  0x0000000000400537 in main () at debug.c:12
```

Şimdi **frame** ve **info locals** komutlarının kullanımlarına bakalım. **frame** ile isteğimiz bir çerçeveyi seçebilir ve **info locals** ile o çerçeveye ait yerel değişken değerlerine ulaşabiliriz.

```
(gdb) frame
#0  bar () at debug.c:2
2               int k = 111;
(gdb) info locals
k = 0
(gdb) frame 2
#2  0x0000000000400537 in main () at debug.c:12
12              foo();
(gdb) info locals
i = 111
```

Komut Dosyaları

Komut dosyaları, gdb komutlarından oluşan yazı dosyalarıdır. # ile başlayan satırlar açıklama olarak ele alınır. gdb çalıştırılırken, komut dosyası argüman olarak geçirilebildiği gibi sonrasında gdb komut satırından da gösterilebilir. Genel kullanımları aşağıdaki gibidir.

Kullanım
$ gdb -x KOMUTDOSYASI
(gdb) source KOMUTDOSYASI

KOMUTDOSYASI dosya ismi ve dizin yolundan oluşabilir. Şimdi, incelediğimiz debug programı için, basit bir örnek yapalım. Aşağıdaki komutları *commands.txt* dosyasında saklayabilir ve sonrasında gdb'yi aşağıdaki gibi çalıştırabilirsiniz.

```
# Komut dosyası
# İlk olarak debug programı yüklenir, kaynak kod listelendikten sonra
# bar fonksiyonuna kadar kod işletilir.

file debug
list
break bar
run
```

```
$ gdb -q -x commands.txt
1       void bar() {
2               int k = 111;
3       }
4
5       void foo() {
6               int j = 111;
7               bar();
8       }
9
10      int main() {
Breakpoint 1 at 0x4004fa: file debug.c, line 2.

Breakpoint 1, bar () at debug.c:2
2               int k = 111;
(gdb)
```

Başlangıç Dosyaları

gdb ilk açılışta bazı öntanımlı dosyaları, bir öncelik sırasına göre, aramaktadır. Aradığı başlangıç dosyaları ve öncelikleri aşağıdaki gibidir.

Dosya	Konum
system.gdbinit	Kullanıcı veya çalışma dizininden bağımsız, sistem genelindeki başlangıç dosyasıdır. Bu özelliğin kullanılabilmesi için gdb --with-system-gdbinit seçeneği ile derlenmiş olmalıdır
~/.gdbinit	Kullanıcı dizinindeki başlangıç dosyasıdır
./.gdbinit	Çalışma dizinindeki başlangıç dosyasıdır

Daha önce komut dosyalarını incelerken kullandığımız örneği şimdi ana dizinde bir .gdbinit dosyası oluşturarak tekrarlayalım. Bu durumda gdb'yi çalıştırdığımızda aşağıdaki gibi başlayan bir uyarı güvenlik uyarısı alıyoruz.

```
$ gdb -q
warning: File "/home/serkan/embedded/gdb/.gdbinit" auto-loading has been declined by y
our `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
    add-auto-load-safe-path /home/serkan/embedded/gdb/.gdbinit
...
```

Bu durumda kullanıcı dizinindeki .gdbinit dosyasına, uyarıda belirtilen **add-auto-load-safe-path /home/serkan/embedded/gdb/.gdbinit** komutunu ekleyip gdb'yi yeniden çalıştırtığımızda, çalışma dizinindeki **.gdbinit** dosyasının içeriğinin okunduğunu görmekteyiz.

```
$ gdb -q
1       void bar() {
2               int k = 111;
3       }
4
5       void foo() {
6               int j = 111;
7               bar();
8       }
9
10      int main() {
Breakpoint 1 at 0x4004fa: file debug.c, line 2.

Breakpoint 1, bar () at debug.c:2
2               int k = 111;
```

Çalışma Modları

Komut satırı seçeneklerini kullanarak gdb'yi farklı modlarda çalıştırmak mümkündür. Desteklenen modlardan birkaçına bakalım.

Seçenek	Mod
-nx	Başlangıç dosyalarındaki komutlar çalıştırılmaz
-q	Başlangıç mesajları basılmaz
-batch	Kullanıcıyla etkileşime geçilmez, başlangıç dosyaları ve komut satırı seçenekleri işletildikten sonra gdb sonlanır

batch mod ile, istediğiniz komutları seçenek olarak geçirerek, gdb komut satırına düşmeksizin gdb'nin istediğiniz işleri yapmasını sağlayabilirsiniz. Çalışmasını istediğimiz komutları ex seçeneğiyle belirtebiliriz. Bu şekilde gdb'nin ürettiği çıktıyı bu şekilde bir dosyaya yönlendirebiliriz. Aşağıdaki örneği inceleyiniz.

```
$ gdb -q -batch -ex "file debug" -ex "list" -ex "break bar" -ex "run" > debug.txt

$ cat debug.txt
1       void bar() {
2               int k = 111;
3       }
4
5       void foo() {
6               int j = 111;
7               bar();
8       }
9
10      int main() {
Breakpoint 1 at 0x4004fa: file debug.c, line 2.

Breakpoint 1, bar () at debug.c:2
2               int k = 111;
```

Kabuk Kullanımı

gdb çalışıyorken, gdb'yi sonlandırmadan ya da askıya almadan, kabuk komutlarını **shell KOMUT** şeklinde çalıştırmak mümkündür. Örneğin, gdb ekranı temizlemek için bir komuta sahip olmadığından ekranı **shell clear** şeklinde temizleyebilirsiniz.

Kesme Noktaları Oluşturulması

Kesme noktaları, program akışının durdurulduğu ve kontrolün kullanıcıya geçtiği noktalardır. Bu durumda tipik kullanım, program verisini incelemek ve kodu, kontrollü bir şekilde, adım adım çalıştırmak şeklinde olmaktadır. Kesme noktaları program kodu üzerinde açık bir şekilde belirtilebildiği gibi data belleği üzerindeki alanlar da izlenebilir. Şimdi bu özelliğe daha yakından bakalım.

Kod Üzerinde Oluşturulan Kesme Noktaları (Breakpoints)

Program kodu üzerinde kesme noktası **break KONUM** şeklinde tanımlanmaktadır. Konum değeri aşağıdaki biçimlerde olabilmekte ayrıca bir koşul ifadesi de eklenebilmektedir.

Komut	Açıklama
break [DOSYAADI:]SATIRNUMARASI	Belirtilen satır kesme noktası olarak işaretlenir
break [DOSYAADI:]FONKSİYONADI	Belirtilen fonksiyon kesme noktası olarak işaretlenir
break ADRES	Belirtilen adres kesme noktası olarak işaretlenir
break KONUM if KOŞUL	Kesme noktasına koşul eklenir

Proje birden çok kaynak dosyadan oluşuyorsa DOSYAADI belirtilmedir. Şimdi bir örnek üzerinde koşul içeren bir kesme noktası oluşturalım. Döngü içerisinde indeks değerinin basıldığı 6. satırı kesme noktası olarak belirliyoruz. Bir koşul belirtmeseydik, döngünün her turunda akış kesilecekti. Koşulu indeks değerinin 5 olması olarak belirlediğimizden akış bu kesme noktasında durduğunda i değişkeni 5 değerine sahiptir.

```
#include <stdio.h>

int main() {
    int i;
    for (i = 0; i < 10; ++i) {
        printf("%d\n", i);
    }
    return 0;
}
```

```
$ gcc -odebug debug.c -g
```

```
$ gdb -q debug
Reading symbols from debug...done.
(gdb) list
1       #include <stdio.h>
2
3       int main() {
4               int i;
5               for (i = 0; i < 10; ++i) {
6                       printf("%d\n", i);
7               }
8               return 0;
9       }
(gdb) break 6 if i == 5
Breakpoint 1 at 0x400547: file debug.c, line 6.
(gdb) run
Starting program: /home/serkan/embedded/gdb/debug
0
1
2
3
4

Breakpoint 1, main () at debug.c:6
6                       printf("%d\n", i);
(gdb) print i
$1 = 5
```

Ayrıca, bir kesme noktasına sonradan da koşul ekleyebilirsiniz. Aynı örnek için 6 numaralı satıra kesme noktası ekleyip, sonrasında **condition** komutu ile bir koşul ifadesi ekleyebiliriz. condition komutunun genel hali aşağıdaki gibidir.

Komut	Açıklama
condition N KOŞUL	N kesme noktası numarasını göstermektedir

Tanımlanmış kesme numaralarını **info breakpoints** komutu ile öğrenebilirsiniz.

```
$ gdb -q debug
Reading symbols from debug...done.
(gdb) break 6
Breakpoint 1 at 0x400547: file debug.c, line 6.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400547 in main at debug.c:6
(gdb) condition 1 i == 5
(gdb) run
Starting program: /home/serkan/embedded/gdb/debug
0
1
2
3
4

Breakpoint 1, main () at debug.c:6
6                       printf("%d\n", i);
```

Bellek Üzerinde Oluşturulan İzleme Noktaları (Watchpoints)

Kod üzerinde tanımlanan kesme noktalarının aksine izleme noktaları veri belleği üzerindedir. Bu sayede, veri belleği üzerinde ilgilendiğimiz bir alanın değeri okunduğunda veya değiştirildiğinde akışın durmasını sağlayabiliriz. İzleme noktaları oluşturduğumuzda, kod üzerinde ilgilendiğimiz bellek alanının değerini kullanan kısımları bulma zorunluluğumuz ortadan kalkmaktadır.

İzleme noktası oluşturmak için kullanılan komutlar aşağıdaki gibidir.

Komut	Açıklama
watch İFADE	Argüman olarak geçirilen ifadenin değeri değiştirildiğinde etkin olur
rwatch İFADE	Argüman olarak geçirilen ifadenin değeri okunduğunda etkin olur
awatch İFADE	Argüman olarak geçirilen ifadenin değeri okunduğunda veya değiştiğinde etkin olur

Genel biçimde gösterilen ifade çoğunlukla bir değişken adı olmaktadır. Bu özelliğin kullanımına basit bir örnek yaparak daha yakından bakalım.

```
int main() {
      int i;
      int j;
      i = 0;
      j = i;
      return 0;
}
```

```
$ gcc -odebug debug.c -g -m32
```

İlk olarak programı main fonskiyonuna kadar çalıştıralım, ardından i değişkenindeki değerin değişimini izlemek için watch i komutuyla bir izleme noktası oluşturduktan sonra continue ile programın çalışmasını devam ettirelim. Bu durumda programın i değerine ilişkin eski ve yeni değerleri vererek sonlandığını görüyoruz, **disas** ile *i* değişkenine (-0x8(%ebp)) 0 değerinin yazıldığını ve akışın sonraki makina komutunda durduğunu görüyoruz. Benzer testleri **rwatch** ve **awatch** komutlarıyla da yapabilirsiniz.

```
$ gdb -q debug
    Reading symbols from debug...done.
    (gdb) break main
    Breakpoint 1 at 0x80483f1: file debug.c, line 6.
    (gdb) run
    Starting program: /home/serkan/embedded/gdb/debug

    Breakpoint 1, main () at debug.c:6
    6               i = 0;
    (gdb) watch i
    Hardware watchpoint 2: i
    (gdb) c
    Continuing.
    Hardware watchpoint 2: i

    Old value = 134513680
    New value = 0
    main () at debug.c:7
    7               j = i;
    (gdb) disas
    Dump of assembler code for function main:
      0x080483eb <+0>:     push   %ebp
      0x080483ec <+1>:     mov    %esp,%ebp
      0x080483ee <+3>:     sub    $0x10,%esp
      0x080483f1 <+6>:     movl   $0x0,-0x8(%ebp)
    =>0x080483f8 <+13>:    mov    -0x8(%ebp),%eax
      0x080483fb <+16>:    mov    %eax,-0x4(%ebp)
      0x080483fe <+19>:    mov    $0x0,%eax
      0x08048403 <+24>:    leave
      0x08048404 <+25>:    ret
    End of assembler dump.
```

Burada bir noktaya dikkatinizi çekmek istiyoruz. İzleme noktası oluşturduğumuzda watch komutunun, *Hardware watchpoint 2: i* şeklinde bir bilgi mesaji verdiğini görmekteyiz. Mesajdaki Hardware ifadesi izleme noktasının donanım desteği alınarak oluşturulduğunu göstermektedir. Bir sonraki bölümde yazılımsal ve donanımsal olarak oluşturulan izleme noktalarına kısaca değineceğiz.

Yazılımsal ve Donanımsal Kesme Noktaları

Bir önceki bölümde, kod ve bellek üzerinde, kesme noktalarının nasıl kullanıldığını inceledik. Kod üzerinde belirlediğimiz noktalar işletilmeye çalışıldığında veya izlediğimiz değişkenler adreslendiğinde, akışın sonlandırıldığını ve kontrolun debugger programına, gdb, geçtiğini gördük. Şimdi bu özelliğin nasıl gerçekleştirildiğe daha yakından bakalım. Kesme noktaları yazılımsal ve donanımsal olmak üzere iki farklı şekilde oluşturulabilmektedir.

Yazılımsal Kesme Noktaları

Yazılımsal kesme noktaları, kod üzerinde oluşturduğumuz kesme noktalarını (breakpoints) oluşturmak için kullanılmaktadır. gdb uygulamasında bu noktaları break komutuyla oluşturmuştuk. Kod üzerinde yazılımsal kesme noktaları oluşturmanın birden çok yolu olmasına karşın burada gdb tarafından da kullanılan yöntemden bahsedeceğiz. Bu yöntemde, kesme noktasındaki makina komutu debugger tarafından değiştirilerek, işlemcinin bir istisna (exception) durumu oluşturması hedeflenmektedir. İstisna durumu oluştuğunda, işlemci normal akışını sonlandıracak ve işletim sistemi tarafından tanımlanan bir ele alım kodunu (interrup handler) çalıştıracaktır. Ele alım kodunun tipik davranışı ise bu duruma neden olan prosese bir sinyal göndermek şeklindedir. gdb, işletim sistemi tarafından gönderilen bu sinyalden haberdar olmakta ve kontrol gdb uygulamasına geçmektedir. gdb, alt proses olarak çalıştırdığı veya daha sonradan bağlandığı prosese gelen sinyalleri takip edebilmektedir. x86 mimarisinde, istisna durumu oluşturmak için, gerçek makina kodu **0xCC** olan, **int 3** sembolik makina kodu kullanılmaktadır. int sembolik makina kodu, yazılım yoluyla kesme (software interrupt) oluşturmak için kullanılmakta ve kesme numarasını operand olarak almaktadır. Normalde komutun kendisi (opcode) ve aldığı operand olmak üzere 2 byte olmasına karşın, int 3 komutu özel olarak bir byte ile gösterilmektedir. Bu durum, *Intel Architecture Software Developer's Manual* dokümanında aşağıdaki gibi ifade edilmiştir.

```
The INT 3 instruction generates a special one byte opcode (CC) that is intended for ca
lling the debug exception handler. (This one byte form is valuable because it can be u
sed to replace the first byte of any instruction with a breakpoint, including other on
e byte instructions, without over-writing other code).
```

İşlemci, int 3 sembolik makina koduyla karşılaştığında normal akışından çıkacak ve işletim sisteminin debug ele alım kodunu çalıştıracaktır. Bu kodda, nihayetinde debugger tarafından alınacak, *SIGTRAP* sinyalini üretecektir. gdb uygulamasında, bir kesme noktası belirlendiğinde karşılık geldiği konumdaki makina komutunun ilk byte değerinin 0xCC değeri ile değiştirildiğini söyledik. Bu durumda incelediğiniz kod üzerinde bir kesme noktası belirleyip sonrasında disas ile makina kodlarına baktığınızda, kesme noktasında, 0xCC değerini görmeyi bekleyebilirsiniz. Fakat gdb, gerçekte bu işlemi yapmasına karşın, disas çıktısında kodun değişmemiş halini göstermektedir. Bu yüzden bu durumu gözleyebilmek için, kendi makina kodlarının bir kısmını basan, aşağıdaki örneği kullanacağız. Örnek kodu break.c adıyla saklayıp, debug sembolleri olmaksızın, aşağıdaki gibi derleyebilirsiniz.

```
#include <stdio.h>
int main() {
    int i,j;
    unsigned char *p = (unsigned char*)main;

    for (j = 0; j < 2; j++) {
    printf("%p: ", p);
    for (i = 0; i < 16; i++)
        printf("%.2x ", *p++);
    printf("\n");
    }
    return 0;
}
```

```
$ gcc -obreak break.c
```

İlk olarak programı, herhangi bir kesme noktası oluşturmaksızın, çalıştıralım. Çıktı olarak, main fonksiyonundan itibaren toplamda 32 byte'tan oluşan makina komutlarını görmekteyiz.

```
$ gdb -q break
Reading symbols from break...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/serkan/embedded/gdb/break
0x400586: 55 48 89 e5 48 83 ec 10 48 c7 45 f8 86 05 40 00
0x400596: c7 45 f4 00 00 00 00 eb 5a 48 8b 45 f8 48 89 c6
[Inferior 1 (process 22438) exited normally]
```

Sonrasında main fonksiyonunu kesme noktası olarak belirleyelim ve kodu main fonksiyonuna kadar işletelim.

```
(gdb) break main
Breakpoint 1 at 0x40058a
(gdb) run
Starting program: /home/serkan/embedded/gdb/break

Breakpoint 1, 0x000000000040058a in main ()
```

Bu noktada sembolik ve gerçek makina kodlarına baktığımızda, main fonksiyonunun başlangıç kodlarından (prologue) sonraki 0x40058a adresindeki ilk komutunun kesme noktası olarak gösterildiğini görüyoruz. Kesme noktasındaki makina komutunun 48 ile başladığını görmekteyiz. Programın bir önceki çalışmasında da 5. sıradaki makina komutunun 48 olarak gösterildiğine dikkat ediniz. Bu noktadan sonra continue komutu ile programı çalıştırıyoruz.

```
(gdb) disas /r
  Dump of assembler code for function main:
    0x0000000000400586 <+0>:     55      push   %rbp
    0x0000000000400587 <+1>:     48 89 e5        mov    %rsp,%rbp
  =>0x000000000040058a <+4>:     48 83 ec 10     sub    $0x10,%rsp
  ...
```

Bu sefer 5. sıradaki makina komutunun, beklediğimiz üzere, 48 yerine cc ile gösterildiğini görmekteyiz. Buna karşın, disas çıktısında bu değer hala 48 ile gösterilmeye devam etmektedir.

```
(gdb) c
  Continuing.
  0x400586: 55 48 89 e5 cc 83 ec 10 48 c7 45 f8 86 05 40 00
  0x400596: c7 45 f4 00 00 00 00 eb 5a 48 8b 45 f8 48 89 c6
  [Inferior 1 (process 22442) exited normally]
```

Yazılımsal kesme noktaları bazı zorluklar barındırmaktadır. Kesme noktasından sonra akış devam ettirilmek istendiğinde, 0xCC makina kodu yerine gerçek makina kodu geri yazılmalı, akış devam ettirilmeli fakat kesme noktasının hala kullanılabilir olması için hemen ardından 0xCC kodu geri yazılmalıdır. Kesme noktasında program askıya alındığında komut göstericisi bir sonraki komutu göstermektedir. Akışın devam ettirilebilmesi için komut göstericisi tekrardan kesme noktasını gösterecek şekilde ayarlanmalı ve işlemci pipeline'ı temizlenmelidir.

Donanımsal Kesme Noktaları

Donamımsal kesme noktalarıyla, yazılımsal kesme noktalarının aksine, veri belleğini de, etkin bir şekilde izlemek mümkündür. Donanımsal kesme noktaları, işlemci bünyesinde, özel gereksinimlere ihtiyaç duymaktadır. x86 mimarisinde bu amaçla, **DRx** şeklinde isimlendirilen, 6 adet debug yazmacı bulunmaktadır. Bu yazmaçlardan 4 tanesi adres yolunu dinlemekte, iki tanesi ise kontrol ve durum bilgisi için kullanılmaktadır. Bu yazmaçlar, aktif olmaları durumunda, kendilerine yüklenen değerleri adres yolundaki değerlerle karşılaştırmaktadırlar. Adreslerin eşleşmesi durumunda, adres üzerinde, okuma, yazma veya çalıştırma işlemleri yapılmasına göre içsel bir kesmeye neden olmaktadırlar. Bu durumda akış durmakta, araya işletim sisteminin ele alım kodu girmekte ve nihayetinde, yazılımsal kesmelerde olduğu gibi, debugger bu durumdan haberdar olmaktadır. Bu özelliğin olmaması durumunda bellek izlenmek istendiğinde debugger her makina komutundan sonra izlenen bellek bölgesiyle ilgili işlem yapılıp yapılmadığına bakmalıdır. Donanımsal kesme noktaları, yazılımsal kesme noktalarının aksine, kısıtlı sayıda olmalarına karşın, değişkenlerin izlenmek istendiği ve kodun, FLASH bellek gibi, değiştirilemediği ortamlarda kod üzerinde de kesme noktaları

oluşturabilmek için oldukça faydalıdırlar. gdb'nin kod üzerindeki kesme noktalarını yazılımsal, bellek üzerindeki izleme noktalarının ise donanımsal olarak gerçekleştirdiğini görmekteyiz.

Kesme Noktalarının Silinmesi ve Etkisizleştirilmesi

Kesme noktalarını **delete** ve **disable** komutlarıyla silebilir veya etkisizleştirebilirsiniz. Kod veya bellek üzerinde oluşturduğunuz kesme noktalarını **info breakpoints** ile listeleyebilir, kesme noktalarının durumlarına ve numaralarına ulaşabilirsiniz. **disable** ve **delete** komutları kesme numaralarını argüman olarak almaktadır, argüman geçirilmemesi durumunda tüm kesme noktaları üzerinde işlem yapılır. Aşağıdaki örnek kullanımı inceleyiniz.

```
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x080483f1 in main at debug.c:6
    breakpoint already hit 1 time
2       hw watchpoint  keep y              i
(gdb) disable 1
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep n   0x080483f1 in main at debug.c:6
    breakpoint already hit 1 time
2       hw watchpoint  keep y              i
(gdb) delete 2
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep n   0x080483f1 in main at debug.c:6
  breakpoint already hit 1 time
```

Strace

Nasıl Çalışır?

Unix tabanlı sistemlerde **strace** gibi bir uygulamanın varolabilmesi için gereken `ptrace` sistem çağrısı uzun yıllardır (**SVr4** ve **4.3BSD**) bulunmaktadır.

Sistem çağrısı ismini **process trace** kavramından alır. `ptrace` sistem çağrısı üzerinden bir uygulama başka bir uygulamanın durumunu takip edebildiği gibi değişiklikler yapma imkanına da sahip olmaktadır.

`ptrace` sistem çağrısı temel olarak **gdb** gibi debug uygulamalarında, **strace**, **ltrace** gibi sistem veya kütüphane çağrılarını takip uygulamalarında, *code coverage* araçlarında, çalışan yazılım koduna dokunmadan bazı hataların giderilmesinde veya güvenlik kontrollerinden geçirilmesinde kullanılır.

`ptrace` çağrısıyla bir uygulamanın kontrolü tamamen başka bir uygulama verilmektedir. Buradaki kontrolden kastımız, uygulamanın kullandığı tüm bellek alanına erişim, sinyallerin alınması, değiştirilmesi, file descriptor'ların yönetimi hatta uygulamanın kod segmentinin değiştirilerek yamalar yapılması dahil aklımıza gelebilecek hemen her türden tehlikeli değişikliklere izin veriliyor olmasıdır.

Bahsettiğimiz bu özelliklerinden ötürü, bir uygulamanın başka bir uygulamayı `ptrace` ile kontrol edebilmesi için, ilgili uygulamaya **sinyal** gönderme yetkisinin bulunması gerekir. Dolayısıyla özel durumlar haricinde her kullanıcının kendi sahibi olduğu diğer uygulamaları ptrace ile kontrol edebileceğini, **root** kullanıcısının da sistemdeki tüm uygulamaları kontrol edebileceğini söyleyebiliriz.

Linux Capabilities API sisteminin geliştirilmesinden sonra yukarıda koşullardan bağımsız olarak, **CAP_SYS_PTRACE** özelliği sayesinde de `ptrace` izni verilebilmektedir.

Tipik Kullanım

Uygulamanızı `strace` ile aşağıdaki biçimde başlatmanız yeterlidir:

```
$ strace ls /tmp
```

Ancak çoğu zaman çok daha önceden başlatılmış ve çalışmaya devam eden, bununla birlikte herhangi bir sorun nedeniyle ek bilgi toplamak istediğiniz durumlar oluşur. `strace` ile herhangi bir çalışan uygulamaya, `-p` parametresine `<PID>` değerini vermek suretiyle *attach* olabiliriz:

```
$ strace -p $(pidof mysqld)
Process 26829 attached - interrupt to quit
select(13, [10 12], NULL, NULL, NULL)   = 1 (in [10])
fcntl64(10, F_SETFL, O_RDWR|O_NONBLOCK) = 0
accept(10, {sa_family=AF_INET, sin_port=htons(33033), sin_addr=inet_addr("192.168.0.15
")}, [16]) = 34
fcntl64(10, F_SETFL, O_RDWR)            = 0
rt_sigaction(SIGCHLD, {SIG_DFL, [CHLD], SA_RESTART}, {SIG_DFL, [CHLD], SA_RESTART}, 8)
 = 0
getpeername(34, {sa_family=AF_INET, sin_port=htons(33033), sin_addr=inet_addr("192.168
.0.15")}, [16]) = 0
...
```

`-f` Parametresi

Strace'in uygulamanın tüm thread'lerini ve uygulamadan *fork()* edilen diğer çocuk süreçlerini takip edebilmesi için `-f` parametresi verilmelidir.

```
$ strace -f ./example

$ strace -f -p $(pidof mysqld)
```

`-e` Parametresiyle Filtreleme

Strace çıktısı zaman zaman takip için oldukça kalabalık olabilir.

Sadece belirli sistem çağrılarını takip etmek istiyorsanız `e` parametresi ile bunu yapabilirsiniz:

```
$ strace -f -e trace=open,write,close,connect,select -p 3245
```

Sadece dosya işlemleriyle ilgili sistem çağrılarını takip etmek için `-e trace=file` kullanılabilir:

```
$ strace -e trace=file 4535
```

Sadece network ile ilgili sistem çağrılarını filtrelemek için `-e trace=network` kullanılabilir:

```
$ strace -e trace=network 23232
```

Zaman Bilgisi Alma: `-tt`

Sistem çağrılarının çıktısı alınırken saniye hassasiyetinde zaman bilgisi de almak istiyorsak `-t` parametresini kullanabiliriz.

Çoğu zaman saniye hassasiyeti yeterli olmayacaktır. Mikrosaniye hassasiyetinde zaman bilgisi almak için `-tt` parametresini kullanabiliriz:

```
$ strace -tt ls /tmp
...
00:07:10.595807 openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
00:07:10.595885 getdents(3, /* 6 entries */, 32768) = 176
00:07:10.595977 getdents(3, /* 0 entries */, 32768) = 0
00:07:10.596041 close(3)
```

İstatistiki Bilgi Alma: `-c`

`-c` parametresi ile istediğimiz süre boyunca sistem çağrılarıyla ilgili istatistik toplayabiliriz:

```
$ strace -f -c -p $(pidof mysqld)
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
42.89    0.592035         275      2149       151 read
28.11    0.388024        2587       150        18 futex
27.82    0.384023        1352       284           select
 1.16    0.016001        3200         5           rt_sigtimedwait
 0.01    0.000154           0      1457           write
 0.01    0.000152          22         7           accept
 0.00    0.000007           0      9478           time
 0.00    0.000000           0       108           open
 0.00    0.000000           0       136           close
 0.00    0.000000           0        40           unlink
 0.00    0.000000           0         5           alarm
 0.00    0.000000           0         9           access
 0.00    0.000000           0        21           ioctl
 0.00    0.000000           0      2409           gettimeofday
 ................
```

`-o` İle Çıktıların Kayıt Edilmesi

`strace` uzun süreli çalıştırılıp oluşan loglar daha detaylı olarak geniş zaman aralığında incelenecekse, logların kayıt edilmesi gerekecektir.

`-o` parametresi ile logların kayıt edileceği dosyayı belirtebilirsiniz:

```
$ strace -f -o /tmp/strace.log -e trace=file ls /tmp
```

Ptrace Engelleme

Linux altında bir uygulamanın, kendisinin **root** harici kullanıcılar tarafından `ptrace` sistem çağrısı ile kontrol edilmesini engelleyebilmesine imkan verilmiştir.

Bu işlem için `prctl` özel sistem çağrısı kullanılır. Uygulama `prctl` aracılığıyla kendisi için `PR_SET_DUMPABLE` bayrağını temizleyecek olursa **root** haricindeki kullanıcıların uygulamaya sinyal gönderme hakkı olsa dahi bu uygulamayı `ptrace` ile kontrol etme şansları olmaz.

Bu özelliğin en tipik kullanımlarından biri, OpenSSH authentication agent yazılımında görülür. Böylelikle kullanıcıların parola girme aşamasında uygulamanın `ptrace` ile başka bir uygulama tarafından kontrolü engellenmiş olur.

Güvenlik

Linux kullanımının yaygınlaşmasıyla birlikte zararlı yazılımlara rastlanma sıklığı da artmaktadır. Geleneksel Linux process modelindeki `ptrace` imkan seti sebebiyle, sisteminizde kendi kullanıcınızla çalıştırdığınız herhangi bir yazılım içerisine zararlı bir kod enjekte edilmiş ise (en basit *xterm* aracından gelişmiş web tarayıcı uygulamalarına kadar), `ptrace` sistem çağrısı sayesinde çalışan diğer tüm uygulamalarınızın kontrolünün bu zararlı yazılım tarafından devralınması ve siz hiç bir şey farketmeden önemli bilgilerin kopyalanması mümkündür.

Pek çok kullanıcının farkında olmadığı bu duruma karşılık, Linux çekirdeği içerisindeki **Yama** kod adlı güvenlik modülüyle bir koruma mekanizması geliştirilmiştir. (Ayrıntılı bilgiler için: https://www.kernel.org/doc/Documentation/security/Yama.txt)

Yama modülü olan Linux çekirdeğinde, `/proc/sys/kernel/yama/ptrace_scope` dosyası üzerinden `ptrace` sistem çağrısına verilecek tepki kontrol altına alınabilmektedir. Öntanımlı olarak bu dosyada **0** değeri yazmaktadır. Dosyada yazan değer aşağıdaki tablo doğrultusunda yorumlanır:

Değer	Anlam
0	**Geleneksel davranış:** önceki bölümde anlatılanlar doğrultusunda `ptrace` yapabilme hakkının bulunduğu tüm uygulamalar kontrol edilebilir
1	**Kısıtlandırılmış `ptrace` :** sadece uygulamanın doğrudan parent process'lerine veya `PR_SET_PTRACER` opsiyonuyla uygulama tarafından izin verilen debug uygulamalarına ait `PID` değerlerinin eşleştiği uygulamalara kontrol izni verilir. Böylece `gdb program_adi` ve `strace program_adi` şeklindeki kullanımlar çalışmaya devam eder ancak çalışan bir uygulamaya sonradan *attach* olmaya izin verilmeyecektir (dolayısıyla `strace -p PID` yöntemi de çalışmayacaktır). Diğer opsiyon da özellikle KDE, Chromium, Wine gibi uygulamaların kullandığı, debug/crash handler'a ait PID değerinin `PR_SET_PTRACER` ile uygulama içerisinden belirtilmesi ve bu sayede spesifik bir uygulamanın `ptrace` yapabilmesi şeklindedir
2	**Sistem yöneticisine `ptrace` :** sadece `CAP_SYS_PTRACE` özelliği tanımlanmış uygulamalar veya `prctl` ile `PTRACE_TRACEME` opsiyonunu tanımlayan çocuk process'ler kontrol edilebilir
3	**Tamamen devre dışı:** hiç bir şart altında `ptrace` yapılmasına imkan tanınmaz. Bu özellik bir defa tanımlandığı takdirde çalışma anında tekrar değişiklik yapılamaz

Her ne kadar uygulamalar `prctl` üzerinden kendilerinin `ptrace` yapılabilmesini **root** kullanıcısı dışında devre dışı bırakabiliyor olsalar da, pek çok yazılımcı bu detayların farkında değildir. OpenSSH agent'ı gibi doğrudan güvenlikle ilgili yazılımlar bu işlemleri yapıyor olmasına karşın, sistemde çalışan tüm yazılımlardan aynı davranışı beklemek doğru olmaz. Bu nedenle sistem genelinde yazılımdan bağımsız çözümlerin üretilmesi önem taşır. Son zamanlarda bazı Linux dağıtımları (Ubuntu vb.) yukarıda tariflediğimiz `ptrace_scope` dosyasının öntanımlı değerini **1** yapmaya başlamışlardır. Böylelikle `ptrace` işlemleri kısıtlandığından sistem genelinde daha güvenli bir çalışma ortamı sağlanmaktadır.

Android kullanan sistemler de düşünüldüğünde bu gibi güvenlik konularında daha hassas olunması gerektiği açıktır.

Örnek strace gerçekleştirimi

Aşağıdaki örnek uygulamayı `ministrace.c` adıyla kaydedip

```
$ gcc -m32 -o ministrace ministrace.c
```

ile derleyebilirsiniz.

> **Not:** Örnek uygulama 32 bitlik sistemler için yazılmış olup mimari farklılıkları gözardı edilmiştir. Bu sebeple 32 bitlik bir sistemde veya `gcc-multilib` kurulu ise `-m32` parametresi ile derlemelisiniz.

```
#include <sys/ptrace.h>
#include <sys/reg.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int wait_for_syscall (pid_t child)
{
    int status;
    while (1) {
        ptrace(PTRACE_SYSCALL, child, 0, 0);
        waitpid(child, &status, 0);
        if (WIFSTOPPED(status) && WSTOPSIG(status) & 0x80)
            return 0;
        if (WIFEXITED(status))
            return 1;
    }
}

int do_child (int argc, char **argv)
{
    char *args [argc+1];
    memcpy(args, argv, argc * sizeof(char*));
    args[argc] = NULL;

    ptrace(PTRACE_TRACEME);
    kill(getpid(), SIGSTOP);
    return execvp(args[0], args);
}

int do_trace (pid_t child)
{
    int status, syscall, retval;
    waitpid(child, &status, 0);
    ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACESYSGOOD);
    while(1) {
        if (wait_for_syscall(child) != 0) break;

        syscall = ptrace(PTRACE_PEEKUSER, child, sizeof(long)*ORIG_EAX);
        fprintf(stderr, "syscall(%d) = ", syscall);

        if (wait_for_syscall(child) != 0) break;

        retval = ptrace(PTRACE_PEEKUSER, child, sizeof(long)*EAX);
        fprintf(stderr, "%d\n", retval);
    }
    return 0;
}
```

```
int main (int argc, char **argv)
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s prog args\n", argv[0]);
        exit(1);
    }

    pid_t child = fork();
    if (child == 0) {
        return do_child(argc-1, argv+1);
    } else {
        return do_trace(child);
    }
}
```

Uygulama derlendikten sonra herhangi bir komutu `ministrace` ile çalıştırıp çıktıyı inceleyebiliriz:

```
$ ./ministrace date
syscall(11) = 0
syscall(12) = 21843968
syscall(21) = -2
syscall(9) = 164245504
...
syscall(9) = 164241408
syscall(1) = Tue Mar  3 13:44:52 EET 2015
29
syscall(3) = 0
...
```

Örnek uygulamamızda **65** satırlık bir kod ile strace uygulamasının temel çalışma prensibi gösterilmeye çalışılmıştır. Daha gelişmiş bir örnekte sistem çağrılarının numaralarından isimlerine ulaşmak, çağrıda kullanılan parametreleri ve geri dönüş kodlarının ilgili sistem çağrısı özelinde anlamlarını göstermek mümkün olabilir.

GNU Build Sistemi Araçları

Bu bölümde uygulamaların kaynak kodlarının derlenmesi ve sisteme kurulması süreçlerinde yardımcı olacak araçlar ele alınmaktadır.

Linux sistemler üzerinde yazılım geliştirme yaparken, sıklıkla bu araçları kullanmakta olduğumuzdan, bu konulardaki farkındalığın artırılmasının sonraki süreçlerde pek çok faydası olacaktır.

Temel olarak `make` uygulaması ve `Makefile` dosyaları kullanımı anlatılacak, daha sonra kod taşınabilirliğini ve kurulum süreçlerini de hedefleyen `autoconf & automake` sistemiyle ilgili genel bilgi verilmeye çalışılacaktır.

Make

Uygulama geliştirirken sıklıkla obje dosyalarımızı yeniden ve yeniden oluşturmak zorunda kalırız. Yerine göre `gcc`, `ld`, `ar` vb. uygulamaları tekrar tekrar aynı parametrelere çağırırız. İşte `make` ugulaması, programların yeniden derlenme sürecini otomatik hale getirmek, sadece değişen kısımların yeniden derlenmesini sağlamak suretiyle zamandan kazanmak ve işlemleri her zaman otomatik olarak doğru sırada yapmak için tasarlanmıştır.

Temel Kurallar

`make` uygulaması çalıştırıldığında, bulunulan dizinde sırasıyla `GNUmakefile`, `makefile` ve `Makefile` dosyalarını arar. Alternatif olarak `-f` seçeneği ile Makefile olarak kullanacağınız dosyayı da belirlemeniz mümkün olsa da standartların dışına çıkmamakta fayda var. `make` neyi nasıl yapacağını bu dosyalardan öğrenecektir. Eğer bulunduğunuz dizinde bir `Makefile` dosyası yoksa aşağıdaki gibi bir çıktı alacaksınız demektir:

```
$ make
make: *** No targets specified and no makefile found.  Stop.
```

> Genel kabul görmüşlüğü ve göz alışkanlığı açısından dosya adı olarak alternatiflerin yerine `Makefile` isminin kullanılması önerilir

Bir `Makefile` aslında işlemlerin nasıl yapılacağını gösteren kural tanımlamalarından oluşmaktadır. Genel olarak dosyanın biçimi aşağıdaki gibidir:

```
hedef1: bağımlılıklar
<TAB> komut
<TAB> komut
<TAB> ...

hedef2: bağımlılıklar
...
```

Burada en sık yapacağımız hata `TAB` tuşuna basmayı unutmak olacaktır. `Makefile` dosyasını hazırladığınız editörden kaynaklanan bir problem de olabilir. `TAB` işlemine dikkat edilmediğinde aşağıdaki gibi bir uyarı alabilirsiniz:

```
$ make
Makefile:6: *** missing separator (did you mean TAB instead of 8 spaces?)
```

Kurallar arasında bir satır boş bırakılması GNU make için zorunlu olmamakla birlikte bazı Unix versiyonlarıyla uyumluluk için boşluk bırakılması gereklidir.

İlk satırda `hedef1` 'in oluşturulmasında etkili olan, dolayısıyla bağımlılık üreten dosyalar birbirinden boşluk ile ayrılmış olarak tek satırda listelenir. Eğer bağımlılık kısmında yer alan dosyalardan en az birinin son değiştirilme tarihi, `hedef1` 'den daha yeni ise, `hedef1` yeniden oluşturulur. Diğer durumda `hedef1` 'in yeniden oluşturulmasına gerek olmadığı anlaşılır, çünkü `hedef1` 'in bağımlı olduğu dosyalarda `hedef1` üretildikten sonra bir değişiklik olmamıştır.

> NOT: Görüldüğü üzere dosyaların son değiştirilme tarihleri üzerinden işleyen bir mekanizma bulunmaktadır. Sistem saatiniz ileri veya geri zıplarsa kurallar doğru çalışamayacaktır. Bu durumda `make clean` ile önce tam bir temizlik yapılıp yeniden süreci başlatabilirsiniz.
>
> Eğer sistem zamanındaki oynamalar nedeniyle, dosyaların son güncellenme zamanları, o anki sistem saatinden daha ileride ise, **make: warning: Clock skew detected. Your build may be incomplete** şeklinde bir uyarı alınacaktır.

Sonraki satırlarda bağımlılık yaratan bu dosyalardan `hedef1` 'in oluşturulabilmesi için gerekli komutlar yer alır. Şimdi basit bir örnek için önce yeni bir dizin oluşturup içerisinde aşağıdaki `Makefile` dosyasını oluşturalım:

```
bolgeler: marmara karadeniz ege
    cat marmara karadeniz ege | sort -u > bolgeler

marmara: istanbul bursa
    cat istanbul bursa | sort -u > marmara

karadeniz: samsun sinop
    cat samsun sinop | sort -u > karadeniz

ege: izmir aydin
    cat izmir aydin | sort -u > ege

clean:
    rm -f ege karadeniz marmara bolgeler
```

Dosyamızı hazırladıktan sonra `make` komutunu çalıştıralım:

```
$ make
make: *** No rule to make target `istanbul', needed by `marmara'.  Stop.
```

Yukarıdaki örnekte neler olduğunu anlamaya çalışalım:

1. `make` uygulamasına `Makefile` içerisindeki bir hedef kural ismini parametre olarak

vermediğimizde öntanımlı olarak dosyada bulduğu ilk hedefi gerçekleme çalışır
2. Dosyamızdaki ilk hedefin `bolgeler` olduğunu gördük
3. `bolgeler` hedefinin bağımlılıkları `marmara`, `karadeniz` ve `ege` dosyaları şeklindeymiş
4. Eğer bulunduğumuz dizinde `bolgeler` dosyası zaten mevcut ve son değiştirilme tarihi, bağımlılıkları olan `marmara`, `karadeniz` ve `ege` dosyalarının her üçünden de daha güncel olsa idi, `make` yeni bir işlem yapmaya gerek olmadığını düşünecekti. Ancak bizim dizinimizde henüz bu dosyaların hiç biri yok
5. Bu nedenle `bolgeler` hedefini gerçeklemek için öncelikle dizinde mevcut olmayan `marmara` hedefini gerçeklemek için işlemlere başlandı
6. `marmara` hedefi de benzer şekilde `istanbul` ve `bursa` dosyalarına bağımlı ve her iki dosya da sistemde yok
7. Bir önceki durumdan farklı olarak, `istanbul` dosyası dizinde mevcut olmadığı gibi bu dosyayı üretecek herhangi bir hedef de tanımlı değil.
8. Bu yüzden ***marmara*** *hedefini üretebilmek için gereken (dizinde mevcut olmayan)* ***istanbul*** *hedefini üretecek kural da yok* şeklinde bir hata mesajı ile `make` süreci sonlanmıştır

Eğer bulunduğumuz dizinde, `istanbul` ve `bursa` dosyalarını oluşturacak olursak, `make` sonrası `marmara` hedefinin üretildiğini görebileceğiz. Aynı şekilde diğer bölgeler için de `Makefile` dosyasında tanımladığımız kurallar doğrultusunda gereken dosyaları ürettiğimizde, onlar da `make` tarafından oluşturulacaktır.

Sonrasında herhangi bir il dosyasını güncellediğimizde, ona bağlı bölge tüm bölgeleri içeren dosya otomatik olarak güncellenecektir.

Şimdi de `C` dilinde yazılmış basit bir uygulamanın derlenmesi sürecine yönelik örnek `Makefile` dosyamıza bakalım:

```
CC     = gcc
CFLAGS = -O2 -Wall -pedantic
LIBS   = -lm -lnsl

all: install

ornek: ornek.o
    $(CC) $(CFLAGS) $(LIBS) -o ornek ornek.o

ornek.o: ornek.c
    $(CC) $(CFLAGS) -c ornek.c

clean:
    rm -f ornek *.o

install: ornek
    cp ornek /usr/local/bin
```

Bu örnekte hedef olarak `ornek` uygulaması derlenecektir. Uygulamanın bağımlı olduğu dosya `ornek.o` şeklinde olup bu dosya da `ornek.c` kaynak kod dosyasına bağımlıdır.

İlk satırda yer alan `CC` değişkeniyle kullanacağımız derleyiciyi belirliyoruz. Makefile dosyaları içerisinde bu şekilde değişken tanımlaması yapıp, değişkeni dosya içerisinde `$(değişken)` şeklinde kullanabiliriz. İkinci satırda ise derleyiciye vereceğimiz bazı seçenekleri `CFLAGS` değişkenine atıyoruz. Üçüncü satırda uygulamamızın linklenmesi gereken kütüphaleri `-l` parametresiyle listeledik. Ardından ilk kuralımız geliyor: `ornek` dosyası `ornek.o` dosyasına bağımlı olarak belirtilmiş ve `ornek.o` 'dan `ornek` 'in oluşturulabilmesi için gerekli komut hemen altında listelenmiştir. Değişkenlerin değerlerini yerine koyduğumuzda komutumuz `gcc -O2 -Wall -pedantic -lm -lnsl -o ornek ornek.o` şeklinde olacaktır.

İkinci kuralımız `ornek.o` 'nun nasıl oluşturulacağını belirtmektedir. `ornek.c` dosyasında bir değişiklik olduğunda `ornek.o` dosyası hemen altında listelenen komutla yeniden oluşturulur:

```
$ gcc -O2 -Wall -pedantic -c ornek.c
```

Üçüncü kuralımızda çalıştığımız dizinde nasıl temizlik yapacağımızı belirtiyoruz. `make clean` komutunu çalıştırdığımızda `ornek` dosyası ve `.o` ile biten obje dosyaları silinecektir.

Dördüncü kuralımız ise `install` şeklinde. Bu kuralda da ornek dosyasında bir değişme olduğunda `cp ornek /usr/local/bin` komutu ile dosyayı /usr/local/bin dizini altına kopyalıyoruz.

Makefile içerisindeki her bir kural `make` uygulamasına seçenek olarak verilebilir ve ayrıca işletilebilir. Yukarıdaki gibi bir Makefile dosyasına sahipsek `make ornek.o` komutuyla sadece ornek.o için verilen kuralın çalıştırılmasını sağlayabiliriz. Veya `make install` komutuyla sadece install kuralının çalışmasını sağlayabiliriz. Ancak `install` hedefi aynı zamanda `ornek` 'e bağımlı olduğundan `ornek` için girilen kurallar da çalışacaktır. Aynı şekilde `ornek` de `ornek.o` 'ya bağımlı olduğundan `ornek.o` kuralı da çalışacaktır.

Şimdi bu `Makefile` dosyasının bulunduğu yerde `ornek.c` kaynak dosyasını da hazırladığımızı varsayarak aşağıdaki çıktıları inceleyelim:

```
$ make ornek.o
gcc -O2 -Wall -pedantic -c ornek.c

$ make clean
rm -f ornek *.o

$ make
gcc -O2 -Wall -pedantic -c ornek.c
gcc -O2 -Wall -pedantic -lm -lnsl -o ornek ornek.o
cp ornek /usr/local/bin
```

Yukarıdaki Makefile örneğimize tekrar dönelim. `make clean` komutunu çalıştırdığımızda derleme sonrasında oluşan dosyalar silinmektedir. Peki, bulunduğumuz dizinde ismi **clean** olan bir dosya mevcut ise ne olur?

```
$ touch clean
$ make clean
make: `clean' is up to date.
```

Gördüğünüz gibi **clean** adında bir dosya var olduğu ve clean için bağımlılık listesi olmadığından dolayı, kuralın güncelliğini koruduğunu ve alttaki komutların çalıştırılmaması gerektiğini düşündü. İşte bu gibi durumlar için özel bir kural mevcuttur: `.PHONY`

Yukarıda anlatılan problemi giderebilmek için Makefile dosyamızın içeriğine aşağıdaki kuralı da eklemeliyiz:

```
.PHONY: clean
```

Böylelikle `make clean` komutunun, bulunulan dizinde **clean** adında bir dosya olsa bile düzgün olarak çalışmasını sağlamış olduk, bir nevi `clean` hedefini korumaya almış olduk.

Soyut Makefile Kuralları Tanımlamak

Önceki bölümde temel olarak make kullanımı üzerinde durduk. Örnek bir Makefile hazırladık. Ancak tek bir kaynak dosyasından oluşturulan bir uygulama için `make` sistemi o kadar da yararlı bir şey değil. Zaten gerçekte de en küçük uygulama bile onlarca kaynak dosyasından oluşur. Şimdi böyle bir uygulama için Makefile hazırlayalım.

Örnek: Soyut kurallar kullanılmamış Makefile

Aşağıdaki bir kısmı ortak kullanılan az sayıda kaynak dosyadan oluşan 2 adet uygulamanın derleme sürecini yöneten Makefile örneğini inceleyiniz:

```
LIBS            = -lm \
                  -lrt \
                  -lpthread \
                  $(shell pkg-config --libs openssl)

INCLUDES        = -I/usr/local/include/custom

all: server client

server: common.o server.o list.o
    $(CC) $(CFLAGS) $(LIBS) -o server common.o server.o list.o

client: common.o client.o
    $(CC) $(CFLAGS) $(LIBS) -o client common.o client.o

common.o: common.c common.h
    $(CC) $(CFLAGS) $(INCLUDES) -c common.c

server.o: server.c server.h common.h list.h
    $(CC) $(CFLAGS) $(INCLUDES) -c server.c

client.o: client.c client.h ortak.h
    $(CC) $(CFLAGS) $(INCLUDES) -c client.c

list.o: list.c list.h
    $(CC) $(CFLAGS) $(INCLUDES) -c list.c

install: client server
    mkdir -p /usr/local/bin
    cp client /usr/local/bin/
    cp server /usr/local/bin/

uninstall:
    rm -f /usr/local/bin/client
    rm -f /usr/local/bin/server

clean:
    rm -f *.o server client

.PHONY: clean
```

Kullandığımız derleyici, derleyici seçenekleri, kütüphaneler gibi değerleri değişkenlere atamakla neler kazandığımıza bir bakalım. Derleyici parametrelerini değiştirmeye karar verdiğimizde değişken kullanmıyor olsaydık 6 farklı yerde bu değişikliği el ile yapmak zorunda kalacaktır. Fakat şimdi ise sadece `CFLAGS` değişkeninin değerini değiştirmemiz yeterli olacaktır.

Ancak gene de yukarıdaki gibi bir Makefile yazmak uzun sürecek bir işlemdir. Eğer uygulamanız 60 adet **.c** dosyasından oluşuyorsa ve 60 farklı obje için tek tek kuralları yazmak zorunda kalıyorsanız bu hoş olmaz. Çünkü tüm **.o** dosyalarını üretebilmek için vereceğimiz komut aynı: `$(CC) $(CFLAGS) $(INCLUDES) -c xxx.c`
Oysa biz 60 defa bu komutu tekrar yazmak zorundayız. İşte bu noktada soyut kurallar (*abstract rules*) imdadımıza yetişir.

Bir soyut kural genel olarak `*.u1` uzantılı bir dosyadan `*.u2` uzantılı bir dosyanın nasıl üretileceğini tanımlar. Kullanımı aşağıdaki gibidir:

```
.u1.u2:
    komutlar
    komutlar
...
```

Burada `u1` kaynak dosyanın uzantısı iken, `u2` hedef dosyanın uzantısıdır. Bu tür kullanımda dikkat ederseniz bağımlılık tanımlamaları yer almamaktadır. Çünkü tanımladığımız soyut genel kural için bağımlılık belirtmek çok anlamlı değildir. Bunun yerine **.u1** uzantılı bir dosyadan **.u2** uzantılı dosya üretmede istisnai olarak farklı bağımlılıkları olan kurallar da ileride vereceğimiz örnekte olduğu gibi belirtilebilir.

Soyut kurallar tanımlarken aşağıdaki özel değişkenleri kullanmak gerekecektir:

Özel Değişken	İşlevi
$<	Değiştiği zaman hedefin yeniden oluşturulması gereken bağımlılıkları gösterir
$@	Hedefi temsil eder
$^	Geçerli kural için tüm bağımlılıkları temsil eder

Bu bilgiler ışığında hemen bir örnek verelim. Uzantısı **.cpp** olan bir kaynak kodundan obje kodunu üretebilmek için aşağıdaki gibi bir kural tanımlayabiliriz:

```
.cpp.o:
    g++ -c $<
```

Şimdi konuya biraz daha açıklık getirelim. Kaynak dosyamızın adı **helper.cpp** ve amacımız **helper.o** obje dosyasını üretmek olsun. Yukarıdaki kural kaynak dosyamız için çalıştığında `.cpp.o:` satırı yüzünden helper.cpp, oluşacak helper.o için bir bağımlılık durumunu alır. Bu nedenle `$<` değişkeni helper.cpp'yi gösterir. Bu sayede helper.o dosyası üretilmiş olacaktır.

Şimdi aynı mantıkla obje dosyalarından çalıştırılabilir programımızı üretelim.

```
.o:
    g++ $^ -o $@
```

Bu biraz daha karışık çünkü çalıştırılabilir dosyamızın uzantısı olmayacak. Eğer tek bir uzantı verilmiş ise bunun birinci uzantı olduğu ve ikincinin boş olduğu düşünülür.

Soyut kurallar tanımladığımızda yapmamız gereken iki işlem daha bulunur. Bunlardan birincisi kullandığımız uzantıların neler olduğunu belirtmektir. Bu işlem için `.SUFFIXES` özel değişkeni kullanılır:

```
.SUFFIXES: .cpp .o
```

Diğer yapmamız gereken işlem ise üretilecek çalıştırılabilir dosyamızın hangi obje dosyalarına, obje dosyalarımızın ise hangi kaynak dosyalarına bağımlı olduğunu belirtmek olacaktır. İşin en güç tarafı da budur. Her zaman doğru değerleri yazmak o kadar kolay olmayabilir. Bu noktada **gcc** derleyicisinin `-M`, **g++** derleyicisinin `-MM` seçenekleriyle bağımlılıkları Makefile dosya biçimine uygun şekilde hesaplatabiliriz. Aşağıdaki ekran çıktısına bakalım:

```
$ gcc -M server.c
server.o: server.c /usr/include/stdio.h /usr/include/features.h \
  /usr/include/x86_64-linux-gnu/bits/wordsize.h \
  /usr/lib/gcc/x86_64-linux-gnu/4.7/include/stddef.h \
  /usr/include/x86_64-linux-gnu/bits/types.h \
  /usr/lib/gcc/x86_64-linux-gnu/4.7/include/stdarg.h \
  /usr/include/x86_64-linux-gnu/bits/stdio_lim.h \
  /usr/include/x86_64-linux-gnu/bits/sys_errlist.h common.h list.h
```

Görüldüğü gibi **server.o** için gerekli Makefile kuralını bizim için hatasız olarak verdi. Tek yapmamız gereken bu satırları kopyalayıp Makefile içerisine yapıştırmaktır. Ancak bağımlılıklar hesaplandığında, esasen pek sık değişmeyen sistem kütüphaneleri içindeki referans gösterdiğimiz başlık dosyalarının da eklenmiş olduğunu görüyoruz. Makefile dosyasına her bir `.o` bağımlılık listesi için bunlara yazarsak dosya bizim için iyice okunmaz hale gelecek. O yüzden genel sistem başlık dosyalarını atlayarak, Makefile dosyamızı bu yöntemler eşliğinde soyut kurallarla yeniden yazmayı deneyelim.

Örnek: Soyut kuralların kullanıldığı Makefile

```
LIBS            = -lm \
                  -lrt \
                  -lpthread \
                  $(shell pkg-config --libs openssl)

INCLUDES        = -I/usr/local/include/custom

.SUFFIXES: .c .o

.c.o:
    $(CC) $(CFLAGS) $(INCLUDES) -c $<

.o:
    $(CC) $(CFLAGS) $(LIBS) $^ -o $@

all: server client

server: common.o server.o list.o
client: common.o client.o
common.o: common.c common.h
server.o: server.c server.h common.h list.h
client.o: client.c client.h ortak.h
list.o: list.c list.h

install: client server
    mkdir -p /usr/local/bin
    cp client /usr/local/bin/
    cp server /usr/local/bin/

uninstall:
    rm -f /usr/local/bin/client
    rm -f /usr/local/bin/server

clean:
    rm -f *.o server client

.PHONY: clean
```

Makro Kütüphaneleri Kullanımı

Önceki örneğimizde tüm bağımlılık kurallarını `gcc` 'ye hesaplattığımızda çıkan uzun listeden pek memnun olmadık. El yordamıyla içlerinden standart kütüphane başlık dosyalarını çıkarıyor olmanın pek uygulanabilir bir çözüm olmadığı ortada. Üstelik dosya sayısı arttıkça Makefile dosyamız içerisindeki karmaşa da artacaktır. Bu sorunları nasıl çözebiliriz?

Çözüm yolu, işleri bizim için kolaylaştıran Makefile makroları yazmaktan veya hazır yazılmış olanları kullanmaktan geçmektedir.

İnternet üzerinde çeşitli Makefile makro kütüphaneleri bulmanız mümkün. Bunlardan bizce fonksiyon seti / kullanım kolaylığı / maksimum fayda dengesinde en iyilerinden biri, **Alper Akcan** tarafından hazırlanmış olan `Makefile.lib` makro kütüphanesidir. Kütüphaneyi https://github.com/alperakcan/libmakefile adresinden indirebilirsiniz.

`Makefile.lib`, çapraz derleme işlemlerini `CROSS_COMPILE_PREFIX` değişkeninin ayarlanması suretiyle yönetebilmektedir. Ayrıca detaylı çıktı vermesi için make sistemlerinde alışageldiğimiz haliyle *verbose* opsiyonu `V` değişkeni üzerinden `V=1` şeklinde bir atama yapmak suretiyle aktifleştirilebilmektedir.

Bu şekilde yazılmış örnek bir Makefile dosyasına bakalım:

```
target-y  = target1
target-y += target2

target1_files-y = \
    target_file_shared.c \
    target1_file_2.c \
    target1_file_3.c

target1_includes-y = \
    ./ \
    /opt/include

target1_libraries-y = \
    ./ \
    /opt/lib

target1_cflags-y = \
    -DUSER_DEFINED \
    -O2

target1_ldflags-y = \
    -luserdefined

target2_files-y = \
    target_file_shared.c \
    target2_file_2.c \
    target2_file_3.c

target2_includes-y = \
    ./ \
    /opt/include

target2_libraries-y = \
    ./ \
    /opt/lib

target2_cflags-y = \
    -DUSER_DEFINED \
    -O2

target2_ldflags-y = \
    -luserdefined

include Makefile.lib
```

Projenin github sayfasından ve `Makefile.lib` dosya içeriğinden kullanımıyla ilgili yardım alabilirsiniz.

Makefile.lib kullandığınızda, tüm bağımlılıklar sistem kütüphaneleri de dahil olarak detaylı biçimde hesaplanır. Bağımlılıkların neler olduğu ve hesaplanmasında hangi komutun kullanıldığı gibi ek bilgiler, bulunulan dizinde nokta ile başlayan (dolayısıyla ön tanımlı `ls` komutunda listelenmeyen ve göz kalabalığı oluşturmayan) bir dizin yapısı altında saklanır. Derleme süreci daha sağlıklı ve temiz bir şekilde ilerler. Aşağıdaki Makefile.lib kullanılan bir projedeki derleme zamanı çıktıları görünmektedir:

```
$ make
  DEP         /home/demirten/embedded/gateway/.gateway/base64.dep
  DEP         /home/demirten/embedded/gateway/.gateway/backend.dep
  DEP         /home/demirten/embedded/gateway/.gateway/database.dep
  DEP         /home/demirten/embedded/gateway/.gateway/common.dep
  DEP         /home/demirten/embedded/gateway/.gateway/ini.dep
  DEP         /home/demirten/embedded/gateway/.gateway/gateway.dep
  CC          /home/demirten/embedded/gateway/.gateway/gateway.o
  CC          /home/demirten/embedded/gateway/.gateway/ini.o
  CC          /home/demirten/embedded/gateway/.gateway/common.o
  CC          /home/demirten/embedded/gateway/.gateway/database.o
  CC          /home/demirten/embedded/gateway/.gateway/backend.o
  CC          /home/demirten/embedded/gateway/.gateway/base64.o
  LINK        /home/demirten/embedded/gateway/.gateway/gateway
  CP          /home/demirten/embedded/gateway/gateway

$ ls .gateway/
backend.dep      backend.o.cmd   base64.o      common.dep.cmd
database.dep     database.o.cmd  gateway.dep   gateway.o.cmd
ini.o            backend.dep.cmd base64.dep    base64.o.cmd
common.o         database.dep.cmd  gateway.dep.cmd  ini.dep
ini.o.cmd        backend.o       base64.dep.cmd  common.dep
common.o.cmd     database.o      gateway.cmd     gateway.o
ini.dep.cmd
```

Görüldüğü üzere make uygulaması çalıştırıldığında önce bağımlılıklar hesaplanmış, hesaplama sonuçları ve kullanılan komutlar hedef uygulama ismi olan `gateway` 'den yola çıkılarak `.gateway` adıyla oluşturulan dizin altında toplanmış, tüm derleme işlemleri sonucu oluşan obje dosyaları da `.gateway` altında biriktirilmeye devam edilmiş ve işlem sonucunda ana dizinde `gateway` hedef uygulaması oluşturulmuştur.

Eğer derleme sürecinde daha detaylı ekran çıktısı almak istersek, `make V=1` şeklinde komutu çalışamız yeterli olacaktır.

Daha karmaşık bir Makefile.lib örneğine bakalım (gerçek zamanlı harita render kütüphanemizin demo uygulaması kısmından alınmıştır):

```
libmakefile Örnek 2

-include ${PLATFORM}.config
```

```
include Makefile.config

MOC ?= moc

target_o-y = \
    libnavigator.o

target_a-y = \
    libnavigator.a

target-$(ENABLE_NAVIGATOR_DEMO) = \
    navigator-demo

libnavigator.o_files-y = \
    tag.h \
    navigator.c \
    ../amalgamation/libamalgamation.o

libnavigator.a_files-y = \
    libnavigator.o \

ifeq (${COMMON_POINT_TYPE}, int)
libnavigator.o_cflags-y += \
    -DNAVIGATOR_COMMON_POINT_TYPE_INT=1
endif

ifeq (${COMMON_POINT_TYPE}, double)
libnavigator.o_cflags-y += \
    -DNAVIGATOR_COMMON_POINT_TYPE_DOUBLE=1
endif

ifeq (${COMMON_BOUND_TYPE}, int)
libnavigator.o_cflags-y += \
    -DNAVIGATOR_COMMON_BOUND_TYPE_INT=1
endif

ifeq (${COMMON_BOUND_TYPE}, double)
libnavigator.o_cflags-y += \
    -DNAVIGATOR_COMMON_BOUND_TYPE_DOUBLE=1
endif

navigator-demo_depends-y = \
    libnavigator.o

navigator-demo_files-y = \
    navigator-qt.cpp \
    navigator-qt-moc.cpp \
    libnavigator.o

navigator-qt-moc.cpp: navigator-qt.h
    ${Q}$(MOC) ${navigator-demo_cflags-y} navigator-qt.h -o navigator-qt-moc.cpp

navigator-demo_cflags-y = \
```

```
    $(shell pkg-config QtGui --cflags)

navigator-demo_cflags-${ENABLE_RENDER_CAIRO} += \
    -DNAVIGATOR_ENABLE_RENDER_CAIRO=1

navigator-demo_cflags-${ENABLE_RENDER_FLOATING_SCALE} += \
    -DNAVIGATOR_ENABLE_RENDER_FLOATING_SCALE=1

navigator-demo_ldflags-y = \
    $(shell echo $(CC) | awk '{ if (/mingw/) { print "-Wl,-Bstatic -static-libgcc -L/m
ingw/Qt/lib -lmman"; } }') \
    $(shell pkg-config freetype2 --libs) \
    -lexpat \
    -lm \
    -lz \
    $(shell pkg-config QtGui --libs) \
    -lpthread

navigator-demo_ldflags-${ENABLE_RENDER_CAIRO} += \
    $(shell pkg-config cairo --libs)

navigator-demo_ldflags-${ENABLE_COMPRESS_SNAPPY} += \
    -lsnappy

navigator-demo_ldflags-${ENABLE_INPUT_TIF} += \
    -lgeotiff \
    -ltiff

navigator-demo_ldflags-${ENABLE_RENDER_PNG} += \
    $(shell pkg-config libpng --libs)

navigator-demo_ldflags-${ENABLE_RENDER_JPEG} += \
    -ljpeg

include Makefile.lib
```

İlk 2 satıra dikkatlice tekrar bakalım:

```
-include ${PLATFORM}.config
include Makefile.config
```

Satır başında yer alan **-** karakteri, ilgili dosya *include* edilmek için arandığında dizinde yer almıyorsa **make** sürecinin hata vermeyip yoluna devam etmesini sağlamak için konulmuştur. Eğer `make` çalıştırılırken `PLATFORM` değişkenine atama yapılırsa, öncelikle ilgili dosya *include* edilecektir.

Ardından her koşulda `Makefile.config` dosyasının *include* edildiğini görmekteyiz.

Şimdi **PLATFORM** değişkenin **Debian** olarak atandığı örnek bir `make` kullanımını ve `Debian.config` ile `Makefile.config` dosyalarının içeriklerini görelim:

```
$ make ENABLE_INPUT_OSM=n PLATFORM=Debian -j 8
```

Yukarıdaki kullanımda öncelikle `ENABLE_INPUT_OSM` değişkeninin değeri `n` olarak, `PLATFORM` değişkenin değeri `Debian` olarak atanmakta ve **8** paralel derleme sürecine imkan verecek şekilde uygulama başlatılmaktadır.

```
# Debian.config
ENABLE_NAVIGATOR_DEMO        ?= y

ENABLE_RENDER_COPYRIGHT      ?= y
RENDER_COPYRIGHT_COLOR       ?= 0x20ddbbcc

ENABLE_COMMON_CLIPPER        ?= y
COMMON_FILE_CACHE_SIZE       ?= 0x1400000

ENABLE_INPUT_TIF             ?= y
ENABLE_INPUT_OSM             ?= y
ENABLE_INPUT_JPEG_TURBO      ?= n
ENABLE_INPUT_TILE            ?= y

ENABLE_RENDER_CAIRO          ?= n
ENABLE_RENDER_PNG            ?= y
ENABLE_RENDER_JPEG           ?= y
ENABLE_RENDER_JPEG_TURBO     ?= n
ENABLE_RENDER_TEXT_ON_PATH   ?= y

ENABLE_COMPRESS_SNAPPY       ?= n

ENABLE_RENDER_FLOATING_SCALE ?= y

ENABLE_EXTERNAL_LOGGER       ?= n

ENABLE_DEBUGF                ?= n
ENABLE_ERRORF                ?= y
ENABLE_INFOF                 ?= y
ENABLE_TODOF                 ?= n
ENABLE_ASSERTF               ?= n

COMMON_POINT_TYPE            ?= int
COMMON_BOUND_TYPE            ?= int
```

```
# Makefile.config

ENABLE_NAVIGATOR_DEMO        ?= y

ENABLE_RENDER_COPYRIGHT      ?= y
RENDER_COPYRIGHT_COLOR       ?= 0x202a77a3

ENABLE_COMMON_CLIPPER        ?= y
COMMON_FILE_CACHE_SIZE       ?= 0x2000000

ENABLE_INPUT_TIF             ?= y
ENABLE_INPUT_OSM             ?= y
ENABLE_INPUT_JPEG_TURBO      ?= y
ENABLE_INPUT_TILE            ?= y

ENABLE_RENDER_CAIRO          ?= n
ENABLE_RENDER_PNG            ?= n
ENABLE_RENDER_JPEG           ?= y
ENABLE_RENDER_JPEG_TURBO     ?= y
ENABLE_RENDER_TEXT_ON_PATH   ?= y

ENABLE_COMPRESS_SNAPPY       ?= y

ENABLE_RENDER_FLOATING_SCALE ?= y

ENABLE_EXTERNAL_LOGGER       ?= n

ENABLE_DEBUGF                ?= n
ENABLE_ERRORF                ?= y
ENABLE_INFOF                 ?= y
ENABLE_TODOF                 ?= n
ENABLE_ASSERTF               ?= y

COMMON_POINT_TYPE            ?= double
COMMON_BOUND_TYPE            ?= double
```

Bu şekildeki konfigürasyon dosyaları yardımıyla, çeşitli değişkenlerin hem öntanımlı değerlerini atayabilmekte, hem de kullanıcı tarafından özellikle belirtilmiş ise, ilgili değeri kullanabilmekteyiz. Bunun için `?=` tanımından faydalanıyoruz. Bu tanım Makefile içerisinde, **eğer değişkene atama yapılmamış ise → ata** şeklinde işlev görmektedir

Bu yapı ile farklı konfigürasyon dosyaları kullanılabildiği gibi, konfigürasyon dosyası içerisinde `?=` şeklinde tanımlanmış değişkenleri aşağıdaki şekilde ezmek de mümkün olmaktadır:

```
$ make ENABLE_DEBUGF=y PLATFORM=Debian
```

Yukarıdaki komut, bir önceki Makefile örneğiyle birlikte değerlendirildiğinde, Debian.config dosyası içerisinde yer alan `ENABLE_DEBUGF ?= n` şeklindeki satırın geçersiz olmasını sağlayacaktır

Eğer Debian.config içerisinde bu tanım `ENABLE_DEBUGF = n` şeklinde doğrudan eşittir karakteri ile yapılmış olsaydı, öncesinde atanan değerden bağımsız olarak bu konfigürasyon dosyası işlendiğinde her zaman dosyanın içindeki atama geçerli olacaktı. Buradaki küçük detaylar dikkatle kullanıldığında, aynı kod üzerinden farklı konfigürasyonlarda derleme işleminiz kolaylaşacaktır.

Make Alternatifleri

CMake

CMake, birden çok platformu destekleyen (Linux, Apple, Windows) güçlü bir inşa aracıdır.

Geliştirilmesi büyük ölçüde **Kitware** firması tarafından yapılmaktadır.

CMake kendi kural dosyalarını işleyerek, hangi platformda çalışıyorsa o platform için doğal inşa sistemine ait kural dosyaları oluşturur (*NIX sistemler için Makefile).

Aşağıdaki örnek CMake dosyasını inceleyiniz:

```
if (${UNIX})
  set (DESKTOP $ENV{HOME})
else()
  set (DESKTOP $ENV{USERPROFILE}/Desktop)
endif()

set  (PRJ      ${DESKTOP}/common/svn )
set  (FILELIST ${PRJ}/src/source.txt )

message(STATUS "CMAKE_GENERATOR : ${CMAKE_GENERATOR}")
message(STATUS "DESKTOP         : ${DESKTOP}")
message(STATUS "PRJ             : ${PRJ}")
message(STATUS "FILELIST        : ${FILELIST}")
message(STATUS "SYSTEM_NAME     : ${CMAKE_SYSTEM_NAME}")

project(project_name)

include_directories(
  ${PRJ}/src
  ${PRJ}/includes
)

# Load SRC Variable from file
file(READ ${FILELIST} SRC)
string(REGEX REPLACE  "#.*$"  ""  SRC  ${SRC})
string(REPLACE        "\n"    ";" SRC  ${SRC})

add_executable(${PROJECT_NAME} ${SRC} )

foreach (f ${SRC})
  set_source_files_properties(${f} PROPERTIES LANGUAGE       CXX)
endforeach(f)

if (${WIN32})
  link_directories(
  )

  add_definitions(
    -DDEFINE1
  )

  target_link_libraries(
    ${PROJECT_NAME}
    wsock32.lib
  )
endif()
```

SCons

SCons, Python dili ile geliştirilmiş, birden çok platformu destekleyen diğer bir inşa aracıdır.

Konfigürasyon betikleri Python dosyalarından oluşur

C, C++ ve Fortran için doğrudan kod bağımlılık analizi desteği sunar.

Versiyon kontrol sistemlerine doğrudan destek verir (SCCS, RCS, CVS, Subversion, BitKeeper, Perforce).

Dosyaların değişimindeki kontroller son değiştirilme tarihi yerine **MD5SUM** değerleri üzerinden yapılır. Parametre vererek dosyanın değiştirilme tarihine bakacak hale de getirmek mümkündür.

Aşağıdaki örnek SCons dosyasını inceleyebilirsiniz:

```
env = Environment()
env.Append(CPPFLAGS=['-Wall','-g'])
env.Program('hello', ['hello.c', 'main.c'])
```

Rake

Ruby ile geliştirilmiş ve daha çok Ruby projelerinde kullanılan bir inşa aracıdır.

Ruby'nin DSL tanımlama noktasındaki güçlü özelliklerini kullanır.

Kurallar `Rakefile` dosyalarında tutulur.

Rakefile içerisinde Ruby dilinde görevler ve kurallar tanımlanabildiği gibi, dilinden kendisinden gelen ekstra özelliklerle örneğin çalışma zamanında yeni sınıflar dahi üretilebilir.

Aşağıdaki örnek Rakefile dosyasını inceleyebilirsiniz:

```
namespace :cake do
  desc 'make pancakes'
  task :pancake => [:flour,:milk,:egg,:baking_powder] do
     puts "sizzle"
  end
  task :butter do
    puts "cut 3 tablespoons of butter into tiny squares"
  end
  task :flour => :butter do
    puts "use hands to knead butter squares into 1{{frac|1|2}} cup flour"
  end
  task :milk do
    puts "add 1{{frac|1|4}} cup milk"
  end
  task :egg do
   puts "add 1 egg"
  end
  task :baking_powder do
   puts "add 3{{frac|1|2}} teaspoons baking powder"
  end
end
```

autoconf, automake Kullanımı

GNU build sistemi iki temel amacın gerçekleştirilebilmesi için geliştirilmiştir: Programları platformlar arası daha rahat taşınabilir hale getirmek ve kaynak koddan program kurulumlarını mümkün olduğu kadar basite indirgeyebilmek.

Taşınabilir kod yazmak gerçekten oldukça zahmetli bir iştir. Hedef mimarinin ayrıntılı olarak özelliklerinin bilinmesi çoğu zaman mümkün değildir. Bir önceki bölümde örnek olarak yazdığımız Makefile dosyasında `mkdir -p /usr/local/bin` komutunu kullanmıştık. Oysa `mkdir` komutunun `-p` seçeneği tüm Unix sistemlerinde aynı şekilde çalışmaz. Bu ve bunun gibi pek çok farklılık yüzünden her Unix sisteminde çalışabilecek bir `Makefile` yazmak oldukça güçtür. Kullanılan kütüphanelerin sistemler arasındaki farklılıkları ise apayrı bir konudur. İşte GNU build sistemi tüm bu zorlukların üstesinden gelebilmek için oluşturulmuştur.

Kdevelop gibi programlar yeni proje oluşturduğunuzda build sistemini de otomatik olarak oluşturmaktadırlar. Ancak oluşan dosyalar fazlasıyla karışık olduğundan bu bölümde çok daha basit örneklerle yapıyı anlatmaya çalışacağız. Buradaki temel bilgilerden yararlandıktan sonra Kdevelop vb. gibi programların ürettiği veya internetten indirmiş olduğunuz herhangi bir uygulamanın kaynak kodu içerisinde gezinerek farklı kullanımları inceleyebilirsiniz.

Gerekli Araçlar

Gnu build sistemi için gerekli araçlar ve kullanım alanları aşağıdaki gibidir:

1. **autoconf**: konfigürasyon için kullanılacak configure betik programını üretir. Kodun taşınabilir olmasını etkileyecek özellikleri, üzerinde çalıştığı platform için denetler. Elde ettiği değerleri, daha önceden belirtilmiş şablonlara uygun şekilde birleştirerek özelleştirilmiş Makefile, başlık dosyaları vb. oluşturur. Bu sayede programı derleyecek kullanıcı tek tek elle bu değişiklikleri yapmak zahmetinden kurtulur.
2. **automake**:, autoconf için kullanılacak Makefile şablonlarını (Makefile.in) Makefile.am dosyalarını temel alarak üretir. Automake tarafından üretilen Makefile dosyaları GNU makefile standartlarına uygun olup, kullanıcıyı elle Makefile dosyası oluşturma zahmetinden kurtarır. Autoconf'un çalışabilmesi için öncelikle automake'in düzgün olarak çalışması gereklidir.

3. **libtool**: özellikle paylaşımlı kütüphanelerin taşınabilir bir yapıda oluşturulabilmesi için gereken pek çok detayı kullanıcıdan soyutlar. Kullanımı için autoconf veya automake gerekli değildir, tek başına da kullanılabilir. Automake ise libtool'u destekler ve onunla birlikte çalışabilir.
4. **autotools**: GNU kodlama standartlarına uygun, taşınabilir kod üretmede yardımcı araçlar içeriz.

GNU build sistemi tarafından gerçekleştirilen temel görevler şunlardır:

1. Çok sayıda alt dizin içeren kaynak kodlardan uygulamaları üretebilir. Her bir dizin için ayrıca make komutunu çağırmak zahmetinden geliştiriciyi kurtarır. Bu sayede tüm kaynak kodları aynı dizinde bulundurmak yerine daha hiyerarşik bir dizin yapısı kullanabilirsiniz.
2. Konfigürasyon işlemini otomatik olarak yapar. Kullanıcıların Makefile dosyalarını düzenlemelerine gerek kalmaz.
3. Makefile dosyalarını otomatik olarak üretir. Makefile yazımı büyük projelerde sürekli tekrar gerektirir ve aynı zamanda hata yapmaya elverişli bir yapıdır. GNU build sistemi için sadece `Makefile.am` şablonunun yazımı yeterlidir. Bu sayede hata yapma olasılığı azalır ve yönetimi kolay hale gelir.
4. Hedef platform için özel testler yapabilme imkanı sunar. Makefile.am dosyasına eklenecek bir kaç satırla hedef platformda programın derlenebilesi için aranan özelliklerin var olup olmadığı kontrol edilebilir.
5. Paylaşımlı kütüphanelerin oluşturulması statik kütüphanelerin oluşturulması kadar kolay hale gelir.

GNU build sistemi için gerekli olan bu araçların sadece geliştirmenin yapıldığı sistemde kurulu olması yeterlidir. Bu programlar çalıştıktan sonra her platformda çalışabilecek betik programları üretirler. Bu sayede uygulamanızın kaynak kodunu indirip kurmak isteyen biri, `autoconf`, `automake` gibi araçları da sistemine kurmak zorunda kalmaz.

İşlemlere başlamadan önce `autoconf`, `automake` ve `libtool` paketlerini sistemimize kuralım:

```
$ sudo apt-get install autoconf automake libtool
```

Zorunlu olmamakla birlikte, `configure.ac` dosyalarımız içerisinde temel `M4` makro seti içerisinde yer almayan ancak muhtemelen kullanmak zorunda kalacağımız ek makro arşivini içeren `autoconf-archive` paketini de kurmamız yerinde olacaktır:

```
$ sudo apt-get install autoconf-archive
```

Örnek Makro: `AX_FUNC_MKDIR`

`mkdir()` fonksiyonu bazı platformlarda `_mkdir()` şeklinde kullanılmaktadır.

Bazı platfomlarda dizin erişim yetkileri için 2. argüman kullanılıyorken diğerlerinde tek argüman kullanıldığı bilinmektedir.

Aşağıkda `mkdir()` fonksiyonunun platformlar arası taşınabilirliğini bizim için kolaylaştıran `AX_FUNC_MKDIR` makro örneği verilmiştir:

```
#if HAVE_MKDIR
#  if MKDIR_TAKES_ONE_ARG
     /* MinGW32 */
#    define mkdir(a, b) mkdir(a)
#  endif
#else
#  if HAVE__MKDIR
     /* plain Windows 32 */
#    define mkdir(a, b) _mkdir(a)
#  else
#    error "Don't know how to create a directory on this system."
#  endif
#endif
```

Örnek Uygulama

Şimdi aşağıdaki örnek uygulamamız için GNU build sistemini nasıl kullanacağımızı öğrenelim.

```
#include <stdio.h>
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

int main (int argc, char *argv[])
{
        printf("Örnek Çalışıyor\n");

        if (argc == 2) {
                const char *target = argv[1];
                int rc;
#ifdef HAVE_MKDIR
#ifdef MKDIR_TAKES_ONE_ARG
                rc = mkdir(target);
#else
                rc = mkdir(target, 0755);
#endif
                if (rc == 0) {
                        printf("%s dizini oluşturuldu\n", target);
                } else {
                        fprintf(stderr, "%s dizini oluşturulamadı\n", target);
                }
#else
                fprintf(sdterr, "mkdir() desteklenmiyor\n");
#endif
        }

    return 0;
}
```

Programımızı **ornek.c** olarak kaydedelim. Şimdi programın derlenmesi işlemlerini autoconf ve automake ile yapmaya başlayalım. Bunun için öncelikle `automake` ile kullanılmak üzere aşağıdaki gibi bir `Makefile.am` dosyasını oluşturalım:

```
AUTOMAKE_OPTIONS = foreign
bin_PROGRAMS = ornek
ornek_SOURCES = ornek.c
```

`foreign` opsiyonu sayesinde ilerleyen aşamalarda GNU uygulamalarında bulunması zorunlu dosya kontrollerini devre dışı bırakıyoruz (Bu seçeneği kullanmadığımız durumdaki senaryoyu test ediniz)

Ardından aşağıdaki gibi bir `configure.ac` dosyasını oluşturalım (eski versiyonlarda dosya ismi `configure.in` şeklindeydi):

```
AC_INIT([ornek], [0.1], [ornek-bugs@yh.com.tr])
AC_CONFIG_SRCDIR([ornek.c])
AM_INIT_AUTOMAKE

AC_CONFIG_HEADERS([config.h])

AC_PROG_CC

AC_CHECK_HEADERS([sys/time.h])

AC_CHECK_FUNCS([gettimeofday])

AX_FUNC_MKDIR

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Dosyaları oluşturup kaydettikten sonra şimdi `aclocal` komutunu çalıştıralım:

```
$ aclocal
$ ls -l
-rw-r--r-- 1 demirten demirten 41928 Dec 27 01:04 aclocal.m4
drwxr-xr-x 2 demirten demirten  4096 Dec 27 01:04 autom4te.cache
-rw-r--r-- 1 demirten demirten   120 Dec 27 01:03 configure.ac
-rw-r--r-- 1 demirten demirten    45 Dec 27 00:51 Makefile.am
-rw-r--r-- 1 demirten demirten    87 Dec 27 00:26 ornek.c
```

`aclocal` çalıştıktan sonra bulunduğumuz dizinde `autom4te.cache` dizini ve `aclocal.m4` dosyası oluştu.

Sonraki adımda autoheader komutunu çalıştırıyoruz. Eğer `configure.ac` dosyasında `AC_CONFIG_HEADERS` belirtilmemişse yapılacak kontroller sonrası elde edilen değerler `config.h` dosyasına yazılmayacak olduğundan bu adımı atlayabilirsiniz:

```
$ autoheader
$ ls -l config.h.in
-rw-r--r-- 1 demirten demirten 1825 Dec 27 02:43 config.h.in
```

İşlem bitiminde `config.h` dosyası için kullanılacak `config.h.in` şablonu üretildi.

Sıradaki işlem autoconf komutuyla configure betiğini oluşturmak:

```
$ autoconf
$ ls -l configure
-rwxr-xr-x 1 demirten demirten 140829 Dec 27 01:04 configure
```

İşlem sonrasında bulununan dizinde `configure` adlı tanıdık betik uygulamasını oluşturmuş oluyoruz.

`configure` betiği de oluştuktan sonra `automake -a` komutuyla gereken diğer dosyaları oluşturuyoruz:

```
$ automake -a
configure.ac:4: installing './compile'
configure.ac:3: installing './install-sh'
configure.ac:3: installing './missing'
Makefile.am: installing './depcomp'
```

Yazılımlarda olması beklenen bazı temel dosyaların (ChangeLog, README vb.) çalışma dizinimizde bulunmaması nedeniyle uyarı mesajlarını görmekteyiz ama şu aşamada bunu dikkate almamıza gerek yok.

`automake -a` komutu sonrası dizin içeriğimiz aşağıdaki hale gelmektedir:

```
$ ls
aclocal.m4
autom4te.cache
compile -> /usr/share/automake-1.14/compile
config.h.in
configure
configure.ac
depcomp -> /usr/share/automake-1.14/depcomp
install-sh -> /usr/share/automake-1.14/install-sh
Makefile.am
Makefile.in
missing -> /usr/share/automake-1.14/missing
ornek.c
```

Bu haliyle `./configure` şeklinde betik uygulamamızı çalıştırıp sonucunu görelim:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking whether gcc understands -c and -o together... yes
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking how to run the C preprocessor... gcc -E
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking sys/time.h usability... yes
checking sys/time.h presence... yes
checking for sys/time.h... yes
checking for gettimeofday... yes
checking for mkdir... yes
checking for _mkdir... no
checking whether mkdir takes one argument... no
checking that generated files are newer than configure... done
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
config.status: executing depfiles commands
```

İşlem bitiminde hem `Makefile` hem de `config.h` dosyası oluşacaktır:

```
/* config.h.  Generated from config.h.in by configure.  */
/* config.h.in.  Generated from configure.ac by autoheader.  */

/* Define to 1 if you have the `gettimeofday' function. */
#define HAVE_GETTIMEOFDAY 1

/* Define to 1 if you have the <inttypes.h> header file. */
#define HAVE_INTTYPES_H 1

/* Define to 1 if you have the `mkdir' function. */
#define HAVE_MKDIR 1

/* Define to 1 if you have the <sys/types.h> header file. */
#define HAVE_SYS_TYPES_H 1

/* Define to 1 if you have the <unistd.h> header file. */
#define HAVE_UNISTD_H 1

/* Define to 1 if you have the `_mkdir' function. */
/* #undef HAVE__MKDIR */

/* Define if mkdir takes only one argument. */
/* #undef MKDIR_TAKES_ONE_ARG */

/* Name of package */
#define PACKAGE "ornek"

/* Define to the address where bug reports for this package should be sent. */
#define PACKAGE_BUGREPORT "ornek-bugs@yh.com.tr"

/* Define to the full name of this package. */
#define PACKAGE_NAME "ornek"

/* Define to the full name and version of this package. */
#define PACKAGE_STRING "ornek 0.1"

/* Define to the one symbol short name of this package. */
#define PACKAGE_TARNAME "ornek"

/* Define to the home page for this package. */
#define PACKAGE_URL ""

/* Define to the version of this package. */
#define PACKAGE_VERSION "0.1"

/* Define to 1 if you have the ANSI C header files. */
#define STDC_HEADERS 1

/* Version number of package */
#define VERSION "0.1"
```

Artık Makefile dosyamız hazır olduğunda göre, `make` komutu ile derleme işlemini yapabiliriz:

```
$ make
make  all-am
make[1]: Entering directory /tmp/deneme
gcc -DHAVE_CONFIG_H -I. -g -O2 -MT ornek.o -MD -MP -MF \
    .deps/ornek.Tpo -c -o ornek.o ornek.c
mv -f .deps/ornek.Tpo .deps/ornek.Po
gcc  -g -O2   -o ornek ornek.o
make[1]: Leaving directory /tmp/deneme
```

Uygulamamızı çalıştıralım:

```
$ ./ornek dizin1
Örnek Çalışıyor
dizin1 dizini oluşturuldu
```

Uygulamanın Kurulumu

autotools bileşenleriyle yaptığımız bu çalışma, uygulamanın sadece derlenmesi sürecini değil, kurulum ve sistemden kaldırma süreçlerini de desteklemektedir. Bu noktada dilerseniz uygulamayı sisteme kurabilirsiniz. Bunun için uygulamanın kurulacağı dizinler üzerinde yazma yetkinizin bulunması gerekecektir.

Uygulamayı sistemimize kurmak için:

```
$ sudo make install
make[1]: Entering directory /tmp/deneme
/bin/mkdir -p /usr/local/bin
/usr/bin/install -c ornek /usr/local/bin
make[1]: Nothing to be done for install-data-am.
make[1]: Leaving directory /tmp/deneme
```

Sistemden kaldırmak için:

```
$ sudo make uninstall
( cd /usr/local/bin && rm -f ornek )
```

`autoreconf`

Zaman içerisinde `Makefile.am` veya `configure.ac` dosyalarında değişiklik yaptığınızda, autotools araçlarını **doğru sırada** yeniden çalıştırmanız gerekecektir.

Alternatif olarak, bu işlemi kolaylaştıran `autoreconf` komutunu `-vfi` parametresi ile kullanabiliriz. Bu şekilde değişiklik var ise gerekli bileşenler doğru sırada çalışacaktır:

```
$ autoreconf -vfi
autoreconf: Entering directory `.'
autoreconf: configure.ac: not using Gettext
autoreconf: running: aclocal --force
autoreconf: configure.ac: tracing
autoreconf: configure.ac: not using Libtool
autoreconf: running: /usr/bin/autoconf --force
autoreconf: running: /usr/bin/autoheader --force
autoreconf: running: automake --add-missing --copy --force-missing
```

Uygulamanın Dağıtıma Hazırlanması

Uygulamanızın artık hazır olduğunu düşündüğünüzde `make distcheck` komutu ile onu paket haline getirebilirsiniz (Ekran çıktısı biraz uzun olduğundan burada listelenmemiştir). Bu komut işini tamamladığında bulunduğunuz dizinde `ornek-0.1.tar.gz` adında bir dosya oluşacaktır. Artık bu dosya ile programınızın dağıtımını yapabilirsiniz.

Şimdi biraz da yaptığımız bu örneği biraz daha açıklayalım.

- Makefile.am içerisinde mantıksal bir dil kullandık. Yazdığımız hiç bir satır çalıştırılmadı.
- Diğer yandan configure.in içerisinde kullandığımız dil prosedüreldir, yazdığımız her satır çalıştırılacak bir komutu göstermektedir.
- Makefile.am dosyası içerisindeki ilk satır programın ismini belirtirken ikinci satır programı oluşturan kaynak kodları belirtmektedir.

Şimdi daha karışık olan configure.in içerisindeki komutlara sırasıyla bakalım:

- **AC_INIT** komutu configure betiği için ilklendirmeleri yapar. Parametre olarak kaynak dosyaların adlarını alır.
- **AM_INIT_AUTOMAKE** komutu, automake kullanacağımızı gösterir. Parametre olarak programın ismini ve versiyonunu alır. Eğer Makefile.in dosyalarını elle hazırlayacak olsaydık bu komutu kullanmamıza da gerek olmayacaktı.
- **AC_PROG_CC** komutu kullanılan C derleyicisinin ne olduğunu belirler.
- **AC_PROG_INSTALL** komutu BSD uyumlu install uygulamasına sahip olup olmadığımızı denetler. Eğer yoksa bu işlem için install-sh'ı kullanır.
- **AC_OUTPUT** komutu configure betik programının Makefile dosyalarını Makefile.in dosyalarından üretmesi gerektiğini belirtir.

Örneğimizdeki configure.in dosyası içerisinde yer almayan ama sıklıkla kullanacağımız bazı komutlar da şunlardır:

- **AC_PROG_RANLIB** komutuyla bir kütüphane geliştiriyorsak ranlib'in sistemde nasıl kullanılacağını öğrenebiliriz.
- **AC_PROG_CXX** komutuyla sistemdeki C++ derleyicisinin ne olduğunu öğrenebiliriz.
- **AC_PROG_YACC** ve **AC_PROG_LEX** komutlarıyla kaynak kodlarımız lex veya yacc dosyaları içeriyorsa bu uygulamaların sistemde varlığını denetleyebiliriz.

Eğer alt dizinlerde başka Makefile dosyalarımız da olacaksa bunu

```
AC_OUTPUT(Makefile         \
          dizin1/Makefile  \
          dizin2/Makefile  \
          )
```

komutlarıyla belirtebiliriz.

Dosyaların içeriğinden bahsettikten sonra şimdi de biraz önc yaptığımız örnekte çalıştırdığımız komutlardan sonra neler olduğuna tekrar bakalım.

- `aclocal` komutu çalıştıktan sonra aclocal.m4 dosyası üretilir. Bu dosya içerisinde autoconf tarafından kullanılacak olan makrolar yer almaktadır (kendi özel makrolarımızı nasıl hazırlayacağımıza ileride değinilecektir).
- `autoconf` komutuyla aclocal.m4 ve configure.ac dosyaları işlenerek configure betik programı oluşturulur.
- `automake` komutu Makefile.am dosyasını temel alan bir Makefile.in oluşturur. Ayrıca GNU kodlama standartlarına göre eksik olan dosyalar için örnek birer kopya üretir.
- `./configure` komutuyla çalıştırılan betik programı daha önceden belirtilen özellikler için sistemimizi test eder ve Makefile.in dosyasını örnek alarak Makefile dosyalarını oluşturur. AC_OUTPUT() ile belirtilen tüm dosyalardaki @FOO@ şeklindeki kayıtları FOO için elde edilen değerlerle değiştirir (örneğin C derleyicisinin ne olduğu gibi).

Konfigürasyon Başlık Dosyalarının Kullanımı

Çoğu zaman derleme anında bazı makrolar tanımlamak isteriz. `-D` seçeneği ile derleyiciye bildirilen bu değerleri programımız içerisinden kullanarak ilgili kod parçacığının çalışma şeklini değiştirebiliriz. autoconf kullandığımız bir uygulama için böylesi seçenekleri kullanmanın yolu konfigürasyon başlık dosyası, `config.h` kullanmaktan geçmektedir.

`config.h` mantığını kullanabilmemiz için test.c programımızın en başına aşağıdaki üç satırı eklemeliyiz:

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

Burada unutulmaması gereken önemli bir nokta, config.h dosyasının mutlaka ilk olarak include edilmesidir.

Program kaynak kodunu bu şekilde değiştirdikten sonra `configure.ac` dosyasına `AC_CONFIG_HEADERS([config.h])` satırını ve kontrol etmek istediğimiz ilgili makroları eklemeliyiz.

Automake Değişkenleri

Projemiz büyüdükçe kaynak kodların bulunduğu yerler gittikçe karışmaya başlar. Bunları düzenleyebilmek amacıyla daha hiyerarşik dizin yapıları kurarız. Ancak bu defa da her bir dizin için uygun Makefile dosyalarını üretmemiz gerekecektir. Bunun için bu bölümde Makefile.am dosyalarından daha detaylı bir şekilde bahsedeceğiz.

Makefile.am dosyalarının genel biçimi `değişken = değer` şeklindedir. Ancak aynı zamanda, geleneksel Makefile mantığındaki gibi hedef ve soyut kural tanımlamalarını da destekler. Şimdi Makefile.am dosyalarında sıklıkla kullanacağımız satırlara bir bakalım.

- **INCLUDES = -I/usr/local/include -I/usr/custom/include ...:** Obje kodlarını oluştururken derleyiciye include edilen dosyaları hangi dizinlerde araması gerektiğini belirtir. Ayrıca proje kaynak kod yapısı içerisindeki bir dizin seçenek olarak verildi ise alt dizinlerde yer alan kaynak programların hepsinin bu dosyalara erişebilmelerini sağlamak amacıyla tanımlama **INCLUDES = -I$(top_srcdir)/src/libxxx** şeklinde yapılmalıdır. Buradaki `$top_srcdir` değişkeni kaynak kod yapısı içerisindeki en üst dizini tutar.
- **LDFLAGS = -L/usr/local/lib ...:** Derleyici çalıştırılabilir dosyaları üretirken ihtiyaç duyduğu kütüphaneleri hangi dizinlerde araması gerektiğini bu tanımla öğrenecektir.
- **LDADD = test.o ... $(top_builddir)/lib/libfoo.a ... -lfoo ...:** Tüm oluşacak çalıştırılabilir dosyalara linklemek istediğiniz sisteme kurulu olan ve olmayan obje dosyaları burada listelenir. Eğer listelenen obje dosyası sistemde kurulu değilse dosyanı tam adresi verilmelidir ($top_builddir/lib/libfoo.a örneğindeki gibi).
- **EXTRA_DIST = dosya1 dosya2 ...:** Kaynak kod paketinizde bulunmasını istediğiniz her türlü dosyayı burada listeleyebilirsiniz.

- **SUBDIRS = dizin1 dizin2 ...:** Bulunulan dizin için işlem yapmadan önce kuralların çalıştırılması gereken dizinlerdir. make uygulaması bulunulan dizinde işleme başlamadan önce, burada belirtilen dizinlerdeki Makefile kurallarını çalıştırır ve listelenen tüm dizinler için işlemleri bitirdikten sonra bu dizine geri döner.
- **bin_PROGRAMS = test test2 ...:** make komutu çalıştıktan sonra üretilecek ve make install komutuyla belirli bir dizin altına kopyalanacak program adları burada listelenir.
- **lib_LIBRARIES = libfoo1.a libfoo2.a ...:** make komutu çalıştıktan sonra üretilecek ve make install komutuyla belirli bir dizin altına kopyalanacak kütüphane dosyalarının adları burada listelenir.
- **check_PROGRAMS = program1 program2 ... :** make komutunun çalışması esnasında üretilmeyip, sadece make check komutuyla üretilecek, programınızın tümünü veya bir kısmını test edecek uygulamaların çalıştırılabilir dosya adları listelenir.
- **TESTS = program1 program2 ...:** make check komutu sonrasında test amaçlı çalıştırılacak dosya adlarını listeler. Çoğu durumda **TEST = $(check_PROGRAMS)** şeklinde bir tanımlama yapabilirsiniz.
- **include_HEADERS = foo1.h foo2.h ... :** /prefix/include dizini altına kurulmasını istediğiniz başlık dosyalarını burada listelemelisiniz.
- **bin_PROGRAMS** değişkeninde listelediğiniz her bir program için aşağıdaki tanımlamaları da yapmalısınız (program kelimesi yerine programın adını yazmalısınız):
- **program_SOURCES = test.c test1.c test2.c test.h test1.h test2.h ...:** Automake programı burada belirtmiş olduğunuz dosya adları için, C, C++ ve Fortran dillerine özgün soyut Makefile kurallarını oluşturur. Eğer başka bir dil kullanılıyorsa gerekli kuralları siz vermelisiniz.
- **program_LDADD = $(top_builddir)/lib/libfoo.a -lnsl ...:** Burada programınıza linklenmesi gereken kütüphaneleri listelemelisiniz.
- **program_LDFLAGS = -L/dizin1 ...:** program_LDADD ile belirttiğiniz kütüphanelerin hangi dizinlerde aranması gerektiği burada listelenir.
- **program_DEPENDENCIES = dep1 dep2 ... :** Programınızın derlenebilmesi için bağımlı olduğu diğer hedefleri burada listelemelisiniz.

Ek Bölümler

Derleyici Optimizasyonları

Clang ve LLVM

LLVM (Low Level Virtual Machine) projesi 2000 yılında University of Illinois'de dinamik derleme ve performans optimizasyonları araştırmalarına yönelik olarak başlatıldı. C++ ile geliştirilen proje, herhangi bir programlama dili için derleme zamanı, linkleme zamanı, çalışma zamanı ve idle zamanı optimizasyonlarına yönelik yeniden kullanılabilir bileşen ve kütüphaneleri iyi tanımlanmış bir arayüzle sunmaktadır.

2005 yılından itibaren Apple tarafından yapılan ciddi katkıların ve geniş kullanımının yanı sıra son yıllarda Sony firmasının da SDK'larında kullanmaya başlamasıyla birlikte önemli bir ivme kazanmıştır.

GCC'ye oranla *internal* olarak çok daha modern ve iyi bir tasarıma sahip LLVM, gün geçtikçe kullanım oranını artırmakta ve genel olarak daha optimize bir kod üretimi sağlamaktadır

Clang, C, C++, Objective-C ve Objective-C++ dilleri için LLVM önyüzü sağlar. GCC derleyicisi ile uyumludur ve uzun vadede GCC'nin yerine geçmesi hedeflenmektedir. **University of Illinois/NCSA Lisansı** ile açık kaynaklı olarak geliştirilen projede özellikle Apple, Google, ARM, Sony ve Intel firmalarının ciddi katkıları göze çarpmaktadır.

Avantajlar

Clang özellikle IDE üzerinden kullanımlarda GCC'nin sahip olduğu pek çok olumsuz özelliğe dair dizaynından kaynaklanan avantajlar sunmaktadır. Bu nedenle IDE entegrasyonu çok daha iyidir.

GCC ile kıyaslandığında Clang ile derleme işlemleri daha hızlı gerçekleşmektedir. Yaklaşık 10 obje dosyası ve toplam 100.000 satır kod içeren C ile yazılmış bir proje için aynı parametrelerle gcc ile karşılaştırdığımızda aşağıdaki tablo ortaya çıkmaktadır:

Derleyici	Toplam Derleme Zamanı	Oluşan Binary Boyutu
Clang	4 saniye	755 Kb
GCC	7 saniye	953 Kb

Ek olarak derleme sürecinde kullanılan bellek miktarı da gcc'ye oranla daha düşüktür.

Clang kod üzerindeki olası hata veya uyarı gerektiren durumları daha agresif biçimde kontrol eder. Gcc ile hiç uyarı almadan derlediğiniz bir projede clang ile yüzlerce uyarı alabilirsiniz. Gcc'den farklı olarak hataların satır içlerindeki konumunu da detaylı bir şekilde

gösterdiğinden düzeltilmesi daha kolaydır. Aynı zamanda uygulamanızın verimliliğine olumlu katkısı olur. Örnek bir hata mesajı aşağıdaki gibidir:

```
backend.c:194:15: error: address of array 'device-
>required_software' will always evaluate to 'true'
      [-Werror,-Wpointer-bool-conversion]
                if (device->required_software) {
                ~~  ~~~~~~~~^~~~~~~~~~~~~~~~~
```

Avantajlı durumların yanında, üretilen kodun çalışma zamanı performansı açısından LLVM 3.5 öncesi versiyonlarda (2014 öncesi) bariz bir handikap bulunmakta ve gcc ile derlenmiş kodlar daha hızlı çalışmaktaydı. Ancak 3.5 versiyonu ve sonrasında bazı testlerde Clang & LLVM daha iyi performans vermeye başlamıştır. LLVM tarafındaki hızlı geliştirme eforu nedeniyle kısa zamanda tüm testlerde LLVM'in daha iyi sonuçlar üretmesi beklenebilir.

Stallman'ın İtirazı

https://gcc.gnu.org/ml/gcc/2014-01/msg00247.html

İçsel ve Anonim Fonksiyonlar

İçsel ve onların isimsiz halleri olan anonim fonksiyonlar, başka fonksiyonların içinde tanımlanan fonksiyonlardır. Global fonksiyonlara göre daha dar bir bilinirlik alanına sahip olan bu fonksiyonlar genel olarak davranış değiştirmek ve olay dinlemek amacıyla başka fonksiyonlara *callback* olarak geçirilirler.

Birçok dilde yaygın bir kullanıma sahip bu fonksiyonlar C standartlarında yer almamaktadır. Buna karşın içsel fonksiyonlar GNU C eklentisi olarak desteklenmektedir. Anonim fonksiyonlar ise C++11 standartları ile C++ diline dahil edilmiştir.

Daha önce söylediğimiz gibi anonim fonksiyonlar (*anonymous functions*), bir isme sahip içsel fonksiyonların (*nested functions*) bir formudur. Tanımlanmaları ve çağrılmalarında bazı farklılıklar bulunmaktadır.

Biz ilk önce içsel fonksiyonların GNU C eklentisi olarak nasıl oluşturulduğuna, ardından anonim fonksiyonların C++ dilinde nasıl ele alındığına bakacağız. İncelemelerimizde GNU geliştirme araçlarından faydalanacağız.

İçsel Fonksiyonlar

GNU C eklentilerine göre bir içsel fonksiyon aşağıdaki gibi tanımlanabilir.

```
int main() {
    int local = 0;
    void inner() {
        ++local;
    }
}
```

> **Not:** İçsel fonksiyonlar GNU C tarafından desteklenmesine karşın GNU C++ tarafından desteklenmemektedir.

İçsel *inner* fonksiyonu tanımlandığı fonksiyon bloğunda çağrılabildiği gibi adresi başka bir fonksiyona geçirilerek dışarıdan da çağrılabilir. Sırasıyla bu iki çağırma biçimini inceleyeceğiz.

Örnek kodda görüldüğü gibi *inner* fonksiyonu kendi yerel bilinirlik alanında olmayan, dıştaki *main* fonksiyonunun yerel değişkeninin değerini değiştirmektedir. Bu doğal olmayan kullanım şeklinin nasıl gerçekleştirildiğini incelemek için derleyicinin ürettiği sembolik makina koduna bakacağız. İncelemelerimizde 32 bit mimari hedefli sembolik makina kodu kullanacağız.

> **Not:** 64 bitlik bir sistem kullanıyorsanız derleyicinize **m32** anahtarı geçirerek 32 bitlik kod üretmesini sağlayabilirsiniz. 64 bitlik sistemde 32 bitlik kod üretebilmek ve çalıştırabilmek için ekstradan paketlere ihtiyaç duyulacaktır. Ubuntu 14.04.1 LTS için **libc6-i386** ve **lib32stdc++-4.8-dev** paketleri sisteme kurulmuştur.

İçsel Fonksiyonun Tanımlandığı Blok İçinde Çağrılması

```
#include <stdio.h>
int main() {
    int local = 0;
    void inner() {
        ++local;
    }
    inner();
    printf("%d\n", local);
}
```

Yukarıdaki kodu *inner.c* adıyla saklayıp aşağıdaki gibi derleyebilirsiniz.

```
gcc -oinner inner.c -m32 --save-temps
```

--save-temps anahtarı ile derleyicinin ürettiği ara kodlar uygun ad ve uzantılarla dosya sistemine kaydedilmektedir. *inner.c* için derleyici aşağıdaki dosyaları üretecektir.

Dosya Adı	İçerik
inner.i	Önişlemcinin ürettiği kod
inner.s	Derleyicinin ürettiği sembolik makina kodları
inner.o	Gerçek makina kodlarını içeren ELF formatlı amaç kod
inner	Çalıştırılabilir ELF formatlı kod

> **Not:** Komut satırından kullandığımız **gcc** uygulaması aslında derleyici değil, derleme sürecinde gerekli olan uygulamaları uygun sıra ve parametrelerle çağıran bir sürücü (*driver*) programdır. Bir C kodu çalıştırılabilir hale gelene kadar, temel olarak, aşağıdaki aşamalardan geçmektedir.
>
> - Önişlem aşaması
> - Derleyeci tarafından sembolik makina kodlarının üretilmesi
> - Assembler tarafından gerçek makina kodlarının üretilmesi
> - Linker tarafından çalıştırılan dosyanın üretilmesi
>
> Fakat biz burada detaya girmeden bütün bu süreçten derleme süreci olarak bahsedeceğiz.

Örnek kod derlenip çalıştırıldıktan sonra terminal ekranına **1** değerini basacaktır.

Sembolik makina kodlarını incelemeye başlamadan önce, **binutils** paketinden çıkan **nm** ve **readelf** araçları ile amaç dosyadaki sembollere bakalım.

```
$ nm inner.o

00000000 t inner.1826
0000000e T main
         U printf
```

```
$ readelf -s inner.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS inner.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000    14 FUNC    LOCAL  DEFAULT    1 inner.1826
     6: 00000000     0 SECTION LOCAL  DEFAULT    5
     7: 00000000     0 SECTION LOCAL  DEFAULT    7
     8: 00000000     0 SECTION LOCAL  DEFAULT    8
     9: 00000000     0 SECTION LOCAL  DEFAULT    6
    10: 0000000e    51 FUNC    GLOBAL DEFAULT    1 main
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
```

Yazdığımız *inner* fonksiyonunun başlangıç adresinin **inner.1826** sembolüyle temsil edildiğini görmekteyiz. Derleyici, global isim alanındaki aynı isimli bir fonksiyondan ayırmak için, fonksiyon adının sonuna ürettiği bir sayı eklemiş ve bu sembolü dışsal bağlanıma(*external linkage*) kapatmış. Bu aşamadan sonra sembolik makina kodlarını inceleyebiliriz. **.cfi** ile başlayan assembler direktiflerini göz ardı ettiğimizde *main* fonksiyonu için derleyicinin aşağıdaki gibi bir kod ürettiğini görmekteyiz. Üretilen sembolik makina kodunun **AT&T** sözdiziminde olduğuna dikkat ediniz.

> **Not:** Derleyiciye **-masm=intel** anahtarını geçirerek Intel sözdizimine uygun sembolik makina kodu üretmesini sağlayabilirsiniz.

```
main:
 1          pushl   %ebp
 2          movl    %esp, %ebp
 3          andl    $-16, %esp
 4          subl    $32, %esp
 5          movl    $0, %eax
 6          movl    %eax, 28(%esp)
 7          leal    28(%esp), %eax
 8          movl    %eax, %ecx
 9          call    inner.1826
 10         movl    28(%esp), %eax
 11         movl    %eax, 4(%esp)
 12         movl    $.LC0, (%esp)
 13         call    printf
 14         leave
 15         ret
```

Baştaki 4 ve sondaki 2 makina kodu derleyici tarafından yazılan başlangıç(*prologue*) ve bitiş(*epilogue*) kodlarıdır. Başlangıç kodları genel olarak yığının ve yazmaçların hazırlanmasından, bitiş kodları ise yazmaçların eski durumlarına yüklenmesinden sorumludur.

32 bit sistemlerde yığının tepe noktası *esp* yazmacında tutulmakta ve yığın genel olarak büyük adresten küçük adrese doğru genişlemektedir. Başlangıç kodlarına baktığımızda 4 numaralı komut ile *main* fonksiyonu için yığında 32 byte'lık bir alan ayrıldığını görmekteyiz.

Bu aşamada başlangıç ve bitiş kodları arasındaki kodlar asıl ilgilendiğimiz kısmı oluşturmaktadır. İlk önce *main* sonrasında ise *inner* fonksiyonuna ait kodları tek tek inceleyelim. Sembolik makina kodlarını yorumlamak bir miktar aşinalık gerektirmektedir, burada mümkün olduğunca detaya girmeden komutların yaptıklarıyla ilgileneceğiz.

Sembolik makina kodu	İşlevi
`movl $0, %eax`	eax yazmacına 0 değeri yerleştirilmiş
`movl %eax, 28(%esp)`	eax yazmacındaki 0 değeri, yığının başlangıcından itibaren 28 byte uzaklıktaki güvenli bir bölgeye yerleştirilmiş. Bu alan C kodundaki yerel lokal değişkenine karşılık gelmektedir. Otomatik ömürlü yerel değişkenlerin sabit(hardcoded) adreslere sahip olmayıp, yazmaç göreli(register relative) adreslere sahip olduğunu hatırlayınız
`leal 28(%esp), %eax`	yığında yerel değişken için ayrılmış alanın adresi eax yazmacına atanmış
`movl %eax, %ecx`	eax yazmacının değeri yani yerel değişken adresi ecx yazmacına kopyalanmış
`call inner.1826`	inner fonksiyonu çağrılmış

Buraya kadar olan sembolik makina komutlarının C dilindeki karşılığının aşağıdaki gibi olduğunu söyleyebiliriz.

```
int local = 0;
inner();
```

Bundan sonraki bitiş kodlarına kadar olan komutlar yerel değişkenin değerinin *printf* ile bastırılmasına ilişkindir.

Son durumda *ecx* yazmacında yerel değişkenin adresi bulunmakta ve *inner* fonksiyonu çağrılmakta. *inner* fonksiyonuna ait sembolik makina kodları ise aşağıdaki gibidir.

```
inner.1826:
1         pushl   %ebp
2         movl    %esp, %ebp
3         movl    %ecx, %eax
4         movl    (%eax), %edx
5         addl    $1, %edx
6         movl    %edx, (%eax)
7         popl    %ebp
8         ret
```

Başlangıç ve bitiş kodları arasındaki kodları adım adım inceleyelim.

Sembolik makina kodu	İşlevi
`movl %ecx, %eax`	ecx yazmacındaki değer eax yazmacına kopyalanmış. eax yazmacı artık *main* fonksiyonunun yerel değişkeninin adresini tutmaktadır
`movl (%eax), %edx`	eax yazmacının gösterdiği bellek adresindeki değer, yani main fonksiyonunun yerel değişkeninin değeri, edx yazmacına yazılmış
`addl $1, %edx`	edx yazmacının değeri 1 arttırılmış
`movl %edx, (%eax)`	edx yazmacındaki değer eax yazmacının gösterdiği bellek adresine yani *main* fonksiyonunun yerel değişkenine yazılmış

Özetleyecek olursak, içsel fonksiyonun tanımlandığı blok içinde çağrıldığı durumda, derleyici dıştaki fonksiyona(outer function) ait yerel değişkenin adresini *ecx* yazmacında saklamakta ve içsel fonksiyonda *ecx* yazmacını kullanarak kendini çağıran fonksiyonun yerel değişkeninin adresine ulaşmaktadır.

İçsel Fonksiyonun Dışarıdan Çağrılması

İncelememize örnek bir kod üzerinden başlayalım.

```
#include <stdio.h>

typedef void (*PF) ();

void foo(PF f) {
    //diğer işlemler..
    f();
}

int main() {
    int local = 0;
    void inner() {
        ++local;
    }
    foo(inner);
    printf("%d\n", local);
    return 0;
}
```

Örnekte içsel *inner* fonksiyonunun adresi *foo* isimli başka bir fonksiyona geçirilmekte ve bu şekilde dışsal olarak çağırılmaktadır. Kod derlenip çalıştırıldığında yine bir önceki *1* sonucunu üretecektir.

İçsel fonksiyonun tanımlandığı fonksiyon içinde çağrıldığı durumda, dıştaki fonksiyona ait yerel değişken adresinin *ecx* yazmacında saklandığını ve içsel fonksiyonun yerel değişken adresine *ecx* üzerinden ulaştığını hatırlayınız. Buradaki örnekte ise içsel fonksiyon başka bir fonksiyon tarafından çağırılmakta. Bu durumda içsel fonksiyonun adresinin geçirildiği *foo* fonksiyonunun, içsel fonksiyon çağrısından önce, *ecx* yazmacındaki değeri bozmayacağının bir garantisi yoktur. *foo* fonksiyonu kendisine geçirilen adresin içsel bir fonksiyona ait olup olmadığı bilgisine sahip değildir, kaldı ki *foo* fonksiyonu bir kütüphane fonksiyonu olabilir. Bu durumda *ecx* yazmacına yerel değişkenin adresin yazmak yeterli olmayacaktır.

Örnek kod için derleyicinin ürettiği sembolik makina koduna bakarak oluşan durumu inceleyelim. Örnek uygulama bir öncekine benzer şekilde aşağıdaki gibi derlenebilir.

```
gcc -oinner inner.c -m32 --save-temps -fno-stack-protector
```

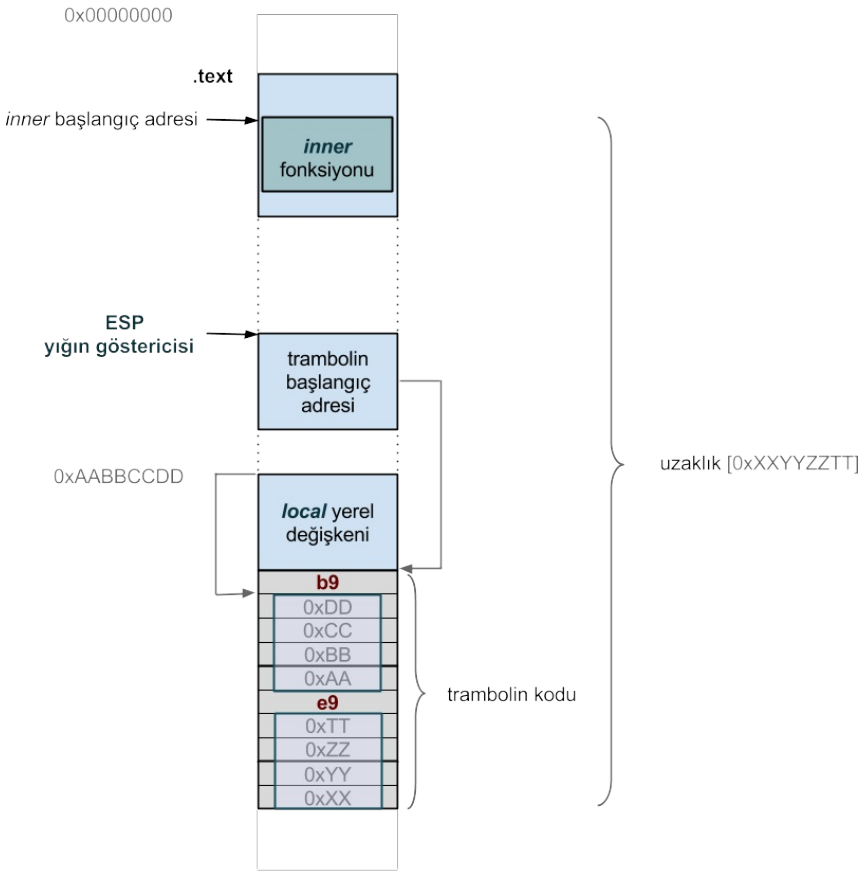
Son argümanı derleyicinin yığın taşmalarını(stack overflow) tespit edebilmek için fazladan yazdığı kodları yazmasını engellemek için ekledik. Derleyici içsel fonksiyon için bir öncekiyle aynı kodu yazmasına karşın, *main* fonksiyonunun kodunun bir hayli değiştiğini görmekteyiz.

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        leal    16(%esp), %eax
        addl    $4, %eax
        leal    16(%esp), %edx
        movb    $-71, (%eax)
        movl    %edx, 1(%eax)
        movb    $-23, 5(%eax)
        movl    $inner.1830, %ecx
        leal    10(%eax), %edx
        subl    %edx, %ecx
        movl    %ecx, %edx
        movl    %edx, 6(%eax)
        movl    $0, %eax
        movl    %eax, 16(%esp)
        leal    16(%esp), %eax
        addl    $4, %eax
        movl    %eax, (%esp)
        call    foo         #foo çağrısı
        movl    16(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

main fonksiyonuna bakıldığında **-71** ve **-23** olmak üzere iki adet negatif değerin *eax* yazmacı referans alınarak belleğe yazıldığı görülmektedir. Negatif sayıların bellekte ikiye tümlenmiş halleriyle tutulduğunu hatırlayınız.

> **Not:** Bir sayının ikiye tümleyenini bulmak için, ikili sayı sisteminde temsil edilen sayının, 1 olan bitleri 0 ve 0 olan bitleri 1 yapılarak önce bire tümleyeni alınır. Sonrasında elde edilen sonuç 1 ile toplanarak ikiye tümleyenine ulaşılır.

-71 ve **-23** sayıları, birer byte ile, bellekte sırasıyla **0xb9** ve **0xe9** şeklinde tutulurlar. *main* için yığında yer ayrılmasından, *foo* fonksiyonu çağrısına kadar olan kodlar işletildiğinde yığının son hali aşağıdaki gibi olacaktır.



Dikkat edilecek olursa, beklentinin tersine, *foo* fonksiyonuna argüman olarak **.text** alanında bulunan içsel fonksiyonun başlangıç adresi geçirilmek yerine, yığında içeriği **0xb9** ile başlayan bölgenin adresi geçirilmiştir.

Derleyicinin *foo* için ürettiği kod ise aşağıdaki gibidir.

```
foo:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %eax
        call    *%eax
        leave
        ret
```

foo fonksiyonu kendisine argüman olarak geçirilen adresi *eax* yazmacına yazmış ve sonrasında o adrese dolaylı çağrı(indirect call) yapmıştır. Bu durumda *foo* fonksiyonu direkt olarak içsel *inner* fonksiyonuna çağrı yapmak yerine yığında güvenli bir bölgeye çağrı

yapmaktadır. Bu noktadan sonra işlemci yığındaki kodları işleyecektir.

> **Not:** Yığındaki bir kodun çalıştırılabilmesi için yığının çalıştırılabilir(*executable stack*) olarak işaretlenmesi gerekmektedir. Bu işlem çoğunlukla bağlayıcı(*linker*) tarafından, sembolik makina kodundaki direktiflere bakılarak yapılır. Bağlayıcı program *ELF* dosya formatı içerisindeki *GNU_STACK* başlık alanına yığının çalıştırılabilir olup olmadığı bilgisini yazar. Yığının çalıştırılabilir olarak işaretlenip işaretlenmediğini sembolik makina komutlarına veya ELF dosyasına bakarak anlayabiliriz. Örneğimiz için aşağıdaki komut çıktılarını inceleyiniz.
>
> ---
>
> $ *cat* inner.s | *grep* -i stack
>
> .section .note.GNU-stack,**"x"**,@progbits
>
> ---
>
> $ *readelf* -lW inner | *grep* -i stack
>
> GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW**E** 0x10
>
> ---
>
> Ayrıca bağlayıcıya yığının çalıştırılabilir olarak işaretlenip işaretlenmeyeceği açık bir şekilde aşağıdaki gibi de geçirilebilir. İçsel fonksiyon içeren örneğimizde, yığın çalıştırılamaz olarak işaretlendiğinde, bellek üzerinde erişim ihlali oluştuğuna dikkat ediniz.
>
> ---
>
> $ *gcc* -oinner inner.c -m32 --save-temps -fno-stack-protector **-z noexecstack**
>
> $./inner
>
> Segmentation fault (core dumped)
>
> ---
>
> **Not:** Yığın üzerinde bir kodun çalıştırılabilmesinin güvenlik açığı oluşturacağına dikkat ediniz.

Yığındaki çalıştırılabilir bu kod **trambolin**(trampoline) olarak adlandırılır. Trambolin kodu içsel fonksiyona dallanmak(*jump*) için kullanılmıştır. Şekilde, yerel değişkenin başlangıç adresi *0xAABBCCDD* ile, içsel fonksiyonun başlangıcıyla trambolin kodunun sonu arasındaki uzaklık ise *0xXXYYZZTT* ile temsil edilmektedir.

Yığında **b9** ile başlayan 10 byte uzunluğundaki blok trambolin kodunu oluşturmaktadır. **b9**, **x86** mimarisinde *ecx* yazmacına kopyalamaya ilişkin gerçek işlem kodu(*opcode*), **e9** ise göreli dallanma(*relative jump*) işlem kodudur. Trambolin kodu *main* fonksiyonuna ait yerel değişkenin adresini *ecx* yazmacına yazmakta ve sonrasında içsel fonksiyona dallanmaktadır. Bu şekilde içsel fonksiyon, güvenli bir şekilde, *ecx* yazmacındaki adresi kullanarak *main* fonksiyonunun yerel değişkenine ulaşabilmektedir.

> **Not:** *x86* mimarisinde, *e9* makina komutu göreli dallanma işleminden sorumludur. *e9* makina kodu, operand olarak, hedef adres ile kendinden sonraki makina komutuna ait adresin farkını almaktadır.
>
> **Not:** Burada neden **ecx** yazmacının kullanıldığı gibi bir soru aklınıza gelebilir. C dili için fiili standart(*de facto standard*) çağırma biçimi(calling convention) olan *cdecl(C declaration)* çağırma biçiminde *eax*, *ecx* ve *edx* yazmaçlarının değerleri çağıran kod tarafından saklanmaktadır(*caller-saved*). *ecx* yazmacının değerinin saklanması çağıran tarafın sorumluluğunda olduğundan, bir içsel fonksiyon çağrısından önce çalışan *trambolin* kodunun, *ecx* yazmacının değerini değiştirmesinde bir mahsur yoktur.

Trambolin kodu *main* için ayrılan yığın alanında bulunmaktadır. Bu durumda, içsel bir fonksiyon tanımı içeren dışsal fonksiyon sonlandığında, yığın alanındaki trambolin koduna ait referans geçerliliğini yitirecektir. Bir fonksiyon sonlandığında ona ait yığın alanı geri verilmektedir.

İçsel fonksiyonun adresinin geçirilerek dışarıdan çağrılma durumunda, çağırma işlemi içsel fonksiyonu sarmalayan fonksiyon sonlanmadan yapılmalıdır. Aksi halde, yığının güvenirliliği kalmadığından, belirsiz davranış(*undefined behaviour*) oluşacaktır.

Daha önce içsel fonksiyonların genel olarak başka kodlara geçirildiğini, bu sayede onların davranışlarını değiştirdiğini veya olan bir olaydan haberdar olmayı sağladığını söylemiştik. Birçok dilde bu amaçlar için kullanılabilmelerine rağmen bir GNU C eklentisi olan içsel fonksiyonlar bir olayı dinlemek üzere asenkron çağrılmaya uygun değillerdir. Buna karşın senkron çağrılmaları durumunda diğer fonksiyonlara güvenle geçirilebilirler. Örnek olarak standart bir C fonksiyonu olan *qsort* verilebilir. *qsort* fonksiyonuna karşılaştırma amaçlı kullanması için, global isim alanında görünmeyen, içsel bir fonksiyon son argüman olarak geçirilebilir.

```
    void qsort(void *base, size_t nmemb, size_t size,
                   int (*compar)(const void *, const void *));
```

İçsel fonksiyonlar ayrıca iç içe çoklu döngülerde istenilen bir noktada tüm döngülerden çıkmak için kullanılabilir. Aşağıdaki örneği inceleyiniz.

```
#include <stdio.h>

int main() {
    int i, j, k;
    void inner() {
        for (i = 0; i < 100; ++i) {
            for (j = 0; j < 100; ++j) {
                for (k = 0; k < 100; ++k) {
                    /*tüm döngülerden tek hamlede çıkılıyor*/
                    if (k > 0) return;
                }
            }
        }
    }
    inner();
    printf("i: %d\tj: %d\tk: %d\n", i, j, k);
}
```

Tüm döngülerden çıkılmak istendiğinde, birden çok *break* deyimi kullanmaksızın tek bir *return* deyimi kullanılabilir.

Son olarak içsel fonksiyonların bir GNU C eklentisi olan *statement expressions* içinde kullanımından bahsedeceğiz. Bu eklenti ile bir bileşik deyim(compound statement), parentezler içine alınarak bir ifade(expression) gibi ele alınabilir. Birleşik deyimlerin küme parentezlerine alınarak oluşturulduğunu hatırlayınız. Bileşik deyimin en sonundaki noktalı virgül ile sonlandırılmış ifade, tüm yapının değeri olarak ele alınır. Genel formu aşağıdaki gibidir.

```
({deyim veya deyimler; dönüş değeri;})
```

Bir fonksiyondan dönen değerin mutlağını alan örnek bir kod aşağıdaki gibidir.

```
#include <stdio.h>

int get_int() {
    return -111;
}

int main() {
    int abs = 0;
    abs = ({    int a; int b;
                a = get_int();
                b = a < 0 ? -a : a;
                b; });
    printf("%d\n", abs);
    return 0;
}
```

İçsel fonksiyonlar da bu yapı içersinde tanımlanabilir. Aşağıdaki örneği inceleyiniz.

```
#include <stdio.h>

typedef void (*PF) (int);

void foo(PF f) {
    f(111);
}

int main() {
    int local = 0;
    PF pf = ({ void inner (int x) { local = x; } inner; });
    foo(pf);
    printf("local: %d\n", local);
    return 0;
}
```

inner isimli içsel fonksiyon, bileşik ifadenin bir parçası olarak tanımlanmakta ve adresi de bu yapının ürettiği sonuç olarak ele alınmaktadır. *inner* fonksiyonunun adresi önce *pf* göstericisine atanmış, ardından *foo* fonksiyonuna geçirilmiş. Aynı işlem tek hamlede aşağıdaki gibi de yapılabilir.

```
#include <stdio.h>

typedef void (*PF) (int);

void foo(PF f) {
    f(111);
}

int main() {
    int local = 0;
    foo(({ void inner (int x) { local = x; } inner; }));
    printf("local: %d\n", local);
    return 0;
}
```

Önişlemci kullanılarak içsel fonksiyonlar görünüşte anonim fonksiyonlarmış gibi kullanılabilirler. Aşağıdaki örnekte içsel fonksiyon açık bir şekilde, isim verilerek, tanımlanmak yerine bu iş için bir makro kullanılmaktadır. İçsel fonksiyona ait geri dönüş türü ve parametre listesiyle fonksiyon gövdesi *lambda* makrosuna argüman olarak geçirilmiş.

Önişlemcinin ürettiği çıktıyı derleyiciye **-E** anahtarı geçirerek görebilirsiniz.

```
#include <stdio.h>

#define lambda(return_type, function_body) \
({ \
      return_type _fn_ function_body \
      _fn_; \
})

typedef void (*PF) (int);

void foo(PF f) {
    f(111);
}

int main() {
    int local = 0;
    lambda(void, (int x) { local = x; }) (111);
    printf("local: %d\n", local);
    return 0;
}
```

Anonim Fonksiyonlar

Anonim fonksiyonlar **C++11** standartları ile dile eklenmiş isimsiz içsel fonksiyonlardır.

Anonim fonksiyonlar, ayrıca **lambda fonksiyonları** olarak da adlandırılır ve *lambda ifadeleri (lambda expression)* kullanılarak tanımlanırlar. Bir lambda ifadesi aşağıdaki forma sahiptir.

```
[capture-list] (parameters) -> return_type {function_body}
```

Örnek bir anonim fonksiyon tanımı ise aşağıdaki gibidir.

```
[] (int x, int y) -> int {return x + y}
```

Anonim fonksiyonlar, kendilerini sarmalayan fonksiyonun yerel değişkenlerine ulaşabilmektedir. Bu özelliğin GNU C eklentisi olarak nasıl gerçeklendiğini bir önceki bölümde incelemiştik. Burada ise benzer özellik *fonksiyon nesneleri (function object)* kullanılarak sağlanmaktadır.

Derleyici *lambda* ifadesini kullanarak yeni bir tür tanımlar ve bu tür için bir *operator()* fonksiyonu yazar. Anonim fonksiyona ilişkin işlemler bu nesne kullanılarak yapılır. Bu isimsiz fonksiyon nesneleri ayrıca **closure** olarak da isimlendirilmektedir.

> **Not:** Bir *operator()* fonksiyonu tanımlayarak, fonksiyon çağrı operatorünü (*function call operator*) yükleyen (*overload*) sınıf örneklerine yani bu türden oluşturulan nesnelere fonksiyon nesneleri (*function object*) veya functor denilmektedir.
>
> Fonksiyon nesneleri, söz dizimsel olarak, görünüşte birer fonksiyon gibi kullanılabilir ve başka fonksiyonlara *callback* olarak geçirilebilir.

Fonksiyon nesneleri sahip oldukları veri elemanları sayesinde *durum (state)* bilgisine sahip fonksiyonlar olarak kullanılabilirler. Durum bilgisinin kullanımını göstermek için aşağıdaki örneği inceleyelim.

```
#include <iostream>

using namespace std;

class Functor {
public:
    Functor(int state) {
        m_member = state;
    }
    bool operator() (int a, int b) {
        return (a - b) > m_member;
    }
private:
    int m_member;
};

int main() {
    /*referans karşılaştırma değeri*/
    int reference = 10;
    Functor f_obj(reference);
    bool result = f_obj(11, 0);
    if (result) {
        cout << "Ok" << endl;
    }
    else {
        cout << "Not Ok" << endl;
    }
    return 0;
}
```

Örneği *functor.cpp* ismiyle saklayıp aşağıdaki gibi derleyebilirsiniz.

```
g++ -ofunctor functor.cpp
```

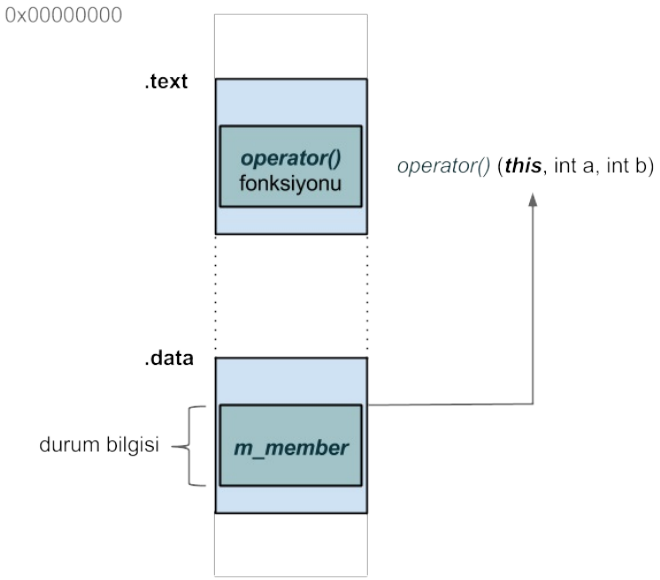
Örnekte temel olarak verilen 2 sayı arasındaki farkın bir referans değerden büyük olup olmadığına bakılmıştır. Yapılan işlemleri daha yakından inceleyelim.

```
1.  Functor f_obj(reference);
2.  bool result = f_obj(11, 0);
```

`1.` satırda sınıfın başlangıç fonksiyonuna (*constructor*) referans değeri geçirilerek *f_obj* nesnesi yapılandırılmış, `2.` satırda ise *f_obj* nesnesi üzerinden sınıfın *operator()* üye fonksiyonu çağrılarak karşılaştırma işlemi yapılmıştır.

Sınıfın başlangıç fonksiyonuna geçirilen referans değeri, sınıfın *m_member* üye değişkeninin değerini değiştirerek bir durum (*state*) bilgisini oluşturmaktadır. Daha sonra yapılacak olan karşılaştırma işleminde bu durum bilgisi kullanılmaktadır. Yukarıdaki örnek için bu durum

bilgisi görsel olarak aşağıdaki gibi gösterilebilir.



m_member sınıfın data belleğindeki görüntüsünü oluşturmaktadır. *operator()* üye fonksiyonuna *m_member* üye değişkeninin adresi ilk argüman olarak geçirilmektedir.

> **Not:** Üye fonksiyonlara ilk argüman olarak üzerinde işlem yapacakları nesnenin adresinin gizli bir biçimde geçirildiğini hatırlayınız. Bu adrese üye fonksiyon içerisinde **this** anahtar sözcüğü ile ulaşmaktayız.

Burada *m_member* değişkeni durum bilgini tutmakta ve *operator()* fonksiyonunun davranışını değiştirmektedir.

Bölümün başında anonim fonksiyonlar için derleyicinin bir tür yazdığını, bu tür için bir *operator()* üye fonksiyonu tanımladığını ve işlemlerin bu türden oluşturulan isimsiz bir nesne üzerinden yapıldığını söylemiştik. Derleyicinin anonim fonksiyonlar için nasıl bir kod yazdığını ve anonim fonksiyonların fonksiyon nesneleriyle olan ilişkisini görmek için aşağıdaki örnek kodu inceleyelim.

```
#include <iostream>

using namespace std;

#ifdef FUNCTOR
class Functor {
public:
#ifdef INLINE
    Functor(int& total) __attribute__((always_inline))
        : m_total(total) {}
#else
    Functor(int& total)
        : m_total(total) {}
#endif

    void operator()(int num) {
        m_total += num;
    }
private:
    int& m_total;
};
#endif

int main()
{
int total = 0;
#ifdef FUNCTOR
    Functor fobj(total);
    fobj(111);
    printf("%d\n", total);
#else
    [&total] (int num) { total += num; } (111);
    printf("%d\n", total);
#endif
    return 0;
}
```

main içinde aynı sonucu üreten, önişlemci direktifleriyle ayrılmış, iki adet kod bloğu bulunmaktadır. Derleme işlemine hangi bloğun girecegine *FUNCTOR* makrosunun varlığına göre karar verilmektedir. *INLINE* makrosunu ne amaçla kullandığımızı daha sonra söyleyeceğiz.

> **Not:** Derleyiciye komut satırında **-D** anahtarı geçirerek bir makro tanımlaması sağlanabilmektedir.

Her iki kod bloğunda da *main* fonksiyonunun yerel değişkeninin değeri başka bir fonksiyon tarafından değiştirilmektedir. İlk olarak bu işlemin bizim yazdığımız bir sınıfa ait fonksiyon nesnesiyle nasıl yapıldığını, sonrasında ise bir anonim fonksiyon kullanılarak nasıl yapıldığını inceleyeceğiz.

Fonksiyon Nesnesinin Açık Kullanımı

Örnek uygulamaya *lambda.cpp* ismini verdikten sonra aşağıdaki gibi derleyebiliriz.

```
g++ -olambda lambda.cpp -m32 -std=c++11 -DFUNCTOR --save-temps
```

> **Not:** *-std* anahtarı ile derleyiciye kullanmasını istediğimiz standardı belirtiyoruz.

lambda.s dosyasını adım adım inceleyerek işe başlayalım. Derleyicinin *main* fonksiyonu için aşağıdaki gibi bir kod ürettiğini görmekteyiz.

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $0, 24(%esp)
        leal    24(%esp), %eax
        movl    %eax, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZN7FunctorC1ERi
        movl    $111, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call    _ZN7FunctorclEi
        movl    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

main için yığın alanından 32 byte'lık yer ayrılmış.

```
subl    $32, %esp
```

Yığının tepe noktasına 24 byte uzaklıktaki alan *total* yerel değişkeni için ayrılmış ve bu alana 0 değeri atanmış.

```
movl    $0, 24(%esp)
```

Yerel değişkenin adresi ilk önce *eax* yazmacına yazılmış ve oradan yığının tepe noktasına 4 byte uzaklıktaki alana kopyalanmış.

```
leal    24(%esp), %eax
movl    %eax, 4(%esp)
```

Yığının tepe noktasına 28 byte uzaklıktaki alanın adresi önce *eax* yazmacına yazılmış ve oradan yığının tepe noktasından başlayan alana yazılmış.

```
leal    28(%esp), %eax
movl    %eax, (%esp)
```

Sonrasında aşağıdaki gibi bir fonksiyon çağrısına ilişkin sembolik makina kodunu görmekteyiz.

```
call    _ZN7FunctorC1ERi
```

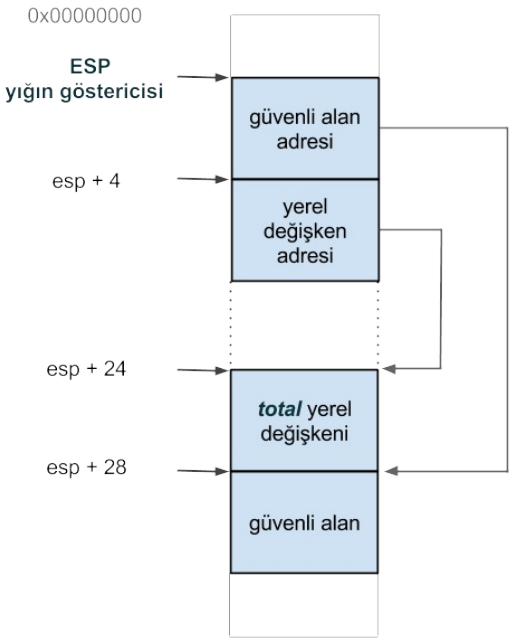
C++ derleyicisinin fonksiyon isimlerini dekore ettiğini hatırlayınız. C++ derleyicisi ürettiği sembolik makina kodunda, kullanıcının tanımladığı isimleri değil, kendi ürettiği aşağı seviyeli *assembler* isimlerini kullanmaktadır.

> **Not:** *binutils* paketinden çıkan **c++filt** aracı ile dekore edilmiş isimler kullanıcının tanımladığı isimlere geri dönüştürülebilir.

c++filt ile çağrı yapılan sembolün hangi fonksiyona ait olduğunu bulabiliriz.

```
$ c++filt _ZN7FunctorC1ERi
Functor::Functor(int&)
```

c++filt çıktısından buradaki çağrının sınıfın başlangıç fonksiyonuna (*constructor*) ait olduğunu görüyoruz. Bu aşamada başlangıç fonksiyonu çağrıldığında yığının durumu aşağıdaki gibidir.



Başlangıç fonksiyonuna ilk argüman olarak yapılandıracağı nesnenin, ikinci argüman olarak ise yerel *total* değişkeninin adresi geçirilmektedir. C++ kodunda, yerel değişken adresinin *referans* yoluyla gizli bir biçimde geçirildiğine dikkat ediniz. Bu durumda yığının tepesinde *güvenli alan adresi* olarak gösterdiğimiz alandaki adres fonksiyon nesnesi için kullanılacak alanı göstermektedir.

> **Not:** gcc derleyicisi, C++ dilinde sınıfın statik olmayan üye fonksiyonları için **thiscall** çağırma biçimini (*calling convention*) kullanmaktadır. *thiscall* çağırma biçimi C dilinde **cdecl** çağırma biçimine oldukça benzemektedir. Çağırılan fonksiyonlara argümanlar yığın yoluyla geçirilmekte ve sağdan sola doğru yığına atılmaktadır. Bu durumda yığının tepesindeki değer çağırılan fonksiyonun en soldaki yani ilk parametresine denk gelmektedir. *thiscall* çağırma biçiminde *cdecl* çağırma biçiminden farklı olarak yığının en tepesi gizli bir *this* göstericisi geçirilmektedir.

Derleyicinin sınıfın başlangıç kodu için ürettiği sembolik makina kodu ise aşağıdaki gibidir.

```
_ZN7FunctorC2ERi:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    12(%ebp), %edx
        movl    %edx, (%eax)
        popl    %ebp
        ret
```

> **Not:** Başlangıç fonksiyonu *main* içinde *_ZN7FunctorC1ERi* adıyla çağırılmasına karşın fonksiyon tanımı *_ZN7FunctorC2ERi* şeklinde yapılmış. Nedeni konumuzun dışında olduğundan yalnız bu detayı söyleyip geçeceğiz.

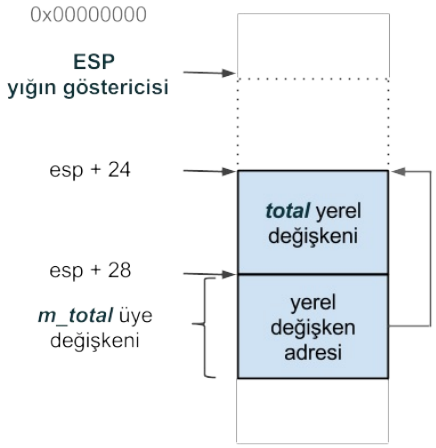
Başlangıç fonksiyonuna geçirilen ilk argüman (*nesnenin adresi*) *eax* yazmacına, ikinci argüman (*yerel değişken adresi*) ise *edx* yazmacına yazılmış.

```
movl    8(%ebp), %eax
movl    12(%ebp), %edx
```

edx yazmacındaki yerel değişken adresi, *eax* yazmacının bellekte gösterdiği alana yazılmış.

```
movl    %edx, (%eax)
```

Bu andan itibaren, fonksiyon nesnesi yapılandırılmış ve *m_total* üye değişkeni *main* fonksiyonunun yerel değişkeninin adresini tutar durumdadır. Başlangıç fonksiyonu döndüğünde yığının durumu aşağıdaki gibidir.



Tekrar *main* fonksiyonuna döndüğümüzde, 111 değerinin ve *m_total* üye değişkeninin adresinin sırasıyla yığına atıldığını görüyoruz. *m_total* değişkeni fonksiyon nesnesinin data belleğinde kapladığı alanı göstermektedir.

```
movl    $111, 4(%esp)
leal    28(%esp), %eax
movl    %eax, (%esp)
```

Sonrasında aşağıdaki fonksiyon çağrısını görmekteyiz.

```
call    _ZN7FunctorclEi
```

Dekore edilmiş sembol adına *c++filt* ile baktığımızda sınıfın *operator()* fonksiyonuna ait olduğunu görmekteyiz.

```
$ c++filt _ZN7FunctorclEi
Functor::operator()(int)
```

operator() fonksiyonuna ait sembolik makina kodu aşağıdaki gibidir.

```
_ZN7FunctorclEi:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    (%eax), %eax
        movl    8(%ebp), %edx
        movl    (%edx), %edx
        movl    (%edx), %ecx
        movl    12(%ebp), %edx
        addl    %ecx, %edx
        movl    %edx, (%eax)
        popl    %ebp
        ret
```

Makina kodlarına yakından bakalım. Fonksiyona geçirilen ilk argüman değeri ilk önce *eax* yazmacına yazılmış, ardından yazmacın gösterdiği adrese karşılık gelen bellek alanındaki değer tekrar *eax* yazmacına kopyalanmış.

```
movl    8(%ebp), %eax
movl    (%eax), %eax
```

Bu işlem C dilinden aşina olduğumuz *pointer dereference* işlemine karşılık gelmektedir. Son durumda *eax* yazmacında *m_total* değişkeninin değeri yani *main* fonksiyonunun yerel değişkeninin (*total*) adresi bulunmaktadır. Sonraki üç komut ile iki defa *dereference* işlemi yapılarak *ecx* yazmacına *total* yerel değişkeninin değeri yazılmış.

```
movl    8(%ebp), %edx
movl    (%edx), %edx
movl    (%edx), %ecx
```

Son durumda, *eax* yazmacında *total* yerel değişkeninin adresi, *ecx* yazmacında ise değeri bulunmaktadır. Daha sonra *operator()* fonksiyonuna açık olarak geçirilen argüman, örneğimiz için 111, ilk önce *edx* yazmacına atılmış, *total* yerel değişkeninin değeriyle toplanarak yerel değişkenin bellek alanına yazılmış.

```
movl    12(%ebp), %edx
addl    %ecx, %edx
movl    %edx, (%eax)
```

main fonksiyonuna geri döndüğümüzde geri kalan komutların yerel değişkenin değerinin standart çıktıya basılmasıyla ilgili olduğunu görmekteyiz.

Özetleyecek olursak, *main* fonksiyonunun yerel değişkeninin adresi bir fonksiyon nesnesinde *durum* bilgisi olarak saklanmış ve *operator()* fonksiyonuyla bu adrese ulaşılarak yerel değişkenin değeri değiştirilmiştir.

Lamdba İfadeleri

Daha önce derleyicinin *lambda* ifadelerini kullanarak bizim için bir tür yazdığından bahsetmiştik. Şimdi bu duruma daha yakından bakalım. Bir önceki konu başlığında incelediğimiz örnek kodu FUNCTOR makrosu tanımlamaksızın aşağıdaki gibi derleyelim.

```
g++ -olambda lambda.cpp -m32 -std=c++11 --save-temps
```

Bu durumda anonim fonksiyon çağrısı derleme sürecine girecektir. Anonim fonksiyonun tanımlandıktan hemen sonra çağrıldığına dikkat ediniz.

```
[&total] (int num) { total += num; } (111);
```

lambda ifadesinin genel formunu yeniden hatırlatarak daha yakından bakalım.

```
[capture-list] (parameters) -> return_type {function_body}
```

Köşeli parantezler boş bırakılabildiği gibi dışsal değişkenler virgül ile ayrılmış bir liste şeklinde geçirilebilir. Bu dışsal değişkenlere değer (*capture by value*) veya adres (*capture by reference*) yoluyla erişilebilir. Örnek bazı kullanımlar aşağıdaki gibi verilebilir.

Kullanım	Açıklama
`[ ]`	Dışsal bir değişkene erişim yoktur
`[&]`	Bütün dışsal değişkenlere adres ile erişilir
`[=]`	Bütün dışsal değişkenlere değer ile erişilir
`[x, &y]`	x değişkenine değerle y değişkenine adres ile erişilir
`[&, x]`	x değişkenine değer ile erişilirken diğer tüm dışsal değişkenlere adres ile erişilir

Burada dışsal değişken ile anonim fonksiyonun içinde tanımlandığı fonksiyona ait yerel değişkenleri kastettiğimizi hatırlatalım.

Köşeli parantezlerden sonra parametre değişkenleri ve fonksiyon gövdesi yazılır. Çoğu durumda geri dönüş değerinin türü derleyici tarafından fonksiyon gövdesine bakılarak tahmin edilmektedir. Buna karşın geri dönüş türü açık bir şekilde de yazılabilir.

> **Not:** Aslında bir *lambda* ifadesinin en genel formu aşağıdaki gibidir.
>
> *[capture-list] (params) mutable(optional) exception attribute -> ret { body }*
>
> Biz burada genel olarak anonim fonksiyonların işleyişiyle ilgilendiğimizden detaya girmeyeceğiz.

Köşeli parantezler içinde geçirdiğimiz dışsal değişkenler fonksiyon gövdesi içinde kullanılabilmektedir. Tekrardan örnek koddaki *lambda* ifadesine baktığımızda *total* yerel değişkenine adres yoluyla erişildiğini ve fonksiyon gövdesinde sol taraf değeri olarak kullanıldığını görmekteyiz.

Derlediğimiz kodu çalıştırdığımızda bir öncekiyle aynı sonucu ürettiğini göreceğiz.

Şimdi derleyicinin anonim fonksiyon için ürettiği kodu bir önceki bölümde incelediğimiz kod ile karşılaştırarak inceleyelim. Bir önceki bölümde bir *functor* sınıfı yazmış ve yerel değişkenin değerini bu sınıftan oluşturduğumuz nesne ile değiştirmiştik.

ANONİM

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $0, 24(%esp)
        leal    24(%esp), %eax
        movl    %eax, 28(%esp)
        movl    $111, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call
_ZZ4mainENKUliE_clEi
        movl    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

FUNCTOR

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $0, 24(%esp)
        leal    24(%esp), %eax
        movl    %eax, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call
_ZN7FunctorC1ERi
        movl    $111, 4(%esp)
        leal    28(%esp), %eax
        movl    %eax, (%esp)
        call
_ZN7FunctorclEi
        movl    24(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Her iki kodda da yığının tepe noktasından itibaren 24 byte uzaklıktaki alan *total* yerel değişkeni için ayrılmış ve 0 ilk değeri verilmiş.

```
movl    $0, 24(%esp)
```

Functor örneğine baktığımızda bundan sonraki 4 sembolik makina komutunun *Functor* sınıfının başlangıç fonksiyonuna geçirilecek argümanlarla ilgili olduğunu görmekteyiz. Yerel değişkenin ve nesne için ayrılmış alanın adresleri sırasıyla yığına atılmış.

```
leal    24(%esp), %eax
movl    %eax, 4(%esp)
leal    28(%esp), %eax
movl    %eax, (%esp)
```

Sonrasında sınıfın başlangıç kodu çağrılarak nesne için ayrılan alana yerel değişkenin adresi yazılmış. Nesne için yığının başından itibaren 28 byte uzaklıktaki alanın ayrıldığına dikkat ediniz. Bu aşamada anonim fonksiyon örneğine baktığımızda aynı işlemin aşağıdaki gibi yapıldığını görmekteyiz.

```
leal    24(%esp), %eax
movl    %eax, 28(%esp)
```

Gerçekten de *functor* örneğinde başlangıç fonksiyonunu *inline* olarak tanımladığımızda, derleyici bir fonksiyon çağrısı yapmak yerine, buradaki kodun aynısını üretecektir. Bunun için bir önceki örnekte derleyiciye **-DINLINE** argümanı geçirerek bu durumu inceleyebilirsiniz.

Sonrasında her iki kod örneğinde de 111 değeri ve yerel değişkenin adresini tutan alanın (*fonksiyon nesnesi*) adresi yığına aktarılmış ve ardından fonksiyon çağrıları yapılmış. *Functor* örneği için yapılan çağrının sınıfın *operator()* üye fonksiyonuna olduğunu hatırlayınız. Anonim fonksiyon örneğinde ise çağrı aşağıdaki gibidir.

```
call    _ZZ4mainENKUliE_clEi
```

c++filt ile sembolün kullanıcı seviyesindeki karşılığına baktığımızda şöyle bir sonuç ürettiğini görmekteyiz.

```
$ c++filt _ZZ4mainENKUliE_clEi
main::{lambda(int)#1}::operator()(int) const
```

Buradan derleyicinin bizim için *const* bir *operator()* fonksiyonu yazdığını ve çağırdığını anlayabiliriz.

main::{lambda(int)#1} bize derleyicinin bizim için yazdığı tür adını göstermektedir. *lambda* ifadesinin *main* fonksiyonu içinde yazıldığını ve *int* türden parametreye sahip olduğunu hatırlayınız. Derleyicinin yazdığı *operator()* fonksiyonuna baktığımızda daha önce bizim yazdığımız *operator()* fonksiyonuyla aynı olduğunu görmekteyiz.

Burada derleyici, bizim yazdığımız *lambda* ifadesinden yola çıkarak, yerel değişkenin adresini tutan bir fonksiyon nesnesi oluşturmuş, ardından *operator()* fonksiyonu içinde bu yerel değişken adresini ve kullanıcının geçirdiği değeri kullanmış. Daha önce de söylediğimiz

gibi burada yerel değişkenin adresini tutan isimsiz nesne **closure** olarak isimlendirilir. İsimsiz fonksiyon nesnesinin otomatik ömürlü olduğuna yani yığında oluşturulduğuna dikkat ediniz.

Bu aşamada anonim fonksiyonların kullanımına birkaç örnek vermek yararlı olacaktır. Anonim fonksiyonlar, şablonlarla (*template*) yoğun bir kullanıma sahip, fonksiyon nesneleri yerine kullanılabilir. Aşağıdaki örneği inceleyiniz.

```
#include <iostream>
#include <vector>

using namespace std;

#ifndef LAMBDA
class AccumulatorFunctor {
public:
    AccumulatorFunctor(int& total)
        : m_total(total) {}

    void operator()(int num) {
        if (num % 2 == 0) {
            m_total += num;
        }
    }
private:
    int& m_total;
};
#endif

template<class InputIt, class UnaryFunction>
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}

int main() {
    vector<int> v = {1, 2, 3, 4, 5, 6};
    int total = 0;
#ifndef LAMBDA
    for_each(v.begin(), v.end(), AccumulatorFunctor(total));
#else
    for_each(v.begin(), v.end(), [&total] (int num) { total += (num % 2 == 0) ? num : 0
; });
#endif
    cout << "total: " << total << endl;
    return 0;
}
```

Örnekte, bir vektördeki çift sayıların toplamının yerel *total* değişkenine yazılması hedeflenmiş. Kod içerisinde aynı işin, hem bir fonksiyon nesnesiyle hem de anonim fonksiyon ile nasıl yapıldığının örneği bulunmaktadır. *lambda* ifadesi kullanılarak oluşturulan isimsiz fonksiyon nesnesi (*closure*) fonksiyon şablonu (*function template*) kullanılarak yazılan *for_each* fonksiyonuna *callback* olarak geçirilmiş. İşlemin anonim fonksiyon ile gerçekleştirilmesi için kodu aşağıdaki gibi derleyebilirsiniz.

```
g++ -Wall -olambda lambda.cpp -m32 --save-temps -std=c++11 -DLAMBDA
```

Ayrıca, anonim fonksiyonlar herhangi bir dışsal değişkenle ilişkilendirilmediği durumda (**[]** içinin boş olduğu durum) gizli bir biçimde (*implicitly*) fonksiyon göstericisine dönüştürülerek *callback* olarak kullanılabilir. Aşağıdaki örneği inceleyiniz.

```
#include <iostream>
#include <vector>

using namespace std;

typedef int (*PF) (int);

void foo(PF f) {
    int result = f(111);
    if (result) {
        cout << "Odd" << endl;
    }
    else {
        cout << "Even" << endl;
    }
}

int main() {
    int total;
    (void)total;
    foo([] (int arg) { return (arg & 1); });
    return 0;
}
```

Örnekte *total* yerel değişkeninin anonim fonksiyon içinde kullanılmadığına dikkat ediniz. Dışarıya geçirilen içsel bir fonksiyon ile yerel bir değişkene ulaşabilmek için GNU C eklentilerince yığında bir *trambolin* kodu yazıldığını hatırlayınız. Daha önce de belirttiğimiz gibi GNU C++ ise bu eklentiyi içermemektedir.

Son olarak kısaca C++ diline eklenen anonim fonksiyon ya da *closure* kavramını, diğer bazı dillerdeki yakın kullanımlarıyla karşılaştıracağız. Buradaki *closure* ifadesi, *Java* diline *Java 8* ile eklenen ve *javascript* dilinde kullanılmakta olan *closure* ile tam olarak aynı anlamı taşımamaktadır. Daha sınırlı bir kullanıma sahiptir.

Daha önce içsel fonksiyonların dışarıdan *asenkron* çağrılmaları durumunda belirsiz davranış oluşacağından bahsetmiştik. Aynı problem burada anonim fonksiyonlar için de geçerlidir. Anonim fonksiyonun içinde tanımlandığı fonksiyon sonlandığında bu foksiyona ait yığın alanı geri verilmekte ve sonraki anonim fonksiyon çağrıları güvenilir olmayan bir alan üzerinde işlem yapmaktadır. Bu durumu, otomatik ömürlü yerel bir değişken adresini dönen bir fonksiyonun geri dönüş değerinin kullanımına benzetebiliriz. *Java* ve *javascript* gibi dillerde ise içinde anonim fonksiyon tanımlanan fonksiyonlara ait yığın alanı bir şekilde saklanmaktadır. C++ dilinde birçok yönden kullanışlı olan bu özellik maalesef şu haliyle *asenkron* olarak gerçekleşen bir olayı dinlemek için uygun değildir.

Derleyicinin anonim fonksiyonları nasıl ele aldığını bilmek, bizim bu özelliğin sınırlarını bilerek daha doğru kullanmamıza yardımcı olacaktır.

FreeTDS ile SqlServer Bağlantısı

Linux tabanlı çözüm kümelerinde her ne kadar pek kullanım alanı bulmasa da, ticari dünyada zaman zaman Microsoft SQL Server veritabanı sunucusuna bağlanmanız ve üzerinde işlem yapmanız gerekebilmektedir.

Java gibi yüksek seviyeli dillerde **ODBC** sürücüleri üzerinden çeşitli çözümler olmakla beraber bu bölümde biz C üzerinden en alt seviyede *native* protokolü kullanarak SqlServer ile haberleşme konusu üzerinde duracağız.

FreeTDS

FreeTDS, **TDS** (Tabular Data Stream) protokolünün **LGPL** lisanslı özgür bir gerçekleştirimidir.

Tabular Data Stream (TDS), bir veritabanı sunucusu ile ona bağlı istemciler arasındaki iletişim ve veri transferini modelleyen, TCP/IP tabanlı bir protokoldür.

Protokol **Sybase Inc.** tarafından geliştirilmiş ve ilk olarak 1984 yılında Sybase SQL Server ürününde kullanılmıştır.

1990 yılında Sybase ve Microsoft firmalarının aralarında yapmış oldukları teknoloji işbirliği anlaşmasını takiben, Microsoft firması da Sybase SQL Server kodunu temel alarak kendi veritabanı sunucusu olan SQL Server ürününü geliştirdi. Bu nedenle Microsoft SQL Server ürününde de TDS protokolü kullanılmaktadır.

Linux platformlarında TDS protokolünün FreeTDS gerçekleştirimi oldukça kararlı durumda olup, C dilinin yanı sıra Php, Ruby, C++ vb. dillerde de alt katmanda FreeTDS kullanan farklı kütüphane alternatifleri mevcuttur.

TDS protokolünün **5.0** versiyonu Sybase tarafından dokümante edilmiş olmakla birlikte, diğer versiyonlarına dair bilgiler genel kullanıma açılmamıştı. 2008 yılında Microsoft, daha önce hayata geçirdiği **Open Specification Promise** doğrultusunda TDS protokol detaylarını genel kullanıma açtı ve bu tarihten sonra kütüphaneler daha güvenilir hale geldi.

> Not: TDS 5.0 versiyonu ile Sybase sunuculara bağlanılabiliyor olmasına karşın bu versiyon Microsoft tarafından desteklenmemektedir. Microsoft SQL Server bağlantıları için protokolün **7.X** versiyonları kullanılmalıdır.

Konsol İstemcisi - Sqsh

Linux sistemlerde kullanılmak üzere, Sybase tarafından geliştirilen **isql** konsol arayüzündeki temel fonksiyonaliye ve ek olarak kullanım kolaylığı açısından bazı yeni fonksiyonlara sahip **sqsh** uygulaması geliştirilmiştir. Uygulamayı paket yöneticinizle aşağıdaki gibi sisteminize kurabilirsiniz:

```
$ sudo apt-get install sqsh
```

Sqsh ile bir sunucuya bağlanırken temel olarak aşağıdaki parametreler kullanılır:

Parametre	Açıklama
-S	Sunucu adresi
-U	Kullanıcı Adı
-P	Parola (parametre olarak girilmez ise konsolda tekrar sorulacaktır)
-D	Veritabanı Adı

Örnek olarak *172.16.2.139* ip adresindeki **example_db** veritabanına *testuser* kullanıcı adı ve *tstpwd123* parolasıyla bağlanalım ve *bolgeler* tablosundaki kayıtları görelim:

```
$ sqsh -S 172.16.2.139 -U testuser -P tstpwd123 -D example_db
sqsh-2.1.7 Copyright (C) 1995-2001 Scott C. Gray
Portions Copyright (C) 2004-2010 Michael Peppler
This is free software with ABSOLUTELY NO WARRANTY
For more information type '\warranty'

1> select * from bolgeler
2> go
 id          isim
 ----------- ---------------------------------------
           1 Akdeniz B�lgesi
           2 Dogu Anadolu B�lgesi
           3 Ege B�lgesi
           4 G�neydogu Anadolu B�lgesi
           5 I� Anadolu B�lgesi
           6 Marmara B�lgesi
           7 Karadeniz B�lgesi
```

Yukarıdaki sonuç kümesine baktığımızda bazı karakterlerin düzgün görüntülenmediğini, bazılarının ise değiştirildiğini görmekteyiz (ğ -> g vb.)

Sorunun çözümü için sunucuya bağlantı kurarken kullanılacak karakter seti kümesi olarak UTF-8'i belirtmemiz gereklidir. Her ne kadar sqsh uygulamasının yardım sayfasında **-J UTF-8** gibi bir parametre geçirmek suretiye bu işlemin yapılabildiği yazsa da kullandığımız

versiyonda (2.1.7) bu şekilde çözüm üretemedik. Karakter problemini, sunucu bazlı genel ayarlamaların yapılmasına imkan veren `freetds.conf` dosyası üzerinden yapacağımız tanımlamalarla çözeceğiz.

freetds.conf

FreeTDS kütüphanesi ile çalışırken öntanımlı olarak `/etc/freetds/freetds.conf` dosyası okunmaktadır.

Bu dosyada genel olarak kütüphanenin davranışını değiştirebilecek tanımlamalar bulunmaktadır. Ayrıca belirli bir SQL sunucu için özel ayarların da buradan yapılmasına imkan verilmektedir.

Dosyanın genel içeriği ve örnek sunucu bazlı tanımlamalar aşağıdaki gibidir:

```
[global]
        # TDS protocol version
;       tds version = 4.2

        # Whether to write a TDSDUMP file for diagnostic purposes
        # (setting this to /tmp is insecure on a multi-user system)
;       dump file = /tmp/freetds.log
;       debug flags = 0xffff

        # Command and connection timeouts
;       timeout = 10
;       connect timeout = 10

        # If you get out-of-memory errors, it may mean that your client
        # is trying to allocate a huge buffer for a TEXT field.
        # Try setting 'text size' to a more reasonable limit
        text size = 64512

# A typical Sybase server
[egServer50]
        host = symachine.domain.com
        port = 5000
        tds version = 5.0

# A typical Microsoft server
[egServer70]
        host = ntmachine.domain.com
        port = 1433
        tds version = 7.0

[mssql]
        host = 172.16.2.139
        port = 1433
        tds version = 7.0
        client charset = UTF-8
```

Yukarıda anlaşılabileceği üzere, tüm sunucuları etkileyecek ayarlar `[global]` bölümü altında yer almakta, aynı zamanda `[mssql]` örneğindeki gibi belirli bir sunucua isim verilerek (DNS ismi olması gerekmiyor), sunucu bazlı ek ayarlamalar yapma şansı da bulunmaktadır.

Örneğimizde **mssql** adında bir sunucu ismi tanımladık ve **client charset** değerini UTF-8 olacak şekilde değiştirdik.

Bu tanım sonrasında hem **sqsh** uygulamasından hem de **freetds** kullanan diğer uygulamalarda, sunucu isim/ip parametresinde **mssql** ismini kullanabilir ve konfigürasyon dosyasında bu bölümde belirtilmiş ayarların aktif olmasını sağlayabiliriz. Bir önceki **select** örneğimizi tekrar edelim:

```
$ sqsh -S mssql -U testuser -P tstpwd123 -D example_db
1> select * from bolgeler
2> go
 id          isim
 ----------- ---------------------------------------
           1 Akdeniz Bölgesi
           2 Doğu Anadolu Bölgesi
           3 Ege Bölgesi
           4 Güneydoğu Anadolu Bölgesi
           5 İç Anadolu Bölgesi
           6 Marmara Bölgesi
           7 Karadeniz Bölgesi
```

.sqshrc

Sqsh ile çalışırken kullanım ortamınızı daha konforlu hale getirmek için ek ayarlamaları ev dizininiz altındaki `.sqshrc` dosyası üzerinden tanımlayabilirsiniz (henüz hiç ayar yapılmadı ise dosyanın oluşturulması gerekecektir)

Örneğin yukarıdaki çıktı formatı yerine öntanımlı MySQL konsol arayüzündekine benzer bir fomrmat kullanılmasını istiyorsanız, **go** komutunu **-m pretty** parametresi ile çalıştırmalısınız. Bu komutu her çalıştırdığımızda parametresini girmek zorunda kalmamak için bir **alias** tanımlayabiliriz.

Aşağıdaki satırı `~/.sqshrc` dosyanıza girin:

```
\alias go='\go -m pretty'
```

Şimdi tekrar bölge listesini sorgulayalım:

```
$ sqsh -S mssql -U testuser -P tstpwd123 -D example_db
1> select * from bolgeler
2> go
+=============+=====================================================+
|          id | isim                                                |
+=============+=====================================================+
|           1 | Akdeniz Bölgesi                                     |
+-------------+-----------------------------------------------------+
|           2 | Doğu Anadolu Bölgesi                                |
+-------------+-----------------------------------------------------+
|           3 | Ege Bölgesi                                         |
+-------------+-----------------------------------------------------+
|           4 | Güneydoğu Anadolu Bölgesi                           |
+-------------+-----------------------------------------------------+
|           5 | İç Anadolu Bölgesi                                  |
+-------------+-----------------------------------------------------+
|           6 | Marmara Bölgesi                                     |
+-------------+-----------------------------------------------------+
|           7 | Karadeniz Bölgesi                                   |
+-------------+-----------------------------------------------------+
```

Diğer bazı kullanışlı örnekler için http://www.sypron.nl/sqsh.html adresine bakabilirsiniz.

Kütüphane Kullanımı

FreeTDS kütüphanesini C uygulamalarında kullanabilmek için aşağıdaki komutla geliştirme paketini sisteminize yükleyebilirsiniz:

```
$ sudo apt-get install freetds-dev
```

Aşağıdaki örnek uygulamayı mssql_connect.c adıyla kaydedip şu şekilde derleyebilirsiniz:

```
$ gcc -o mssql_connect mssql_connect.c -lsybdb
```

Örnek kodumuzu listeleyip önemli yerlerini detaylandırmaya çalışalım:

```c
/* mssql_connect.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sybfront.h>
#include <sybdb.h>
#include "../common/debug.h"

struct mssql_column {
```

```
    char *name;
    char *buffer;
    int type;
    int size;
    int status;
};

void display_usage (const char *name)
{
    printf("Usage: %s SERVER USER PASS DATABASE QUERY\n", name);
}

int database_mssql_errhandler (DBPROCESS * dbproc, int severity, int dberr,
        int oserr, char *dberrstr, char *oserrstr)
{
    (void) dbproc;
    (void) oserr;
    (void) oserrstr;

    if (dberr) {
        errorf("Sqlserver Msg %d, Level %d", dberr, severity);
        errorf("%s", dberrstr);
    } else {
        debugf("%s", dberrstr);
    }
    return INT_CANCEL;
}

int main (int argc, char *argv[])
{
    LOGINREC *login;
    DBPROCESS *dbproc;
    struct mssql_column *columns = NULL;
    struct mssql_column *pcol = NULL;
    int row_code;
    int ncols;
    int nrows;
    int ret;
    int c;

    if (argc != 6) {
        display_usage(argv[0]);
        exit(1);
    }

    const char *server      = argv[1];
    const char *username    = argv[2];
    const char *password    = argv[3];
    const char *database    = argv[4];
    const char *query       = argv[5];

    if (dbinit() == FAIL) {
        errorf("Couldn't init MSSQL library");
```

```
        exit(1);
    }
    /* Maximum 5 seconds for sql login */
    dbsetlogintime(5);

    /* Set error handler callback function */
    dberrhandle(database_mssql_errhandler);

    login = dblogin();
    dbsetluser(login, username);
    dbsetlpwd(login, password);
    if ( (dbproc = dbopen(login, server)) == NULL) {
        errorf("Couldn't open sqlserver db connection");
        exit(1);
    }
    if (dbuse(dbproc, database) == FAIL) {
        errorf("Couldn't change to db: %s", database);
        exit(1);
    }
    if (dbfcmd(dbproc, query)  == FAIL) {
        errorf("Couldn't create sql statement");
        exit(1);
    }
    if (dbsqlexec(dbproc) == FAIL) {
        errorf("Couldn't execute sql query");
        exit(1);
    }

    ncols = dbnumcols(dbproc);
    infof("%d columns found", ncols);

    while ( (ret = dbresults(dbproc)) != NO_MORE_RESULTS) {
        if (ret == FAIL) break;

        if ( (columns = calloc(ncols, sizeof(struct mssql_column))) == NULL) {
            errorf("Couldn't allocate columns");
            break;
        }

        for (pcol = columns; pcol - columns < ncols; pcol++) {
            c = pcol - columns + 1;
            pcol->name = dbcolname(dbproc, c);
            pcol->type = dbcoltype(dbproc, c);
            pcol->size = dbcollen(dbproc, c);
            if (pcol->type != SYBCHAR) {
                pcol->size = dbwillconvert(pcol->type, SYBCHAR);
            }
            debugf("col: %d, type: %d, size: %d, name: %s", c, pcol->type, pcol->size,
pcol->name);
            if ( (pcol->buffer = malloc(pcol->size + 1)) == NULL) {
                errorf("Couldn't allocate space for row buffer");
                break;
            }
```

```
            if (dbbind(dbproc, c, NTBSTRINGBIND, pcol->size + 1, (BYTE*)pcol->buffer)
== FAIL) {
                errorf("Couldn't bind to %s", pcol->name);
                break;
            }
            if (dbnullbind(dbproc, c, &pcol->status) == FAIL) {
                errorf("Couldn't make null bind to %s", pcol->name);
                break;
            }
        }

        while ((row_code = dbnextrow(dbproc)) != NO_MORE_ROWS) {
            switch (row_code) {
            case REG_ROW:
                for (pcol=columns; pcol - columns < ncols; pcol++) {
                    char *buffer = pcol->status == -1 ? "NULL" : pcol->buffer;
                    printf("%s: %s\t", pcol->name, buffer);
                }
                printf("\n");
                break;

            case BUF_FULL:
                errorf("buffer full");
                break;

            case FAIL:
                errorf("failed");
                exit(1);
                break;

            default:
                printf("Data for computeid %d ignored\n", row_code);
            }

        }
    }

    /* Free metadata and data */
    for (pcol = columns; pcol - columns < ncols; pcol++) {
        free(pcol->buffer);
    }
    free(columns);

    /* Get row count if available */
    if ( (nrows = dbcount(dbproc)) > -1) {
        debugf("Affected rows: %d", nrows);
    }

    dbclose(dbproc);
    dbfreebuf(dbproc);
    dbloginfree(login);

    return 0;
```

```
}
```

Kütüphanenin İlklendirilmesi: `dbinit`

FreeTDS kütüphanesinin kullanıldığı uygulamalarda, kütüphane içerisinden herhangi bir fonksiyon çağrılmadan önce, `dbinit()` fonksiyonunun çağrılmış olması şarttır.

`dbinit()` dahili bazı veri yapılarının doldurulmasını ve yerel spesifik tarih vb. format bilgilerini okumak için freetds içerisinden çıkan -varsa- `/etc/freetds/locales.conf` dosyasını okur.

> `locales.conf` dosyasının bu şekilde okunması *deprecated* bir özellik olmuştur. Güncel kütüphane versiyonları sistemin yerel (locale) ayarlarından bu bilgileri temin etmektedir. Ancak gene de `locales.conf` dosyası bulunursa işlenmektedir.

Bu işlemin uygulamanın *main* fonksiyonu içerisinde yapılmasında fayda vardır. Ancak herhangi bir sebeple *dbinit* işleminin bir fonksiyon içerisinden koşullu olarak sonradan yapılması gerekiyorsa, mutlaka statik bir değişkenle kütüphanenin ilklendirme işleminin daha önce yapılıp yapılmadığını tutmanız zorunludur. İlklendirme işleminin tekrar edilmesi, takibi zor hatalara yol açabilmektedir.

Hata İşleme: `dberrhandle`

Kütüphanenin hata ve uyarı durumlarında çağıracağı *calback* fonksiyonunu, `dberrhandle()` fonksiyonu ile belirtilmelidir.

Bu fonksiyonun prototipi aşağıdaki gibidir:

```
typedef int (*EHANDLEFUNC) (DBPROCESS * dbproc, int severity,
            int dberr, int oserr, char *dberrstr, char *oserrstr);
```

Fonksiyon çağrıldığında `dberr` parametresi 0'dan farklı ise, kritik bir veritabanı hatası olduğu anlaşılır.

Bağlantı Kurma ve Veritabanı Seçimi

Bağlantı kurmak için öncelikle kullanıcı adı ve parola bilgileri `LOGINREC` veriyapısı içerisine doldurulmalıdır.

Bunun için öncelikle `LOGINREC` tipinde bir değişken, `dblogin()` fonksiyonu ile ilklendirilir, ardından*dbsetluser* ve *dbsetlpwd* fonksiyonları ile ilgili parametreleri ayarlanır.

Sonraki adımda hazırlanan `LOGINREC` veri yapısı ve sunucu bilgisini (burada IP adresi, DNS üzerinden çözülebilen bir hostname veya `freetds.conf` içerisinde tanımlanmış bir sunucu adı kullanılabilir) parametre olarak alıp, geriye sürecin ilerleyen aşamalarında sürekli kullanılacak olan `DBPROCESS` handle döndürecek olan `dbopen()` fonksiyonu çağrılır.

Herhangi bir sebeple hata alınırsa, ilgili hata mesajının detayı hata işlemeleri için önceden belirlenmiş olan *callback* fonksiyonundan alınabilir.

Bağlantı zaman aşımı süresini kontrol altına almak isterseniz, `dbopen()` fonksiyonu çağırmadan önce `dbsetlogintime(int seconds)` prototipindeki fonksiyonu kullanarak saniye cinsinden bir limit de tanımlayabilirsiniz.

Bağlantı gerçekleştikten sonra `dbuse()` fonksiyonu ile üzerinde çalışılacak olan veritabanı seçimi işlemi yapılmaktadır.

Sorgu Çalıştırma ve Yanıt İşleme

Veritabanı üzerinde çalıştırılacak olan sorgu, öncelikle `dbfcmd()` fonksiyonu ile hazırlanır. Ardından `dbsqlexec()` fonksiyonu ile çalıştırılır.

Sorgu bu şekilde işletildikten sonra geriye dönen değerlerin saklanacağı uygun veri yapıları oluşturulmalıdır. Bunun için uygulama kaynak kodumuzun ilk bölümünde, `struct mssql_column` şeklinde bir yapı tanımladık. Bu yapıyı ihtiyaçlarınız doğrultusunda genişletebilirsiniz.

Tanımladığımız yapıyı, işletmiş olduğumuz sorgunun yanıt kümesindeki her bir sütun ile ilgili veri tipi, uzunluk ve sütun ismi bilgilerini işlemek için kullanacağız.

Örneğimizi geri dönen sütun sayısını önceden bilemeyeceğimiz, her türlü sorgu için çalışacak şekilde hazırladık. Dolayısıyla öncelikle göndermiş olduğumuz sorgu yanıtının kaç sütundan oluştuğunu öğrenmemiz gerekiyor. Bu işlem için `dbnumcols()` fonksiyonunu kullanıyoruz.

Sütun sayısını öğrendikten sonra ilgili bilgileri hazırlamış olduğumuz `struct mssql_column` veri yapısında saklamak üzere bellekte yer ayırıyoruz.

Ardından sorgu yanıtındaki satırları işlemeye geçmeden hemen önce, sütunlarla ilgili sütun ismi, tipi ve veri uzunluğu bilgilerini sırasıyla `dbcolname()` , `dbcoltype()` ve `dbcollen()` fonksiyonlarıyla elde ediyoruz.

Sütun ile ilgili bilgileri bu şekilde öğrendikten sonra, hazırlamış olduğumuz veri yapısında yanıtları saklayacağımız yerleri hazırlıyoruz. Bu noktada kodumuzu kısa tutmak adına, metin dışındaki tiplerin, `dbwillconvert()` fonksiyonuyla metin tabanlı bir formata dönüştürüldüğünde gereken uzunluğu hesaplatıp, metne dönüştüğü zamanki uzunluğu için yetecek kadar bellekte alan açıyoruz. Örnek olarak 4 byte'lık `INT` tipindeki bir sütun için

`dbwillconvert()` sonrası sütun boyutunun **11** olarak geleceğini göreceğiz zira 4 byte'lık bir işaretli INT değerini metne dönüştürüp saklayabilmek için 11 byte uzunluğunda bir alan gereklidir.

Gerçek ortamda gönderdiğiniz sorgularla ilgili bilgi sahibi olacağınızdan, tüm sütunları metin tabanlı dönüşüme zorlamak yerine, `struct mssql_column` veri yapısı içerisindeki genel amaçlı `buffer` değişkenini bir `union` yapısı ile değiştirip, sütun tipine göre `union` içerisinde INT, FLOAT vb. veri tipleri kullanabilir ve metin dönüşümü yapmadan doğrudan bu alanların içerisine yazılmasını sağlayabilirsiniz.

Bellekteki alanlar hazır edildikten sonra her bir sütunu `dbbind()` fonksiyonu ile yanıt setine nasıl bağladığımızı belirtmemiz gerekiyor. Örneğimizde *bind* tipi olarak hep `NTBSTRINGBIND` değerini kullandık. Yukarıdaki ek notumuz doğrultusunda eğer metin dönüşümü uygulamayacaksanız bunun yerine `INTBIND` , `REALBIND` , `BIGINTBIND` vb. diğer veri tipleri için uygun *binding* değerlerini de kullanabilirsiniz.

Her bir sütun için gerekli bind işleminin yanı sıra *NULL* değerler için de `dbnullbind()` fonksiyonuyla bir adet *binding* işleminin daha yapılması gereklidir.

> Not: Sütun ve binding tipleri için sabitler, kütüphane içerisinden çıkan `sybdb.h` dosyası içerisinde yer almaktadır.

Şimdi artık sıra satırları işlemeye geldi. Bunun için `dbnextrows()` fonksiyonu `NO_MORE_ROWS` değeri döndürmediği müddetçe iterasyonla tüm bilgileri alabiliriz.

Örnek uyguladığımızda her bir satırda aldığımız değerleri, sütun ismi ile birlikte ekrana bastırdık.

Yanıt satırlarının işlenmesi tamamlandıktan sonra sistem kaynaklarını serbest bırakıyoruz.

Bağlantının Sonlandırılması

Veritabanı ile ilgili işlemlerimiz tamamlandıysa açık olan bağlantımızı `dbclose()` fonksiyonuyla kapatmamız gerekir.

Son olarak kullandığımız `DBPROCESS` ve `LOGINREC` değişkenlerini de `dbfreebuf()` ve `dbloginfree()` fonksiyonlarıyla da geride artık kalmayacak şekilde temizliyoruz.

Örnek Kullanım

Hazırlamış olduğumuz uygulama ile bir miktar veri içeren `bolgeler` ve `iller` adında 2 tablo üzerinde INNER JOIN sorgusu çalıştıralım:

```
$ ./mssql_connect mssql testuser tstpwd123 example_db \
    "SELECT iller.*, bolgeler.isim AS bolge_adi FROM iller \
    INNER JOIN bolgeler ON iller.bolge_id=bolgeler.id ORDER BY
isim"
info: 5 columns found (main mssql_connect.c:91)
debug: col: 1, type: 56, size: 11, name: id (main
mssql_connect.c:110)
debug: col: 2, type: 47, size: 8, name: plaka (main
mssql_connect.c:110)
debug: col: 3, type: 56, size: 11, name: bolge_id (main
mssql_connect.c:110)
debug: col: 4, type: 47, size: 400, name: isim (main
mssql_connect.c:110)
debug: col: 5, type: 47, size: 200, name: bolge_adi (main
mssql_connect.c:110)
id: 13  plaka: 06  bolge_id: 5  isim: Ankara    bolge_adi: İç
Anadolu Bölgesi
id: 11  plaka: 07  bolge_id: 1  isim: Antalya   bolge_adi:
Akdeniz Bölgesi
id: 1   plaka: 08  bolge_id: 7  isim: Artvin    bolge_adi:
Karadeniz Bölgesi
id: 9   plaka: 16  bolge_id: 6  isim: Bursa     bolge_adi:
Marmara Bölgesi
id: 4   plaka: 28  bolge_id: 7  isim: Giresun   bolge_adi:
Karadeniz Bölgesi
id: 7   plaka: 34  bolge_id: 6  isim: İstanbul  bolge_adi:
Marmara Bölgesi
id: 2   plaka: 53  bolge_id: 7  isim: Rize      bolge_adi:
Karadeniz Bölgesi
id: 6   plaka: 55  bolge_id: 7  isim: Samsun    bolge_adi:
Karadeniz Bölgesi
id: 15  plaka: 58  bolge_id: 5  isim: Sivas     bolge_adi: İç
Anadolu Bölgesi
id: 3   plaka: 61  bolge_id: 7  isim: Trabzon   bolge_adi:
Karadeniz Bölgesi
id: 8   plaka: 77  bolge_id: 6  isim: Yalova    bolge_adi:
Marmara Bölgesi
debug: Affected rows: 15 (main mssql_connect.c:159)
```

> Not: `debug.h` dosyasını Kaynak Dosyalar bölümünden edinebilirsiniz.

Kaynak Dosyalar

debug.h

```
#include <stdio.h>

#define COLOR_DEFAULT            "\033[0m"
#define COLOR_BLACK               "\033[0;30m"
#define COLOR_BLACK_BOLD        "\033[1;30m"
#define COLOR_RED                "\033[0;31m"
#define COLOR_RED_BOLD            "\033[1;31m"
#define COLOR_GREEN               "\033[0;32m"
#define COLOR_GREEN_BOLD        "\033[1;32m"
#define COLOR_BROWN               "\033[0;33m"
#define COLOR_BROWN_BOLD        "\033[1;33m"
#define COLOR_BLUE                "\033[0;34m"
#define COLOR_BLUE_BOLD           "\033[1:34m"
#define COLOR_MAGENTA            "\033[0;35m"
#define COLOR_MAGENTA_BOLD        "\033[1;35m"
#define COLOR_CYAN                "\033[0;36m"
#define COLOR_CYAN_BOLD           "\033[1;36m"
#define COLOR_LIGHTGRAY           "\033[0;37m"
#define COLOR_LIGHTGRAY_BOLD    "\033[1;37m"

#define errorf(a...) { \
        fprintf(stderr, "%serror:%s ", COLOR_RED_BOLD, COLOR_DEFAULT); \
        fprintf(stderr, a); \
        fprintf(stderr, " %s(%s %s:%d)%s\n", COLOR_BLACK_BOLD, __FUNCTION__, __FILE__,
 __LINE__, COLOR_DEFAULT); \
        fflush(stderr); \
}

#define infof(a...) { \
        fprintf(stderr, "%sinfo:%s ", COLOR_CYAN, COLOR_DEFAULT); \
        fprintf(stderr, a); \
        fprintf(stderr, " %s(%s %s:%d)%s\n", COLOR_BLACK_BOLD, __FUNCTION__, __FILE__,
 __LINE__, COLOR_DEFAULT); \
        fflush(stderr); \
}

#define debugf(a...) { \
        fprintf(stderr, "%sdebug:%s ", COLOR_MAGENTA, COLOR_DEFAULT); \
        fprintf(stderr, a); \
        fprintf(stderr, " %s(%s %s:%d)%s\n", COLOR_BLACK_BOLD, __FUNCTION__, __FILE__,
 __LINE__, COLOR_DEFAULT); \
        fflush(stderr); \
}
```