# UNIX-Linux Sistem Programlama Kursu Notları
# C ve Sistem Programcıları Derneği

# Kaan ASLAN

# Güncelleme Tarihi: 11/09/2019

# (Notlar Baştan Yazılmaktadır)



---

**Tarihsel Arka Plan ve Temel Kavramlar**

Kursumuzun başında tarihsel arka plan ve bazı temel kavramlar üzerinde duracağız.

**Sistem Programlama Nedir?**

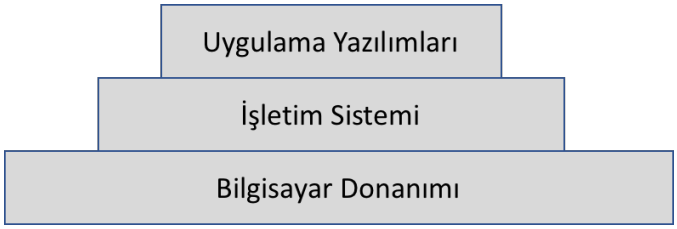
Bilgisayar donanımıyla arayüz oluşturan, uygulama programlarına çeşitli bakımlardan hizmet veren programlara "sistem programları", programlamanın bunlarla ilgili alanına da "sistem programlama" denilmektedir. Sistem programlarma faaliyetleri aşağı seviyeli olma eğilimindedir. Bunları yazmak için önemli ölçüde teorik bilgiye ve uygulama becerisine gereksinim duyulmaktadır. Sistem programlama yazılımın ağır sanayisi niteliğindedir. IT sektöründeki Microsoft, Apple gibi pek çok büyük firma geliştirdikleri sistem programlarıyla bu hale gelmişlerdir. Tipik sistem programlarından bazıları şunlardır:

- İşletim Sistemleri
- Derleyiciler ve yorumlayıcılar
- Editörler
- Debug Programları
- Virüs ve Antivirüs yazılımları
- Haberleşme programları
- Gömülü sistem programları
- Çevre birimlerinin ve diğer donanımsal aygıtların programlanması ve aygıt sürücüleri
- Veritabanı motorları
- Sanallaştırma yazılımları ve emülatör yazılımları
- Oyun motorları

Sistem programlama faaliyetleri için en çok kullanılan programlama dilleri C, C++ ve Sembolik Makina Dilleridir (Assembly Langauges). Tabii bazı sistem programları C#, Java ve hatta Python gibi dillerle de yazılabilmektedir. Fakat C/C++ dillerinin asıl uzmanlık alanı sistem programlamadır.

**İşletim Sistemi Nedir?**

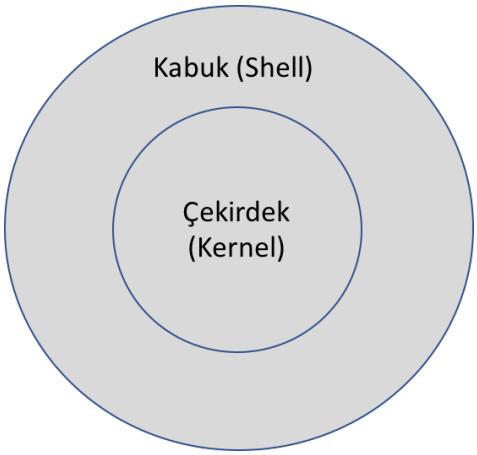
İşletim sistemleri bilgisayar donanımının kaynaklarını yöneten, bilgisayar donanımı ile kullanıcı arasında arayüz oluşturan temel bir sistem programıdır. Pek çok teorisyen işletim sistemini bir kaynak yöneticisi (resource manager) olarak tanımlamıştır.




İşletim sistemlerinin yönettiği kaynakların en önemlileri şunlardır:

- CPU: Hangi program ne zaman, ne kadar süre için CPU'ya atanacak?
- RAM: Programlar RAM'in neresine yüklenecek?
- Disk: Dosyaların parçaları diskte hangi sektörlerde ve nasıl tutulacak?
- Çevre birimleri (klavye, fare, ağ, yazıcı vs.): Fare, klavye gibi birimler nasıl yönetilecek? Network işlemleri nasıl yapılacak?

İşletim sistemleri yapı olarak iki kısımdan oluşmaktadır: Çekirdek (kernel) ve kabuk (shell). Çekirdek işletim sisteminin donanımını kontrol eden ve kaynakları yöneten kabuk ise kullanıcı ile arayüz oluşturan kısmıdır. Örneğin UNIX/Linux sistemlerinde komut satırı, Gnome, KDE gibi pencere yöneticileri, Windows'taki masasüstü (Explorer) kabuk kısımdır. Kullanıcılar genellikle çekirdeğik görmezler.




İşletim sistemleri çeşitli biçimlerde sınıflandırılabilmektedir.

Proses Yönetimine Göre: Aynı anda tek bir programı çalıştıran işletim sistemlerine "tek prosesli (single processin)", aynı anda birden fazla programı çalıştırabilen işletim sistemlerine "çok prosesli (multiprocessing) işletim sistemleri denilmektedir. Örneğin DOS işletim sistemi tek prosesli bir sistemdi. Bir programı çalıştırırdık sonra o sonlanınca başkasını çalıştırabilirdik. Halbuki Windows, UNIX/Linux, Mac OS X çok prosesli işletim sistemleridir.

Kullanıcı Sayısına Göre: Birden fazla farklı kullanıcının çalışabildiği sistemlere "çok kullanıcılı (multiuser)", tek bir kullanıcının çalışabildiği sistemlere "tek kullanıcılı (singleuser)" sistemler denilmektedir. Genellikle çok prosesli işletim sistemleri aynı zamanda çok kullanıcılı sistemlerdir. Birden fazla kullanıcının söz konusu olduğu sistemlerde kullanıcıların yetkilerinin yarlanması, birbirlerinin alanlarına erişememesi, sistem kaynaklarını belli oranlarda bölüşmesi gerekebilmektedir. Örneğin DOS tek kullanıcılı bir sistemdi. Halbuki Windows, UNIX/Linux ve Mac OS X sistemleri çok kullanıcılı sistemlerdir.

Çekirdek Yapısına Göre: İşletim sistemleri çeekirdek yapısına göre "monolithic kernel" ve "microkernel" olmak üzere ikiye ayrılmaktadır. Monolithic çekirdeklerde işletim sisteminin büyük kısmı çekirdek modunda çalışır. Microkernel sisteminde ise çekirdek modunda çalışan kısım minimize edilmeye çalışılmıştır. Aslında monolithic ve microkernel çekirdekleri bir spektrum olarak düşünebiliriz. (Örneğin spektrumun 0-100 arasında olabilir ve işletim sistemleri bu spektrum arasında herhangi bir noktada olabilir.)

Dışsal Olaylarla Yanıt Verebilme Özelliğine Göre: İşletim sistemleri dışsal olaylara yanıt verme bakımından gerçek zamanlı olan(real time) ve gerçek zamanlı olmayan (nonrealtime) sistemler olmak üzere ikiye ayrılabilir. Dışsal olaylara hızlı bir biçimde yanıt verebilecek çekirdek yapısına sahip olan işletim sistemlerine "gerçek zamanlı (realtime) işletim sistemleri denilmektedir. Gerçek zamanlı işletim sistemleri de kendş aralarında "hard realtime" ve "soft realtime" işletim sistemleri olmak üzere ikiye ayrılabilmektedir. Hard realtime sistemlerde dışsal olaylara yanıt verme bakımından çok güvenilir olma iddiasındadır. Soft realtime sistemler ise bu konuda daha gevşektir.

Dağıtıklık Durumuna Göre: İşletim sistemleri dağıtıklık durumuma göre "dağıtık olan (distributed)" ve "dağıtık olmayan (nonedistributed)" sistemler biçiminde ikiye ayrılabilmektedir. Dağıtık işletim sistemlerinde sistem birden fazla bilgisayardan oluşan tek bir sistem gibi davranmaktadır. Örneğin 10 tane makineyi tek bir sistem olarak düşünebilirsiniz. Bu durumda bu bilgisayarların kaynakları (örneğin diskleri ve CPU'ları) bu 10 makine tarafından paylaşılmaktadır. Windows, UNIX/Linux ve Mac OS X dağıtık işletim sistemleri değildir. Ancak bu sistemlerde dağıtık uygulamalar yapılabilmektedir.

Donanım Özelliğine Göre: Masaüstü, mobil, embedded Neredeyse her yaygın masaüstü işletim sisteminin bir mobil versiyonu da oluşturulmuştur. Windows'un mobil versiyonuna genel olarak Windows CE denilmektedir. Windows CE'nin akıllı telefonlar ve tabletler için özelleştirilmiş biçimine ise Windows Mobile denilmektedir. IOS (Iphone Operating System) Apple firmasının (yani Mac OS X'lerin) mobil işletim sistemidir. Android bir çeşit mobil Linux sistemi olarak değerlendirilebilir. Android projesinde Linux alınmış, biraz özelleştirilmiş, bazı parçaları atılmış, buna bir arayüz giydirilmiş ve akıllı telefonlara uygun hale getirilmiştir. Nokia eskiden Symbian sistemlerinde büyük bir pazar payına sahipti. Ancak bu firma akıllı telefon geçişini çok iyi yönetemedi. MeeGo ve Maemo sistemlerini denedi. Sonra büyük bölümünü Microsoft'a satmak zorunda kaldı. Bugün Nokia artık akıllı telefon olarak Windows Mobile sistemlerini üretmektedir. Ancak yavaş yavaş Android telefon üretimine de başlamıştır.

Bugün için (2018 Aralık) en yaygın kullanılan mobile işletim sistemi Android'tir (%72'den fazla). Bunu IOS (%20 civarı), onu da Windows Mobil izlemektedir (%0.5 civarı).

Mobil işletim sistemlerinin doğal programlama ortamları sistemden sisteme değişebilmektedir. Android'in doğal programla ortamı Java, IOS'un Objective-C ve Swift, Windows Mobile'in ise C#'tır. Tabii bu ortamlarda bu dillerin dışında başka dillerle de uygulamalar geliştirilebilmektedir.

Kaynak Kod Lisansına Göre: Kaynak kod lisansına göre işletim sistemlerini kabaca "açık kaynak kodlu (open source)" ve mülkiyete bağlı (proprieatary) olmak üzere ikiye ayırabiliriz. Açık kaynak kodlu işletim sistemleri değişik açık kaynak kod lisanslarına sahip olabilmektedir. Bunların kaynak kodları indirilip üzerinde değişiklikler yapılabilmektedir. Örneğin Windows işletim sistemi mülkiyete sahiptir. Oysa Linux, BSD sistemleri, Solaris, Android gibi sistemler açık kaynak kodludur. Mac OS X sistemlerinin çekirdeği açık diğer kısımları (örnein kabuk kısmı) kapalıdır.

Kaynak Kodun Özgünlüğüne Göre: Bazı işletim sistemleri bazı işletim sistemlerinin kodları alınıp değiştirilerek oluşturulmuştur (örneğin Android ve Mac OS X'te olduğu gibi). Bazı işletim sistemlerinin kodları ise sıfırdan yazılmıştır. Kodları sıfırdan yazılan yani orijinal kod temeline dayanan işletim sistemlerinden bazıları şunlardır:

- AT&T UNIX
- DOS
- Windows
- Linux
- BSD (belli bir yıldan sonra)
- Solaris
- XENIX
- VMS

GUI Çalışma Desteğine Göre: Bazı işletim sistemleri GUI çalışma modelini doğrudan desteklerken bazıları desteklememektedir. Örneğin Windows sistemleri çeekirdekle entegre edilmiş bir GUI çalışma modeli sunmaktadır. UNIX/Linux sistemleri de XWindows (ya da X11) denilen bir katmanla benzer bir model sunmaktadır. Fakat örneğin DOS işletim sisteminin böyle bir doğal GUI desteği yoktu.

Ağ Üzerinde Hizmet Alıp Verme Rollerine Göre: İşletim sistemlerini ağ altında hizmet alıp verme rollarine göre client ve server biçiminde de iki gruba ayırabiliriz. Bazı işletim sistemlerinin istemci versiyonları birbirlerindem ayrılmıştır. Bazılarında ise bu ayrım yapılmamıştır. Örneğin Windows 7, 8, 10 sistemleri bu bakımdan istemci (client) sistemleridir. Halbuki Windows Server 2012, 2016, 2019 sunucu sistemleri oalrak piyasaya sürülmüştür. Eskiden Mac OS X'in istemci ve sunucu versyonalrı farklıydı. Fakat MAc OS X 10.7 (Lion) ile birlikte istemci ve sunucu versiyonları birleştirildi. Linux dağıtımlarının çoğu da hem istemci hem de sunucu olarak kullanılabilmektedir. Ancak bazı dağıtımların ise istemci ve sunucu versyonları farklıdır. Pekiyi işletim sistemlerinin istemci ve sunucu versiyonları arasındaki farklılıklar nelerdir? Kabaca iki tür farklılığın olduğunu söyleyebiliriz. Birincisi çekirdekle ilgili farklılıklar. Genellikle sunucu sistemlerinde çizelgeleyici alt sistemde istemci sistemlerine göre farklılıklar bulunmaktadır. İkincisi işletim sistemlerinin sunucu versiyonları hazır bazı sunucu programlarını da içermektedir.

## Bilgisayar Donanımlarının Tarihsel Gelişimi

Elektronik düzeyde bugün kullandığımız bilgisayarlara benzer ilk aygıtlar 1940'lı yıllarda geliştirilmeye başlanmıştır. Ondan önce hesaplama işlemlerini yapmak için pek çok mekanik aygıt üzerinde çalışılmıştır. Bunların bazıları kısmen başarılı olmuş ve belli bir süre kullanılmıştır. Mekanik bilgisayarlarlardaki en önemli girişim Charles Babbage tarafından yapılan "Analytical Engine" ve "Diffrenece Engine" aygıtlarıdır. "Analitical Engine" tam olarak bitirilememiştir. Fakat bunlar pek çok çalışmaya ilham kaynağı olmuştur. Hatta bir dönem Babbage'in asistanlığını yapan Ada Lovelace bu "Analytical Engine" üzerindeki çalışmalarından dolayı dünyanın ilk programcısı kabul edilmektedir. Şöyle ki: Rivayete göre Babbage Ada'dan "Analytical Engine" için Bernolli sayılarının bulunmasını sağlayan bir yönerge yazmasını istemiştir. Ada'nın yazdığı bu yönergeler dünyanın ilk programı kabul edilmektedir. (Gerçi bu yönergelerin bizzat Babbage'in kendisi tarafından yazılmış olduğu neredeyse ispatlanımış olsa bile böyle atıf vardır.) Daha sonra 1800'lü yılların oratalarından itibaren elektronikte hızlı bir ilerleme yaşanmıştır. Bool cebri ortaya atılmış, çeşitli devre elemanları kullanılmaya başlanmış ve mantık devreleri üzerinde çalışmalar başlatılmıştır. 1900'lü yılların başlarında artık yavaş yavaş elektromekanik bilgisayar fikri belirmeye başlamıştır. 1930'lu yıllarda Alan Turing konuya matematiksel açıdan yaklaşmış ve bugünkü bilgisayar benzeri bir makinenin hangi matematik problemleri çözebileceği üzerine kafa yormuştur. Turing bir şerit üzerinde ilerleyen bir kafadan oluşan ve ismine "Turing Makinesi" denilen soyut makine tanımlamıştır ve bu makinenin neler yapabileceği üzerinde kafa yormuştur. ACM Turing'in anısına bilgisayarın Nobel ödülü gibi kabul edilen Turing ödelleri vermektedir.

Dünyanın ilk elektronik bilgisayarının hangisi olduğu konusunda bir fikir birliği yoktur. Bazıları Konrad Zuse'nin 1941'de yaptığı Z3 bilgisayarını ilk bilgisayar olarak kabul ederken bazıları 1944'te yapılan Harward Mark 1 bilgisayarını bazıları da 1945'te yapılan ENIAC'ı ilk bilgisayar olarak kabul etmektedir.

Modern bilgisayar tarihi üç döneme ayrılarak incelenebilir:

1) Transistör öncesi dönem (1940-1950'lerin ortalarına kadar)
2) Transistör dönemi (1950'lerin ortalarından 1970'lerin ortalarına kadar)
3) Entegre devre dönemi (1970'lerin ortalarından günmüze kadarki dönem)

İlk bilgisayarlar vakum tüplerle yapılmıştı. Vakum tüpler hem büyük yer kaplıyordu hem de çok ısınıyordu dolayısıyla da çok güç harcıyordu. Ayrıca güvenilir elemanlar değildi. Bu nedenle bu devirdeki bilgisayarlar bir salon büyüklüğündeydi.

Transistör i1947 yılında John Bardeen, William Schockley ve Walter Brattain tarafından Bell Lab'ta icad edildi. Fakat 1950'li yılların oratalarına doğru kullanıma girdi. İlk transistörlü radyo ve ilk transistörlü bilgisayar (TRADIC) 1954 yılında yapıldı. Transistörler 1950'li yıllarda yavaş yavaş bilgisayar devrelerine de girmeye başladı. Bu sayede bilgisayar devreleri küçüldü ve kuvvetlendi. O zamanların en önemli firmaları IBM, Honeywell, DEC gibi firmalardı.

Entegre devreye benzer ilk çalışma aslında ilk olarak 1949 yılında Alman mühendis Werner Jacobi tarafından yapıldı. Ancak entegre devre fikri 1952 yılında İngiliz Geoffrey Dummer tarafından ortaya atıldı. Fakat gerçek anlamda ilk gerçekleştirimi 1958 yılında Texas Instruments şirketi çalışanı Jack Kilby tarafından yapıldı. Kilby'den habersiz olarak yaklaşık altı ay sonra benzer entegre devre gerçekleştirimi Fairchild Semiconductor firmasında Robert Noyce tarafından da yapıldı. Kilby ile Noyce patent konusunda mahkemelik olmuşlarsa da sonra anlaşma sağlanmış ve her iki kişi adına patentleme yapılmıştır. Robert Noyce aslında transistörü bulan ekipteki William Schockley'nin yanında çalışıyordu. Bu ekipte Gordon Moore da vardı. Schockley'nin yönetiminden memnun olmayan bu ekip Fairchild Semiconductor şirketine

geçmiştir. Noyce şirketin genel müdürü, Moore da arge müdürü olmuştur. Daha sonra 1968 yılında Robert Noyce, Gordon Moore Fairchild Semiconductor firmasından ayrılarak Intel'i kurdu. İkili Intel'i kurduktan sonra şirkete Fairchild Semiconductor'dan Andrew S. Grove da yanlarına aldı. Dünyanın entegre devre olarak üretilen ilk mikroişlemcisi Intel'in 8080'i kabul edilmektedir. Intel daha önce 4004, 8008 gibi entegre devreler yaptıysa da bunlar tam bir mikroişlemci olarak kabul edilmemektedir. Entegre devreler kullanılarak mikroişlemciler yapılmaya başlanınca artık bilgisayar dünyası yeni bir döneme girmiş oldu.

Intel 8080'i tasarladığında bundan bir kişisel bilgisayar yapılabileceği onların aklına gelmemiştir. Kişisel bilgisayar fikri Ed Roberts isimli bir girişimci tarafından ortaya atıldı. Ed Roberts 8080'i kullanarak Altair isimli ilk kişsel bilgisayarı yaptı ve "Popular Electronics" isimli dergiye kapak oldu. Altair makine dilinde kodlanıyordu. Roberts buna Basic derleyicisi yazacak kişi aradı ve Popular Electronics dergisine ilan verdi. İlana o zaman Harward'ta öğrenci olan Bill Gates ve Paul Allen başvurdular. Böylece Altair daha sonra Basic ile piyasaya sürüldü. Gates ve Allen okuldan ayrıldılar ve 1975 yılında Microsoft firmasını kurdular. (O zamanlar bu yeni kişisel bilgisayarlara mikrobilgisayarlar denilmekteydi). Amerika'da bu süreç içerisinde bilgisayar kulüpleri kuruldu ve pek çok kişi kendi kişisel bilgisayarlarını yapmaya çalıştı. Steve Jobs ve Steve Wozniak Apple'ı 1976 yılında böyle bir süreçte kurmuştur.

IBM kişsel bilgisayar konusunu hafife aldı. Fakat yine de bir ekip kurarak bugün kullandığımız PC'lerin donanımını tasarlamıştır. Ancak IBM küçük iş olduğu gerekçesiyle bunlara işletim sistemini kendisi yazmadı, taşeron bir firmaya yazdırmak istedi. Bu süreç içerisinde Microsoft IBM ile anlaşarak DOS işletim sistemini geliştirdi. İlk PC'lerin donanımı IBM tarafından, yazılımı Microsoft tarafından yapılmıştır. Microsoft IBM'le iyi bir anlaşma yaptı. IBM uzağı göremedi. Anlaşmaya göre başkalarına DOS'un satışını tamamen Microsoft yapacaktı. IBM ikinci bir hata olarak PC için donanım patentlerini almayı ihmal etti. Bunun sonucunda pek çok firma IBM uyumlu daha ucuz PC'ler yaptılar. Fakat bunların hepsi işletim sistemini Microsoft'tan satın alıyordu. Böylece Microsoft 80'li yıllarda çok büyüdü.

## İşletim Sistemlerinin Doğuşu ve İlk İşletim Sistemleri

1940'lı yıllarda ilk elektronik bilgisayarlar yapıldığında henüz bir işletim sistemi kavramı yoktu. Bu bilgisayarlara program yazacak olanlar işletim sistemi faaliyetlerini de kendileri yapmak zorunda kalıyordu. (Yani şimdi mikrodenetleyicilere kod yazanlarda olduğu gibi.) Transistör bulunduktan sonra 1950'li yıllarda artık elektronik bilgisayarlar yavaş yavaş transistörlerle yapılmaya başlandı. Transistörlerin ortaya çıkması hep bilgisayarların kapasitelerini ve güvenilirliklerini artırmış, hem de güç harcamalarını düşürmüştür.

1950'li yıllarda IBM gibi pek çok bilgisayar üreticisi firma yalnızca donanım satıyordu. İşletim sistemi gibi programları yazmak kullanıcıların yapması gereken bir işti. Böylece donanımı satın alan her kurum işletim sistemine benzeyen programları da kendisi yazıyordu. Bu anlamda standart bir işletim sistemi yoktu. Bugünkü anlamda ilk işletim sisteminin General Motors'un 1956 yılında IBM'in 701 sistemi için yazdığı NAA IO olduğu söylenebilir.

1960'lara gelindiğinde IBM System/360 isminde yeni bir bilgisayar donanımı geliştirme işine girişti ve artık donanımla işletim sistemini birlikte satma fikrini benimsedi. Bu donanım 1964 yılında duyuruldu ve 1965 yılında gerçekleştirildi. İlk System/360 Model 30 bilgisayarları o zamanın "Solid Logic Technoloy (SLD)" teknolojisiyle üretilmişti. Hem öncekilerden daha güçlüydü hem de daha az yer kaplıyordu. Saniyede 34500 işlem yapabiliyordu ve 8 ila 64K belleğe sahipti. 1967 yılında System/360'ın Model 60'ı piyasaya sürüldü. Bu model saniyede 16.6 milyon komut çalıştırabiliyordu ve ana belleği de tipik olarak 512K, 768K ve 1 MB idi. IBM Sistem 360 donanımları için 1964 yılında ilk kez OS/360 işletim sistemini oluşturdu. IBM daha sonra 1967 yılında OS/360 Model 67 için OS/360'ın TSS 360 isminde zaman paylaşımlı (time sharing) bir versiyonunu daha geliştirdi. IBM'in System/360 makineleri ve işletim sistemleri önemli ticari başarı kazanmıştır. System/360'ı System/370 izledi. System/360 ve System/370 için başka kurumlar da işletim sistemleri geliştirmiştir. Michigan Terminal System (MTS) ve MUSIC/SP bunlar arasında önemli olanlardandır.

1960'lı yıllarda başka firmalarda işletim sistemleri geliştirmiştir. Örneğin Control Data Corporation firmasının SCOPE işletim sistemi batch işlemler yapabiliyordu. Aynı firma MACE isminde bu işletim sisteminin zaman paylaşımlı bir versiyonunu da yazmıştır. Firma bu çalışmalarını 1970'li yıllarda Kronos işletim sistemiyle devam ettirmiştir. Burroughs firması 1961 yılında MCP iişletim sistemi ile B5000 bilgisayarlarını, GE firması da 1962 yılında GECOS işletim sistemiyle GE-600 serisi bilgisayarlarını piyasaya sürdü. UNIVAC dünyanın ilk ticari bilgisayarlarını üreten firmadır. Bu firma da 1962 yılında UNIVAC 1107 için EXEC I işletim sistemini yazdı. Bu işletim sistemini sırasıyla Exec 2 ve Exec 8 izledi.

DEC (Digital Equipment Corporation) eskilerin en önemli bilgisayar üretici firmalarından biriydi. (DEC 1998 yılında Compaq firması tarafından Compaq' firması da 2002 yılında HP firması tarafından ssatın alındı.) Firmanın en önemli ürünleri PDP (Programmed Data Processor) isimli bilgisayarlarıdır. Firma PDP-1'den başlayarak PDP-16'ya kadar PDP makinelerinin 16 versiyonunu piyasaya sürmüştür. DEC'in PDP-8'inin mini bilgisayar devrimini başlattığı söylenebilir. Bu model 50000'in üzerinde satışa ulaşmıştır. UNIX işletim sistemi 1969 yılında ilk kez DEC'in PDP-7 modeli üzerinde yazılmıştır. DEC PDP -7 18 bitlik bir makineydi. Makine DECsys denilen işletim sistemi benzeri bir yönetici programla beraber satılıyordu. DEC'in PDP-10 modelinde DEC işletim sistemi olarak TOPS-10 isimli bir sisteme geçti. PDP-10 26 bitlik bir makineydi.

1960 yıllarda DEC firması da "mini" bilgisayarları için de bazı işletim sistemleri yazmıştır. O zamanlar işletim sistemleri ağırlıklı olarak sembolik makine diliyle yazılıyordu. 1960'lı yılların sonlarında AT&T Bell Lab. tarafından UNIX işletim sistemi geliştirildiğinde önemli bir devrim yaşandı. UNIX işletim sistemi 1973 yılında C ile yeniden yazılmıştır. Böylece artık işletim sistemlerinin yüksek seviyeli dillerle de yazılabildiği görülmüştür. PDP-11'i 16 bitlik PDP-12 izledi. PDP-12 Intel'in x86 ve Morola'nın 6800 işlemcileri için ilham kaynağı olmuştur.

1970'li yılların ikinci yarısında entegre devrelerin de geliştirilmesiyle "ev bilgisayarları (home computer)" ortaya çıkmaya başladı. Bunlarda genellikle BASIC yorumlayıcıları ile iç içe geçmiş CP/M tya da GEOS işletim sistemleri kullanılıyordu. 1970'li yıllarda pek çok firma farklı ev bilgisayarları üretmiştir. BBC Micro, Commodore 64, Apple II, Atari, Amstrad, ZX Spectrum dönemin en ünlü ev gilgisayarlarındandı. Bu makinelerde kullanılan işlemciler Intel'in 8080'i, Zilog'un Z80'i, Motorola'nın 6800'ü gibi 8 bitlik işlemcilerdi.

Apple firması 1976 yılında kuruldu. Apple'ın ilk bilgisayarı Apple I idi. Bunu 1977'de Apple II, 1980'de de Apple III izledi. Bu ilk Apple bilgisayarlarında AppleDOS isimli işletim sistemleri kullanılıyordu. Daha sonra Apple 1983'te Lisa modelini piyasaya sürdü. 1983'ün sonlarında da ilk Machintosh bilgisayarını çıkarttı. Lisa ile birlikte Apple grafik tabanlı işletim sistemlerine geçiş yaptı. Lisa ve sonraki Apple bilgisyaraların hepsi grafik bir arayüze sahiptir. Machintosh markası daha sonra Mac olarak teleffuz edilmeye başlandı. Lisa bilgisayarlarında kullanılan işletim sistemi LisaOS ismindeydi. Apple daha sonra Machintosh bilgisayarlarının değişik versiyonlarını piyasaya sürdü. Bunlardaki işletim sistemini "System Software 1 (1984), System Software 2 (1985), System Software 3 (1986), System Software 4 (1987), System Software 5 (1987), System Software 6 (1988), ve System Software 7 (1991)" olarak isimlendirdi. Apple System Softwre 7.5'ten sonra işletim sisteminin ismini "System Software" yerine Mac OS olarak değiştirdi ve System Software 7.6 versiyonu Mac OS 7.6 ismiyle çıktı. Daha sonra Apple 1997 yılında Mac OS 8'i, 1999 yılında da Mac OS 9'u çıkarmıştır.

1980'li yıllarda Mac bilgisayarlarının fiyatı çok yüksekti ve satışları da iyi gitmiyordu. Çünkü Steve Jobs bilgisayarların program yazmak için değil kullanmak için alınması gerektiğini düşünüyordu. Nihayet Apple'daki çalkantılar sonucunda Steve Jobs 1985 yılında Apple'dan ayrılmak zorunda kaldı (kovuldu da denebilir) ve NeXT firmasını kurdu. NeXT firması NeXT isimli bilgisayarları geliştirdi. Bu bilgisayarlarda NeXTSTEP isimli işletim sistemi kullanılıyordu. Daha sonta bu sistem açık hale getirildi ve OPENSTEP ismini aldı. Dünyanın ilk Web tarayıcısı Tim Barners Lee tarafından Cern'de NeXT bilgisayarları üzerinde gerçekleştirilmiştir.

Steve Jobs 1997 yılında Apple'a geri döndü. Apple da NeXT firmasını 200 milyon dolara satın aldı. Sonra piyasya iMac ve Power Mac serileri çıktı. Daha sonra Steve Jobs Mac'lerin çekirdeklerini tamamen değiştirme kararı aldı. Mac'ler Mac OS X ile birlikte yeni bir çekirdeğe geçtiler.

DOS işletim sistemi text ekranda çalışıyordu. Microsoft da geleceğin grafik taabanlı işletim sisteemlerinde olduğunu gördü ve yavaş yavaş DOS'u bırakarak grafik tabanlı bir sisteme geçmeyi planladı. Bunun için Windows isimli grafik arayüzün birinci versiyonunu 1985'te çıkardı. Bunu 1987'de Windows 2, 1990'da Windows 3.0 ve 1992'de de Windows 3.1 izledi. Bu 16 bit Windows sistemleri işletim sistemi değildi. DOS üzerinden çalıştırılan birer grafik arayüz gibiydi. Microsoft daha sonra Windows'u Windows NT 3.1 ile bağımsız bir işletim sistemi haline getirdi. Microsot bundan sonra sırasıyla 1994 yılında Windows NT 3.5'i, 1995 yılında Windows NT 3.51'i ve Windows 95'i, 1998 yılında Windows 98'i, 2000 yılında Windows 2000 ve Windows ME'yi, 2001 yılında Windows XP'yi, 2006 yılında Windows Vista'yı, 2012 yılında Windows 8'i, 2015 yılında da Windows 10'u çıkarmıştır.

Linux işletim sistemi 1992 yılında bir dağıtım olarak piyasaya çıkmıştır. Linux işletim sisteminin hikayesi daha geniş olarak izleyen bölümlerde ele alınmaktadır.

## UNIX Türevi İşletim Sistemlerinin Tarihsel Gelişimi

UNIX İşletim sistemi AT&T Bell Laboratuvarlarında 1969-1971 yılları arasında geliştirildi. Proje ekibinin lideri Ken Thompson'du. Çalışma ekibinde Dennis Ritchie, Brian Kernighan gibi önemli isimler de vardı. Ekip daha önce General Electrics'in GE-645 main frame bilgisayarı için Multics işletim sistemi üzerinde çalışıyordu. Multics işletim sisteminin geliştirilmesine 1964 yılında başlandı. Projede General Electric, MIT ve Bell Lab birlikte çalışıyordu. Sonra proje Honeywell şirketi tarafından devralınmıştır.

AT&T 1969 yılında bu projeden çekilerek kendi işletim sistemini geliştirmek istemiştir. Geliştirme çalışmasına DEC'in PDP-7 makinelerinde başlanmıştır. UNIX ismi 1970 yılında Brian Kernighan tarafından Multics'ten kelime oyunu yapılarak isimlendirilmiştir. Proje ekibi AT&T'yi DEC PDP-11 almaya ikna etti ve böylece geliştirme çalışmaları buarad devam etti. UNIX'in resmi olarak ilk sürümü Ekim 1971'de ikinci sürümü ise Aralık 1972'de, Üçüncü ve dördüncü sürümleri de 1973 yılında yayınlanmıştır. UNIX işletim sistemi büyük ölçüde PDP'nin sembolik makine dili ve Ken Tompson'ın B isimli programlama diliyle gelitirilmiştir. B programalam dili fonksiyonları alıp DEC'in makine diline dönüştürüyordu. Aslında tam bir derleyici olarak değerlendirilip değerlendirilmeyeceği tartışmalıdır. İşte 1972 yılında Dennis Ritchie Ken Thompson'ın B programlama dilinden hareketle C Programlama dilini geliştirmiştir. UNIX işletim sisteminin dördüncü sürümü 1973 yılında yeniden C Programlama Dili ile yazılmıştır. 1974 yılında UNIX'in beşinci sürümü oluşturuldu. Bu sürümlerin hepsi araştırma amaçlıydı ve "educational license" ismiyle lisanslanmıştı. UNIX işletim sistemi bir araştırma projesi olarak organize edilmişti. Bu nedenle kaynak kodlarını araştırma kuruluşlarına ücretsiz dağıtılmıştır. Örneğin 1974 yılında Kaliforniya Üniversitesi (Berkeley) işletim sisteminin kopyasını Bell Lab'tan aldı. 1975 yılında UNIX'in altıncı sürümü şirketlere yönelik hazırlandı. UNIX'in altıncı versiyonunun kaynak kodları 20000$'a (şimdikinin 95000 $'ı) şirketlere sunuldu. 1977 yılında Bell Lab, UNIX'i Interdata 7/32 isimli 32 bit mimariye port etti. Bunu 1978'de VAX portu izledi.

1978 yılında Kaliforniya Üniversitesi "Berkeley Software Distribution (1BSD)" ismiyle AT&T dışındaki ilk UNIX dağıtımını gerçekleştirdi. Bu dağıtım hayatını hala FreeBSD, OpenBSD ve NetBSD olarak devam ettirmektedir. 1979'da BSD'nin ikinci versiyonu (2BSD) ve 1979'un sonlarına doğru da üçüncü versiyonu (3BSD) piyasaya sürüldü. Bunu 1980 yılında versiyon 4 (4BSD) izlemiştir. 1991 yılında BSD UNIX'ten AT&T kodları tamamen arındırılmış ve kod bakımından özgün hale getirilmiştir. BSD'nin son versiyonu 1995'te 4.4BSD Lite Release 2 olarak çıkmıştır.

1980'li yıllarda pek çok kurum ve ticari firma AT&T'den UNIX kodlarını alıp kendilerine yönelik UNIX sistemleri oluşturmuştur. Bunların önemli olanları şunlardır:

AIX: IBM tarafından geliştirilmiş olan UNIX türevi sistemlerdir. İlk kez 1986 yılında piyasaya sürülmüştür. IBM AIX'i System/370, RS/6000 PS2 bilgiyaralarında kullanıyordu. Bu sistemlerAT&T UNIX System 5 kodları temel alınarak geliştirilmiştir. AIX hala kullanılmaktadır.

IRIX: SGI firması tarafından AT&T ve BSD kodları değiştirilerek 1988'de oluşturulmuştur. 2006'da bırakılmıştır.

HP-UX: HP 9000 bilgisayarları için 1982'de oluşturulmuştur. Hala devam ettirilmektedir.

ULTRIX: DEC firmasının PDP-7, PDP-11 ve VAX donanımları için geliştirdiği UNIX sistemiydi. İlk versiyonu 1984 yılında çıktı. 1995 yılında piyasadan çekildi.

XENIX: Microsoft tarafından 1980 yılında geliştirilmeye başlanmıştır. İlk versiyonu 1980'in sonlarına doğru çıkmıştır. Daha sonra SCO firması Maicrosoft'la bu konuda işbirliği yapmış 1987 yılında da Microsoft sistemi tamamen SCO'ya devretmiştir. Bu sistemi daha sonra SCO firması ,SCO-UNIX olarak devam ettirmiştir.

SCO-UNIX: SCO firması XENIX'i Microsoft'tan alınca bunu SCO-UNIX olarak devam ettirdi. SOC-UNIX'in ilk versiyonu 1989 yılında çıktı.SCO sonra bunu OpenServer ismiyle devam ettirmiştir.

FreeBSD, NETBSD ve OpenBSD: 4.3BSD sistemleri temel alınarak geliştirilmiştir. FreeBSD NetBSDve 1993 yılında, OpenBSD ise 1996 yılında piyasaya çıkmıştır. Sürüdürülmeye devam etmeketdir. Önemli bir UNIX varyantı durumundadır. Bu üç sistem de birbirlerine çok benzemektedir. FreeBSD genel amaçlı client ve server işletim sistemi

olma niyetindedir. NetBSD daha taşınabilirdir ve geniş bir porta sahiptir. Daha çok bilimsel çalışmalarda tercih edilmektedir. OpenBSD güvenliğin önemli olduğu alanlarda tercih edilmektedir.

SunOS (Solaris): Sun firmasının BSD kodlarıyla oluşturduğu UNIX türevi işletim sistemiydi. İlk versiyonu 1982 yılında çıktı. SunOS işletim sistemi 5.2 versiyonundan sonra (1992) Solaris ismiyle pazarlanmaya başlamıştır.

Linux: Linux Torvalds'ın öncülüğünde geliştirilmiş en yaygın UNIX türevi işletim sistemidir. İlk versiyonu 1991 yılında çıkmıştır. Hala devam ettirilmektedir. Linux'un tarihsel gelişimi izleyen bölümde ayrıntılı bir biçimde açıklanmaktadır.

Mac OS X: Carneige Mellon üniversitesinin Mach isimli çekirdeği ile BSD Unix sisteminin biraraya getirilmesiyle oluşturulmuşturç ilk versiyonu 2001 yılında piyasaya sürülmüştür. İzleyen bölümlerde Mac OS X'in tarihsel gelişimi ayrıntılı olarak ele alınmaktadır.

### Linux Sistemlerinin Tarihsel Gelişimi

Linux Torvalds Helsinki Üniversitesinde öğrenciyken bir işletim sistemi yazmaya niyetlenmiştir. O zamanlarda telif uygulanmayan UNIX türevi bir işletim sistemi kalmamıştı ve GNU projesinin işletim sistemi de (GNU Hurd) bitirilememişti. MINIX sisteminin lisansı yalnızca akademik kullanımlar için sınırlandırılmıştır. Linus projesini USENET haber gruplarında duyurdu ve zamanla kendisine gönüllü yardım edecek sistem programcıları buldu. Yazılım dünyasında bu tür girişimlerle sık karşılaşıldığı halde başarı olasılığı nispeten düşük olmaktadır. Linus Torvalds'ın bu girişimi başarıya ulaşmıştır.

1992 yılında Linux'un 0.01 sürümü oluşturuldu. 1994 yılında stabil bir biçimde Linux 1.0 versiyonu dağıtılmaya başlandı. Bunu 1996 yılında Linux 2.0, 1999 yılında 2.2, 2000 yılında 2.4 ve 2003 yılında 2.6 izledi. Daha sonra Linux versiyon numaralandırma sistemi değiştirilmiştir. 2011 yılında 3.0, 2015 yılında 4.0, 2019 yılında 5.0 versiyonları çıkmıştır. Kursun yapıldığı zamandakşi son çekirdek sürümü 5.5'tir.

Linux monolithic bir çekirdek yapısına sahiptir. Büyük ölçüde POSIX uyumu bulunmaktadır.

### Mac OS X Sistemlerinin Tarihsel Gelişimi

Mac OS X UNIX türevi bir işletim sistemidir. Çekirdeğine Darwin denilmektedir. Darwin açık kaynak kodlu bir sistemdir. Ancak Mac OS X tam anlamıyla açık bir sistem değildir. Yani MAC OS X çekirdeği açık geri kalanı mülkiyete (proprietary) bağlı bir işletim sistemidir.

Darwin'in hikayesi 1989 yılında NeXT'in NeXTSTEP işletim sistemiyle başladı. NeXTSTEP daha sonra OPENSTEP oldu. Apple NeXT firmasını 1996'nın sonunda 1997'nin başında satın aldı ve sonraki işletim sistemini OPENSTEP üzerine kuracağını açıkladı. Bundan sonra Apple 1997'de OPENSTEP üzerine kurulu olan Rapsody'yi çıkardı. 1998'de de yeni işletim sisteminin Mac OS X olacağını açıkladı. Daha sonra Rapsody'den Darwin projesi türedi. Darwin projesi ayrı bir işletim sistemi olarak da yüklenebilmektedir. Ancak Darwin grafik arayüzü olmadığı için Mac programlarını çalıştıramaz.

Darwin'den çeşitli projeler türetilmiştir. Bunlardan biri Apple tarafından 2002'de başlatılan OpenDarwin'dir. Bu proje 2006'da sonlandırılmıştır. 2007'de PureDarwin projesi başlatılmıştır.

Darwin'in çekirdeği XNU üzerine oturtulmuştur. XNU bir çekirdektir ve NeXT firması tarafından NEXTSTEP işletim sisteminde kullanılmak üzere geliştirilmiştir. XNU, Carnegie Mellon ("Karnegi" diye okunuyor) üniversitesi'nin Mach 3 mikrokernel çekirdeği ile 4.3BSD karışımı hibrit bir sistemdir.

Mac OS X sistemlerinin versiyonları şunlardır:

- Mac OS X 10.0 (Cheetah, 2001)
- Mac OS X 10.1 (Puma, 2001)
- Mac OS X 10.2 (Jaguar, 2002)
- Mac OS X 10.3 (Panther, 2003)
- Mac OS X 10.4 (Tiger, 2005)

- Mac OS X 10.5 (Leopard, 2007)
- Mac OS X 10.6 (Snow Leopard, 2009)
- Mac OS X 10.7 (Lion, 2011)
- Mac OS X 10.8 (Mountain Lion, 2012)
- Mac OS X 10.9 (Maverics, 2013)
- Mac OS X 10.10 (Yosemite, 2014)
- Mac OS X 10.11 (El Capitan, 2015)
- Mac OS X 10.12 (Sierra, 2017)
- Mac OS X 10.13 (High Sierra, 2017)
- Mac OS X 10.14 (Mojave, 2018)
- Mac OS X. 10.15 (Catalina, 2019)

Mac OS X büyük ölçüde POSIX uyumlu bir sistemdir.

## GNU Projesi, Özgür Yazılım ve Açık Kaynak Kod Akımları

1970'lerdeki mikro bilgisayarlar devrimine kadar yazılımda bir telif anlayışı yoktu. Yani yazılımın dağıtılması konusunda sözleşmeler ve hukuki yaptırımlara gerek duyulmamıştı. Yazılım zaten donanımla birlikte satılıyordu ya da kuruma özel yapılıyordu. 1969 yılında IBM yazılımı donanımla birlikte verdiği için rekabet kurallarına uymadığı gerekçesiyle mahkemeye verilmiştir ve cezaya çarptırılmıştır. 1970'li yıllarda yazılım maliyetleri artmış, yazılım sektörü genişlemiş ve lisanslama politikaları da uygulamaya sokulmuştur. Pek çok yazılım bu yıllarda özel lisanslarla piyasaya sürülmeye başlanmıştır. 1980'li yıllarda bu lisanslama faaliyetleri hız kazanmıştır.

1980'li yıllarda tüm UNIX türevi sistemler çeşitli biçimlerde sınırlandırıcı lisanslara sahipti. Yani 1980'li yıllarda sınırlaması olmayan UNIX türevi sistemler kalmamıştı. Bu nedenle bedava ve sınırlamasız UNIX türevi bir işletim sistemine gereksinim duyulmaya başlandı. İşte durumdan vazife çıkaran ünlü Emacs editörünün yazarı Richard Stallman 1983 yılının sonlarına doğru GNU projesini başlattı ve özgür yazılım (free software) fikrini oraya attı. GNU projesinin amacı açık kaynak kodlu UNIX benzeri bir işletim sistemini ve geliştirme araçlarını yazmaktı. Proje fiilen 1984 yılında başlamıştır. Stallman 1985 yılında özgür yazılım kavramını yaygıınlaştırmak amacıyla Free Software Foundation (www.fsf.org) isimli kurumu kurdu ve atık GNU projesi bu kurum tarafındsan yürütülmye başlandı. FSF özgür yazılım modeli için GPL (GNU Public Liscence) denilen lisansı ortaya çıkardı. Özgür yazılım akımında oluşturulan bir yazılım istenildiği gibi çalıştırılabilir, kopyalanabilir, incelenebilir, dağıtılabilir, değiştirilebilir ve iyileştirilebilir. Daha açık bir biçimde özgür yazılım tipik olarak aşağıdaki dört özgürlükle tanımlanmıştır:

Özgürlük 0: Programı her türlü amaç için çalıştırma özgürliği
Özgürlük 1: Programın kaynak kodunu inceleme ve değiştirebilme özgürlüğü
Özgürlük 2: Programın kopyalarını çıkartabilme ve yeniden dağıtabilme özgürlüğü
Özgürlük 3: Programı iyileştirebilme ve iyileştirilmiş programı yayınlama özgürlüğü

GNU projesi bağlamında pek çok temel araç (gcc derleyici, ld bağlayıcı vs.) geliştirilmiştir. Fakat hedeflenen bir çekirdek bir türlü oluşturulamamştır.

Aslında özgür yazılım (free software) ile açık kaynak kod (open source) akımları arasında bazı farklar olmakla birlikte her iki akımın da hedefleri benzerdir. Özgür yazılım bir sosyal harekete benzetilirken açık kaynak kod akımı bir geliştirme metodolojisine benzetilmektedir. Biz kursumuzda tüm bu akımları "açık kaynak kod (open source)" olarak nitelendireceğiz. Özgür yazılımım akımının temel lisansı GPL'dir (GNU Public Liscence). Bunun yumuşatılmış LGPL (Lesser GPL) biçiminde bir versiyonu da oluşturulmuştur. Ayrıca Apache, MIT, BSD gibi açık kaynak kodlu başka lisanslar da vardır. Şüphesiz bu lisansların aralarında birtakım farklılıklar bulunmakla birlikte pek çok yönleri de ortaktır.

### Dağıtım Kavramı ve Linux Dağıtımları

Açık kaynak kodlu yazılımlar değiştirilerek biraraya getirilip paketlenerek istenildiği gibi dağıtılabilmektedir. Dağıtım (distribution) bu anlamda genelk bir terimdir ve her türlü açık kaynak kodlu yazılım için dağıtım söz konusu olabilir. Ancak biz burada Linux dağıtımları üzerinde duracağız.

Linuz temel olarak bir çekirdek geliştirme projesidir. Linux kaynak kodlarına baktığınızda tüm kodların çekirdekle ilgili olduğunu görürsünüz. Çekirdeğin dışındaki tüm yazılımlar (örneğin init prosesinden başlayarak, kabuk yazılımları, paket yöneticileri, pencere yöneticileri vs.) hepğ başka proje grupları taarafından gerçekleştirilmiş açık kaynak kodlu yazılımlardır. İşte tüm bu açık kaynak kodlu yazılımların Linux çekirdeği temelinde bir araya getirilmesi ve doğrudan kullanıcının yükleyip çalıştırabileceği biçimde paketlenmesine Linux dağıtımları denilmektedir. Linux dağıtımları pencere yöneticileri (KDE, GNOME gibi), paket yötecilieri (APT, RPM, YUM, DPKG, PACMAN, ZYPPER gibi), içerilen yazılımlar bakımından farklılıklar gösterebilmektedir.

Toplamda 200'ün üzerinde Linux dağıtımının olduğu söylenebilir. Ancak bunlar arasında az sayıda dağıtım çok popüler olmuştur. Bazı dağıtımlar bazı dağıtımlardan oluşturulmuştur. Burada en çok kullanılan Linux dağıtımlarından bahsedeceğiz.

Debian Dağıtımı: En önemli ve en eski Linux dağıtımlarından biridir. Red Hat Enterprise Linux en önemli Fedora türevidir. Knoppix, Mint, Ubuntu Debian türevi dağıtımlardır.

Fedora: Red Hat firması trafından çıkarılmış olan dağıtımdır. İlk kez 2003 yılında oluşturulmuştur. RPM paket yöneticisini kullanır. yum ve rpm paket yöneticilerini kullanır. Centos ve Scientific Linuz en önemli Fedora türevi dağıtımlardır. 2000 yılında ilk sürümü yapılan Red Hat Enterprise Linux (RHEL) en önemli Fedora türevidir. Ondan da CentOS, Scientific Linux gibi dağıtımlar türetilmiştir. CentOS server makinelerde en yaygın kullanılan Linux versiyonudur.

OpenSUSE: Alman SUSE firmasının desteklediği dağıtımdır. SUSE LinuxEnterprise isminde ticari bir versiyonu da vardır. ZYpp, YaST ve RPM paket yöneticileri kullanır.

Slackware: En eski Linux dağıtımıdır. 1993 yılında oluşturulmuştur. Sürdürümü yavaş olmakla birlikte hala devam etmektedir.

### UNIX Türevi Sistemlerde Kullanılan Standartlar

1980'li yıllarda AT&T ya da BSD kodlarından türetilmiş olan ve çoğunluğu şirketlere ait olan pek çok UNIX türevi sistem oluşturuldu. Bu sistemler birbirlerine çok benzemekle birlikte aralarında bazı farklılıklara da sahipti. İşte IEEE durumdan vazife çıkartarak bu UNIX türevi sistemleri standardize etmek için kolları sıvadı ve bunun sonucu olarak da POSIX standartları oluşturuldu.

POSIX süzcüğü Richard Stallman tarafından önerilmiştir. "Portable Operating System Interface for UNIX" sözcüklerinden kısaltılarak uydurulmuştur ve "poziks" biçiminde okunmaktadır. POSIX standartları üzerinde çalışmalar 1985 yılında başlamıştır ve ilk standartlar 1988 yılında "IEEE Std 1003.1-1988" kod numarasıyla oluşturulmuştur. POSIX her ne kadar UNIX türevi sistemler için düşünülmüşse de UNIX türevi mimariye sahip olamayan sistemler için de kullanılabilecek bir standarttır. Örneğin Windows sistemleri Interix denilen alt ssistemle POSIX uyumlu olarak da kullanılabilmektedir.

POSIX standartları 4 bölümden oluşmaktadır:

1) Base Definitions: Bu bölümde temel tanımlamalar bulunmaktadır.
2) Shell & Utilities: Bu bölümde kabuk komutları ve standart utility programlar ele alınmaktadır.
3) System Interfaces: Bu bölümde C programcıları için hazır bulunan POSIX fonksiyonları tanıtılmaktadır.
4) Rationale: Çeşitli kuralların ve özelliklerin gerekçeleri bu bölümde açıklanmaktadırç

POSIX standartlarının zaman içerisinde çeşitli versiyonları çıkartılmıştır. Bu versiyonlarda hem yeni POSIX fonksiyonları kütüphaneye eklenmiş hem de standartlardaki bazı bozukluklar ve uyumsuzluklar düzeltilmiştir. Standardın önemli versiyonları şu senelerde yayınlanştır: 1992, 1993, 1995, 1997, 2001, 2004, 2008, 2017. Standartlardaki en önemli değişim 1993 yılında "POSIX 1.b" diye de isimlendirilen "Realtime-extensions" ile 1995 yılında "POSIX 1.c" diye isimlendirilen "Thread-extensions" isimli eklemelerdir. Bu eklemelerle POSIX'e gerçek zamanlı işlemler için çeşitli özelliklerle thread kullanımı eklenmiştir.

Single UNIX Specification UNIX türevi sistemler için oluşturulmuş diğer önemli standarttır. Bis sistemin UNIX olarak değerlendirilebilmesi için bu standartlara uygun olması gerekmektedir. Standartlar "Austin Group" isimli toplulukla

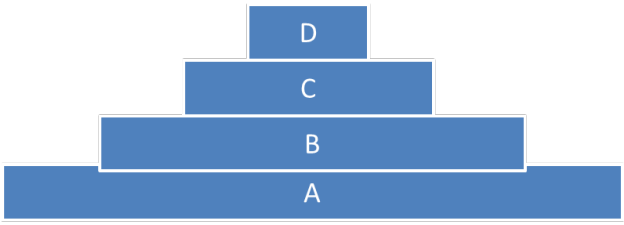
"Open Group" isimli dernek tarafından geliştirilmiştir. Sürdürümü Open Group tarafından yapılmaktadır. Open Group hali hazırda UNIX sistemlerinin isim haklarını elinde bulundurmaktadır. Single UNIX Specification isimli standardın zamanla pek çok versiyonu oluşturulmuştur.

POSIX standartları ile Single UNIX Specification standartları arasında eskiden daha fazla farklılılar vardı. Ancak bugün itibari ile bu iki standart birbirlerine yaklaştırılmış ve neredeyse aynı hale getirilmiştir. Single UNIX Specification dokümanlarına Internet'ten Open Group'un web sitesinden erişilebilir:

https://pubs.opengroup.org/onlinepubs/9699919799/

### Programlamadaki Katmanlı Yapılar

Yazılımda genel olarak kod tekrarı istenmez. Bu nedenle yazılım sistemleri katmanlı bir yapıya sahip olur. Örneğin B kütüphanesi A kütüphanesinin fonksiyonlarını kullanarak yazılmış olabilir. C kütüphanesi de B'yi kullanarak yazılmış olabilir. D de C'yi kullanmış olabilir:




Kod tekrarının iki önemli dezavantajı vardır: Gereksiz kod büyümesi oluşur ve test ve bakım işlemlerini zorlaştırır.

Yazılımda kod tekrarının engellenmesi için başvurulan tipik yöntem kodu alt programlara ayırıp onları çağırmaktır. Örneğin C'de proje içerisinde bir kod parçasının çeşitli yerlerde yinelendiğini düşünelim. Bu kod parçasını bir fonksiyon olarak bildirip tekrarlanan yerlerde o fonksiyonu çağırabiliriz. Aslında nesne yönelimli programlama tekniğinde türetme işlemine de kod tekrarını engellemek için başvurulmaktadır. Bu teknikte iki sınıfın birtakım ortak elemanları varsa bu ortak elemanlar bir taban sınıfta toplanır, bu iki sınıf da o taban sınıftan türetilerek gerçekleştirilir.

#### API (Aplication Programming Interface) Kavramı

Bir yazılım sisteminde (bu bir işletim sistemi olabilir, framework olabilir, ya da başka bir yazılımlar olabilir) uygulama programcılarının doğrudan çağırabileceği, o sistem ile uygulama programcısı arasında köprü oluşturan fonksiyon ya da sınıf kümesine API denilmektedir. API aslında lastik bir terimdir. Hangi fonksiyonlara API denilebileceği tartışılabilir. Fakat genel olarak API uygulama programcılarının ilgili sistem üzerinde birtakım faydalı işlemler yapabilmek için kullandıkları fonksiyon ya da sınıflardır. Örneğin Java API'leri denildiğinde Java sınıflarını, Windows API'leri denildiğinde Windows işletim sisteminde temel işlemleri yapmak için kullanılan fonksiyonları anlarız.

### Kütüphane (Library) ve Framework Kavramları

Kütüphane ve Framework kavramlarının sınırları tam belli değildir. Değişik kaynaklar bu sınırları değişik biçimde çizebilmektedir. Fakat bir sistemin framework olarak tanımlanabilmesi için şu iki özelliğin bulunması gerektiği yönünde bir eğilim vardır:

1) Karmaşıklığın kullanıcıya daha basit gösterilmesi ve yük oluşturan bazı hammaliye işlemlerin kullanıcının üzerinden alınması.

2) Kod akışının ele geçirilmesi ve duruma göre programcıya belli zamanlarda verilmesi (inversion of control).

Halbuki kütüphanelerde arka planda birtakım işlemleri bizim için yapmak ve bir akışı ele geçirmek gibi bir amaç yoktur. Kütüphanelerde programın akışı bizdedir. Biz istersek kütüphane fonksiyonlarını çağırırız. Onlar da faydalı işlemleri yaparlar. Şüphesiz pek çok framework aynı zamanda birtakım kütüphanelere de (API'lere de) sahiptir.

Bazı ara durumlarda o şeyin framework olarak mı yoksa kütüphane olarak mı adlandırılacağı konusunda tereddütler olabilir. (Örneğin Qt için ona kütüphane diyenler de framework diyenler de vardır.)

## CPU, Mikroişlemci, Mikrodenetleyici ve SOC Kavramaları

Bir bilgisayar sisteminde aritmetik, mantıksal, bitsel işlemler ve karşılaştırma işlemleri mikroişlemci (microprocessor) denilen birim tarafından yapılmaktadır. Mikroişlemciler entegre devre biçiminde üretilmişlerdir. Mikroişlemcilere kavramsal olarak CPU (Central Processing Unit) de denilmektedir. Yani CPU mikroişlemcilerin kavramsal ismidir. Aslında bir bilgisayar sisteminde komut çalıştıran pek çok işlemci bulunabilmektedir. CPU bu işlemcileri de programlayan ana (merkezi) işlemcidir. (Bilgisayar sisteminde yerel bazı işlemlerden sorumlu yardımcı işlemciler de vardır. Örneğin "kesme denetleyicisi (interrupt controller)", "disk denetleyicisi (disk controller)", "DMA denetleyicisi (DMA controller)" gibi.)

Kendi içerisinde CPU'su, RAM'i, ROM'u ve bazı çevre üniteleri de bulunan entegre devrelere "mirodenetleyici (microcontroller)" denilmektedir. Mikrodenetleyicilerin işlem kapasiteleri ve içerdikleri bellek miktarları düşük olma eğilimindedir. Ancak bunlar çok kolay programlanıp uygulamaya sokulabilmektedir. Mikro denetleyicilere "tek çiplik bilgisayar (single chip computer)" da denilmektedir. Mikrodenetleyiciler özellikle gömülü sistemlerde tercih edilirler. Bunların düşük güç harcaması ve ucuz olmaları en büyük avantajlarındandır.

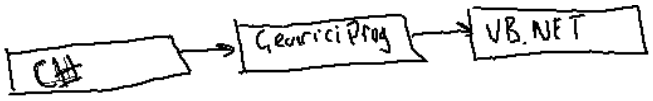
Bazı firmalar ayrı birimler olarak tasarlanmış mikroişlemcileri, RAM'leri, ROM'ları ve diğer bazı üniteleri tek bir entegre devrenin içerisine sıkıştırmaktadır. Bunlara genel olarak "SOC (System On Chip)" denilmektedir. SOC mikrodenetleyicilere benzese de aslında onlardan farklıdır. SOC'lar içerisindeki işlemcilerin ve belleklerin kapasiteleri yüksektir. Bunlar özel amaçlı üretilirler ve bunların devrelerde kullanılmaları mikrodenetleyiciler kadar kolay değildir. Bunların en önemli avantajları "az yer kaplamasıdır". Örneğin Raspberry Pi kitlerinde Broadcom isimli firmanın 2835, 2836, 2837 numaralı SOC entegreleri kullanılmıştır. Bunların içerisinde Cortex A serisi ARM işlemcileri, RAM ve ROM bellekler bulunmaktadır.

### Gömülü Sistemler (Embedded Systems)

Asıl amacı bilgisayar olmayan fakat bilgisayar devresi içeren sistemlere genel olarak gömülü sistemler denilmektedir. Örneğin elektronik tartılar, biyomedikal aygıtlar, GPS cihazları, turnike geçiş sistemleri, müzik kutuları vs. birer gömülü sistemdir. Gömülü sistemlerde en çok kullanılan programlama dili C'dir. Ancak son yıllarda Raspberry Pi gibi, Banana Pi gibi, Orange Pi gibi güçlü ARM işlemcilerine sahip kartlar çok ucuzlamıştır ve artık gömülü sistemlerde de doğrudan kullanılır hale gelmiştir. Bu kartlar tamamen bir bilgisayarın işlevselliğine sahiptir. Bunlara genellikle Linux işletim sistemi ya da Android işletim sistemi yüklenir. Böylece gömülü yazılımların güçlü donanımlarda ve bir işletim sistemi altında çalışması sağlanabilmektedir. Örneğin Raspberry Pi'a biz Mono'yu yükleyerek C#'ta program yazıp onu çalıştırabiliriz.

## Çevirici Programlar (Translators), Derleyiciler (Compilers) ve Yorumlayıcılar (Interpreters)

Bir programlama dilinde yazılmış olan programı eşdeğer olarak başka bir dile dönüştüren programlara çevirici programlar (translators) denilmektedir. Çevirici programlarda dönüştürülmek istenen programın diline kaynak dil (source language), dönüşüm sonucunda elde edilen programın diline de hedef dil (target/destination language) denir. Örneğin:




Burada kaynak dil C#, hedef dil VB.NET'tir.

Eğer bir çevirici programda hedef dil aşağı seviyeli bir dil ise (saf makine dili, arakod ve sembolik makine dilleri alçak seviyeli dillerdir) böyle çevirici programlara derleyici (compiler) denilmektedir. Her derleyici bir çevirici programdır fakat

her çevirici program bir derleyici değildir. Bir çevirici programa derleyici diyebilmek için hedef dile bakmak gerekir. Örneğin arakodu gerçek makine koduna dönüştüren CLR bir derleme işlemi yapmaktadır. Sembolik makine dilini saf makina diline dönüştüren program da bir derleyicidir.

Bazı programlar kaynak programı alarak hedef kod üretmeden onu o anda çalıştırırlar. Bunlara yorumlayıcı (interpreter) denilmektedir. Yorumlayıcılar birer çevirici program değildir. Yorumlayıcı yazmak derleyici yazmaktan daha kolaydır. Fakat programın çalışması genel olarak daha yavaş olur. Yorumlayıcılarda kaynak kodun çalıştırılması için onun başka kişilere verilmesi gerekir. Bu da kaynak kod güvenliğini bozar.

Bazı diller yalnızca derleyicilere sahiptir (C, C++, C#, Java gibi). Bazıları yalnızca yorumlayıcılara sahiptir (PHP, Perl gibi). Bazılarının hem derleyicileri hem de yorumlayıcıları vardır (Basic, Swift, Python gibi). Genel olarak belli bir alana yönelik (domain specific) dillerde çalışma yorumlayılar yoluyla yapılmaktadır. Genel amaçlar diller daha çok derleyiciler ile derlenerek çalıştırılırlar.

### Decompiler'lar ve Disassembler'lar

Alçak seviyeli dillerden yüksek seviyeli dillere dönüştürme yapan (yani derleyicilerin yaptığının tam tersini yapan) yazılımlara "decompiler" denilmektedir. Örneğin C#'ta yazılıp derlenmiş olan .exe dosyadan yeniden C# programı oluşturan bir yazılım "decompiler"dır. Saf makine dilini decompile etmek neredeyse mümkün değildir. Ancak .NET'in arakodu olan "CIL (Common Intermediate Language)" ve Java'nın ara kodu olan "Java Byte Code" kolay bir biçimde decompile edilebilmektedir. C#'ta derlenmiş ve çalıştırılabilir hale getirilmiş dosyayı yeniden C#'a dönüştüren pek çok decompiler vardır (örneğin Salamander, Dis#, Reflector, ILSpy gibi). İşte bu tür durumlar için C# ve Java programcıları kendileri bazı önlemler almak zorundadırlar. Ancak C, C++ gibi doğal kod üreten derleyicilerin ürettiği kodlar geri dönüştürülememektedir.

### IDE (Integrated Development Environment) Kavramı

Derleyiciler komut satırından çalıştırılan programlardır. Bir programlama faaliyetinde program editör denilen bir program kullanılarak yazılır. Diske save edilir. Sonra komut satırından derleme yapılır. Bu yorucu bir faaliyettir. İşte yazılım geliştirmeyi kolaylaştıran çeşitli araçları içerisinde barındıran (integrated) özel yazılımlara IDE denilmektedir. IDE'nin editörü vardır, menüleri vardır ve çeşitli araçları vardır. IDE'lerde derleme yapılırken derlemeyi IDE yapmaz. IDE derleyiciyi çalıştırır. IDE yardımcı bir araçtır, mutlak gerekli bir araç değildir.

Microsoft'un ünlü IDE'sinin ismi "Visual Studio"dur. Apple'ın "X-Code" isimli IDE'si vardır. Bunların dışında başka şirketlerin malı olan ya da "open source" olan pek çok IDE mevcuttur. Örneğin "Eclipse" ve "Netbeans" yaygın kullanılan cross-platform "open source" IDE'lerdir. Linux altında Mono'da "Mono Develop" isimli bir IDE tercih edilmektedir. Bu IDE'nin Windows versiyonu da vardır.
UNIX/Linux sistemlerinde C ve C++ IDE'si olarak QtCreator, KDevelop, Eclipse (CDT), NetBeans, ve MonoDevelop kullanılabilir. Ancak kursumuzdaki uygulmalarda bir IDE kullanmayacağız. Kodları bir Kate, Visual Studio Code gibi bir editörde yazıp komut satırından derleyeceğiz.

### Doğal Kodlu Çalışma, Arakodlu Çalışma ve JIT Derlemesi

Kullandığımız CPU'lar ikilik sistemdeki makine komutlarını çalıştırmaktadır. Bir kodun CPU tarafından çalıştırılabilmesi için o kodun o CPU'nun makine diline dönüştürülmül olması gerekir. Zaten derleyiciler de bunu yapmaktadır. Eğer bir çevirici program (yani derleyici) o anda çalışılmakta olan makinenin CPU'sunun işletebileceği makine kodlarını üretiyor CPU da bunları çalıştırıyorsa buna doğal kodlu (native code) çalışma denilmektedir. Örneğin C ve C++ programlama dillerinde doğal kodlu çalışma uygulanmaktadır. Biz bu dillerde bir programı yazıp derlediğimizde artık o derlenmiş program ilgili CPU tarafından çalıştırılabilecek doğal kodları içermektedir.

Bazı sistemlerde derleyiciler doğrudan doğal kod üretmek yerine hiçbir CPU'nun makine dili olmayan (dolayısıyla hiçbir CPU tarafından işletilemeyen) yapay bir kod üretmektedir. Bu yapay kodlara genel olarak "ara kodlar (intermediate codes)" denilmektedir. Bu arakodlar doğrudan CPU tarafından çalıştırılamazlar. Arakodlu çalışma Java ve .NET dünyasında ve daha başka ortamlarda kullanılmaktadır. Java dünyasında Java derleyicilerinin ürettikleri ara koda "Java Bytecode", .NET (CLI) dünyasında ise "CIL (Common Intermediate Language)" denilmektedir. Pekiyi bu arakodlar ne işe

yaramaktadır? İşte bu arakodlar çalıştırılmak istendiğinde ilgili ortamın alt sistemleri devreye girerek önce bu arakodları o anda çalışılan CPU'nun doğal makine diline dönüştürüp sonra çalıştırmaktadır. Bu sürece (yani arakodun doğal makine koduna dönüştürülmesi sürecine)" tam zamanında derleme (just in time compilation)" ya da kısaca "JIT derlemesi" denilmektedir. Java ortamında bu JIT derlemesi yapıp programı çalıştıran alt sisteme "Java Sanal Makinesi (Java Virtual Machine)", .NET ortamında ise CLR (Common Language Runtime)" denilmektedir.

Şüphesiz doğal kodlu çalışma arakodla çalışmnadan daha hızlıdır. Pek çok benchmark testleri aradaki hız farkının %20 civarında olduğunu göstermektedir. Pekiyi arakodlu çalışmanın avantajları nelerdir? İşte bu çalışma biçimi derlenmiş kodun platform bağımsız olmasını sağlamaktadır. Buna "binary portability" de denilmektedir. Böylece arakodlar başka bir CPU'nun ya da işletim sisteminin bulunduğu bir bilgisayara götürüldüğünde eğer orada ilgili ortam (framework) kuruluysa doğrudan çalıştırılabilmektedir.

### UNIX/Linux Ortamlarının Windows ve Mac OS X Sistemlerinde Oluşturulması

Windows'ta Linux ortamının oluşturulması için iki yöntem kullanılabilmektedir.

1) Cygwin Ortamı İle Oluşturma: Cygwin isimli ortam yapay biçimde bize Linux çalışma ortamını (genel olarak POSIX çalışma ortamını) sunmaktadır. Cygwin bir sanal makine değildir. Bize yapay biçimde UNIX/Linux ortamı sunan bir yazılımdır. Biz bu ortamda UNIX/Linux kabuk komutlarını, gcc, g++ gibi derleyicilerle POSIX fonksiyonlarını kullanabiliriz. Burada geliştirdiğimiz programlar ilgili UNIX/Linux ortamına götürülerek yeniden derlenmek suretiyle çalıştırılabiir. Ancak Cygwin ortamının bir sanallaştırma yapmadığına arka planda Windows'un olanaklarıyla UNIX/Linux ortamını emüle ettiğine dikkat ediniz.

2) Sanallaştırma Yoluyla: Bugün VmWare, VirtualBox, Xen gibi sanallaştırma ve hypervisor yazılımlarla orijinal işletim sistemi tamamen sanallaştırma yoluyla çalıştırılabilmektedir. Artık Cygwin kullanımı Windows sistemlerinde bu nedenle çok azalmıştır. Sanallaştırmada "host" ve "guest" sistemler arasında "copy-paste" işlemleri de yapılabilmektedir. Sanallaştırma yazılımları "host" olarak Windows, Linux ve Mac OS X sistemlerinde bulunmaktadır.

Burada bir noktaya dikkatinizi çekmek istiyoruz: Mac OS X sistemleri aslında belli derecede POSIX uyumu olan UNIX türevi bir sistemdir. Dolayısıyla kursumuzda UNIX/Linux sistemi denildiğinde Mac OS X sistemi de anlaşılmalıdır. Kursumuzdaki UNIX/Linux sistemi için verilen örnekler Mac OS X sistemlerinde doğrudan derlenerek çalıştırılabilir.

### UNIX/Linux Sistemlerinde C Programlarının Derlenerek Çalıştırılması

Bir C programını derlemek için önce programın bir metin editöründe yazılıp bir dosya biçiminde diskte saklanması gerekir. Bundan sonra dosya ismi derleyicilere komut satırı argümanı biçiminde verilerek derleme gerçekleştirilmektedir. UNIX/Linux sistemlerinde ağırlıklı olarak GCC ve Clang derleyicileri kullanılmaktadır. Bu iki derleyicinin komut satırı seçenekleri birbirleriyle uyumludur. Program bir metin editörde yazılıp saklandıktan sonra derleme işlemi komut satırından şöyle yapılmaktadır:

```
gcc -o <çalıştırılabilen dosya ismi> <kaynak dosya ismi>
```

ya da :

```
clang -o <çalıştırılabilen dosya ismi> <kaynak dosya ismi>
```

Örneğin:

```
gcc -o sample sample.c
```

ya da örneğin:

```
clang -o sample sample.c
```

Eğer -o seçeneği kullanılmamışsa çalıştırılabilen dosyanın ismi "a.out" olacaktır.

gcc ve clang default durumda derleme sonrasında bağlayıcıyı (linker) çalıştırmaktadır. Bağlama işlemi bittikten sonra gcc oluşturulmuş olan amaç dosyayı (object file) da kendisi siler. UNIX/Linux sistemlerinde bağlama (linking) işlemi GNU projesi kapsamında geliştirilmiş olan "ld" isimli bağlayıcı programıylla yapılmaktadır. Aslında biz derleme ve bağlama işlemini ayrı ayrı iki aşamada da yapabiliriz. gcc ve clang derleyicilerinde -c seçeneği "yalnızca derle (only compile), fakat bağlama" anlamına gelmektedir. Biz bir C programını yalnızca derleyip ondan amaç dosya elde edebiliriz. Örneğin:

```
csd@csd-virtual-machine ~/Study/SysProg-2019 $ gcc -c sample.c
csd@csd-virtual-machine ~/Study/SysProg-2019 $ ls -l sample.o
-rw-r--r-- 1 csd study 1512 Şub  9 09:46 sample.o
csd@csd-virtual-machine ~/Study/SysProg-2019 $
```

Amaç dosyayı bağlamak için ld bağlayıcısı kullanılabilir. Ancak bu durumda bazı başlangıç dosyalarının (start-up object files) da bağlama işlemine dahil edilmesi gerekir. Biz bağlama işlemini de gcc (ya da clang ile) ile yapabiliriz. gcc aslında bu durumda arka planda ld bağlayıcı programını çalıştırmaktadır. Örneğin:

```
csd@csd-virtual-machine ~/Study/SysProg-2019 $ gcc -o sample sample.o
csd@csd-virtual-machine ~/Study/SysProg-2019 $ ./sample
This is a test
csd@csd-virtual-machine ~/Study/SysProg-2019 $
```

Bu biçimde gcc başlangıç dosyalarını da ld bağlayıcısına vererek bağlama işlemini ona yaptırmaktadır.

UNIX/Linux sistemlerinde bulunulan dizindeki bir programı komut satırından çalıştırabilmek için yalnızca dosyanın ismi yazılmaz. Onun dizini de belirtilmelidir. Tipik çalıştırma şöyle yapılır:

```
./sample
```

"." karakterinin "bulunulan dizini temsil ettiğini anımsayınız.

Örneğin:

```
csd@csd-virtual-machine ~/Study/SysProg-2019 $ sample
'sample' komutu bulunamadı, şunu mu demek istediniz:
 'yample' paketinden 'yample' komutu (universe)
 'simple' paketinden 'meryl' komutu (universe)
 'ample' paketinden 'ample' komutu (universe)
sample: komut bulunamadı
csd@csd-virtual-machine ~/Study/SysProg-2019 $ ./sample
This is a test
csd@csd-virtual-machine ~/Study/SysProg-2019 $
```

GCC derleyicisi pek çok sisteme port edilmiştir. GCC'nin Windows port'una MinGW denilmektedir. GCC ve Clang derleyicileri varsayılan durumda durumda eğer sistem 32 bit ise 32 bit derleme, 64 bit ise 64 bit derleme yapmaktadır. Örneğin makinemizdeki Linux sistemi 64 bit ise biz aşağıdaki gibi bir derlemeden 64 bit ELF formatına sahip bir çalıştırılabilir dosya elde ederiz:

```
gcc -o sample sample.c
```

64 bit Linux sistemlerinde 32 bit derleme yapmak için -m32 seçeneği kullanılmalıdır. Örneğin:

```
gcc -m32 -o sample sample.c
```

Pek çok 64 bit Linux sisteminde 32 bitlik derleme paketleri hazır olarak bulunmamaktadır. Bu nedenle 32 bit derleme için ek pakletlerin yüklenmesi gerekebilmektedir. Ubuntu türevi sistemlerde bu paketlerin yüklenmesi aşağıdaki komutla yapılabilir:

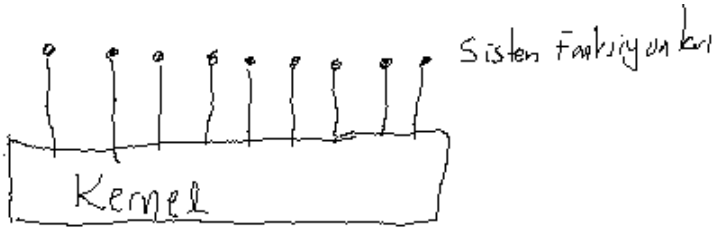
```
sudo apt-get install g++-multilib libc6-dev-i386
```

**İşletim Sisteminin Sistem Fonksiyonları, POSIX Fonksiyonları, Windows API Fonksiyonları ve Standart C Fonksiyonları**

İşletim sistemleri genellikle prosedürel teknik kullanılarak C Programlama Diliyle yazılmaktadır. İşletim sistemleri kabaca çekirdek ve kabuk kısımlarından oluşur.


Çekirdek (kernel) işletim sisteminin ana işlevlerini gerçekleştiren motor kısmıdır. Kabuk (shell) ise işletim sisteminin kullanıcıyla arayüz oluşturan kısmıdır. Örneğin Windows'ta masaüstü, UNIX/Linux sistemlerinde bash gibi komut satırı programları bu sistemlerin kabuk kısımlarını oluşturmaktadır. Tabii UNIX/Linux sistemlerinde de Windows'taki gibi grafik kabuk sistemleri de (pencere yöneticisi sistemler) bulunmaktadır.

İşletim sistemlerinin çekirdeklerinde binlerce fonksiyon bulunur. Bunların küçük bir kısmı dışarıdan da (kullanıcı modundan) önemli bazı işleri yapmak için çağrılabilmektedir. Bunlara sistem fonksiyonları (system call) denilmektedir. Her işletim sisteminin sistem fonksiyonlarının isimleri, parametrik yapıları farklı olabilmektedir. Biz de C Programcısı olarak bu sistem fonksiyonlarını doğrudan çağırabiliriz. Ancak her işletim sisteminin sistem fonksiyonları farklı olduğu için sistem fonksiyonları taşınabilir değildir.

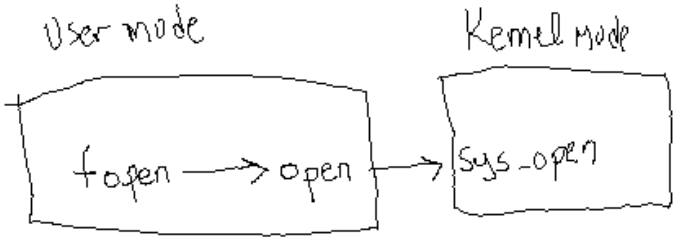



Programlarda sık karşılaşılan bazı işlemler aslında aşağı seviyede tamamen işletim sisteminin kontrolü altındadır. O faaliyetleri gerçekleştirmek isteyen herkes aslında eninde sonunda işletim sisteminin ilgili sistem fonksiyonunu çağırmak zorundadır. Örneğin bir dosyayı silmek için bir sistem fonksiyonu vardır. Kullandığımız dil ne olursa olsun, eninde sonunda dosya bu fonksiyonla silinir. Çünkü bunun başka yolu yoktur. Biz hangi dili, kütüphaneyi ya da ortamı (framework) kullanıyor olursak olalım. Bu işi yapmak için bize sunulan fonksiyonlar eninde sonunda işletim sisteminin dosyayı silen sistem fonksiyonunu çağırarak bu işi yaparlar.

POSIX standartları temelde hem kabuk komutlarını hem de C'den çağrılacak ortak fonksiyonları belirlemektedir. POSIX fonksiyonları UNIX türevi sistemlerdeki ortak fonksiyonlardır. POSIX fonksiyonları Linux gibi, BSD gibi, Solaris gibi hatta MAC OS X gibi sistemlerde aynı biçimde kullanılabilmektedir. Bazı POSIX fonksiyonları doğrudan o sistemdeki bir sistem fonksiyonu çağırır. Bazı POSIX fonksiyonları ise hiçbir sistem fonksiyonunu çağırmaz. Bazıları da birden fazla sistem fonksiyonunu çağırabilmektedir. Örneğin dosya açmak için open isimli bir POSIX fonksiyonu kullanılmaktadır. Bu fonksiyon Linux sistemlerinde sys_open isimli sistem fonksiyonunu çağırmaktadır.

Standart C fonksiyonları ise tüm C derleyicilerinde bulunan fonksiyonlardır. İşletim sistemi ne olursa olsun C derleyicileri bu standart C fonksiyonlarını bizim için hazır durumda bulundurmaktadır. Bu üç grup fonksiyon içerisinde şüphesiz en geniş taşınabilirliğe sahip olan standart C fonksiyonlarıdır.

Örneğin Linux sistemlerde bir standart C fonksiyonu olan fopen fonksiyonunu çağırmış olalım. Bu sistemlerde fopen fonksiyonu birtakım işlemlerden sonra open POSIX fonksiyonunu çağırmaktadır. open POSIX fonksiyonu da sys_open isimli sistem fonksiyonunu çağırır. Dosyanın açılması aslında sys_open isimli istem fonksiyonu tarafından yapılmaktadır.




POSIX fonksiyonları UNIX/Linux sistemlerinde standart C fonksiyonlarının bulunduğu kütüphane içerisine yerleştirilmiştir. Bu kütüphameye libc denilmektedir. Bu libc kütüphanesi GNU tarafından glibc ismiyle gerçekleştirilmiştir. Bu kütüphane gcc ile bağlama işlemi yapılırken otomatik biçimde bağlama sürecine katılmaktadır. Yani UNIX/Linux sistemlerinde POSIX fonksiyonlarını çağırmak için ek bir işlem yapmaya gerek yoktur. POSIX fonksiyonlarının prototipleri çeşitli başlık dosyaları içerisinde bulundurulmuştur. Pek çok POSIX fonksiyonunun prototipi <unistd.h> dosyası içerisinde bulunmaktadır. Ancak bu dosyanın dışında POSIX fonksiyonlarının prototiplerini barındıran pek çok başlık dosyası da vardır.

Pekiyi bazen işletim sisteminin sistem fonksiyonlarını doğrudan çağırmamız gerekebilir mi? Taşınabilirlik sağlamak için ortak özelliklere hitap etmek gerekmektedir. Yani örneğin Linux'ta olan fakat BSD'de olmayan bir özellik POSIX fonksiyonunun konusu olamaz. Çünkü POSIX fonksiyonları tüm UNIX türevi sistemler için düşünülmüştür. İşte biz bazen belirli bir sisteme özgü işlemler yapmak isteyebiliriz. Bunun için doğrudan o sistemin sistem fonksiyonlarını çağırmak zorunda kalabiliriz. Şimdi şöyle bir soru soralım: Linux için fopen mı, open mı, yoksa sys_open mı daha geniş olanaklara sahiptir? İşte Linux'un sys_open fonksiyonu Linux'a özgü yazılmıştır. open ise UNIX türevi tüm sistemleri hedef alacak biçimde tasarlanmıştır. Halbuki fopen fonksiyonu tüm sistemlerde olabilecek ortak özelliklere göre tanımlanmıştır.

Windows sistemlerinde genellikle sisteme yönelik birtakım işlemler için hazır biçimde bulunan fonksiyonlar vardır. Bunlara "Windows API Fonksiyonları" denilmektedir. Windows'un API fonksiyonları düzey olarak POSIX fonksiyonlarına benzetilebilir. Nasıl POSIX fonksiyonları tüm UNIX türevi sistemlerdeki ortak fonksiyonları betimliyorsa Windows'un API fonksiyonları da tüm Windows sistemlerinde kullanabileceğimiz ortak fonksiyonları betimlemektedir. (Bazı Windows API fonksiyonlarının belli bir Windows versiyonundan sonraki versiyonlarda kullanılabildiğini de belirtelim.) Windows'un API fonksiyonlarından bazıları Windows'un belli bir sistem fonksiyonunu doğrudan çağırabilmekte, bazıları birden fazla sistem fonksiyonunu çağırabilmekte, bazıları ise hiçbir sistem fonksiyonunu çağırmayabilmektedir. Windows'un API fonksiyonlarının pek çoğunun prototipi <Windows.h> isimli başlık dosyasındadır. Bu API fonksiyonlarının bulunduğu dinamik kütüphaneler (Kernel32.dll, User32.dll, Gdi32.dll gibi) Microsoft'un derleyicileri ve bağlayıcıları tarafından doğrudan işleme sokulurlar. Dolayısıyla Windows'un API fonksiyonlarını çağırmak için Microsoft derleyicilerinde <windows.h> dosyasını include etmek dışında yapılacak başka bir şey yoktur.

### UNIX/Linux Sistemlerinde Programların Komut Satırı Argümanları

UNIX/Linux dünyasında genellikle kabuk komutları birer çalıştırılabilen program biçimindedir. Bu komutların çeşitli seçenekleri komut satırı argümanları ile işleme sokulmaktadır. Örneğin dizin listesini elde etmek için "ls" komutu kullanılmaktadır. Ancak bu komut default durumda yalnızca dizindeki dosyaların isimlerini yazdırır. Fakat örneğin bu komut "-l" seçeneği ile kullanıldığında (long form) dosyaların yalnızca isimleri değil ayrıntılı bilgileri de yazdırılmaktadır. İşte genel olarak UNIX/Linux sistemlerinde programlar bu biçimde komut satırı argümanları alırlar.

UNIX/Linux sistemlerinde komut satırı argümanları için ağırlıklı olarak GNU stili kullanılmaktadır. Bu stilin anahtar noktaları şunlardır:

- GNU stilinde komut satırı argümanları üç biçimde bulunabilmektedir:

1) Seçeneksiz argümanlar: Bu argümanlarda '-' karakteri ile başlayan bir seçenek kullanılmaz. Örneğin:

```
cat sample.c
```

Burada "cat" prog ismini, "sample.c" ise seçeneksiz argümanı belirtmektedir.

2) Argümansız seçenekler: Bu tür argümanlar '-' karakteri ve onun yanında tek bir karakter ile belirtilirler. Örneğin:

```
ls -l
```

Burada "-l" argümansız bir seçeneği belirtmektedir. Birden fazla argümansız seçenek ayrı ayrı belirtilebileceği gibi tek bir '-' karakteri ile birlikte de belirtilebilir. Örneğin:

```
ls -l -i
```

ile:

```
ls -li
```

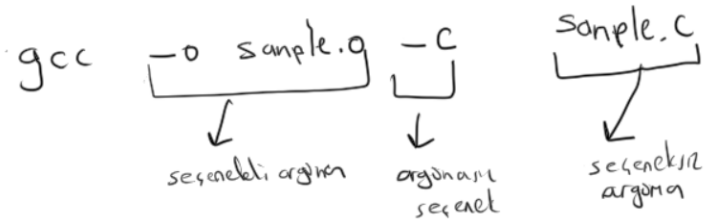
aynı anlamdadır. Eğer seçenek birden fazla karakterden oluşuyorsa bu durumda '-' yerine '--' kullanılmaktadır. Örneğin:

```
ls --version
```

Komut satırı seçeneklerinin büyük harf küçük harf duyarlılığı olduğuna dikkat ediniz. İki tane '-' karakteri ile uzun seçeneklendirme daha sonraları GNU stiline dahil edilmiştir. Klasik GNU stilinde eskiden yalnızca bir tane '-' karakteri kullanılıyordu.

3) Seçenekli Argümanlar: Bazı komut satırı argümanları hem seçenek hem de argüman alabilmektedir.

Örneğin:




Argümanlı seçeneklerde seçenek ile argüman bitişik de yazılabilir. Örneğin bu durumda gcc'nin -o seçeneği ile argümanı bitişik de yazılabilirdi:

```
gcc -osample.o -c sample.c
```

Genel olarak GNU stilinde komut satırı argümanları herhangi bir sırada girilebilir. Ancak bazı özel programlarda komut satırı argümanlarındaki sıra önemli olabilmektedir. Yine genel olarak programlar bir seçenek birden fazla kez girildiğinde bu durumu bir hata olarak değil o seçenek yalnızca bir kez girilmiş gibi ele almaktadırlar.

Komut satırı argümanlarının parse edilmesi programcılar için biraz sıkıntılı olabilmektedir. Bu nedenle programcılar bu işlemler için başkaları tarafından yazılmış çeşitli fonksiyonları ve sınıfları kullanabilmektedir. UNIX/Linux dünyasında getopt isimli standart POSIX fonksiyonu kısa seçenekler için (tek '-' karakterli seçenekler için) parse işlemini yapmaktadır. Bunun uzun seçenekler için ('--' ile başlayan seçenekler için) için kullanılan getopt_long isimli bir biçimi de vardır.

### getopt Fonksiyonunun Kullanımı

getopt yukarıda da belirtildiği gibi GNU stilindeki komut satırı argümanlarını ayrıştırmak için (parse etmek için) kullanılmaktadır. getopt bir POSIX fonksiyonudur. Bu nedenle örneğin Windows sistemlerinde bulunmamaktadır. getopt fonksiyonunun prototipi şöyledir:

```
#include <unistd.h>
```

```
int getopt(int argc, char **const argv, const char *optstring);
```

Fonksiyonun birinci parametresi komut satırı argümanlarının sayısını, ikinci parametresi ise komut satırı argümanlarının bulunduğu gösterici dizisinin adresini alır. Bu iki parametre tipik olarak main fonksiyonundan alınıp getopt fonksiyonuna verilmektedir. Fonksiyon kendi içerisinde bazı global değişkenleri kullanıp onları güncellemektedir. Bunların listesi şöyledir:

```
extern char *optarg;
extern int optind, opterr, optopt;
```

Bu değişkenler <unistd.h> içerisinde zaten extern biçimde bildirilmişlerdir. Dolayısla programcının ayrı bir extern bildirimi yapmasına gerek kalmamaktadır.

Fonksiyonun üçüncü parametresi tek karakterli seçenekleri belirtmektedir. Eğer karakterin yanında ':' karakteri varsa bu argümanlı seçenek anlamına gelir. Örneğin:

```
result = getopt(argc, argv, "abc:");
```

Burada -a ve -b argümansız seçenek -c ise argümanlı seçenektir. getopt bir döngü içerisinde çağrılmalıdır. getopt her çağrılmada kullanıcının girmiş olduğu bir seçeneği bize verir. getopy tüm seçenekleri bulduktan sonra işini bitirince -1 değerine geri döner. O halde fonksiyonun tipik kullanımı şöyle olmalıdır:

```
while ((result = getopt(argc, argv, "abc:")) != -1) {
    ...
}
```

Yukarıda da belirtiğimiz gibi getopt fonksiyonu her çağrıldığında kullanıcının girmiş olduğu bir seçenekle geri dönmektedir. Dolayısıyla tipik olarak getopt fonksiyonunun geri dönüş değeri switch içerisine alınarak işlenmelidir. getopt programcının üçüncü parametreyle belirlemediği bir seçenekle karşılaşırsa ya da argümanlı bir seçenekte argümanın girilmediğini görürse '?' karakteriyle geri döner. Tabii programcının bu tür hatalı girişleri kullanıcıya birer mesajla bildirmesi uygun olur. Aslında default durumda getopt hatalı giriş kontrolünü kendisi yapıp uygun hata mesajını stderr dosyasına kendisi mesaj olarak yazdırmaktadır. (Yani default durumda getopt hem hata mesajını stderr dosyasına yazar hem de '?' karakterine geri döner.) Ancak istersek bu hata yazdırma işlemini getopt'un otomatik olarak yapmasını engelleyebiliriz. İşte opterr isimli global değişken getopt fonksiyonunun hataları etderr dosyasına otomatik yazdırıp yazdırmayacağını belirlemek için kullanılmaktadır. Eğer işin başında opterr değişkenine 0 değeri atanırsa artık getopt hata durumlarında stderr dosyasına herhangi bir hata mesajı yazmaz. opterr değişkeninin işin balında sıfır dışı bir değere sahip olduğuna dikkat ediniz.

Argümanlı seçeneklerde seçenek bulunduğunda optarg isimli global char türden gösterici seçeneğin argümanını gösterecek biçimde (tabii argümanın sonu '\0' ile bitmektedir) ayarlanmaktadır. Programcı bu nedenle argümanlı bir seçenekle karşılaştığında bu argümanı bir göstericide saklamalıdır.

getopt fonksiyonu kullanılırken seçenekler için birer bayrak değişkeni tutulması ve bu bayrak değişkenlerinin seçenek belirlendiğinde set edilmesi uygun olur. Böylece programın belli bir noktalarında bu bayrak değişkenlerine bakılarak uygun işlemler yapılabilecektir. Örneğin:

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

int main(int argc, char **argv)
{
    int ch;
    char *carg;
    int aflag = 0, bflag = 0, cflag = 0;

    while ((ch = getopt(argc, argv, "abc:")) != -1) {
        switch (ch) {
        case 'a':
```

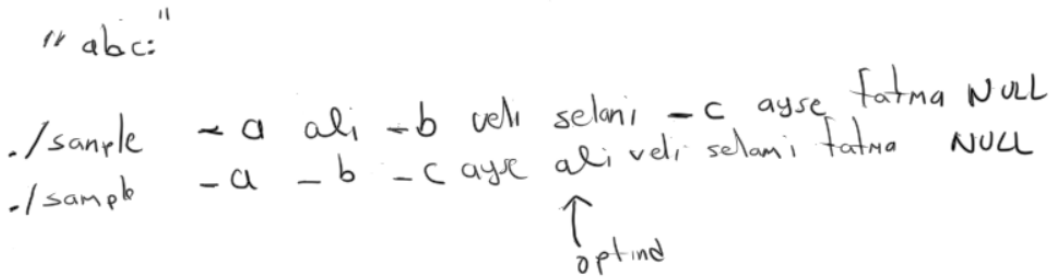
```
            aflag = 1;
            break;
        case 'b':
            bflag = 1;
            break;
        case 'c':
            carg = optarg;
            cflag = 1;
            break;
        }
    }

    if (aflag)
        printf("-a switch is given\n");

    if (bflag)
        printf("-b switch is given\n");

    if (cflag)
        printf("-c switch is given and its arument is: %s\n", carg);

    return 0;
}
```

getopt fonksiyonu çeşitli biçimlerde kullanılabilse de biz burada şöyle bir tavsiyede bulunacağız:

1) Fonksiyonda belirttiğiniz her argümanlı ya da argümansız seçenek için bir flag değişkeni, her argümanlı seçenek için de o argümanları gösteren bir gösterici bulundurun.

2) getopt döngüsü bittikten sonra da bu flag değişkenlerine bakarak istediğiniz işlemleri yapabilirsiniz.

getopt fonksiyonun her çağrıldığında bize bir seçeneği verdiğini söylemiştik. Pekiyi seçeneksiz argümanların yerlerini (yani normal argümanların yerlerini) nasıl bulacağız? İşte getopt aldığı argv dizisinin elemanlarını seçeneksiz argümanlar sonda kalacak biçimde yer değiştirmektedir. Seçeneksiz argümanların başladığı yerin indeksini de optind global değişken ile belirlemektedir. Örneğin:




Geçersiz bir seçenek girildiğinde getopt fonksiyonunun '?' karakteriyle geri döndüğünü söylemiştik. Pekiyi bu geçersiz seçeneğe ilişikin karakter nedir? İşte optopt isimli global değişken bu geçersiz seçeneği bize vermektedir. Benzer biçimde eğer bir seçenekli argümanın argümanı girilmemişse getopt bu durumda da '?' karakterine geri dönmektedir. Yine bu durumda da biz seçenek karakterinin kendisini optopt değişkeninden elde edebiliriz. Burada bir noktaya dikkatinizi çekmek istiyoruz. getopt fonksiyonunda seçenekleri "a:bc" biçiminde girmiş olalım ve kullanıcı da a seçeneği için aşağıdaki gibi argüman girmeyi unutmuş olsun:

```
./sample -a -b -c ali veli selami
```

Butada getopt -a seçeneğinin argümanını "-b" sanacaktır.

Tipik bir getopt kullanımı şöyle olabilir:

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

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

int main(int argc, char *argv[])
{
    int result;
    int a_flag = 0, b_flag = 0, c_flag = 0, d_flag = 0;
    char *c_arg, *d_arg;
    int i;

    opterr = 0;
    while ((result = getopt(argc, argv, "abc:d:")) != -1) {
        switch (result) {
        case 'a':
            a_flag = 1;
            break;
        case 'b':
            b_flag = 1;
            break;
        case 'c':
            c_flag = 1;
            c_arg = optarg;
            break;
        case 'd':
            d_flag = 1;
            d_arg = optarg;
            break;
        case '?':
            if (optopt == 'c' || optopt == 'd')
                fprintf(stderr, "-%c option given without an argument\n", optopt);
            else
                fprintf(stderr, "invalid switch: %c\n", optopt);
            exit(EXIT_FAILURE);
        }
    }

    printf("Arguments without switch:\n");
    for (i = optind; i < argc; ++i)
        puts(argv[i]);

    if (a_flag) {
        printf("-a is given\n");
    }
    if (b_flag) {
        printf("-b is given\n");
    }
    if (c_flag) {
        printf("-c is given, its argument: %s\n", carg);
    }
    if (d_flag) {
        printf("-d is given, its argument: %s\n", darg);
    }
    /* ... */

    return 0;
}
```

**Sınıf Çalışması:** Komut satırı argümanları alarak dört işlem yapan mycalc isimli programı aşağıda belirtildiği gibi yazınız:

Programın kullanımı şöyledir:

```
mycalc [-masd] <operand1> <operand2>
```

- Programın dört seçeneği şöyledir:

```
-m (multiply)
-a (add)
-s (subtract)
-d (divide)
```

Program iki argümanın dört işlem değerini alarak stdout dosyasına yazdırmalıdır.

### getopt_long Fonksiyonun Kullanımı

getopt_long fonksiyonu getopt fonksiyonunun uzun seçenekleri de dikkate alan yeni biçimidir. Uzun seçenekler -- ile başlatılır ve seçenek birden fazla karakterden oluşabilir. Uzun seçenekler de argümana sahip olabilirler. Uzun seçeneklerin argümanları boşlukla ayrılarak ya da '=' ile bitişik bir biçimde girilebilir. Örneğin:

```
./sample -a --color --country turkey --type=binary
```

Burada -a arümansız kısa seçenek, --color argümasız uzun seçenek, --country ve --type da argümanlı uzun seçeneklerdir. Uzun seçeneklerin isteğe bağlı olarak argüman alabilmesi de mümkündür. Tabii bu durumda uzun seçeneğin isteğe bağlı argümanları '=' ile girilmek zorundadır. (Çünkü isteğe bağlı argüman alan uzun seçeneklerde argüman boşlukla ayrılırsa bu durumda bu argümanın seçeneksiz argüman mı yoksa uzun seçeneğin argümanı mı olduğu anlaşılamaz.) Ancak önceki pragrafta da belirttiğimiz argümanlı uzun seçeneklerin argümanları hem ayrı biçimde hem de '=' ile bitişik biçimde girilebilir.

getopt_long fonksiyonu işlevsel olarak getopt fonksiyonunu kapsamaktadır. Fakat getopt_long fonksiyonunun kullanımı getopt fonksiyonunun kullanımından biraz daha zahmetlidir. Önce getopt_long fonksiyonunun prototipini inceleyelim:

```
#include <getopt.h>

int getopt_long(int argc, char *const argv[], const char *optstring,
           const struct option *longopts, int *longindex);
```

getopt_long fonksiyonunun ilk üç parametresi getopt fonksiyonu ile aynıdır. Fonksiyonun dördüncü parametresi struct option isimli bir yapı dizisinin adresini almaktadır. Bu yapı dizisi programcı tarafından uzun seçenekleri belirtmek amacıyla doldurulmaktadır (bu parametrenin const bir gösterici olduğuna dikkat ediniz. Programcının doldurduğu yapı dizisinin son elemanı 0'lardan oluşmalıdır. option yapısı şöyle bildirilmiştir:

```
struct option {
    const char *name;
    int has_arg;
    int *flag;
    int val;
};
```

option yapısnın name elemanı uzun seçeneğin yazısını belirtir.  has_arg elemanı üç değerden birini alabilmektedir:

```
#define no_argument             0
#define required_argument       1
#define optional_argument       2
```

no_argument uzun seçeneğin argüman almadığını, required_argument aldığını belirtmektedir. optional_argument ise uzun seçeneğin isteğe bağlı olarak (optional) argüman alabileceği anlamına gelir. Eğer yapının has_arg elemanı optional_argument olarak girilmişse isteğe bağlı argümanın seçenek=argüman biçiminde '=' ile girilmesi gerekmektedir. Uzun seçeneklerin argümanlar ıyine getopt fonksiyonunda olduğu gibi optarg global değişkeninden elde edilmektedir. Eğer optional_argument için kullanıcı girişte argüman belirtilmemişse bu durumda optarg global değişkenine NULL adres yerleştirilir.

Yapının flag elemanı int türden bir nesnenin adresini almaktadır. Bu eleman NULL adres olarak geçilebilir. Eğer bu eleman NULL adres olarak geçilmemişse getopt_long fonksiyonu girişte ilgili seçenek belirtidiğinde bu adresin gösterdiği nesneye yapının val elemanındaki değeri atar ve 0 ile geri döner. Yapının bu flag elemanına NULL adres atanırsa

getopt_long doğrudan yapının val elemanı ile belirtilen değere geri dönmektedir. Özellikle kısa seçenekle eşdeğer olan uzun seçenekler söz konusu olduğunda programcı yapının bu flag elemanına NULL değerini, val elemanına da kısa seçeneğin karakter değerini yerleştirip durumu daha kolay ele alabilmektedir.

getopt_long fonksiyonunun son parametresi int türden bir nesnenin adresini almaktadır. getopt_long bu adresin gösterdiği nesneye o anda bulduğu uzun seçeneğin option yapı dizisindeki indeksini yerleştirir. Eğer getopt_long uzun değil kısa seçenek bulmuşsa ya da bir hata durumuyla karşılaşmışsa (örneğin argüman girilmemiş bir uzun seçenek) bu nesneye herhangi bir değer yerleştirmez (yani bu durumda buradaki değeri güncellememektedir.) Bu son parametreye genellikle pek gereksinim duyulmamaktadır. Eğer bu parametre kullanılmayacaksa argüman olarak NULL adres geçilebilir.

getopt_long fonksiyonunun geri dönüş değerinin beş biçimde olabileceğine dikkat ediniz:

1) Eğer getopt_long komut satırı argüman incelemesinin sonuna geldiyse -1 değerine geri döner.

2) Eğer getopt_long uzun bir seçeneği bulduysa ve o seçeneğe ilişkin option yapı nesnesinin flag elemanı NULL ise bu durumda o yapı nesnesinin val elemanındaki değerle geri döner.

3) Eğer getopt_long uzun bir seçeneği bulduysa ve o seçeneğe ilişkin yapı nesnesinin flag elemanı NULL değilse fonksiyon yapı nesnesinin val elemanındaki değeri flag elemanı ile belirtilen int nesneye yerleştirir ve sıfır ile geri döner.

4) Eğer getopt_long kısa bir seçeneği bulduysa tıpkı getopt gibi ilgili seçenek karakterinin sayısal değerine geri döner.

5) Eğer getopt_long fonksiyonu olmayan bir seçenek ile karşılaşırsa ya da argümanlı bir seçenekle karşılaştığı halde argüman belirtilmediyse tıpkı getopt fonsiyonunda olduğu gibi '?' karakterine geri dönmektedir. Uzun bir seçeneğin argümanı olmadığında fonksiyon '?' karakteri ile geri döndüğü zaman programcı yine optopt değişkenine bakarak hangi uzun seçeneğin argümanının girilmediğini anlayabilir. Bu durumda getopt_long fonksiyonu optopt değişkenine option yapısının val elemanındaki değeri atamaktadır.

getopt_long fonksiyonunda olmayan bir uzun seçenek girildiğinde bunu elde etmenin dokümante edilmiş bir yolu yoktur. (optopt değişkeninin char türdne olduğuna ve bu amaçla kullanılamayacağına dikkat ediniz.) Ancak GNU kaynak kodlarına bakıldığında olmayan bir seçeneğin bulunduğu durumda bu seçeneğin argv[optind - 1] adresinde olduğu görülmektedir.

getopt_long için değişik kullanımları içeren aşağıdaki örneği verebiliriz:

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

int main(int argc, char *argv[])
{
    int result;
    int a_flag = 0, b_flag = 0, c_flag = 0, d_flag = 0, ccc_flag = 0, bbb_flag = 0;
    char *c_arg, *d_arg, *ccc_arg;
    int option_index;
    struct option options[] = {
        { "aaa", no_argument, NULL, 'a' },
        { "bbb", no_argument, &bbb_flag, 1 },
        { "ccc", required_argument, NULL, 2},
        { 0, 0, 0, 0 }
    };
    int i;

    opterr = 0;
    while ((result = getopt_long(argc, argv, "abc:d:", options, &option_index)) != -1) {
        switch (result) {
            case 2:
```

```
                if (option_index == 2) {
                    ccc_flag = 1;
                    ccc_arg = optarg;
                }
                break;
            case 'a':
                a_flag = 1;
                break;
            case 'b':
                b_flag = 1;
                break;
            case 'c':
                c_flag = 1;
                c_arg = optarg;
                break;
            case 'd':
                d_flag = 1;
                d_arg = optarg;
                break;
            case '?':
                if (optopt == 'c' || optopt == 'd')
                    fprintf(stderr, "-%c option given without an argument\n", optopt);
                else if (optopt == 2)
                    fprintf(stderr, "--%s option given without an argument\n",
options[optopt].name);
                else
                    fprintf(stderr, "invalid switch!\n");
                exit(EXIT_FAILURE);
        }
    }

    if (argv[optind] != NULL) {
        printf("Arguments without switch:\n");
        for (i = optind; i < argc; ++i)
            puts(argv[i]);
    }

    if (a_flag) {
        printf("-a or --aaa is given\n");
    }
    if (b_flag) {
        printf("-b is given\n");
    }
    if (c_flag) {
        printf("-c is given, its argument: %s\n", c_arg);
    }
    if (d_flag) {
        printf("-d is given, its argument: %s\n", d_arg);
    }
    if (bbb_flag) {
        printf("--bbb is given\n");
    }
    if (ccc_flag) {
        printf("--ccc is given, its argument: %s\n", ccc_arg);
    }

    /* ... */

    return 0;
}
```

Bu örnekte kısa seçenekler yine "abc:d:" argümanı kullanılmıştır. Fakat uzun seçenekler için aşağıdaki yapı organize edilmiştir:

```
struct option options[] = {
    { "aaa", no_argument, NULL, 'a' },
    { "bbb", no_argument, &bbbflag, 1 },
    { "ccc", required_argument, NULL, 0 },
    { 0, 0, 0, 0 }
};
```

Bu yapıda birinci uzun seçeneğin ismi "aaa" olduğu görülmektedir. Bu uzun seçenek argüman almaz. Birinci seçeneğin flag değişkenine NULL adres atandığını görüyorsunuz. Dolayısıyla flag değişkeni getopt_long fonksiyonunun kendisi tarafından set edilmeyecektir. getopt_long birinci uzun seçenek için 'a' değeri ile geri dönecektir. Görüldüğü gibi -a kısa seçeneğinin eşdeğeri --aaa biçimindedir. Bir kısa seçeneğin eşdeğer uzun seçeneği olmasına GNU stili ile yazılan programlarda çok sık karşılaşılmaktadır. Yapının ikinci elemanında uzun seçeneğin "bbb" isminde olduğuna ve bunun da argüman almadığına dikkat ediniz. Bu seçenek belirtildiğinde yapının val elemanı içerisindeki 1 değeri bbbflag değişkenine atanacak ve getopt_long 0 ile geri dönecektir. Yapının üçüncü elemanı argüman almaktadır. İlgili argümanın değeri (ismi) getopt fonksiyonunda olduğu gibi yine optarg göstericisinden elde edilir.

getopt_long fonksiyonu da çeşitli biçimlerde kullanılabilse de biz burada yine şöyle bir tavsiyede bulunacağız:

1) Fonksiyonda belirttiğiniz her argümanlı ya da argümansız seçenek için bir flag değişkeni, her argümanlı seçenek için de o argümanları gösteren bir gösterici tutabilirisiniz.

2) Hem uzun hem de kısa biçimi olan seçeneklerde yapının flag elemanı NULL değerinde tutulup fonksiyonun kısa seçenek ile geri dönmesi sağlanabilir.

3) Yalnızca uzun biçimi olan seçeneklerde yapının flag elemanına doğrudan o seçenek için tanımladığınız flag değişkeninin adresini atayabilirsiniz.

4) Fonksiyondan çıktıktan sonra switch içerisinde flag değişkenlerinin durumuna bakabilirsiniz.

Şimdi bir ASCII dosyayı çeşitli biçimlerde ekrana yazdıran mycat isimli bir program yazmak isteyelim. Programımızın kullanımı şöyle olsun:

```
mycat [seçenekler] [dosyalar]
```

Seçenekler:

```
-t  (default)
-o
-x
--top [n], default n = 10
--header
```

Burada -t "text olarak yazdır", -o "ocatal olarak yazdır, -x "hex olarak yazdır anlamına gelmektedir. Bu seçeneklerden yalnızca bir tanesi belirtilebilir ve bu seçenklerden hiçbiri belirtilmemişse -t seçeneği belirtilmiş gibi işlem yapılmalıdır. --top isteğe bağlı (optional) argüman alabilen uzun bir seçenektir. Dosyanın başındaki ilk n satırı yazdırır. Bu uzun seçeneğin default değeri 10'dur. --heder seçeneği birden fazla dosyanın yazdırıldığı durumda dosya isimlerinin de basılmasını sağlamaktadır. Programın örnek bir gerçekleştirim şöyle olabilir:

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

/* Symbolic Constans */

#define DEF_LINE                10
#define HEX_OCTAL_LINE_LEN      16
```

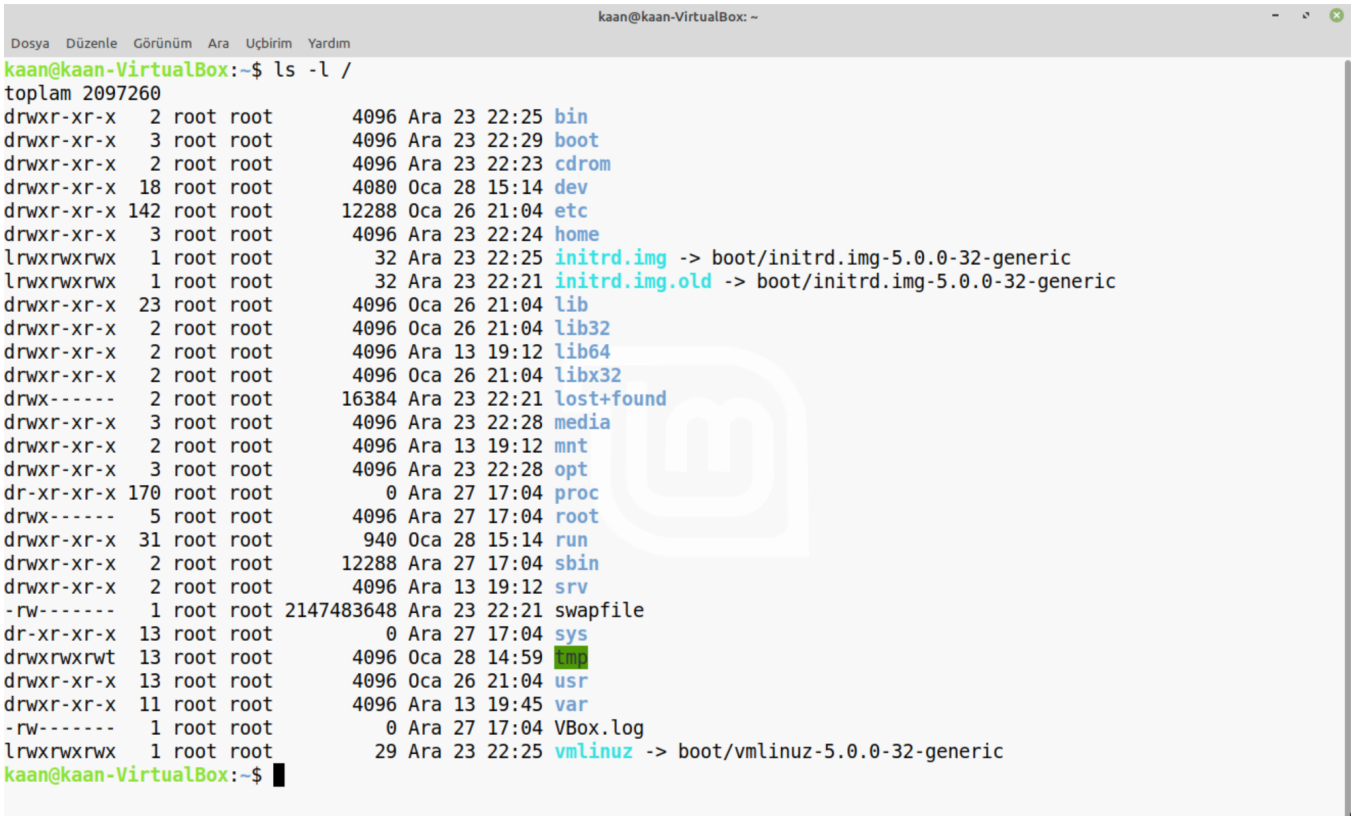
```c
/* Function Prorotypes */

int print_text(FILE *f, int nline);
int print_hex_octal(FILE *f, int nline, int hexflag);

int main(int argc, char *argv[])
{
    int result, err_flag = 0;
    int x_flag = 0, o_flag = 0, t_flag = 0, top_flag = 0, header_flag = 0;
    char *top_arg;
    struct option options[] = {
        {"top", optional_argument, NULL, 1},
        {"header", no_argument, NULL, 'h'},
        {0, 0, 0, 0}
    };
    FILE *f;
    int i, nline = -1;

    opterr = 0;
    while ((result = getopt_long(argc, argv, "xoth", options, NULL)) != -1) {
        switch (result) {
            case 'x':
                x_flag = 1;
                break;
            case 'o':
                o_flag = 1;
                break;
            case 't':
                t_flag = 1;
                break;
            case 'h':
                header_flag = 1;
                break;
            case 1:
                top_flag = 1;
                top_arg = optarg;
                break;
            case '?':
            if (optopt != 0)
                fprintf(stderr, "invalid switch: -%c\n", optopt);
            else
                fprintf(stderr, "invalid switch: %s\n", argv[optind - 1]);  /* argv[optind - 1]
dokümante edilmemiş */
            err_flag = 1;
        }
    }

    if (err_flag)
        exit(EXIT_FAILURE);

    if (x_flag + o_flag + t_flag > 1) {
        fprintf(stderr, "only one option must be specified from -o, -t, -x\n");
        exit(EXIT_FAILURE);
    }
    if (x_flag + o_flag + t_flag == 0)
        t_flag = 1;

    if (top_flag)
        nline = top_arg != NULL ? (int) strtol(top_arg, NULL, 10) : DEF_LINE;

    if (optind == argc)  {
        fprintf(stderr, "at least one file must be specified!..\n");
```

```
        exit(EXIT_FAILURE);
    }

    for (i = optind; i < argc; ++i) {
        if ((f = fopen(argv[i], "rb")) == NULL) {
            fprintf(stderr, "cannot open file: %s\n", argv[i]);
            continue;
        }

        if (header_flag)
            printf("%s\n\n", argv[i]);

        if (t_flag)
            result = print_text(f, nline);
        else if (x_flag)
            result = print_hex_octal(f, nline, 1);
        else
            result = print_hex_octal(f, nline, 0);

        if (i != argc - 1)
            putchar('\n');

        if (!result)
            fprintf(stderr, "cannot read file: %s\n", argv[i]);

        fclose(f);
    }

    return 0;
}

int print_text(FILE *f, int nline)
{
    int ch;
    int count;

    if (nline == -1)
        while ((ch = fgetc(f)) != EOF)
            putchar(ch);
    else {
        count = 0;
        while ((ch = fgetc(f)) != EOF && count < nline) {
            putchar(ch);
            if (ch == '\n')
                ++count;
        }
    }

    return !ferror(f);
}

int print_hex_octal(FILE *f, int nline, int hexflag)
{
    int ch, i, count;
    const char *off_str, *ch_str;

    off_str = hexflag ? "%07X " : "%012o";
    ch_str = hexflag ? "%02X%c" : "%03o%c";

    if (nline == -1)
        for (i = 0; (ch = fgetc(f)) != EOF; ++i) {
            if (i % HEX_OCTAL_LINE_LEN == 0)
                printf(off_str, i);
```

```
            printf(ch_str, ch, i % HEX_OCTAL_LINE_LEN == HEX_OCTAL_LINE_LEN - 1 ? '\n' : ' ');
        }

    else {
        count = 0;
        for (i = 0; (ch = fgetc(f)) != EOF && count < nline; ++i) {
            if (i % HEX_OCTAL_LINE_LEN == 0)
                printf(off_str, i);
           printf(ch_str, ch, i % HEX_OCTAL_LINE_LEN == HEX_OCTAL_LINE_LEN - 1 ? '\n' : ' ');
            if (ch == '\n')
                ++count;
        }
    }

    if (i % HEX_OCTAL_LINE_LEN != 0)
        putchar('\n');

    return !ferror(f);
}
```

**Linux Sistemlerinde Temel Dizin Yapısı**

UNIX türevi sistemlerin dizin yapısı birbirlerine çok benzemektedir. Biz burada bu dizin yapısı üzerinde bazı önemli dizinler hakkında temel açıklamalar yapacağız. Aşağıda Mint dağıtımının kök dizin listesini görüyorsunuz:




Linux Foundation isimli grup UNIX türevi sistemlerdeki dizin yapısını standardize etmeye çalışmıştır. Bu standarda "File System Hiearchy Standard" denilmektedir. Son standartlar 3.0 numarasıyla 2015 yılında oluşturulmuştur.

/bin: Burada kabuk komutlarına ilişin çalıştırılabilir dosyalar ve çeşitli utility programlar bulunur.

/sbin: Sisteme ilişkin aşağı seviyeli çalıştırılabilir dosyalar ve utility'ler bu dizinde bulunmaktadır. Örneğin sistemin boot edilmesi için gereken dosyalar buradadır. Genel olarak /sbin içerisindeki dosyalar normal kullanıcılar için değil sistem yöneticileri içindir.

/lib: /bin ve /sbin içerisinde bulunan programların kullandığı kütüphaneler burada tutulmaktadır.

/boot: Bu dizinde "boot loader" (örneğin grub) ve bazı çekirdek dosyaları (kernel image) bulunur.

/dev: Bu dizinde aygıt sürücülere ilişkin aygıt dosyaları bulunmaktadır. Aygıt dosyalarının ne anlam ifade ettiği kursumuzda ele alınmaktadır.

/etc: Bu dizinin ismi İngilizce "vesaire" anlamına gelen "etcetera" dan kısaltılmıştır. İlk yıllarda bu dizin diğer dizinlerin içerisinde olamayacak her şeyi içeriyordu. Ancak sonraki yıllarda içeriği biraz daha belirginleşmeye başlamıştır. Bu dizinde genel olarak konfigürasyon bilgileri tutulmaktadır. Bu nedenle "etc" ismi de yeniden "editable text configuration" biçiminde isimlendirilmek istenmiştir.

/home: Kullanıcılar için ayrılan dizinler. Normal olarak her kullanıcının bu dizin altında kullanıcı ismine ilişkin bir dizini vardır. Örneğin:

```
kaan@kaan-VirtualBox:~$ ls -l /home
toplam 4
drwxr-xr-x 23 kaan kaan 4096 Ara 27 17:05 kaan
```

/mnt: Kullanıcaların mount işlemi için kullanabilecekleri genel bir dizindir.

/media: Bu dizin çıkarılabilir aygıtların (CDROM gibi, flash EPROM gibi) mount edildiği ana dizindir.

/root: Bu dizin root kullanıcısı için home dizini görevindedir.

/usr: Kullanıcıların yerleştirdiği ya da kurduğu tüm yazılımlara ilşin çalıştırılabilir dosyalar, kütüphaneler ve başlık dosyaları ve diğer birtakım bilgiler bu dizinde tutulmaktadır. Bu dizinin altında da pek çok dizin vardır. Örneğin kullanılan dağıtımların kendi utility programları genellikle /usr/bin dizininde bulundurulur. /usr/local dizini server sistemlerinde lokal makinedeki programları saklamak için düşünülmüştür. Bunun altında da /usr/local/bin dizini bulunmaktadır. /usr/local/bin dizini kullanıcıların programlarını yerleştireceği tipik dizindir.

/var: Bu dizin log dosyaları gibi sistemin çalışması sırasında sürekli güncellenen dosyaların tutulduğu ana dizindir. Bu dizinin altında da pek çok dizin vardır.

/sys: Aygıt sürücülerin, ve çekirdeğe ilişkin çeşitli dosyaların bulunduğu dizindir.

/temp: Geçici dosyalar için bulundurulan bir dizindir. Genellikle sistem kapatılırken silinmektedir.

### Proses Kavramı ve Proseslerin Kontrol Blokları

"Program" kaynak kodlar için ya da çalıştırılabilen dosyalar için kullanılan bir terimdir. Bir program çalıştırıldığında ona artık proses (process) denilmektedir. Yani proses çalışmakta olan programlar için kullanılan bir terimdir.

Bir proses yaratıldığında (yani bir program çalıştırıldığında) işletim sistemi onu izlemek için çekirdek alanında bir veri yapısı oluşturur. Bu veri yapısına kavramsal olarak "Proses Kontrol Bloğu (Process Control Block)" denilmektedir. Örneğin Linux kaynak kodlarında proseslerin kontrol blokları "/include/linux" dizinindeki "task_struct" isimli yapıyla temsil edilmektedir. Pekiyi Proces Kontrol Bloğunda hangi bilgiler saklanmaktadır? İşte tipik bazı bilgiler şunlardır:

- Prosesin o anki durumu
- Prosesin erişim hakları
- Prosesin bellek alanı ile ilgili bilgiler
- Prosesin çizelgelemeyle ilgili bilgileri
- Prosesin çeşitli istatistiksel bilgileri
- Prosesin açmış olduğu dosyaların kayıtları
- Prosesin çalışma dizini (current working directory)

- ...

Proses terimi ile task terimi pek çok bağlamda aynı anlamda kullanılmaktadır. (Fakat bazı sistemlerde bu iki sözcük arasında bazı arklar söz konusu olabilmektedir. Fakat genel olarak bu iki terimi eşdeğer kabul edebiliriz.) Proses Kontrol Bloğu çekirdek alanı içerisinde tutulmaktadır. Böylece bu veri yapısına "kullanıcı modunda (user mode)" çalışan sıradan prosesler erişemezler. Şüphesiz işletim sistemi tüm Proses Kontrol Bloklarını birbirlerine bağlı listelerle bağlamıştır. Böylece işletim sistemi istediği zaman bu listeyi dolaşarak prosesler üzerinde kontrol işlemlerini yapabilir. Prosesler sonlandığında (istemli bir biçimde ya da zorla) işletim sistemi prosesin tutmuş olduğu kaynakları boşaltır ve proses bloğunu da yok eder.

Aslında Proses Kontrol Bloğunu içerisinde göstericilerin de bulunduğu bir ağaç gibi (daha doğru bir deyişle graf gibi) düşünebiliriz. Yani proses kontrol bloğunun içerisindeki bazı elemanlar başka birtakım yapıları gösteriyor olabilir. O yapılar da başka yapıları gösteriyor olabilir. Biz kursumuzda bir bilginin Proses Kontrol Bloğunda olduğunu söylediğimizde o bilginin doğudan ya da dolaylı olarak Proses Kontrol Bloğu Yoluyla erişilebilecek bir yerde olduğunu kastetmiş olacağız. Aşağıda Linux'un 2.6 .11 çekirdek sürümündeki task_struct yapısını görüyorsunuz:

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;    /* per process flags, defined below */
    unsigned long ptrace;

    int lock_depth;     /* Lock depth */

    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;

    unsigned long sleep_avg;
    unsigned long long timestamp, last_ran;
    int activated;

    unsigned long policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;

#ifdef CONFIG_SCHEDSTATS
    struct sched_info sched_info;
#endif

    struct list_head tasks;
    /*
     * ptrace_list/ptrace_children forms the list of my children
     * that were stolen by a ptracer.
     */
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;

/* task state */
    struct linux_binfmt *binfmt;
    long exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */
    /* ??? */
    unsigned long personality;
    unsigned did_exec:1;
    pid_t pid;
    pid_t tgid;
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively.  (p->father can be replaced with
     * p->parent->pid)
     */
```

```
    struct task_struct *real_parent; /* real parent process (when being debugged) */
    struct task_struct *parent; /* parent process */
    /*
     * children/sibling forms the list of my children plus the
     * tasks I'm ptracing.
     */
    struct list_head children;  /* list of my children */
    struct list_head sibling;   /* linkage in my parent's children list */
    struct task_struct *group_leader;   /* threadgroup leader */

    /* PID/PID hash table linkage. */
    struct pid pids[PIDTYPE_MAX];

    struct completion *vfork_done;      /* for vfork() */
    int __user *set_child_tid;      /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid;        /* CLONE_CHILD_CLEARTID */

    unsigned long rt_priority;
    unsigned long it_real_value, it_real_incr;
    cputime_t it_virt_value, it_virt_incr;
    cputime_t it_prof_value, it_prof_incr;
    struct timer_list real_timer;
    cputime_t utime, stime;
    unsigned long nvcsw, nivcsw; /* context switch counts */
    struct timespec start_time;
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
    unsigned long min_flt, maj_flt;
/* process credentials */
    uid_t uid,euid,suid,fsuid;
    gid_t gid,egid,sgid,fsgid;
    struct group_info *group_info;
    kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
    unsigned keep_capabilities:1;
    struct user_struct *user;
#ifdef CONFIG_KEYS
    struct key *session_keyring;    /* keyring inherited over fork */
    struct key *process_keyring;    /* keyring private to this process (CLONE_THREAD) */
    struct key *thread_keyring; /* keyring private to this thread */
#endif
    int oomkilladj; /* OOM kill score adjustment (bit shift). */
    char comm[TASK_COMM_LEN];
/* file system info */
    int link_count, total_link_count;
/* ipc stuff */
    struct sysv_sem sysvsem;
/* CPU-specific state of this task */
    struct thread_struct thread;
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespace */
    struct namespace *namespace;
/* signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    struct sigpending pending;

    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);
    void *notifier_data;
    sigset_t *notifier_mask;

    void *security;
    struct audit_context *audit_context;

/* Thread group tracking */
```

```
    u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
    spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock, write_lock_irq(&tasklist_lock); */
    spinlock_t proc_lock;
/* context-switch lock */
    spinlock_t switch_lock;

/* journalling filesystem info */
    void *journal_info;

/* VM state */
    struct reclaim_state *reclaim_state;

    struct dentry *proc_dentry;
    struct backing_dev_info *backing_dev_info;

    struct io_context *io_context;

    unsigned long ptrace_message;
    siginfo_t *last_siginfo; /* For ptrace use.  */
/*
 * current io wait handle: wait queue entry to use for io waits
 * If this thread is processing aio, this points at the waitqueue
 * inside the currently handled kiocb. It may be NULL (i.e. default
 * to a stack based synchronous wait) if its doing sync IO.
 */
    wait_queue_t *io_wait;
/* i/o counters(bytes read/written, #syscalls */
    u64 rchar, wchar, syscr, syscw;
#if defined(CONFIG_BSD_PROCESS_ACCT)
    u64 acct_rss_mem1;  /* accumulated rss usage */
    u64 acct_vm_mem1;   /* accumulated virtual memory usage */
    clock_t acct_stimexpd;  /* clock_t-converted stime since last update */
#endif
#ifdef CONFIG_NUMA
    struct mempolicy *mempolicy;
    short il_next;
#endif
};
```

Bu yapıdan bir şey anlamazsanız kendinizden şüphelenmeyin lütfen. Çekirdek kodlarının anlaşılması sandığınız kadar kolay bir şey değil.

### Proseslerin ID Değerleri

UNIX/Linux sistemlerinde her prosesin sistem genelinde tek olan (unique) bir proses id değeri vardır. Proses id değeri çekirdek tarafından proses kontrol bloğuna erişmek için bir handle değeri olarak kullanılmaktadır. Bazı sistem fonksiyonları parametre olarak prosesin id değerini alarak proses kontrol bloğuna erişmektedir.

Genel olarak UNIX/Linux çekirdekleri her yaratılan prosese artan sırada bir id numarası verirler. Sistemdeki ilk proses boot kodundan gelerek proses olan ve 0'ıncı id'ye sahip "swapper" prosesidir. İşletim sistemi belleğe yüklenip ilkdeğerlendiği (zaman initialize edildiği zaman) artık "init" prosesi denilen 1 numaralı id'ye sahip proses yaratılır. init prosesi sistemin çalışması için çok önemlidir. init prosesinden hareketle diğer prosesler yaratılarak proses id'ler hızla artmaktadır.

UNIX/Linux sistemlerinde her yaratılan prosese artan sırada bir id veridiğine göre aylarca hatta yıllarca açık kalan server makinelerde proses id sayıları tükenmez mi? İşte prosesler sürekli hayatta kalmazlar. Pek çoğu işini bitirip sonlanmaktadır. Böylelikle sonlanan proseslerin kullandığı proses id değerleri de sisteme iade edilirler. İşletim sistemi çekirdeği maksimum proses id sayısına ulaştığında yeniden başa döner ve orada sonlanmış olan proseslerin id değerlerini yeni prosesler için yeniden kullanır. Yani başka bir deyişle aynı proses id farklı zamanlarda farklı prosesler için kullanılıyor olabilir. Ancak belli bir anda her zaman tüm proseslerin proses id'leri tektir. Bugün kullandığımız Linux çekirdeklerinde default olarak maksimum proses id değeri 32767'dir. Bu değer 32 bit Linux sistemlerinde yükseltilememektedir. Ancak

64 bit Linux sistemlerinde sistem yöneticisi bu değeri $2^{22}$'ye kadar yükseltebilmektedir. değiştirebilir. Sisteminizdeki maksimum proses id değerini proc dosya sistemi sayesinde şöyle öğrenebilirsiniz:

```
cat /proc/sys/kernel/pid_max
```

Pekiyi mademki proseslerin id değerleri bir tamsayı ile temsil edilmektedir, o halde işletim sistemi çekirdeği kendisine bir proses id verildiğinde o id'ye ilişkin prosesin proses kontrol bloğunu nasıl bulmaktadır? İşte UNIX türevi sistemler genellikle bunun için bir hash tablosu kullanmaktadır. Bu hash tablosunda anahtar prosesin id'si değer ise prosesin kontrol bloğunu gösteren adrestir. Aşağıda Linux'ubn 2.6 çekirdeğinde proses id'ye karşılık gelen proses kontrol blok adresinin (task_struct) nasıl bulunduğunu görüyorsunuz:

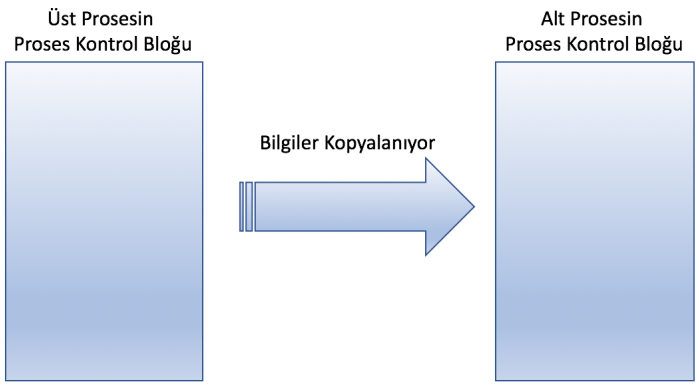
```
static inline struct task_struct *find_task_by_pid(int pid)
{
        struct task_struct *p, **htable = &pidhash[pid_hashfn(pid)];

        for(p = *htable; p && p->pid != pid; p = p->pidhash_next)
                ;

        return p;
}
```

Burada hash tablosu olarak "ayrı zincir oluşturma (seperate chaining)" yöntemi kullanılmıştır. pidhash task_struct adreslerinin oluşturduğu bir dizidir. Gördüğünüz gibi prosesin id değeri hash fonksiyonuna sokulup o hash değerine ilişkin bağlı liste elde edilmiş sonra da bu bağlı listede dolaşım uygulanmıştır.

### Prosesler Arasında Altlık-Üstlük (Parent-Child) İlişkisi

UNIX/Linux sistemlerinde tüm prosesler işletim sisteminin sistem fonksiyonuyla yaratılır ve genellikle işletim sisteminin bir sistem fonksiyonuyla da sonlandırılır. (Proses yaratmak için fork isimli POSIX fonksiyonu kullanılmaktadır. fork POSIX fonksiyonu Linux sistemlerinde sys_fork isimli sistem fonksiyonunu çağırır. Benzer biçimde prosesin sonlandırılması _exit POSIX fonksiyonuyla yapılmaktadır. Linux sistemlerinde bu fonksiyon sys_exit sistem fonksiyonunu çağırmaktadır.)

Prosesler arasında altlık-üstlük (parent-child) ilişkisi de vardır. Prosesi yaratan prosese üst-proses (parent process), yeni oluşturulan prosese de alt-proses (child process) denilmektedir. UNIX/Linux sistemlerinde prosesler arasındaki altlık-üstlük ilişkisi çok belirgindir. Alt prosesin yaratımı sırasında üst prosesin kontrol bloğundaki pek çok bilgi alt prosesin kontrol bloğuna aktarılmaktadır. Böylece alt proses de pek çok bakımdan (örneğin yetki bakımdan, yetenek bakımından, çalışma dizini bakımından vs.) üst prosesle aynı özelliklere sahip olur.

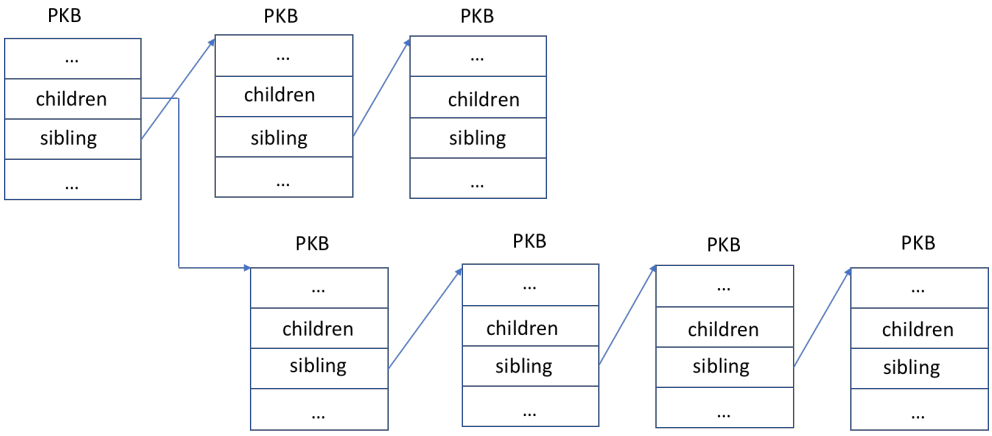



fork işlemi sırasında (yani alt prosesin yaratımı sırasında) prosesin hangi bilgilerinin alt prosese aktarıldığını ileride çeşitli konularda göreceksiniz.

UNIX türevi sistemlerde proseslerin kontrol blokları altlık-üstlük ilişkisini de içerecek biçimde bağlı listelerle birbirlerine bağlanmıştır. Örneğin Linux çekirdeklerinde bu task_struct yapısındaki children ve sibling elemanları bu amaçla kullanılmıştır:

```
struct task_struct {
    /* ... */
    struct list_head children;  /* list of my children */
    struct list_head sibling;   /* linkage in my parent's children list */
    /* ... */
};
```

Buradaki children prosesin alt proses listesini, sibling ise aynı üst prosese bağlı olan proseslerin (yani kardeş prosesleri) listesini tutmaktadır. Bu bağlı listelerin nasıl işlev gördüğü aşağıdaki şekilde açıklanmaktadır:




Buradaki şekil yalnızca bir fikir versin diye çizilmiştir. Bu nedenle Linux bağlı listelerindeki bazı ayrıntılar göz ardı edilmiştir. (Örneğin aslında çekirdekteki bu bağlı listeler "çift (double)" bağlı listedir.)

Linux sistemlerinde boot kodundan gelen 0 numaralı id'ye sahip olan proses 1 numaralı id'ye sahip olacak init prosesini yaratmaktadır. Sonra init prosesi de diğer prosesleri, diğer prosesler de diğer prosesleri yaratarak sistemde çok sayıda proses yaratılmış olur.

UNIX/Linux sistemlerindeki ps POSIX komutu sistemdeki prosesler hakkında bilgi vermektedir. ps komutunun pek çok seçeneği vardır. Bu komut default durumda o andaki terminale ilişkin prosesler hakkında ıs abilgi verir. Komutun önemli birkaç seçeneği şöyledir:

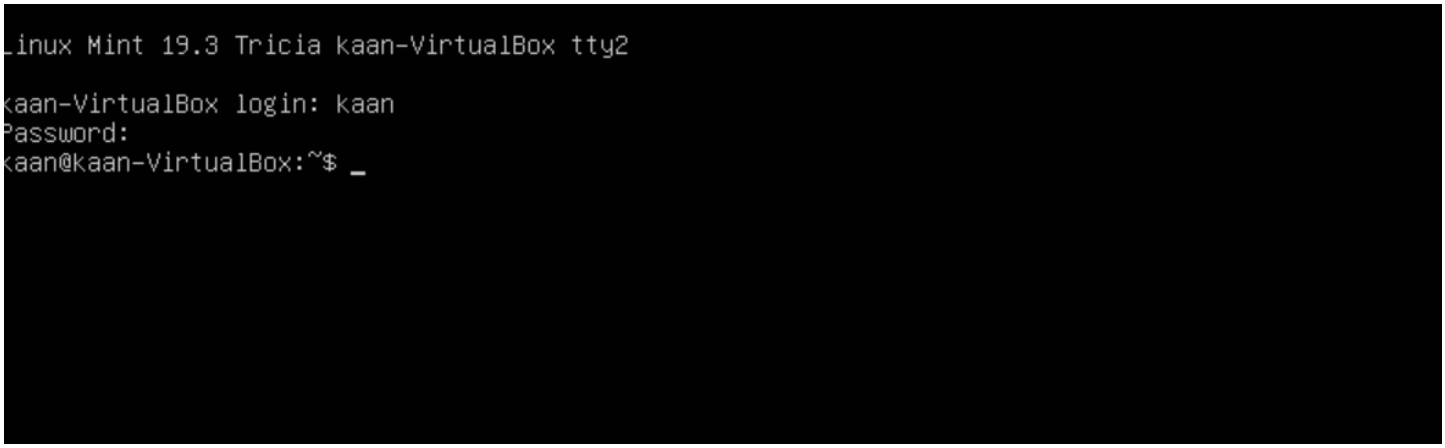
-u [kullanıcı listesi]: Belli bir kullanıcıya ilişkin tüm proseslerin listesi
-e: Tüm proseslerin listesi
-l: Detaylı bilgi (long list)
--forest: Altlık üstlük ilişkisiiçiminde ağaç biçiminde gösterilir.

ps komutunun ayrıntılarını çeşitli kaynaklardan öğrenebilirsiniz.

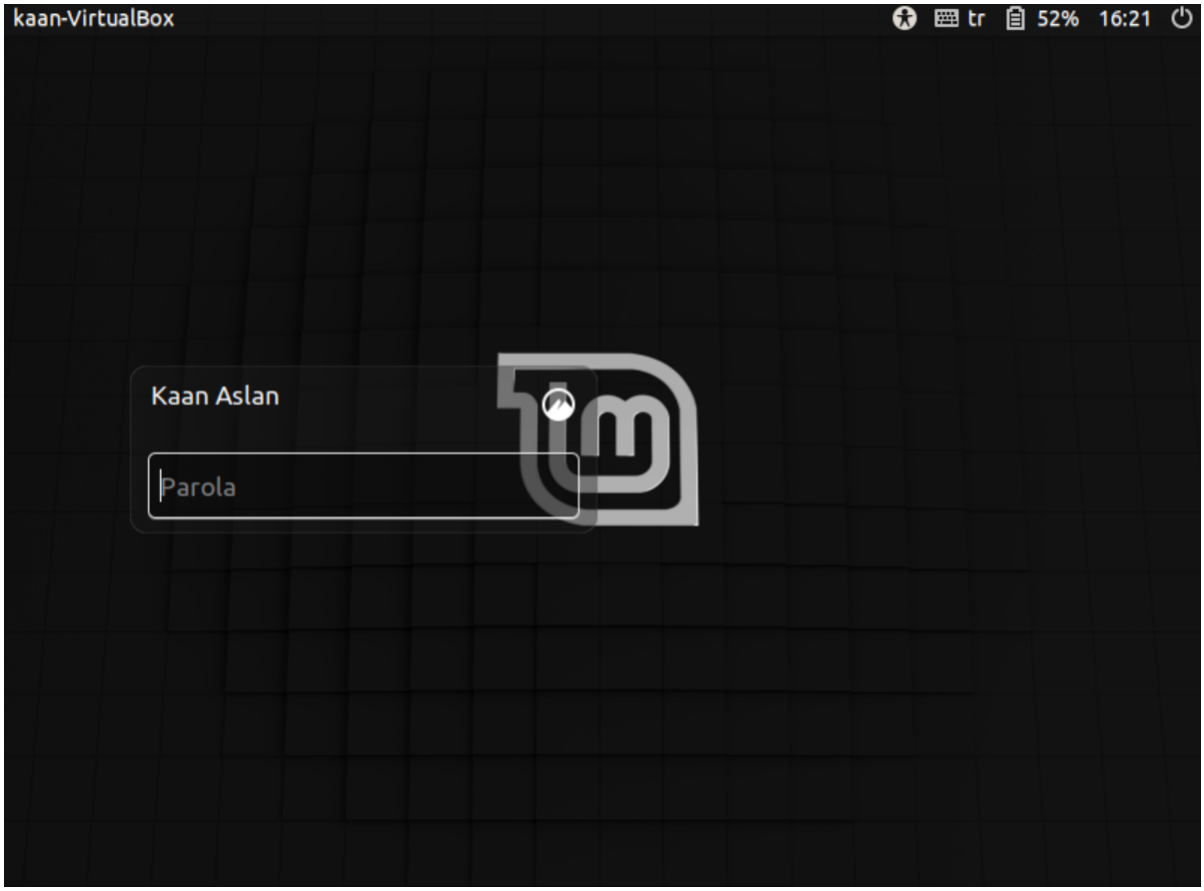
### Sisteme Giriş (Login Olma)

UNIX türevi sistemlerine giriş için her kullanıcıya bir kullanıcı ismi (user name) ve bir parola (password) verilmektedir. Kullanıcılar da bu kullanıcı isimlerini, ve parolalarını yazarak sisteme giriş yaparlar. Sisteme giriş genellikle üç yoldan biriyle gerçekleşmeketdir:

1) Text tabanlı terminal login programları ile: Eğer sistemimizde XWindow sistemi ve bir pencere yöneticisi yoksa (örneğin tipik sunucu sistemlerinde olduğu gibi) sisteme giriş text tabanlı bir biçimde "login" programı yoluyla yapılır. Örneğin:




2) XWindow Sistemleri yoluyla: Eğer sistemimizde XWindow sistemi ve pencere yöneticisi varsa login işlemi bu sistemle de yapılabilmektedir. Örneğin:




3) Uzak bağlantı yoluyla: Uzak bağlantı yoluyla tipik olarak ssh ya da telnet gibi bir protokolle text tabanlı olarak, VNC gibi bir protokolle de GUI tabanlı olarak sisteme giriş yapılabilir. Örneğin:

```
Kagan-MacBook-Pro:~ KaanAslan$ ssh kaan@34.77.157.105
kaan@34.77.157.105's password:
Welcome to Ubuntu 19.10 (GNU/Linux 5.3.0-1011-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Thu Jan 30 13:31:42 UTC 2020

  System load:  0.0               Processes:           101
  Usage of /:   3.8% of 38.60GB   Users logged in:     1
  Memory usage: 15%               IP address for ens4: 10.132.0.3
  Swap usage:   0%

 * Overheard at KubeCon: "microk8s.status just blew my mind".

     https://microk8s.io/docs/commands#microk8s.status

0 updates can be installed immediately.
0 of these updates are security updates.


Last login: Thu Jan 30 13:30:31 2020 from 212.2.212.159
kaan@linux:~$
```

## UNIX/Linux Sistemlerinde Yeni Kullanıcıların ve Grupların Yaratılması

UNIX sistemlerinin çoğunda kullanıcılara ilişkin bilgiler bir dosyada text olarak tutulmaktadır. Bu text dosyanın her satırı bir kullanıcı bilgisinden oluşmaktadır. Örneğin Linux ve BSD sistemlerinde /etc/passwd dosyası kullanıcı bilgilerini tutan yegane dosyadır. Her kullanıcının bilgisi /etc/passwd dosyasında bir satırda tutulmaktadır. Böylece bu sistemlerde yeni bir kullanıcı eklemek aslında bu dosyaya bir satır eklemekten ibarettir. /etc/passwd dosyası sıradan prosesler için "read-only" durumdadır. Yani bu dosyayı sıradan kullanıcılar görüntüleyebilirler ancak değişiklik yetkili kullanıcı (süper kullanıcı ya da kök kullanıcısı) tarafından yapılabilmektedir. Aşağıda örnek bir /etc/passwd dosyasını görüyorsunuz:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network Management,,,:/run/systemd/netif:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd/resolve:/usr/sbin/nologin
syslog:x:102:106::/home/syslog:/usr/sbin/nologin
messagebus:x:103:107::/nonexistent:/usr/sbin/nologin
_apt:x:104:65534::/nonexistent:/usr/sbin/nologin
uuidd:x:105:111::/run/uuidd:/usr/sbin/nologin
cups-pk-helper:x:106:112:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
kernoops:x:107:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin
rtkit:x:108:113:RealtimeKit,,,:/proc:/usr/sbin/nologin
avahi-autoipd:x:109:114:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/nologin
usbmux:x:110:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
systemd-coredump:x:111:117:systemd core dump processing,,,:/run/systemd:/usr/sbin/nologin
lightdm:x:112:118:Light Display Manager:/var/lib/lightdm:/bin/false
dnsmasq:x:113:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
saned:x:114:121::/var/lib/saned:/usr/sbin/nologin
nm-openvpn:x:115:122:NetworkManager OpenVPN,,,:/var/lib/openvpn/chroot:/usr/sbin/nologin
avahi:x:116:123:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin
colord:x:117:124:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
speech-dispatcher:x:118:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
pulse:x:119:125:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin
hplip:x:120:7:HPLIP system user,,,:/var/run/hplip:/bin/false
geoclue:x:121:127::/var/lib/geoclue:/usr/sbin/nologin
kaan:x:1000:1000:Kaan Aslan,,,:/home/kaan:/bin/bash
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
csd:x:1001:1001:,,,:/home/csd:/bin/bash
```

Dosyadaki her satırın bir kullanıcı hakkında bilgi içerdiğine dikkat ediniz. Bu dosyadaki her bir satır ':' karakterleriyle ayrılmış olan 7 alandan oluşmaktadır. Dosyanın son satırını dikkatlice ibceleyiniz:

```
csd:x:1001:1001:,,,:/home/csd/:/bin/bash
```

Buradaki 7 sütunun ne anlama geldiğini kabaca açıklamak istiyoruz. Birinci sütun kullanıcının ismidir. Daha önce de belirttiğimiz gibi her kullanıcının sistem genelinde tek olan bir kullanıcı ismi (user name) vardır. Kullanıcı yaratırken bu ismi kullanıcının istediği gibi alabilirsiniz. İkinci sütun kullanıcının şifrelenmiş parolasını belirtmektedir. Kullanıcıların şifrelenmiş parolaları eskiden doğrudan bu alanda ASCII karakterleriyle saklanıyordu. O yıllarda herhangi bir kişinin bu şifrelenmiş parolaları görmesinde bir sakınca görülmemişti. (Şifreleme "tek yönlü (one way)" yapıldığı için şifrelenmiş paroladan asıl parolayı bulmak mümkün olmamaktadır.) Ancak zamanla bu durum sakıncalı görüldüğü için şifrelenmiş parolaların başka bir dosyada (/etc/shadow dosyasında) saklanması yoluna gidilmiştir. İşte burada 'x' karakterinin olması şifrelenmiş parolanın /etc/shadow dosyasında bulunduğu anlamına gelmektedir. Siz yeni bir kullanıcı yaratırken bu alana 'x' karakteri yerleştirebilir sonra da parolayı da /bin/passwd programı ile belirleyebilirsiniz. Eğer parola alanı boş bırakılırsa bu durum giriş için parolaya gereksinim duyulmadığıu anlamına gelmektedir. Bu durumda login olunurken parola istendiğinde hiç parola girilmeden ENTER tuşuna basılarak giriş yapılabilir. Üçüncü sütun kullanıcının "gerçek kullanıcı id'sini (real user id)" belirtmektedir. Her kullanıcının bir ismi bir de gerçek kullanıcı id'si vardır. İşletim sistemi çekirdeği kullanıcıların isimleriyle değil id'leriyle işlem yapmaktadır. Dördüncü sütun ise kullanıcının "gerçek grup id'sini (real group id)" belirtir. Her kullanıcının bir gerçek kullanıcı id'sinin yanı sıra bir de gerçek grup id'si vardır. Kullanıcıların

gerçek kullanıcı ve gerçek grup id'lerinin ne olduğu ve ne işe yaradığı sonraki bölümde eele alınmaktadır. Beşinci alanda kullanıcıya ilişkin bilgiler bulunmaktadır. Bu alanın da ',' karakterleriyle 4 parçadan oluştuğunu görüyorsunuz. Bu alan boş geçilebilmektedir. Altıncı alanda kullanıcı dizini belirtmektedir. UNIX/Linux sistemlerinde geleneksel olarak her kullanıcı için /home dizininin altında bir kullanıcı dizinini bulundurulmaktadır. Bu dizin sisteme girildiğinde ilk çalıştırılacak programın da çalışma dizini (current working directory) yapılmaktadır. Nihayet yedinci ve son alanda kullanıcı terminal yoluyla sisteme giriş yaptıktan sonra çalıştırılacak program belirtilmektedir. Yukarıdaki örnekte bu alanda /bin/bash yazdığını görüyorsunuz. Bu durum "eğer terminal yoluyla giriş yapılıyorsa" otomatik olarak çalıştırılacak programın "bash (Bourne Again Shell)" isimli kabuk programı olduğu anlamına gelmektedir.

Şimdi /etc/passwd dosyasına aşağıdaki satırı ekleyelim:

```
student:x:1002:1001:,,,:/home/student/:/bin/bash
```

Burada biz yeni kullanıcımızın ismini "student", onun gerçek kullanıcı id'sini 1002 ve gerçek grup id'sini de 1001 olarak belirlemiş olduk. Parola alanının 'x' olduğuna dikkat ediniz. Kullanıcıya ilişkin dizin /home/student biçiminde, terminal girişinde ilk çalıştırılacak program da /bin/bash biçiminde seçilmiştir.

UNIX/Linux sistemlerinde bir grup kullanıcının ortak çalışma yapabilmesi için grup kavramı oluşturulmuştur. Sistemdeki tüm grup bilgileri /etc/group isimli bir dosyada tutulmaktadır. Aşağıda bu dosyanın neye benzediğini görüyorusunuz:

```
systemd-journal:x:101:
systemd-network:x:102:
systemd-resolve:x:103:
input:x:104:
crontab:x:105:
syslog:x:106:
messagebus:x:107:
netdev:x:108:
mlocate:x:109:
ssl-cert:x:110:
uuidd:x:111:
lpadmin:x:112:kaan
rtkit:x:113:
avahi-autoipd:x:114:
ssh:x:115:
bluetooth:x:116:
systemd-coredump:x:117:
lightdm:x:118:
nopasswdlogin:x:119:
scanner:x:120:saned
saned:x:121:
nm-openvpn:x:122:
avahi:x:123:
colord:x:124:
pulse:x:125:
pulse-access:x:126:
geoclue:x:127:
kaan:x:1000:
sambashare:x:128:kaan
vboxsf:x:999:
study:x:1001:
```

/etc/group dosyasının da satırlardan oluştuğunu görüyorsunuz. Buradakşi her bir satır bir grup hakkında bilgi vermektedir. Her satır yine ':' karakterleriyle 4 alana ayrılmıştır. ÖBir örnek olarak dosyanın son satırını dikkatle inceleyiniz:

```
study:x:1001:
```

Burada ilk alan grubun ismini belirtmektedir. İkinci alanda grup parolası bulunur. Grupların da parolaları vardır fakat çok nadir bu parola bir yerlerde gerekmektedir. Üçüncü sütunda grubun grup id'si bulunur. Her grubun kullanıcılar kişiler tarafından kolay algılanan bir ismi ve çekirdek tarafından kullanılan bir id'si olduğuna dikkat ediniz. Son sütunda bu gruba dahil olan kullanıcıların listesi tutulmaktadır. Buradaki liste ',' karakterleriyle ayrılmış kullanıcı isimlerinden oluşmaktadır. Bu isimlere "kullanıcıların ek grupları (supplementary groups)" denilmektedir ve bu konu ileride ele alınacaktır.

Şimdi yeniden /etc/passwd dosyasına eklediğimiz satıra geri dönelim. Bu satırı yeniden aşağıda veriyoruz:

```
student:x:1002:1001:,,,:/home/student/:/bin/bash
```

Buradan student isimli kullanıcının gerçek kullanıcı id'sinin 1002 ve gerçek grup id'sinin 1001 olduğunu görüyoruz. İşte /etc/passwd dosyasında kullanıcının gerçek grup ismi değil, gerçek grup id'si tutulmaktadır. Kullanıcının gerçek grup ismini bulabilmek için önce onun /etc/passwd dosyasındaki gerçek grup id'sini bulmak sonra da /etc/group dosyasından bu grubun ismini elde etmek gerekir.

Yeni bir kullanıcı eklemek için aslında tek yapılacak şeyin /etc/passwd dosyasına bir satır eklemek olduğunu söyledik. Ancak bu kullanıcının işlevsel olabilmesi için iki şeyin daha yapılması gerekmektedir: Kullanıcı için /home dizininin altında bir dizin yaratmak ve kullanıcı için bir parola oluşturmak. (Eğer oluşturduğunuz kullanıcı için /etc/passwd dosyasında belirttiğiniz kullanıcı dizinini oluşturmadıysanız login programı sizi kök dizinde bırakacaktır.) Kullanıcı için /home dizininin altında dizin yaratma işlemi şöyle yapılabilir:

```
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$ sudo mkdir /home/student
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$ ls -l /home
toplam 12
drwxr-xr-x  2 csd  study 4096 Şub  4 13:41 csd
drwxr-xr-x 23 kaan kaan  4096 Şub  4 13:42 kaan
drwxr-xr-x  2 root root  4096 Şub  4 13:50 student
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$
```

Burada yeni yaratılan studnet dizininin kullanıcı id'si ve grup id'sinin root olduğu görülüyor. (Dosyaların ve dizinlerin kullanıcı ve grup id'lerinin ne olduğu ilerideele alınmaktadır.) Bu durumu aşağıdaki komutla düzeltebiliriz:

```
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$ sudo chown student:study /home/student
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$ ls -l /home
toplam 12
drwxr-xr-x  2 csd     study 4096 Şub  4 13:41 csd
drwxr-xr-x 23 kaan    kaan  4096 Şub  4 13:56 kaan
drwxr-xr-x  2 student study 4096 Şub  4 13:50 student
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$
```

Son olarak oluşturduğumuz kullanıcıya bir parola atamamız gerekir. Bu işlem de şöyle yapılabilir:

```
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$ sudo passwd student
Yeni parolayı girin:
Yeni parolayı tekrar girin:
passwd: şifre başarıyla güncellendi
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$
```

Artık kullanıcınız hazır durumda.

Aslında UNIX/Linux sistemlerinde yeni bir kullanıcı yaratma işlemi pratik olarak useradd ve adduser isimli komutlarla yapılabilmektedir. useradd komutu gerçek bir çalıştırılabilen binary dosyadır. add user ise arka planda useradd

komutunu çalıştıran bir perl script dosyasıdır. Yani addser komutunu useradd komutunun daha kullanıcı dostu (user friendly) biçimi olarak düşünebilsiniz. Aslında bu komutlar arka planda yukarıda adım adım yaptığımız işleri yapmaktadır. adduser komutunun kullanımına bir örnek vermek istiyoruz:

```
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$ sudo adduser ali
"ali" kullanıcısı ekleniyor ...
Yeni grup "ali" ekleniyor (1002) ...
Yeni kullanıcı "ali" (1003) "ali" grubuyla ekleniyor ...
"/home/ali" başlangıç dizini oluşturuluyor ...
"/etc/skel" dizininden dosyalar kopyalanıyor ...
Yeni parolayı girin:
Yeni parolayı tekrar girin:
passwd: şifre başarıyla güncellendi
ali için kullanıcı bilgileri değiştiriliyor
Yeni değeri girin, veya varsayılan değer için ENTER'a basın
        Tam İsim []: Ali Serce
        Oda Numarası []:
        İş Telefonu []:
        Ev Telefonu []:
        Diğer []:
Bilgiler doğru mu? [E/h] E
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$
```

Bu komut sonrasında /etc/passwd dosyasına bakınız:

```
systemd-network:x:100:102:systemd Network Management,,,:/run/systemd/netif:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd Resolver,,,:/run/systemd/resolve:/usr/sbin/nologin
syslog:x:102:106::/home/syslog:/usr/sbin/nologin
messagebus:x:103:107::/nonexistent:/usr/sbin/nologin
_apt:x:104:65534::/nonexistent:/usr/sbin/nologin
uuidd:x:105:111::/run/uuidd:/usr/sbin/nologin
cups-pk-helper:x:106:112:user for cups-pk-helper service,,,:/home/cups-pk-helper:/usr/sbin/nologin
kernoops:x:107:65534:Kernel Oops Tracking Daemon,,,:/:/usr/sbin/nologin
rtkit:x:108:113:RealtimeKit,,,:/proc:/usr/sbin/nologin
avahi-autoipd:x:109:114:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/nologin
usbmux:x:110:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
systemd-coredump:x:111:117:systemd core dump processing,,,:/run/systemd:/usr/sbin/nologin
lightdm:x:112:118:Light Display Manager:/var/lib/lightdm:/bin/false
dnsmasq:x:113:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
saned:x:114:121::/var/lib/saned:/usr/sbin/nologin
nm-openvpn:x:115:122:NetworkManager OpenVPN,,,:/var/lib/openvpn/chroot:/usr/sbin/nologin
avahi:x:116:123:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin
colord:x:117:124:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
speech-dispatcher:x:118:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
pulse:x:119:125:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin
hplip:x:120:7:HPLIP system user,,,:/var/run/hplip:/bin/false
geoclue:x:121:127::/var/lib/geoclue:/usr/sbin/nologin
kaan:x:1000:1000:Kaan Aslan,,,:/home/kaan:/bin/bash
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
csd:x:1001:1001:CSD,,,:/home/csd:/bin/bash
student:x:1002:1001:Student,,,:/home/student:/bin/bash
ali:x:1003:1002:Ali Serce,,,:/home/ali:/bin/bash
```

adduser komutunun burada bir satır yarattığını görüyorsunuz. Şimdi de /etc/group dosyasına bakalım:

```
syslog:x:106:
messagebus:x:107:
netdev:x:108:
mlocate:x:109:
ssl-cert:x:110:
uuidd:x:111:
lpadmin:x:112:kaan
rtkit:x:113:
avahi-autoipd:x:114:
ssh:x:115:
bluetooth:x:116:
systemd-coredump:x:117:
lightdm:x:118:
nopasswdlogin:x:119:
scanner:x:120:saned
saned:x:121:
nm-openvpn:x:122:
avahi:x:123:
colord:x:124:
pulse:x:125:
pulse-access:x:126:
geoclue:x:127:
kaan:x:1000:
sambashare:x:128:kaan
vboxsf:x:999:
study:x:1001:
ali:x:1002:
```

Burada da ali isimli bir grubun eklendiğini görüyorsunuz. Son olarak şimdi de /home dizinine bakalım:

```
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$ ls -l /home
toplam 16
drwxr-xr-x  4 ali     ali    4096 Şub  4 14:08 ali
drwxr-xr-x  2 csd     study  4096 Şub  4 13:41 csd
drwxr-xr-x 23 kaan    kaan   4096 Şub  4 13:56 kaan
drwxr-xr-x  2 student study  4096 Şub  4 13:50 student
kaan@kaan-VirtualBox:~/Unix-Linux-SysProg$
```

Aslında grupların yaratılması için de groupadd ve addgroup isimli komutlar da vardır. Yine groupadd komutu asıl binary çalıştırılabilir dosyadır. addgrup ise bir perl script dosyasıdır. addgroup komutunu groupadd komutunun kullanıcı dostu (user frienly) biçimi olarak düşünebilirsiniz.

**UNIX/Linux Sistemlerinde Kullanıcıların ve Grupların Silinmesi**

Sistem yöneticisi nasıl bir kullanıcıyı yratabiliyorsa aynı zamanda onu silebilmektedir. Aslında manuel silmek için yapılacak şeyler yaratırken yapılanların tersidir. Yani bir kullanıcıyı manuel olarak şöyle silinebilir:

1) /etc/passwd dosyasından kullanıcı ile ilgili satır silinir.
2) /home dizininden kullanıcı dizini silinir.

Kullanıcı silmek için de benzer biçimde userdel ve deluser isimli komutlar da vardır. userdel binary çalıştırılabilir bir dosyadır, deluser ise userdel programını çalıştıran bir perl script dosyasıdır.

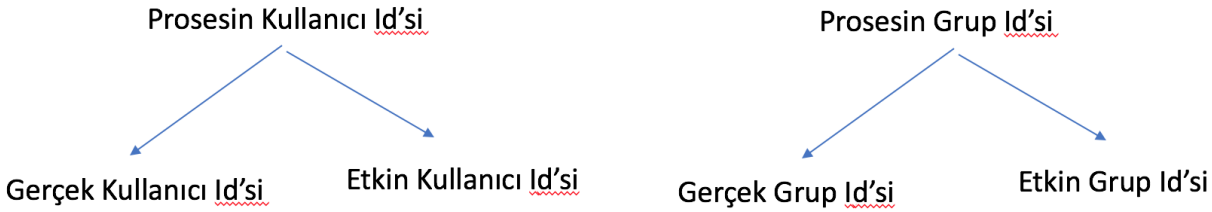
**UNIX/Linux Sistemlerinde Proseslerin Kullanıcı ve Grup ID'leri**

UNIX türevi sistemlerde sisteme girmek isteyen her kullanıcıya bir kullanıcı ismi ve parola verildiğini söylemiştik. İşte sistem genelinde her kullanıcı ismine karşılık o kullanıcı için bir kullanıcı id'si (real user id) karşılık getirilmiştir. İşletim

sisteminin çekirdeği yazısal olan kullanıcı isimleriyle değil sayısal olan kullanıcı id'leriyle işlemleri yapmaktadır. Kullanıcı isimleri aslında kullanıcı id'lerini temsil eden insanlar tarafından kolay algılansın diye uydurulmuş isimlerdir. Kullanıcı isimleriyle kullanıcı id'leri arasındaki ilişkiyi IP protokol ailesindeki "host ismi" ile "ip numarası" arasındaki ilişkiye benzetebiliriz. Yukarıda da belirttiğimiz gibi UNIX/Linux sistemlerinde her kullanıcı aynı zamanda bir gruba da atanmaktadır. Grup belli kullanıcıların bir arada çalışmaları için düşünülmüş bir kavramdır.

UNIX/Linux sistemlerinde her prosesin proses kontrol bloğunda saklanan bir "gerçek kullanıcı id'si (real user id)" ve "gerçek grup id'si (real user id)" vardır. Gerçek kullanıcı id'leri o prosesin sahibini, gerçek grup id'leri ise o prosesin grubunu belirtmektedir.

UNIX/Linux sistemlerinde her prosesin gerçek kullanıcı id'sinin ve gerçek grup id'sinin yanı sıra bir de "etkin kullanıcı id'si (effective user id)" ve "etkin grup id'si (effective group id)" de vardır. İleride göreceğiniz gibi pek çok teste aslında prosesin etkin kullanıcı id'si ve etkin grup id'si girmektedir. Bu nedenle biz kursumuzda "kullanıcı id"si dediğimiz default olarak "etkin kullanıcı id"si, "grup id"si dediğimizde de "etkin grup id"si anlaşılmalıdır. Aslında istisnai birkaç durum dışında proseslerin gerçek kullanıcı id'leri ile etkin kullanıcı id'leri, gerçek grup id'leri ile etkin grup id'leri aynı değerdedir.




Peki proseslerin gerçek ve etkin kullanıcı ve grup id'leri nasıl oluşturulmaktadır? İşte bu id değerleri aslında proses yaratılırken üst proses tarafından alt prosese aktarılmaktadır. Yani bir prosesin gerçek ve etkin kullanıcı ve grup id'si aslında üst prosesten gelmektedir. Pekiyi o zaman üst prosesin kullanıcı ve gerçek kullanıcı id'leri nereden gelmektedir? Tabii onlar da kendi üst prosesinden gelmektedir. Biz burada bu konuda temel bir açıklama yapacağız. Bu konudaki ayrıntılar sistemin boot edilmesinin ele alındığı bölümde açıklanacaktır.

Biz sisteme terminal yoluyla giriş yaptığımızda bizi sisteme sokan login isimli prosestir. Bu proses bizden kullanıcı ismini ve parolayı ister. Eğer bunu doğrularsa bir alt proses yaratarak o alt prosesin gerçek ve etkin kullanıcı id'sini /etc/passwd dosyasındaki kullanıcı id'si olacak biçimde, gerçek ve etkin grup id'sini de yine bu dosyadaki grup id'si olacak biçimde değişirir. Yani bizi sisteme sokan login programı bizi temsil eden prosesin gerçek ve etkin kullanıcı id'sini, gerçek ve etkin grup id'sini /etc/passwd dosyasında belirtildiği gibi ayarlamaktadır. XWindow ve uzak login işleminde de benzer süreç işlemektedir. Biz sisteme girdikten sonra prosesin gerçek ve etkin kullanıcı id'sinin aynı değerde, gerçek ve etkin grup id'sinin de aynı değerde olduğuna dikkat ediniz.

UNIX sistemleri baştan itibaren yetki bakımından "ya hep ya hiç" tarzı bir mimari uygulamıştır. Bu sistemlerde ismine "kök kullanıcısı (root user)" ya da "süper kullanıcı (super user)" ya da "ayrıcalıklı kullanıcı (priveleged user)" denilen bir kullanıcı sistemde hiçbir erişim engeline takılmayıp her şeyi yapabilmektedir. Ancak diğer kullanıcılar için erişim kontrolleri yapılmaktadır. Pek çok UNIX türevi sistem hala bu biçimde bir tasarıma sahiptir. Ancak Linux sistemlerine belli bir zamandan sonra "yetenek (capability)" denilen özellik eklenmiş ve kullanıcı kök kullanıcısı olmasa da bazı yetkilere sahip olabilmiştir. İşte UNIX/Linux sistemlerinde 0 numaralı kullanıcı id'sine sahip olan prosesler "kök (root) proseslerdir" ve bu prosesler diğer proseslerde olmayan bir yetkiye sahiptirler. 0 numaralı kullanıcı id'si özeldir ve bu id'ye sahip olan prosesler sistemde hiçbir erişim engeliyle karşılaşmazlar. Ancak yukarıda da belirttiğimiz gibi Linux sistemlerinde prosesin kullanıcı id'si 0 olmasa bile ona bazı yetenekler verilebilmektedir. Proses yetenekleri kitabımızda ayrı bir bölüm olarak ele alınmaktadır.

UNIX/Linux sistemlerinde eskiden her kullanıcı tek bir gruba üye ilişkin olabiliyordu. Yani kullanıcıların yalnızca gerçek grup id'leri vardı. Zamanla kullanıcıların birden fazla grupla ilişkili olmalarının gerektiği anlaşıldı ve böylece "ek grup (supplementary groups)" kavramı ortaya atıldı. Artık uzun süredir UNIX türevi sistemlerde proseslerin gerçek grup id'lerinin yanı sıra ek grup id'leri de bulunmaktadır. İleride de görüleceği gibi ek grup id'leri erişimlerdeki test işlemlerinde gerçek grup id'leriyle eşdeğer biçimde işleme sokulmaktadır.

Linux sistemlerinde prosesin ek grupları /etc/group dosyasında tutulmaktadır. Anımsayacağınız gibi bu dosya satırlardan, satırlar da ':' karakterleriyle ayrılmış sütunlardan oluşuyordu. İşte her satırdaki son sütun o gruba ek grup olarak dahil olan kullanıcıların listesini tutmaktadır. Daha önce vermiş olduğumuz örnek /etc/group dosyasına bir daha bakınız:

```
syslog:x:106:
messagebus:x:107:
netdev:x:108:
mlocate:x:109:
ssl-cert:x:110:
uuidd:x:111:
lpadmin:x:112:kaan
rtkit:x:113:
avahi-autoipd:x:114:
ssh:x:115:
bluetooth:x:116:
systemd-coredump:x:117:
lightdm:x:118:
nopasswdlogin:x:119:
scanner:x:120:saned
saned:x:121:
nm-openvpn:x:122:
avahi:x:123:
colord:x:124:
pulse:x:125:
pulse-access:x:126:
geoclue:x:127:
kaan:x:1000:
sambashare:x:128:kaan
vboxsf:x:999:
study:x:1001:
```

Burada kaan isimli kullanıcının lpadmin ve sambashare gruplarına da üye olduğunu görüyorsunuz. Yani kaan kullanıcısına ilişkin proseslerin ek grup id'lerinde bu gruplar da bulunacaktır. Bir kullanıcının gerçek kullanıcı id'sinin ve gerçek grup id'sinin /etc/passwd dosyasında tutulduğuna ama ek grup id'lerinin /etc/group dosyalarında tutulduğuna dikkat ediniz. (Yukarıdakş örnek /etc/group dosyasının sondan dördüncü satırında ayrıca kaan isimli bir grup da vardır. Bu grubun ek kullanıcı id'leriyle bir ilgisi yoktur. Bir kullanıcı yaratıldığında default olarak kullanıcı ile aynı isimli bir grup da yaratılmaktadır.)

### UNIX/Linux Sistemlerindeki POSIX Fonksiyonlarının Başarısızlık Nedenlerinin Elde Edilmesi

Sistem programlama sırasında pek çok durumda hatalarla karşılaşılabilmektedir. Sistem programlama faaliyeti sırasında genellikle hatalar çağırdığımız aşağı seviyeli fonksiyonlar tarafından belirlenip geri dönüş değeri ile bize iletilirler. Çağırdığımız fonksiyonların geri dönüş değerlerine bakarak biz de hata durumlarını tespit ederiz.

Hata kontrolü bakımından fonksiyonları iki gruba ayırabiliriz:

1) Her zaman hata kontrolünün yapılması gerektiği fonksiyonlar: Bunlar sistemin o anki durumuyla ilgili biçimde başarısız olabilecek fonksiyonlardır. Bu tür fonksiyonlar çağrılırken kesinlikle hata kontrolü yapılmalıdır. (Örneğin fopen, malloc, ... gibi fonksiyonlar)

2) Eğer programcı her şeyi düzgün yapmışsa, başarısız olma olasılığı olmayan fonksiyonlar: Bu tür fonksiyonlar için hata kontrolü yapılmayabilir. Örneğin biz bir dosyayı fopen fonksiyonu ile düzgün bir biçimde açmışsak bu dosyanın fclose ile kapatılamamasının makul bir nedeni olamaz. (Zaten böyle bir durumda bizim yapabileceğimiz birşey de yoktur.) Tabii bu tür fonksiyonların başarı durumları debug amacıyla test edilebilir. Örneğin biz programımız için "debug" ve "release" versiyonları oluşturmuşsak bu tür fonksiyonların başarısını yalnızca programın "debug" versiyonlarında yapabiliriz.

POSIX fonksiyonlarının çok büyük çoğunluğunun geri dönüş değeri int türdendir. Bu int türden geri dönüş değeri bize fonksiyonun başarılı mı yoksa başarısız mı olduğu bilgisini verir. POSIX fonksiyonları başarı durumunda sıfır (dikkat

ediniz), başarısızlık durumunda ise -1 değerine geri dönerler. Böylece tipik olarak bir POSIX fonksiyonunun başarısızlığı şöyle tespit edilmektedir:

```
if (some_posix_function(...) == -1) {
    ...
}
```

Fakat bazı programcılar kontrolü aşağıdaki gibi de yapabilmektedir:

```
if (some_posix_function(...) < 0) {
    ...
}
```

Bu biçimde kontrolün mikro mertebede daha etkin olduğunu söyleyebiliriz. Fakat bunun bir önemi yoktur. Bazı POSIX fonksiyonlarının geri dönüş değeri bir adres türündendir. Bunlar pek çok sistemde olduğu gibi başarısızlık durumunda NULL adres değerine geri dönerler.

Pekiyi POSIX sistemlerinde bir fonksiyonun neden başarısız olduğunu nasıl anlayabiliriz? İşte bir POSIX fonksiyonu başarısız olduğunda başarısızlığın nedenine ilişkin değeri errno isimli int türden bir global değişkene yerleştirmektedir. Biz de başarısızlık durumunda doğrudan bu errno değerine bakabiliriz. errno değişkeni glibc kütüphanesinde tanımlanmıştır; bunun extern bildirimi <errno.h> dosyası içerisinde yapılmıştır. Bu durumda UNIX/Linux sistemlerinde hata şöyle ele alınabilir:

```
if (some_posix_function(...) == -1) {
    fprintf(stderr, "error: %d\n", errno);
    exit(EXIT_FAILURE);
}
```

Ayrıca (tıpkı Windows sistemlerinde olduğu gibi) errno değişkenin alabileceği tüm hata değerleri de <errno.h> dosyası içerisinde EXXX biçiminde sembolik sabitlerle define edilmiştir. Böylece programcı isterse aşağıdaki gibi bir kod yazabilir:

```
if (errno == EACCESS) {
    ...
}
```

Ya da örneğin şöyle bir kod da yazabilir:

```
switch (errno) {
    case EACCESS:
        .....
        break;
    case EPERM:
        .....
        break;
    case EINTR:
        .....
        break;
    ...
}
```

POSIX standartlarında hangi hatalar için hangi errno değerlerinin (sayısal değerleri kastediyoruz) kullanılacağı standart olarak belirlenmemiştir. Yani aynı hata durumu için errno değişkenin sayısal değerleri farklı sistemlerde farklı olabilmektedir. Fakat hata kodlarına ilişkin EXXX biçimindeki sembolik sabit isimleri standart olarak belirlenmiştir. Biz de taşınabilirlik için errno değişkenindeki sayısal değerleri değil EXXX biçimindeki sembolik sabitleri kullanmalıyız.

UNIX/Linux sistemlerinde bir POSIX fonksiyonu başarısız olursa errno değişkeninde hangi değerlerin bulunabileceği kesin olarak listelenmiştir. Bu listeye POSIX standartlarından ya da man sayfalarından ulaşılabilir. Halbuki Windows sistemlerinde bir API fonksiyonu başarısız olduğunda başarısızlığın tüm nedenleri listelenmemiştir.

errno değişkeninin kullanımı ile ilgili iki önemli noktayı daha açıklamak istiyoruz: Birincisi POSIX standartları bir POSIX fonksiyonlarının başarı durumunda da errno değişkeni değiştirmesine izin vermiştir. Yani errno değişkenine bir değer atadıktan sonra çağırdığınız fonksiyon başarısız olsa bile çıkışta errno değişkeninin değeri değişmiş olabilir. (Tabii aslında genel olarak POSIX kütüphanelerini yazanlar başarı durumunda errno değişkenine genellikle dokunmamaktadır.) İkincisi 0 değeri errno değişkeni için özel bir değerdir. Hiçbir POSIX fonksiyonu errno değişkenine 0 yerleştirmemektedir. (Böylece başarı durumunda errno değerini değiştirmediğini garanti eden POSIX fonksiyonlarının bazılarında çağrıdan önce errno değişkenine 0 atanıp çağrı sonucunda bu değer kontrol edilebilmektedir.)

UNIX/Linux sistemlerinde ayrıca hata kodunu yazıya dönüştüren strerror isimli bir fonksiyon da vardır. strerror fonksiyonunun prototipi şöyledir:

```
#include <string.h>

char *strerror(int errnum);
```

Fonksiyon errno değerini parametre olarak alır, onun yazısını static bir alana yerleştirerek o alanın adresini bize verir. Örneğin:

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

int main(void)
{
    int fd;

    if ((fd = open("xxxxxx.yyy", O_RDONLY)) == -1) {
        fprintf(stderr, "open:%s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
    printf("Ok\n");

    return 0;
}
```

POSIX sistemlerinde bir kademe daha ileriye gidilerek errno değişkendeki değeri strerror fonksiyonuyla elde edip onu stderr dosyasına yazdıran perror isimli bir fonksiyon da bulundurulmuştur:

```
#include <stdio.h>

void perror(const char *s);
```

Fonksiyon önce parametresiyle belirtilen yazıyı, sonra ':' karakterini, sonra da errno değişkenine karşı gelen hata yazısını stderr dosyasına yazdırmaktadır. Böylece perror kullanılarak hata tespiti şöyle yapılabilir:

```
if ((fd = posix_func(...)) == -1) {
    perror("posix_func");
    exit(EXIT_FAILURE);
}
```

Örneğin:

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

```
int main(void)
{
    int fd;

    if ((fd = open("xxxxxx.yyy", O_RDONLY)) == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    printf("Ok\n");

    return 0;
}
```

Kursumuzda genellikle bir POSIX fonksiyonu başarısız olduğunda hata mesajını ekrana yazıp prosesi sonlandırmak için aşağıdaki gibi yazılmış bir fonksiyonu kullanacağız:

```
void exit_sys(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

exit_sys fonksiyonunun kullanımı da şöyle olabilir:

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;

    if ((fd = open("xxxxxx.yyy", O_RDONLY)) == -1)
        exit_sys("open");

    printf("Ok\n");

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

### POSIX Fonksiyarında Kullanılan typedef Edilmiş Tür İsimleri

Pek çok POSIX fonksiyonunun prototipinde sonu _t ile biten xxxxx_t biçiminde typedef isimleri kullanılmıştır. Örneğin:

```
pid_t fork(void);
int chmod(const char *path, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
```

UNIX/Linux sistemlerinde bu sonu xxxxx_t ile biten typedef isimlerinin hepsi <sys/types.h> dosyası içerisinde bildirilmiştir. Dolayısıyla bu tür isimlerinin kullanılması için bu dosyanın include edilmesi gerekmektedir. Ancak burada bir noktayı daha belirtmek istiyoruz. Aslında bu xxxxx_t biçimindeki tür isimlerinin bazıları <sys/types.h> dosyasının dışında başka birtakım dosyalarda da typedef edilmiştir. Dolayısıyla kullanmak istediğiniz tür isimleri zaten include etmiş

olduğunuz dosyalarda bildirilmiş olabilir. Hangi tür isimlerinin hangi dosyalarda typedef edildiğini POSIX standartlarından ya da SUS (Single UNIX Specification) dokümanlarından öğrenebilrisiniz. Örneğin aşağıdaki bağlantıda SUS'ta <sys/types.h> dosyası hakkında açıklamaları göreceksiniz. Bu dokümanı şöyle bir incelemenizi tavsiye ediyouz:

https://pubs.opengroup.org/onlinepubs/9699919799/

Sonu xxxxx_t ile biten bu tür isimleri UNIX/Linux sistemini oluşturan kişiler tarafından belli koşullara uymak koşulu ile istenildiği gibi typedef edilebilmektedir. Örneğin POSIX ve SUS dokümanlarında ssize_t türünün için "işaretli bir tamsayı" türü olması mode_t türünün haehangi bir tamsayı türü olması bir koşul olarak belirtilmiştir. Bu durumda ilgili kişiler tarafından ssize_t türü işaretli herhangi bir tamsayı türü olarak, mode_t türü ise herhangi bir tamsayı türü olarak typedef edilmiş olabilir.

Aslında UNIX/Linux programcısının programını yazarken bu tür isimlerinin aslında hangi tür olarak typedef edilmş olduğunu bilmesine gerek yoktur. Programcı programında bu belirtilen tür isimlerini kullanmalıdır. Örneğin fork POSIX fonksiyonunun prototipine bakınız:

```
pid_t fork(void);
```

Siz bu pid_t türünün sizin siteminizde int olarak typedef edilmiş olduğunu biliyor olsanız bile kodunuzda yine pid_t tür ismini kullanmalısınız. Örneğin:

```
pid_t pid;

pid = fork();
```

Pekiyi POSIX kütüphanesini oluşturanlar prototiplerde ve başka yerlerde bu türlerin gerçek hallerini kullanmak yerine neden bu typedef isimlerini oluşturmuşlardır? Bunun en temel nedeni taşınabilirliği (portability) sağlamaktır. POSIX standartları oluşturulduğunda (ve sonrasında) pek çok sistem aynı fonksiyonları değişik türlerle gerçekleştirmiş durumdaydı. Sonraki sistemler de kendi çekirdek yapıları nedeniyle bu fonksiyonları farklı türlerle gerçekleştirmek istemiştir. Bu nedenle örneğin bazı sistemlerde mode_t türü unsigned short olarak, bazı sistemlerde, unsigned int türü olarak bazı sistemlerde de signed int türü olarak kullanılmış olabilmektedir. Eğer bu typedef isimleri olmasaydı ve biz de programımızda bunları kullanıyor olmasaydık bu durumda yazdığımız kodlar yalnızca bir sistem için geçerli olurken diğerleri için geçerli olmayabilecekti. Halbuki mevcut durumda ilgili sistem söz konusu türleri nasıl seçmişse zaten <sys/types.h> içerisinde uygun biçimde typedef etmiş olacaktır. Dolayısıyla bizim yazdığımız kod sistemelerdeki bu tür değişikliklere karşı geçerliliğini koruyacaktır. Yinelemek gerekirse biz programlarımızda eğer bir fonksyon ya da başka bir yerde xxxxx_t biçiminde bir typedef tür ismi görürsek taşınabilirliği sağlamak için programımız içerisinde bu typedef tür ismini kullanmalıyız.

## İşletim Sistemlerinin Dosya Sistemleri

İçerisinde bilgilerin bulunduğu ikincil belleklerdeki alanlara "dosya (file)" denilmektedir. İkincil bellekler aslında sektör denilen bloklardan oluşmaktadır. Sektörler ikincil belleklerden okunabilen ya da yazılabilen en küçük birimlerdir. Dosyaların parçaları ikincil belleklerdeki sektörlerde tutulmaktadır. İşletim sistemleri bu organize edilmiş sektörleri dış dünyaya "dosya (file)" kavramı ile göstermektedir. Yani aslında dosya yapay bir kavramdır. İşletim sistemi tarafından yapılan bir organizasyondur. İşte işletim sistemlerinin bu organizasyonu yapan alt sistemlerine "dosya sistemi (file system)" denilmektedir. Dosya sistemi UNIX/Linux sistemlerinin kalbidir. Bu sistemlerde pek çok kavram (örneğin borular, aygıtlar vs.) dosya gibi ele alınmaktadır. Böylece farklı kavramlar dosya kavramı ile ifade edilip ortak bir arayüz ile (dosya fonksiyonlarını kastediyoruz) işleme sokulabilmektedir.

UNIX/Linux sistemleri bir çeşit çokbiçimli (polymorphic) mekanizmayla farklı kavramların dosya kavramı biçiminde ele alınmasına olanak vermektedir. Linux sistemlerinde dosya sistemi gerçekleştirimine bu nedenle "Sanal Dosya Sistemi (Virtual File System)" denilmektedir.

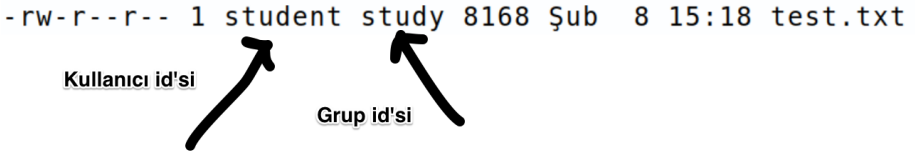
## UNIX/Linux Sistemlerinde Dosyaların Erişim Hakları

UNIX/Linux sistemlerinde bir dosyayla ilgili işlem yapmak için ya da bir dosyayı yaratabilmek için o dosyanın önce açılması gerekmektedir. Dosya açma işlemi "open" isimli POSIX fonksiyonuyla yapılır. ("open" fonksiyonun yanı sıra "openat" ve "creat" isimli iki fonksiyon da vardır. Ancak genellikle açış için "open" fonksiyonu tercih edilmektedir.) open POSIX fonksiyonu pek çok UNIX türevi sistemde doğrudan işletim sisteminin dosya açan sistem fonksiyonunu çağırmaktadır. Örneğin Linux sistemlerinde open fonksiyonu neredeyse doğrudan sys_open isimli sistem fonksiyonunu çağırmaktadır.) İşte ileride göreceğimiz gibi open fonksiyonuyla biz bir dosyayı açarken hangi işlemi yapmak üzere onu açmak istediğimizi belirtiriz. Niyet edilen üç işlem türü şunlardır:

- Yalnızca okuma yapmak (read only)
- Yalnızca yazma yapmak (write only)
- Hem okuma hem de yazma yapmak (read/write)

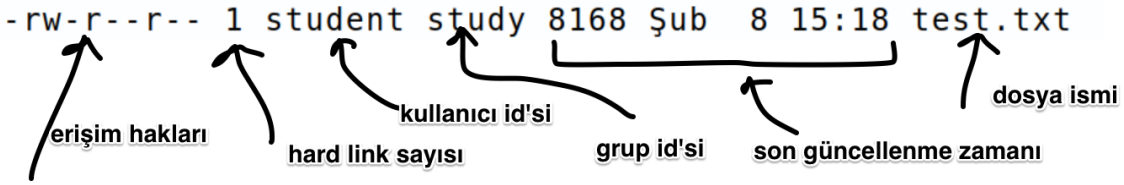
UNIX/Linux sistemlerinde her proses sistemdeki her dosya üzerinde yukarıda belirtilen işlemleri yapamaz. Bir dosyanın belli bir niyetle açılıp açılmaması o dosyanın erişim bilgilerine bağlıdır. Bu bölümde biz bu mekanizma üzerinde açıklamalar yapacağız.

Daha önce her prosesin gerçek ve etkin kullanıcı id'si ile gerçek ve etkin grup id'lerinin olduğunu görmüştük. İşte UNIX/Linux sistemlerinde aynı zamanda her dosyanın da bir kullanıcı id'si ve grup id'si vardır. (Dosyaların gerçek ve etkin kullanıcı id'lerinin, gerçek ve etkin grup id'lerinin olmadığına yalnızca kullanıcı ve grup id'lerinin olduğuna dikkat ediniz.) Bir dosyanın kullanıcı ve grup id'sinin ne olduğunu ls -l komutunu uygulayarak görebilirsiniz. Örneğin:




Burada test.txt dosyasının kullanı id'si student, grup id'si study biçimindedir. (Tabii aslında student kullanıcı ismi, study ise grup ismidir. Ancak bu isimlere karşı birer id geldiğini ve aslında çekirdeğin hep bu id'lerle işlem yaptığını anımsayınız.)

ls -l ile dosyaları görüntülediğinizde en soldaki sütun dosyanın erişim haklarını belirtmektedir:

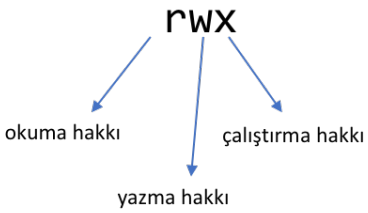



Dosyanın erişim hakları kimlerin dosya üzerinde hangi iişlemleri yapıp yapayacağını belirtmektedir. Erişim haklarının 10 karakterden oluştuğuna dikkat ediniz. Erişim haklarındakş en soldaki karakter dosyanın türünü belirtmektedir. Buradaki '-' dosyanın sıradan bir dosya olduğunu (regular file) olduğunu göstermektedir. Dosya türü olarak en çok bu '-' karakterini görürüz. En soldaki karakter '-'nin dışında şunlardan biri de olabilir:

'd': Dizin dosyası
'l': Sembolik link dosyası
'p': Boru (pipe) dosyası
's': Soket dosyası
'c': Karakter aygıt sürücü dosyası
'b': Blok aygıt sürücü dosyası

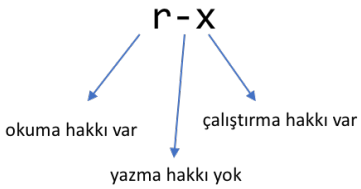
Dosya türünü belirten bu ilk karakterden sonraki 9 karakter üçerli üç gruba ayrılmaktadır. Bu üçerli gruplar "rwx" biçiminde oluşturulmaktadır. Eğer dosyaya okuma izni varsa burada 'r' karakterini, yoksa '-' karakterini, eğer dosyaya yazma hakkı varsa burada 'w' karakterini yoksa '-' karakteri bulunur. Benzer biçimde eğer dosya çalıştırılabilir

(executable) bir dosyaysa ve bu dosyanın belirttiği programı çalıştırma hakkı varsa burada 'x' karakteri, dosya çalıştırılabilir değil ya da çalıştırılabilir olduğu halde çalıştırma hakkı yoksa burada da '-' karakteri bulunacaktır.

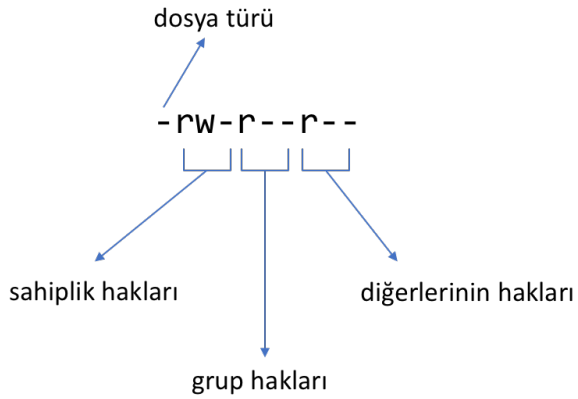



Aşağıdaki erişim haklarına bakınız:






Erişim haklarının üçerli üç gruptan oluştuğunu söylemiştik. Bu üçerli grubun ilkine "sahiplik (owner) hakları ", ikincisine "grup (group) hakları" ve üçüncüsüne de "diğerlerinin (other) hakları" denilmektedir. Örneğin:




Peki "sahiplik hakları", "grup hakları" ve "diğerlerinin hakları" ne anlam ifade etmektedir? Yukarıda da belirttiğimiz gibi bir dosyayı open fonksiyonuyla açmak isteyen programcı açış niyetini "yalnızca okuma amaçlı", "yalnızca yazma amaçlı" ya da "hem okuma hem de yazma amaçlı" biçiminde open fonksiyonunda belirtmektedir. İşte open fonksiyonu çağrıldığında open fonksiyonunun çağırdığı sistem fonksiyonu öncelikle dosyaya erişmek isteyen kişinin kim olduğuna bakar. Dosyayı açmak isteyen kişi şunlardan biri olabilmektedir:

- Dosyanın sahibi
- Dosyanın sahibi değil fakat dosyanın grubuyla aynı gruptan birisi
- Ne dosyanın sahibi ne de aynı gruptan birisi. Yani herhangi birisi.

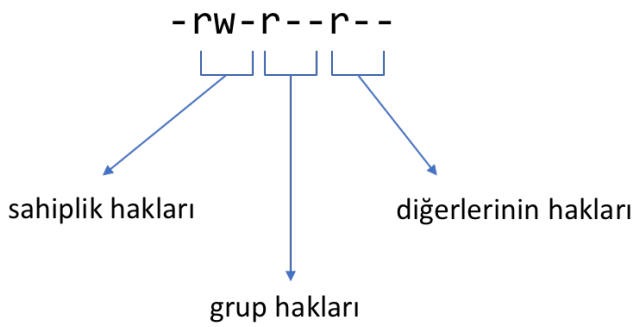
İşletim sistemi dosyayı açmak isteyen kişinin kim olduğuna ise şöyle karar vermektedir:

- Eğer dosyayı açmak isteyen prosesin etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynıysa dosyayı açmak isteyen kişi dosyanın sahibidir.

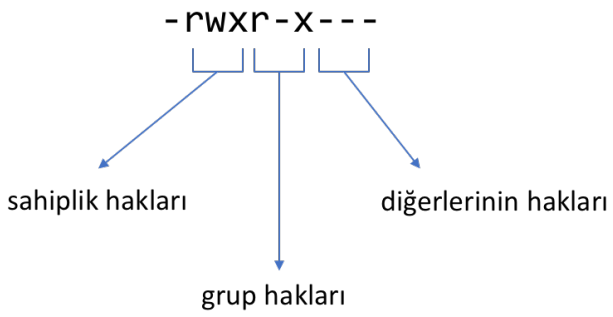
- Eğer dosyayı açmak isteyen prosesin etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynı değil fakat etkin grup id'si dosyanın grup id'si ile aynı ise bu durumda dosyayı açmak isteyen kişi dosya ile aynı grupta olan birisidir.

- Eğer dosyayı açmak isteyen prosesin etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynı değil ve etkin grup id'si de dosyanın grup id'si ile aynı değilse dosyayı açmak isteyen kişi diğer bir kişidir.

İşletim sistemi dosyayı açmak isteyenin kim olduğunu belirledikten sonra erişim haklarıyla açma niyetini karşılaştırmaktadır. Eğer dosyayı açmak isteyen kişi (yani proses) dosyanın sahibi ise dosyanın sahiplik haklarını, dosyayı açmak isteyen kişi (yani proses) dosyanın grubuyla aynı gruptan birisi ise dosyanın grup haklarını, dosyayı açmak isteyen herhangi bir kişi ise dosyanın diğerleri haklarını karşılaştırma işlemine sokar. Yani aslında dosyanın erişim hakları bu dosya için dosya sahibinin, aynı gruptan birisinin ve herhangi bir kişinin ne yapabileceğini belirtmektedir. Örneğin:

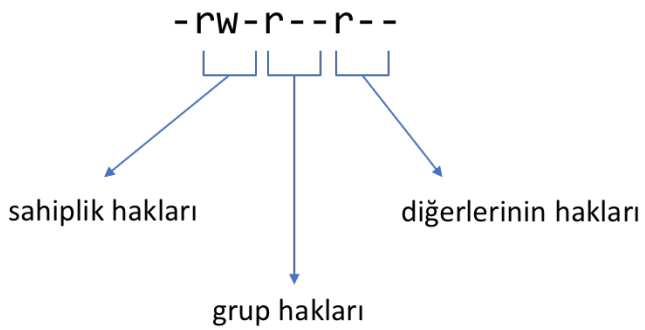



Bu dosyaya dosyanın sahibi okuma ve yazma işlemi, dosyanın grubuyla aynı gruptan birisi yalnızca okuma işlemi ve herahangi birisi de yalnızca okuma işlemi yapabilecektir. Aşağıdaki örneği inceleyiniz:




Bu dosyaya dosyanın sahibi okuma ve yazma işlemleri yapabilir ve bu dosyayı dosyanın sahibi çalıştırabilir. (Dosyanın çalıştırılabilir bir dosya olduğu anlaşılıyor.) Dosyanın grubuyla aynı olan kişi bu dosyadan yalnızca okuma yapabilir ve dosyayı çalıştırabilir. Ancak herhangi birisi bu dosya üzerinde hiçbir işlem yapamaz.

Dosyanın erişim hakları işletim i,stemi tarafından büyük ölçüde dosyanın açılması dırasında yapılmaktadır. Örneğin aşağıdaki haklara sahip bir dosya bulunuyor olsun:




Bu dosyayı open fonksiyonuyla dosyanın sahipi "okuma ve yazma" amacıyla açabilir. Ancak dosyanın grubuyla aynı gruptan bir kişi ya da herhangi bir kişi "okuma ve yazma amacıyla" açamaz. Eğer bu kişiler dosyayı open fonksiyonu ile okuma ve yazma amaçlı açmak isterlerse open fonksiyonu başarısız olacaktır. Ancak dosya ile aynı gruptan kişiler ya da

diğer kişiler bu dosyayı "yalnızca okuma" modunda açabilirler. (Bu durumda örneğin yalnızca okuma hakkınızın bulunduğu bir dosyayı "okuma ve yazma" amaçlı açmak istediğinizde daa açma sırasında başarısız olacaksınız, açtıktan sonra yazma işleminde değil.)

Önceki konularda da UNIX türevi sistemlerde bir kök kullanıcısının olduğunu ve buna "süper kullanıcı (super user)" ya da "ayrıcalıklı kullancı (priveleged user)" da denildiğini de belirtmiştirk. İşte kök kullanıcısı için çekirdek hiçbir erişim kontrolü yapmamaktadır. Yani kök kullanıcısı her türlü dosyadan okuma ve her türlü dosyaya yazma yapabilir. (Ancak kök kullanıcısı bile eğer dosyanın kullanıcı ya da grup ya da diğerleri kısmında hiç 'x' hakkı yoksa dosyayı çalıştıramamaktadır.)

**UNIX/Linux Sistemlerinde Dizinlerin Erişim Hakları**

UNIX/Linux sistemlerindeki dosya sistemlerinde aslında dizinler (directories) de birer dosya gibi ele alınmaktadır. Gerçekten de dizinlerin hem disk organizasyonu bakımından hem de diğer bakımlardan dosyalardan pek farkı yoktur. Dizinler aslında "içinde hangi dosyaların bulunduğu bilgilerini tutan dosyalar" biçiminde düşünülebilir. Nasıl normal bir dosyanın (regular file) içerisinde o dosyaya ilişkin bilgiler varsa dizin dosyalarının içerisinde de o dizinin içindeki dosyalara ilikin bilgiler vardır. Dizin dosyalarının içerisindeki bilgilerin organizasyonu kitabımızın "i-node tabanlı dosya sistemlerinin disk organziasyonlarının" anlatıldığı bölümde ele alınmaktadır. Örneğin dizinimizde doc isimli aşağıdaki gibi bir dizin bulunuyor olsun:

```
drwxr-xr-x 2 kaan study 4096 Şub 12 15:06 doc
```

Dizinin içerisinde de şu dosyaların bulunduğunu varsayalım:

```
-rw-r--r-- 1 kaan study   8168 Şub 12 15:05 a.txt
-rw-r--r-- 1 kaan study    316 Şub 12 15:05 b.txt
-rw-r--r-- 1 kaan study 115703 Şub 12 15:06 c.txt
```

Burada aslında doc isimli dizin bir dosya gibi organize edilmiştir. doc dizininin içini okuduğumuzda biz o dizindeki dosyalara ilişkin dizin girişlerini (directory entries) elde ederiz. Yani yukarıdaki örnekte aslında doc dosyası a.txt, b.txt ve c.txt dosyalarının bilglerini tutan bir dosya gibidir. Burada iki noktaya dikkat etmenizi isteyeceğiz:

1) Dizin dosyalarının içerisinde dosya bilgileri ls -l komutunda gördüğünüz gibi metin tabanlı (yani yazı biçiminde) bulunmaz. Dizin içindeki dosyaların girişleri dosya sistemine bağlı olarak ikili (binary) bir formatta tutulmaktadır.

2) Dizin içerisindeki dosyaların kendi verileri dizin dosyasında tutulmaz. Dizin dosyasında yalnızca o dosyalara ilişkin isim, i-node numarası gibi temel meta data bilgileri tutulmaktadır. (Yani örneğin yukarıdaki doc dizinindeki a.txt, b.txt, c.txt dosyalarının içerisindeki bilgiler dizin dosyasının içerisinde değildir. Yalnızca o dosyaları belirten temel bilgiler dizin dosyalarının içerisindedir.)

Pekiyi mademki dizinler de aslında her bakımdan birer dosya gibi işlem görüyor, o halde dizinler için "read" hakkı ve "write" hakkı ne anlam ifade etmektedir? İşte dizinler bir dosya olduğuna göre ve dizin dosyaları dizindeki dosyaların bilgilerini tuttuğuna göre o dizinde yeni bir dosya yaratıldığında ya da o dizindeki bir dosya silindiğinde dizin dosyasında bir güncelleme yapılacaktır. O halde dizinler için "write" hakkı aslında "o dizin içerisinde yeni bir dosya oluşturulabilir" ya da "o dizinden bir dosya silinebilir" anlamındadır. Gerçekten de "w" hakkına sahip olmadığımız bir dizinde yeni bir giriş yaratamayız ve mevcut bir dizin girişini de silemeyiz. Dilerseniz bunu basit bir biçimde şöyle test edelim. Yukarıdaki doc dizininden önce sahiplikteki "w" hakkını kaldıralım:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ chmod u-w doc
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -ld doc
dr-xr-xr-x 2 kaan study 4096 Şub 12 15:06 doc
```

Şimdi dizine geçip bir dosyayı silmeye çalışalım:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ cd doc
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg/doc$ rm a.txt
rm: 'a.txt' silinemedi: Erişim engellendi
```

Bir dizinden dosya silmek için dosyaya "write" hakkımızın olmasının gerekmediğine, o dosyanın içinde bulunduğu dizine "write" hakkımızın olması gerektiğine dikkat ediniz.

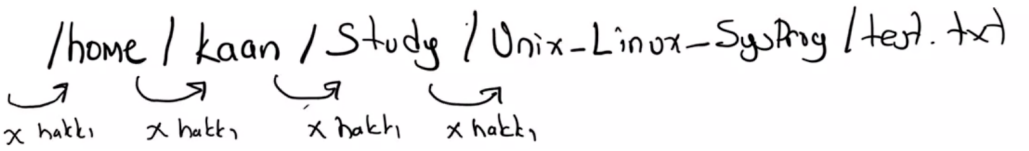
Şimdi de dizin içerisinde bir dosya yaratmak isteyelim:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg/doc$ touch d.txt
touch: `touch' 'd.txt' yapılamadı: Erişim engellendi
```

Yaratmış olduğunuz bir dizindeki 'x' hakları dikkatinizi çekmiş olabilir. Dosya erişim haklerındaki 'x' haklarının "çalıştırılabilir dosyayı çalıştırma" hakkı olduğunu belirtmiştik. Ancak dizinlerdeki 'x' haklarının bununla bir ilişkisi yoktur. Dizinlerdeki 'x' hakkı "yol ifadesinde içinden geçilebilirlik" anlamına gelmektedir. Örneğin aşağıdaki yol ifadesi ile bir dosyayı açmak isteyelim:

```
"/home/kaan/Study/Unix-Linux-SysProg/test.txt"
```

Bu yol ifadesi ile belirtilen test.txt dosyasına ulaşılabilmesi için prosesin bu yol ifadesindeki bütün dizinlere 'x' hakkının olması gerekmektedir. Eğer bir yol ifadesindeki bir dizin bileşenine prosesin 'x' hakkı yoksa yol ifadesini parametre alarak işlem yapmak isteyen tüm fonksiyonlar yol ifadesinin çözümlenmesi (pathname resolution) aşamasında EACCESS isimli hata ile geri dönerler.




Tabii daha önceden de ifade ettiğimiz gibi işletim sistemi çekirdeği 0 numaralı kullanıcı id'sine sahip olan kök proseslerde herhangi bir erişim kontrolü uygulamamaktadır. (Yani örneğin 0 numaralı kullanıcı id'sine sahip olan bir proses yukarıdaki yol ifadesindeki bazı dizinler için 'x' hakları olmasa bile söz konusu dosyaya erişebilmektedir.

Dizinler için 'x' hakları göreli yol ifadeleri için de geçerlidir. Yani örneğin:

```
"test.txt"
```

biçimindeki göreli bir yol ifadesinde prosesin çalışma dizinine ilişkin dizin için 'x' hakına sahip olması gerekir. (Ancak çalışma dizinine kadarki mutlak yol ifadelerindeki dizinlere o anda 'x' hakkına sahip olunmamasının bir önemi yoktur.) Benzer biçimde örneğin:

```
"doc/a.txt"
```

gibi bir yol ifadesinde hedeflenen "a.txt" dosyasına erişilebilmesi için prosesin hem çalışma dizinine hem de "doc" dizinine 'x' hakkının olması gerekir. Bu yol ifadesini "./dos/a.txt" biçiminde de düşünebilirsiniz.

Dizinlerdeki 'x' haklarının her türlü yol ifadesi için uygulandığını vurgulayalım. Başka bir deyişle işletim sisteminin yol ifadelerini parametre alan tüm sistem fonksiyonları bu 'x' hakkı kontrollerini yapmaktadır. Pekiyi dizinlerde 'x' hakkının amacı ne olabilir? Aslında bu hak başkalarının belli bir dizinde daha ileriye gidememelerini sağlamak için düşünülmüştür. Bu sayede kullanıcı kendi dizin ağacında ana bir dizinin 'x' hakkını başkaları için kaldırarak onların bu ağaca erişmesini engelleyebilmektedir.

**Mutlak ve Göreli Yol İfadeleri (Absolute and Reletive Paths)**

Dosyaların yerlerini belirten yazısal ifadelere "yol ifadeleri (path names)" denilmektedir. İşletim sistemleri dosyaları ikincil belleklerde yol ifadelerini kullanarak bulurlar. Bu nedenle dosyalar üzerinde işlemler yapan pek çok POSIX fonksiyonu ve dolayısıyla da sistem fonksiyonu işlem yapacakları dosyaların yol ifadelerini bizden parametre yoluyla isterler. UNIX/Linux sistemlerinde yol ifadelerinde dizin geçişleri '/' karakteriyle belirtilmektedir. (Windows sistemlerinde bunun için '\' karakterinin kullanıldığını anımsayınız.) Yol ifadelerinde dizin geçişleri sırasında bir'den fazla '/' karakteri de kullanılabilmektedir. Ayrıca UNIX/Linux sistemlerinde Windows sistemlerinde olduğu gibi "sürücü (drive)" kavramı da yoktur. Dolayısıyla bu sistemlerde toplamda yalnızca bir tane dizin ağacı vardır. Dizin ağacındaki en dıştaki dizine "kök dizin (root directory)" denir ve tüm dizinler doğrudan ya da dolaylı olarak kök dizinin içerisindedir.

Yol ifadeleri "mutlak (absolute)" ve "göreli (relative)" olmak üzere ikiye ayrılmaktadır. İlk karakterleri '/' olan (yani '/' karakteri ile başlayan) yol ifadeleri mutlak, ilk karakterleri '/' olmayan (yani '/' karakteri ile başlamayan) yoli ifadeleri görelidir. Mutlak yol ifadeleri için şu örnekleri verebiliriz:

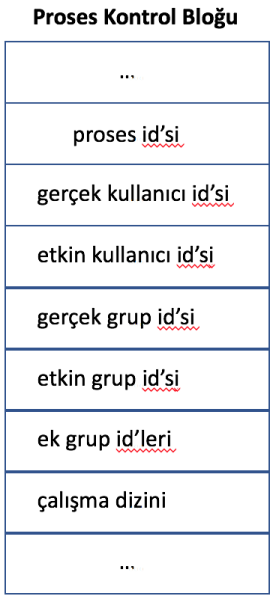
```
"/home/kaan/Study/Unix-Linux-SysProg/test.txt"
"/usr/include/stdio.h"
"/bin/ls"
"/proc/version"
"/dev/tty0"
```

Göreli yol ifadeleri için de şöyle örnekler verebiliriz:

```
"Study/test.txt"
"sample.c"
"Study/SysProg/sample.c"
```

Mutlak yol ifadeleri kök dizinden itibaren bir yol belirtmektedir. Örneğin "/usr/include/stdio.h" biçimindeki yol ifadesi bize kök dizinin altında "usr" dizininin olduğunu, "usr" dizinini altında "include" dizininin olduğunu, "include dizinin de altında "stdio.h" isimli dosyanın bukunduğunu belirtmektedir.

Pekiyi göreli yol ifadeleri nereden itibaren yol belirtmektedir? İşte UNIX/Linux sistemlerinde göreli yol ifadeleri prosesin çalışma dizininden itibaren yol belirtirler. Her prosesin proses kontrol bloğunda saklanan (Linux sistemlerinde "include/linux/sched.h" dosyasındaki task_struct yapısı) bir "çalışma dizini (current working directory)" bilgisi vardır. Prosesin çalışma dizini o anda ne ise (yani prosesin kontrol bloğunda belirtilen çalışma dizini ne ise) göreli yol ifadeleri orası orijin yapılarak yer belirtmektedir. Başka bir deyişle proseslerin çalışma dizinleri göreli yol ifadeleri için orijin noktası görevini yapmaktadır. UNIX/Linux sistemlerinde proseslerin çalışma dizinleri chdir isimli POSIX fonksiyonuyla değiştirilebilmektedir. Şimdiye kadar gördüğümüz kavramlar eşliğinde proses kontrol bloğunda tutulan bilgileri aşağıdaki şekille özetlemek istiyoruz:



Örneğin prosesimizin çalışma dizini "/home/kaan" olsun. Biz de "Study/test.txt" biçiminde göreli bir yol ifadesi verelim. Bu durumda bu göreli yol ifadesinin mutlak yol ifadesi karşılığı "/home/kaan/Study/test.txt" olacaktır. Benzer biçimde "test.txt" biçiminde yalnızca isimle belirtilen göreli yol ifadelerindeki dosyaların prosesin çalışma dizinlerinde aranacağına dikkat ediniz.

Pekiyi bir program çalışmaya başladığında yani proses yaratıldığında onun çalışma dizini neresidir? İşte UNIX/Linux sistemlerinde prosesin çalışma dizini pek çok diğer özellikte olduğu gibi proses yaratılırken üst prosesten alınmaktadır. Yani proses yaratıldığı sırada üst prosesin çalışma dizini neyse alt prosesin de çalışma dizini o olmaktadır. Daha önce terminal yoluyla ya da XWindow sistemi yoluyla login olduğumuzda bizi sisteme sokan prosesin çalışma dizinimizi bu dosyada belirtilen biçimde ayarladığını söylemiştik.

Son olarak yol ifadelerinde kullanılan "." ve ".." dizinlerinden bahsetmek istiyoruz. Yol ifadelerindeki "." dizini soldaki dizin ile aynı dizin anlamına gelmektedir. Örneğin:

```
"/home/./kaan/./Study/sample.c"
```

Bu yol ifadesindeki '.' dizinlerinin aslında bir etkisi yoktur. Yani yukarıdaki yol ifadesi aslında aşağıdaki ile eşdeğerdir:

```
"/home/kaan/Study/sample.c"
```

Eğer göreli yol ifadesi "." dizini ile başlatılırsa bu "." prosesin çalışma dizinini belirtir. Örneğin:

```
"./sample.c"
```

yol ifadesi ile:

```
"sample.c "
```

yol ifadesi eşdeğerdir. Tabii siz normal olarak yukarıdaki örneklerden yol ifadelerindeki "." dizinlerinin gereksiz olduğunu düşünebilirsiniz. Ancak bu "." dizinine bazen gereksinim duyulabilmektedir. Örneğin kabuk programında kabuk programının çalışma dizinindeki sample isimli bir programı yalnızca ismini yazarak değil "." dizinini belirterek aşağıdaki gibi çalıştırdığımızı anımsayınız:

```
./sample
```

Halbuki bu yol ifadesi "sample" yol ifadesi ile eşdeğer olduğu halde biz bu programı kabuk üzerinden şöyle çalıştıramayız:

```
sample
```

Bunun nedenini ilerideki bölümlerde anlayacaksınız.

Yol ifadelerindeki ".." dizini onun solundaki dizinin üst dizini anlamına gelmektedir. Örneğin:

```
"usr/include/../include/stdio.h"
```

yol ifadesi ile aşağıdaki yol ifadesi eşdeğerdir:

```
"usr/include/stdio.h"
```

Göreli yol ifadeleri ".." ile başlatılırsa bu ".." prosesin çalışma dizininin üst dizinini belirtir. Örneğin:

```
"../ali/sample.c"
```

yol ifadesi prosesin çalışma dizininin üzet dizini içerisindeki "ali" dizininin altındaki "sample.c" dosyasını belirtmektedir.

## UNIX/Linux Sistemlerinde Dosya İşlemleri

UNIX türevi işletim sistemlerinin belki de en önemli alt sistemi dosya sistemidir. Bu nedenle bu sistemlerde programlama yapmak isteyen kişilerin ilk öğrenmesi gereken şey temel dosya işlemleridir. UNIX/Linux sistemlerinde dosya işlemleri için pek çok POSIX fonksiyonu bulundurulmuştur. Bu fonksiyonların çoğu ilgili sistemde o sistemin sistem fonksiyonlarını çağırarak işlemlerini yaparlar. UNIX türevi sistemlerde hangi programlama dilinde ya da platformda çalışıyor olursanız olun neticede eninde sonunda işlemler aşağı seviyeli olarak ilgili POSIX fonksiyonları çağrılarak gerçekleşmektedir. Daha önce de belirttiğimiz gibi örneğin C Programlama Dilindeki fopen, fclose, fread, fwrite, fseek gibi fonksiyonlar eninde sonunda aslında bu sistemlerde open, close, read, write, lseek gibi POSIX fonksiyonlarını çağırmaktadır. UNIX türevi sistemlerde sistem programlama faaliyetleri yapacak olan programcıların dosya sistemine ilişkin bu aşağı seviyeli POSIX fonksiyonlarını iyi bir biçimde öğrenmesi gerekmektedir.

Şimdi bir C programcısı olarak aklınıza şu soru gelebilir: Biz dosya işlemlerini başı f ile başlayan standat C fonksiyonlarıyla yapmak yerine neden POSIX fonksiyonları ile yapalım? Standart C fonksiyonlarını kullanmak yerine POSIX fonksiyonlarını kullanmanın bizim için ne gibi ek faydaları var? Bu klasik sorunun yanıtını şöyle vermek istiyoruz:

1) C'nin standart dosya fonksiyonları taban fonksiyonlar değildir. Daha yüksek seviyeli fonksiyonlardır. Oysa sistem programlama yapabilmek için daha aşağı seviyeli çalışmak gerekir.

2) C'nin dosya fonksiyonları tüm sistemlerde olabilecek ortak özellikler dikkate alınarak dar bir işlevsellikle tasarlanmışlardır. Halbuki örneğin UNIX/Linux'un dosya sistemi (ya da Windows'un dosya sistemi) daha geniş olanaklar sunmaktadır. Bu olanaklardan faydalanabilmek için daha aşağı seviyeli POSIX fonksiyonlarıyla çalışmak gerekir.

3) UNIX/Linux sistemlerindeki pek çok dosya işlemi standart C fonksiyonlarıyla yapılamamaktadır. Yani bazı dosya işlemleri için mecburen POSIX fonksiyonlarının kullanılması gerekmektedir.

Biz bu bölümde dosya fonksiyonlarını işlemlerini iki gruba ayırarak inceleyeceğiz:

**1) Temel Dosya Fonksiyonları:** Bunlar dosyayı açmak, kapamak, dosyadan okuma yazma yapmak ve dosya göstericisini konumlandırmak için kullanılan POSIX fonksiyonalrıdır. Bu aşağı seviyeli dosya fonksiyonları pek çok UNIX türevi sistemde bire bir o işletim sisteminin sistem fonksiyonlarını çağırmaktadır. UNIX/Linux sistemlerinde dosya temel olarak open isimli POSIX fonksiyonuyla açılır ve close isimli POSIX fonksiyonuyla kapatılır. Dosyadan read POSIX fonksiyonuyla okuma yaparız ve dosyaya write POSIX fonksiyonuyla yazma yaparız. Dosya göstericisinin konumlandırılması için lseek isimli POSIX fonksiyonu kullanılmaktadır.

**2) Yardımcı Dosya Fonksiyonları:** Temel dosya fonksiyonlarının dışında dosyalar üzerinde bütünsel işlemler yapan pek çok POSIX fonksiyonu vardır. Örneğin yaratılmış bir dosyanın erişim haklarının değiştirilmesi. Dosyanın bilgilerinin elde edilmesi, dosyanın isminin değiştirilmesi, dosyanın silinmesi bu tür yardımcı POSIX fonksiyonlarıyla yapılmaktadır.

## UNIX/Linux Sistemlerinde Temel Dosya Fonksiyonları

UNIX türevi sistemlerde dosyanın açılması ve kapatılması, dosyadan okuma yapılması, dosyaya yazma yapılması, dosya göstericisinin konumlandırılması temel dosya işlemlerini oluşturmaktadır. Sistem programcının bu işlemler için kullanılan POSIX arayüzünü çok iyi bilmesi gerekmektedir. Yukarıda da belirttiğimiz gibi aslında bu işlemler için kullanılan POSIX fonksiyonları pek çok sistemde doğrudan ilgili sistem fonksiyonlarını çağırmaktadır.

## open Fonksiyonu

Bir dosya üzerinde işlem yapabilmek için öncelikle o dosyanın açılması gerekir. Dosyanın açılması sırasında işletim sistemi açılacak dosya ile ilgili bazı hazırlık işlemleri yapmaktadır. Bu hazırlık işlemlerinin neler olduğu ileriki bölümlerde ayrıntılı olarak ele alınacaktır. Bir dosya ile işlemlerimiz bittiğinde de normal olarak o dosyayı kapatmamız gerekir. Dosyanın kapatılması sırasında açılması ile yapılan hazırlık işlemleri geri alınmaktadır. Yani dosya kapatıldıktan sonraki durum açılmadan önceki durum ile aynı olur. Aslında işletim sistemlerinin çoğunda olduğu gibi UNIX/Linux sistemlerinde de dosyaların kapatılması mutlak zorunlu bir işlem değildir. İşletim sistemi hangi proseslerin hangi dosyaları açtığının kaydını tuttuğu için prosesin sonlanması durumunda o prosesin açmış olduğu dosyaları kendisikapatabilmektedir. Tabii

ne olursa olsun programcının bir dosya ile işi bittiğinde o dosyayı kapaması iyi bir tekniktir. Çünkü açık dosyalar belli miktarlarda sistem kaynaklarının harcanmasına yol açmaktadır.

UNIX türevi sistemlerde dosyalar temel olarak open isimli POSIX fonksiyonuyla açılır ve close isimli POSIX fonksiyonuyla da kapatılır. open fonksiyonu yalnızca olan dosyayı açmak için değil aynı zamanda yeni bir dosyayı yaratmak için de kullanılmaktadır. open fonksiyonunun prototipi şöyledir:

```
#include <fcntl.h>

int open(const char *path, int oflag, ...);
```

Fonksiyonun prototipindeki ... (ellipsis) fonksiyonun değişken sayıda parametre alabileceğini belirtmektedir. Fakat aslında open fonksiyonu ya iki argümanla ya da üç argümanla çağrılabilmektedir. Fonksiyonun üçten fazla argümanla çağrılması tanımsız davranışa yol açmaktadır. open fonksiyonunun prototipi <fcntl.h> dosyası içerisindedir.

open fonksiyonun birinci parametresi açılacak dosyanın yol ifadesini belirtir. İkinci parametre ise açış modunu belirtmektedir. Dosya açış modları O_XXXXX biçiminde bayrak (flag) diye isimlendirilen çeşitli sembolik sabitlerin bit düzeyine veya (bitwise or) işlemine sokulmasıyla oluşturulmaktadır. Bu sembolik sabitler genel olarak tüm bitleri 0 yalnızca tek bitleri 1 olan sayılar biçimindedir. Bunlar bir düzeyinde veya işlemine sokulduklarında elde edilen değerin birden fazla biti 1 olur. Fonksiyon da bu değeri yeniden bitlerine ayrıştırarak hangi bayrakların kullanıldığını anlayabilmektedir.

Fonksiyonun ikinci parametresi <fcntl.h> dosyası içerisinde bildirilmiş olan aşağıdaki açış bayraklarından yalnızca bir tanesini içermek zorundadır:

```
O_RDONLY
O_WRONLY
O_RDWR
```

O_RDONLY dosyanın yalnızca okuma yapmak için açılacağını, O_WRONLY dosyanın yalnızca yazma yapmak için açılacağını, O_RDWR ise dosyanın hem okuma hem de yazma yapmak amacıyla açılacağını belirtmektedir. Bu bayraklar bir arada kullanılmaz ve bunların bir tanesinin mutlaka kullanılması gerekmektedir. POSIX standartları yalnızca bir tane kullanılması gereken açış bayraklarına zamanla aşağıdaki iki eklemeyi de yapmıştır:

```
O_EXEC
O_SEARCH
```

Aslında bu bayraklar yalnızca birkaç fonksiyon için gerek ve çok seyrek gereksinim duyulan bayraklardır. (Linux çekirdeği henüz bu açış bayraklarını desteklememektedir.) Bu açış bayraklarına yeri geldiğinde başka konularda değinilecektir. Yukarıda belirttiğimiz mutlaka kullanmanız gereken açış bayraklarının dışında bunlarla birlikte en çok kullanılan bayraklar da şunlardır:

O_CREAT: Bu bayrak "dosya yoksa yarat varsa olanı kullan" anlamına gelmektedir. Yani eğer dosya yoksa bu bayrak dosyanın yaratılmasına yol açacaktır. Ancak eğer dosya varsa bu bayrağın bir etkisi olmayacaktır. Yeni bir dosya yaratmak için bu bayrağın mutlaka kullanılması gerekmektedir.

O_EXCL: Bu bayrak tek başına kullanılamaz. O_CREAT bayrağı ile birlikte kullanılabilir. (Bu bayrağın tek başına kullanılması tanımsız davranışa yol açmaktadır.) O_CREAT moduna bu mod eklenirse "eğer dosya varsa açma işlemi başarısız" olmaktadır. Yani O_CREAT|O_EXCL yalnızca olmayan dosyanın yaratılmasını sağlar.

O_TRUNC: Bu mod dosya açıldıktan sonra içinin sıfırlanacağı anlamına gelir. Bu bayrak açış moduna eklenirse dosyanın içi tamamen silinerek dosya açılmaktadır. Bu bayrağın kullanılabilmesi için dosyanın yazmaya izin veren biçimde yani O_WRONLY ya da O_RDWR bayraklarıyla açılması gerekmektedir.

O_APPEND: Bu bayrak da eğer dosya yazma amacıyla açılmışsa anlamlıdır. Eğer bu bayrak açış moduna eklenirse her türlü yazma işlemi öncesinde dosya göstericisi otomatik olarak EOF durumuna konumlandırılır. Yani bu bayrak dosyaya yapılan yazma işlemlerinin hep dosyanın sonuna yapılmasına yol açmaktadır.

O_DIRECTORY: Aslında dizinlerin de dosyalardan hiçbir farklı yoktur. Yani biz dizinleri de dosyalar gibi open fonksiyonuyla açabiliriz. (Ancak open fonksiyonu dizinlerin yaratılmasında kullanılamamaktadır. Dizinlerin yaratılması mkdir isimli başka bir POSIX fonksiyonuyla yapılmaktadır.) İşte O_DIRECTORY bayrağı belirtildiğinde eğer açılmak istenen dosya bir dizin değilse open fonksiyonu başarısız olur. Yani O_DIRECTORY bayrağı yalnızca dizin dosyalarını açmak için kullanılmaktadır. Eğer bu bayrak kullanılmazsa biz open ile normal dosyaları da dizin dosyalarını da açabiliriz.

Yukarıdaki açış bayraklarının dışında aslında başka açış açış bayrakları da vardır. Ancak bu bayraklar başka konularla ilişkili olduğu için o konuların anlatıldığı yerlerde ele alınacaktır. Biz burada bu diğer açış bayraklarını listelemekle yetinelim:

```
O_CLOEXEC
O_DSYNC
O_NOCTTY
O_NOFOLLOW
O_NONBLOCK
O_RSYNC
O_SYNC
O_TTY_INIT
```

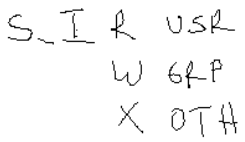
Şimd standart fopen fonksiyonundaki açış modlarının open bayrak karşılıklarını listeleyelim:

| fopen Açış Modu | open Karşılığı |
|---|---|
| "r" | O_RDONLY |
| "r+" | O_RDWR |
| "w" | O_WRONLY\|O_CREAT\|O_TRUNC |
| "w+" | O_RDWR\|O_CREAT\|O_TRUNC |
| "a" | O_WRONLY\|O_CREAT\|O_APPEND |
| "a+" | O_RDWRD\|O_CREAT\|O_APPEND |

Şimdi gelelim open fonksiyonunun üçüncü parametresine. Yukarıda da belirttiğimiz gibi biz open fonksiyonunu iki argümanla ya da üç argümanla çağırabiliriz. Daha önceki konularda bir dosyanın erişim haklarını ilk kez dosyayı yaratan kişinin belirlediğini söylemiştik. İşte open fonksiyonunun üçüncü parametresi eğer dosya yaratılacaksa yaratılacak dosyanın erişim haklarını almaktadır. Yani biz eğer dosyayı yaratacaksak bu üçüncü parametre için argüman girmeliyiz. Dosyanın açış modunda O_CREATE bayrağının "dosya yoksa yarat" anlamına geldiğini söylemiştik. O halde fonksiyonun ikinci parametresinde O_CREAT belirtilmişse bizim üçüncü parametre için argüman girmemiz gerekir. open fonksiyonunun bu üçüncü parametresi mode_t türündendir. POSIX standartları mode_t türünün bir tamsayı türü olarak typedef edilmesi gerektiğini belirtmektedir. open fonksiyonunun bu üçüncü parametresi de birtakım sembolik sabitlerin bit veya işlemine sokulmasıyla oluşturulur. Bu sembolik sabitler <sys/stat.h> dosyası içersinde bildirilmiştir. (Bu durumda dosyayı yaratma gibi bir olasılığınız varsa <sys/stat.h> dosyasını da include etmeniz gerekir.) Erişim hakları için kullanılan sembolik sabitler şunlardır:

```
S_IRUSR
S_IWUSR
S_IXUSR

S_IRGRP
S_IWGRP
S_IXGRP

S_IROTH
S_IWOTH
S_IXOTH
```

Bu sembolik sabitlerin isimleri ilk bakışta size karmaşık gelebilir. Bu nedenle sizin daha kolay algılamanızı sağlmak için bir açıklama yapmak istiyoruz. Buradaki tüm sembolik sabitlerin S_I öneki ile başladığına dikkat ediniz. S_I önekini R, W ya da X karakterleri izlemektedir. R (read), W (write) ve X (execute) anlamına gelir. R, W, X karakterlerini de USR, GRP ya da OTH karakterleri izlemektedir. USR (user) dosyanın sahiplik bilgisini, GRP (group) dosyanın grup bilgisini, OTH (other) ise dosyanın diğer kişiler anlamına gelmektedir.




Bu durumda örneğin:

```
S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH
```

Açış modu aslında "rw-r--r--" anlamına gelmektedir. Örneğin:

```
S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
```

erişim hakları ise "rw-rw-rw-" anlamına gelir. Eğer bu parametre için 0 değerini argüman olarak geçersek bu da "---------" açış modu anlamına gelecektir.

open fonksiyonunun son parametresi konusunda iki uyarıda bulunmak istiyoruz: Birincisi bu parametre için chmod komutundaki gibi octal bir değer girmeyiniz. Örneğin 0644 gibi octal bir değer chmod komutunda "rw-r--r--" anlamına gelir. Halbuki genel olarak POSIX sistemlerinde bu değerlerin yukarıdaki sembolik sabitlere karşılık geleceğinin bir garantisi yoktur. İkinci uyarımız ise open fonksiyonundaki bu argümanı girmeyi unutmakla ilgili. Eğer open fonksiyonunun ikinci parametresi için O_CREAT bayrağı girilmişse dosyanın yaratılma olasılığı olduğu için mutlaka üçüncü parametre için argüman girmelisiniz. Eğer dosya yaratılacak olduğu halde bu üçüncü parametre için argüman girmezseniz fonksiyon stack'ten rastgele çektiği çöp değerlerden bu argümanı oluşturacaktır. Bunun da teknik anlamı tanımsız davranıştır.

Üçüncü parametre için kullanılacak üç sembolik sabit daha vardır:

S_IRWXU
S_IRWXG
S_IRWXO

S_IRWXU sahiplik için tüm hakların, S_IRWXG grup için tüm hakların ve S_IRWXO ise diğer kişiler için tüm hakların olduğunu belirtmektedir. Bu sembolik sabitler aşağıdaki gibi oluşturulmuştur:

```
#define S_IRWXU     (S_IRUSR|S_IWUSR|S_IXUSR)
#define S_IRWXG     (S_IRGRP|S_IWGRP|S_IXGRP)
#define S_IRWXO    (S_IROTH|S_IWOTH|S_IXOTH)
```

open fonksiyonu başarı durumunda ismine "dosya betimleyicisi (file descriptor)" denilen, açılan dosyaya ilişkin bir handle değerine geri döner. open fonksiyonunun geri verdiği bu dosya betimleyicisi çekirdek için açılan dosyayı temsil etmektedir. Dosya betimleyicisi read, write, lseek, close gibi fonksiyonlara parametre olarak geçirilir. Bu fonksiyonlar böylece hangi dosya üzerinde işlem yapacaklarını anlamış olurlar. Ayrıca POSIX standartlarında open fonksiyonunun ilk boş betimleyici değerine (yani en düşük değerli betimleyici ile) geri dönceği garanti edilmiştir. Betimleyicilerin ne anlam ifade ettiği ve ilk boş betimleyicinin ne olduğu ilerideki bölümlerde ele alınmaktadır.

Şimdi open fonksiyonun kullanımına birkaç örnek verelim:

```
int fd;
```

```
if ((fd = open("test.txt", O_RDONLY)) == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

Burada olan dosya yalnızca okuma amacıyla açılmak istenmiştir. Örneğin:

```
int fd;

if ((fd = open("test.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

Burada dosya hem okuma hem de yazma yapma niyetiyle açılmak istenmiştir. Dosya yoksa yaratılacaktır, varsa olan dosya açılacaktır. Dosya yoksa yaratılacak dosya için erişim haklarının "rw-r--r--" biçiminde verildiğine dikkat ediniz. Örneğin:

```
int fd;

if ((fd = open("test.txt", O_WRONLY|O_TRUNC)) == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

Burada olan dosya yalnızca yazma amaçlı açılmak istenmiştir. Dosya varsa sıfırlanacaktır.

Aklınıza şöyle bir soru gelebilir: "open fonksiyonunda yarattığım dosyanın erişim haklarıyla çelişen bir açış modu verirsem dosya açılabilir mi?" Örneğin:

```
int fd;

if ((fd = open("test.txt", O_RDWR|O_CREAT, 0)) == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

open fonksiyonun üçüncü parametresi için verdiğiniz erişim hakları bu çağrı üzerinde etkili olmamaktadır. Yani yukarıdaki örnekte dosya yoksa onu hem okuma hem de yazma amacıyla yaratabileceksiniz.

Şimdi bir prosesin open fonksiyonu ile bir dosyayı açmak istediğinde bu işlemi yapan sistem fonksiyonunun yaptığı kontrolleri adım adım açıklamak istiyoruz. Aşağıdaki maddeler "değilse (yani else if)" biçiminde yorumlanmalıdır. Yani örneğin ikinci maddeye geçmek için birinci maddenin başındaki koşulun sağlanmıyor olması olması gerekir. Başka bir deyişle bir madde için o maddenin başındaki koşul sağlanıyorsa diğer maddelere bakılmaz.

1) Dosya açma işlemini yapan sistem fonksiyonu dosyayı açmak isteyen prosesin etkin kullanıcı id'sinin 0 olup olmadığına bakar. Eğer prosesin etkin kullanıcı id'si 0 ise (yani söz konusu proses root prosesi ise) istek kabul edilir. Herhangi bir kontrol yapılmaz.

2) Dosya açma işlemini yapan sistem fonksiyonu dosyayı açmak isteyen prosesin etkin kullanıcı id'sinin dosyanın kullanıcı id'si ile aynı olup olmadığına bakar. Eğer prosesin etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynıysa dosyanın sahiplik hakları (yani 9'luk erişim haklarının ilk üçü) ile open fonksiyonunda belirtilen bayraklara bakarak kontrol uygular. Eğer dosya O_RDONLY bayrağıyla açılmak istenmişse sahiplik haklarının 'r' özelliğine, dosya O_WRONLY bayrağıyla açılmak istenişse 'w' özelliğine ve dosya O_RDWR bayrağıyla açılmak istenmişse de 'rw' özelliklerine sahip olması gerekir. Ayrıca yol fadesindeki bütün dizinler için prosesin 'x' hakkına sahip olması gerekmektedir.

3) Dosya açma işlemini yapan sistem fonksiyonu dosyayı açmak isteyen prosesin etkin grup id'sinin ya da ek grup id'lerinden herhangi birinin dosyanın kullanıcı id'si ile aynı olup olmadığına bakar. Eğer prosesin etkin grup id'si ya da ek

grup id'lerinden herhangi biri dosyanın grup id'si ile aynıysa dosyanın grup hakları (yani 9'luk erişim haklarının ikinci üçlüğü) ile open fonksiyonunda belirtilen bayraklara bakarak kontrol uygular. Eğer dosya O_RDONLY bayrağıyla açılmak istenmişse sahiplik haklarının 'r' özelliğine, dosya O_WRONLY bayrağıyla açılmak istenişse 'w' özelliğine ve dosya O_RDWR bayrağıyla açılmak istenmişse de 'rw' özelliklerine sahip olması gerekir. . Ayrıca yol fadesindeki bütün dizinler için prosesin 'x' hakkına sahip olması gerekmektedir.

4) Dosya açma işlemini yapan sistem fonksiyonu dosyanın diğerleri için haklarına (yani 9'luk erişim haklarının üçüncü üçlüğü) ile open fonksiyonunda belirtilen bayraklara bakarak kontrol uygular. Eğer dosya O_RDONLY bayrağıyla açılmak istenmişse sahiplik haklarının 'r' özelliğine, dosya O_WRONLY bayrağıyla açılmak istenişse 'w' özelliğine ve dosya O_RDWR bayrağıyla açılmak istenmişse de 'rw' özelliklerine sahip olması gerekir. . Ayrıca yol fadesindeki bütün dizinler için prosesin 'x' hakkına sahip olması gerekmektedir.

Şimdi etkin kullanıcı id'si ali ve etkin grup id'si study olan aşağıdaki gibi bir dosya bulunduğunu varsayalım:

```
-rw-r--r-- 1 ali study 4 Şub  7 12:40 notes.txt
```

Etkin kullanıcı id'si kaan olan prosesin dosyayı aşapıdaki gibi açmaya çalıştığını varsayalım:

```
int fd;

if ((fd = open("test.txt", O_RDWR)) == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

Burada open fonksiyonunu çağıran proses dosyanın sahibi değildir ancak dosyanın grubuyla aynı gruptadır. Bu durumda erişim haklarının grup kısmı kontrole sokulacaktır. Erişim haklarının grup kısmının "r--" biçiminde olduğunu görüyorsunuz. Dosya O_RDWR modunda açılmaya çalışıldığı için open fonksiyonu başarısız olacak ve errno değeri EACCESS ile set edielecektir.

Şimdi ilginç bir doru soralım: Bir dosyanın sahibi kendi dosyasına erişim hakkını kaldırır da başkalarına açarsa bu durumda o dosyayı aaçabilir mi? Örneğin test.txt dosyasının erişim hakları şöyle olsun:

```
----r--r-- 1 kaan study 27 Şub 10 17:56 test.txt
```

Bu dosyayı etkin kullanıcı id'si kaan olan prosese O_RDONLY moduyla açamaz. Ancak aynı gruptan prosesler ve herhangi diğer prosesler bu modda açabilirler. Burada etkin kullanıcı id'si kaan olan prosesin grup id'si study olsa bile bu proses dosyayı O_RDONLY modunda açamayacaktır. (Çünkü eğer prosesin etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynıysa artık başka bir kontrol yapılmayacaktır.)

Kitabımızda bundan sonra bir prosesin bir dosyaya "w", "r" ya da "x" hakkının olması denildiğinde o prosesin o dosya için "w", "r" ya da "x" hakkına sahip olması ya da prosesin kök proses olması ya da Linux sistemlerinde prosesin bu işlemi yapacak yeteneğe sahip olması kastedilecektir.

### creat ve openat Fonksiyonları

creat isimli POSIX fonksiyonu aslında open fonksiyonunun özel bir biçimidir. Pek çok UNIX türevi sistemde bu fonksiyon da ayrı bir sistem fonksiyonu olarak bulundurulmaktadır. Fonksiyonun prototipi şöyledir:

```
#include <fcntl.h>
#include <sys/stat.h>

int creat(const char *path, mode_t mode);
```

Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesini ikinci parametresi de erişim haklarını belirtmektedir. Fonksiyonun prototipi <fcntl.h> dosyası içerisinde erişim haklarına ilişkin sembolik sabitler de <sys/stat.h> dosyası

içerisinde bulunmaktadır. creat fonksiyonu her zaman yeni bir dosya yaratma iddiasındadır. Fonksiyonun open eşdeğeri şöyledir:

```
int creat(const char *path, mode_t mode)
{
    return open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);
}
```

creat fonksiyonunun ilk boş betimleyici ile (yani en düşük değerli boş betimleyici ile) geri dönmesi POSIX standartlarında garanti altına alınmıştır.

openat fonksiyonu POSIX standartlarına sonradan eklenmiştir. openat fonksiyonu çok thread'li ortamlarda göreli yol ifadelerini thread ggvenli bir biçimde oluşturmak için düşünülmüştür. Fonksiyonun prototipi şöyledir:

```
#include <fcntl.h>

int openat(int fd, const char *path, int oflag, ...);
```

openat fonksiyonu genel davranış olarak open fonksiyonu ile aynıdır. Ancak bu fonksiyon open fonksiyonundan bir parametre daha fazladır. openat fonksiyonunun fazla olan birinci parametresi için normal olarak bir dizine ilişkin dosya betimleyicisi argüman olarak girilir. Fonksiyonun birinci parametreden sonraki parametreleri tür ve anlam olarak open fonksiyonu ile aynıdır. Fonksiyonun çalışması şöyledir:

1) Eğer fonksiyonun ikinci parameresinde belirtilen dosyanın yol ifadesi mutlak bir yol ifadesi ise fonksiyonun birinci parametresi dikkate alınmaz. Dolayısıyla fonksiyon tamamen open fonksiyonu gibi davranır.

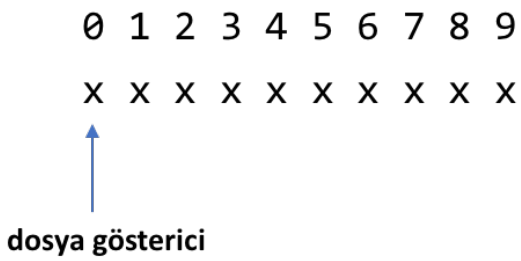
2) Eğer fonksiyonun ikinci parametresi göreli bir yol ifadesi belirtiyorsa bu durumda birinci parametresi bir dizine ilişkin dosya betimleyicisi olmak zorundadır. Bu durumda bu göreli yol ifadesi prosesin çalışma dizini orijin alınarak değil fonksiyonun birinci parametresi orijin alınarak çözülmektedir. Bu sayede farklı thread'ler sanki farklı çalışma dizinlerine sahipmiş gibi bir etki oluşturulmak istenmiştir.) Fonksiyonun ikinci argümanına göreli yol ifadesi girildiğinde fonksiyonun birinci argümanına ilişkin dizin betimleyicisinin O_SEARCH açış moduyla açılıp açılmadığında da bakılmaktadır. Eğer bu betimleyici O_SEARCH moduyla açılmışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaz. Ancak bu dizin O_SEARCH moduyla açılmamışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaktadır.

3) Eğer fonksiyonun ikinci parametresi göreli bir yol ifadesi belirtiyorsa ve birinci parametresine de AT_FDCWD özel değeri girilmişse bu durumda fonksiyon ikinci parametresiyle belirtilen göreli yol ifadesini prosesin çalışma dizinini orijin alarak çözer. Yani fonksiyonun bu durumdaki davranışı da open fonksiyonuyla aynı olur.

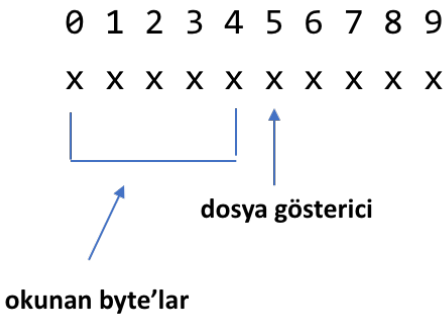
### Dosya Göstericisi (File Pointer) Kavramı

İşletim sistemlerinin büyük bölümünde okuma yazma işlemlerinii kullanıcı uygulama programcısı için kolaylaştırmak amacıyla "dosya göstericisi (file pointer)" kavramı kullanılmaktadır. Dosya göstericisi o anda okuma yazma işleminin dosyanın neresinden yapılacağını belirten bir offset değeridir. Bu anlamda dosya göstericisinin C ve C++ dillerindeki nesnelerin adreslerini tutan göstericilerle bir ilgisi yoktur. Dosya göstericisi dosya işlemleri için bir imleç (yani kalemin ucu) görevini görmektedir.

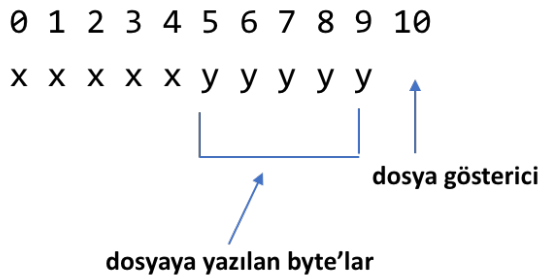
İşletim sistemlerinde dosyalar uygulama programcılarına ardışıl byte toplulukları biçiminde gösterilirler. Dosya içerisindeki her byte'ın -ilk byte 0 olmak üzere- dosyanın kaçıncı byte'ı olduğunu gösteren bir offset numarası vardır. Aşağıdaki şekilde x'ler dosyadaki byte'ları temsil ediyor olsun ve dosyada 10 byte bilgi olduğunu varsayalım:



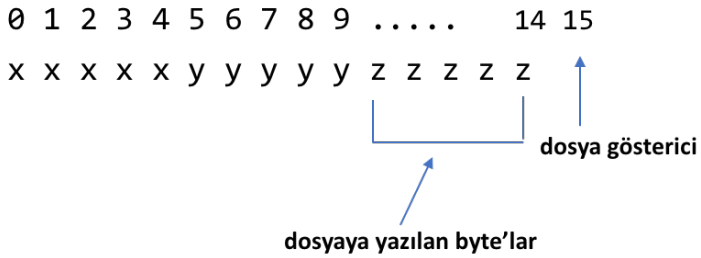

Bu şekilde dosyada bulunan 10 byte x'lerle temsil edilmiştir. Dosya göstericisi de 0 değerindedir. Yani dosya göstericisi dosyanın 0'ıncı offset'ini göstermektedir. Yukarıda da belirttiğimiz gibi işletim sistemlerinde okuma yazma işlemleri her zaman dosya göstericisinin gösterdiği yerden itibaren yapılmaktadır. İşletim sisteminin okuma ve yazma fonksiyonları okunan ya da yazılan miktar kadar dosya göstericisini otomatik olarak ilerletirler. . Örneğin biz yukarıdaki durumda bu dosyadan 5 byte okumak istesek dosyanın başındaki 5 byte'ı okumuş oluruz ve dosya göstericisi de 5 byte ilerletilir:




Dosyaya yapılan yazmalar dosya göstericisinin gösterdiği yerdeki byte'ların güncellenmesine yol açmaktadır. Yukarıdaki örnekte dosya dosyaya 5 byte yazalım. Aşağıdaki gibi bir durum elde edilecektir:




Dosya göstericisinin dosyanın sonundaki byte'tan sonraki byte'ı göstermesi durumuna EOF (End Of File) durumu denilmektedir. Dosya göstericisi EOF durumundayken dosyadan okuma yapılmak istenirse herhangi bir şey okunamaz. Ancak dosya göstericisi EOF durumundayken dosyaya yazma yapılırsa yazılanlar dosyaya eklenir. Şimdi EOF durumunda dosyaya 5 byte daha yazmak isteyelim:




Bir dosyaya ekleeme yapmak için dosya göstericisinin EOF durumunda olması gerektiğine dikkat ediniz. Dosya göstericisi EOF pozisyonundan gerideyse yazılanlar dosyaya ekleme anlamına değil mevcut byte'ların değiştirilmesi anlamına gelmektedir.

Bir dosya open ya da openat fonksiyonlarıyla açıldığında dosya gösterici her zaman 0'ıncı offsettedir. Yani yeni açılmış bir dosyada dosya göstericisi dosyanın başındadır. Eğer dosya yeni yaratılmışsa ya da içi sıfırlanarak açılmışsa bu durumda dosya açıldığında dosyada hiçbir bilgi yoktur. Böylesi bir durumda dosya göstericisinin 0'ıncı offsette olması aynı zamanda EOF durumunda olması anlamına gelecektir. Dolayısıyla içi boş bir dosyaya yazma yapıldığında aslında dosyaya ekleme yapılmış olur.

Pek çok kişi işletim sisteminin dosyanın sonunda özel bir byte (ya da karakter) tuttuğunu sanmaktadır. Halbuki işletim sistemleri dosyanın sonunda dosyanın sonunu gösteren özel bir byte bulundurmazlar. İşletim sissistemleri dosyanın sonunu ve dolayısıyla da dosya göstericisinin EOF durumunda olup olmadığını dosyanın o anki uzunluğuna bakarak anlamaktadır.

UNIX/Linux sistemlerinde tıpkı diğer işletim sistemlerinde olduğu gibi aynı dosya aynı proses ya da başka bir proses tarafından birden fazla kez açılabilmektedir. Dosyanın her farklı açılışından elde edilen dosya betimleyicisi farklı bir dosya göstericisi ile ilişkilidir. Yani dosya göstericisi bir dosya için toplamda bir tane değildir, her açıştan elde edilen dosya betimleyicisi için ayrı bir dosya göstericisi vardır. (Başka bir deyişle aynı dosyaya ilişkin bir dosya betimleyicisi ile okuma yazma yapıldığında diğer betimleyiciye ilişkin dosya göstericisi konum değiştirmez.) Bu konunun ayrıntıları ilerideki bölümlerde ele alınmakatadır.

Bir betimleyiciye ilişkin dosya göstericisinin okuma yazma işlemlerinde otomatik ilerletildiğini belirttik. Dosya göstericisi aynı zamanda lseek isimli POSIX fonksiyounuyla da konumlandırılabilmektedir. lseek fonksiyonu izleyen bölümlerde ele alınmaktadır.

### read ve write Fonksiyonları

UNIX/Linux sistemlerinde dosyadan okuma yapan ve dosyaya yazma yapan iki temel POSIX fonksiyonu vardır: read ve write. Bu POSIX fonksiyonları pek çok UNIX türevi sistemlerde doğrudan okuma ve yazma işlemlerini yapan sistem fonksiyonlarını çağırmaktadır. (Örneğin Linux sistemlerinde read fonksiyonu doğrudan sys_read sistem fonksiyonunu, write fonksiyonu ise sys_write sistem fonksiyonunu çağırır.) read fonksiyonunun prototipi şöyledir:

```
#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);
```

Fonksiyonun birinci parametresi okuma işleminin yapılacağı dosyaya ilişkin dosya betimleyicisini belirtmektedir. Fonksiyon dosya göstericisinin gösterdiği yerden üçüncü parametresiyle belirtilen miktarda byte'ı okuyarak ikinci parametre ile belirtilen adresten itibaren belleğe yerleştirir. Dosya göstericisini okunan byte sayısı kadar otomatik olarak artırır. read fonksiyonu ile dosya göstericisinin gösterdiği yerden idosya sonuna kadar olan byte miktarından daha fazla byte okunmaya çalışılırsa bu durum hata olarak değerlendirilmez. read fonksiyonu bu durumda okuyabildiği kadar byte'ı okur ve okuyabildiği byte sayısı ile geri döner. Fonksiyonun üçüncü parametresinin size_t türünden, geri dönüş değerinin de ssize_t türünden olduğuna dikkat ediniz. size_t türü POSIX standartlarına göre POSIX kütüphanesini yazanlar tarafından o sisteme uygun bir biçimde işaretsiz bir tamsayı türü olarak, ssize_t türü de işaretli bir tamsayı türü olarak <sys/types.h> dosyası içerisinde typedef edilmek zorundadır. (size_t türünün aynı zamanda C Programlama Dilinde de standart bir tür ismi olduğunu anımsayınız. ssize_t trünün başındki 's' harfi "signed" sözcüğünden kısaltılmıştır.)

read fonksiyonu başarı durumunda okuyabildiği byte sayısı ile başarısızlık durumunda da -1 değeri ile geri dönmektedir. Eğer dosya göstericisi EOF durumundaysa read fonksiyonu hiç okuma yapamayacağından dolayı 0 ile geri döner. Fonksiyonun 0 değeri ile geri dönmesinin bir hata olarak değerlendirilmediğine normal bir durum olarak karşılandığına dikkat ediniz. (Özel bir durum olarak read fonksiyonu ile 0 byte okunmak istendiğinde de fonksiyon 0 değeri ile geri dönmektedir.) Pekiyi read fonksiyonu neden başarısız olabilir ve -1 değerine geri dönebilir? İşte bazı nedenler şunlar olabilir:

- Fonksiyona geçirilen dosya betimleyicisi geçersiz olabilir. (Bu durumda errno değişkenine EBADF değeri yerleştirilmektedir.)

- Fonksiyona geçirilen dosya betimleyicisine ilişkin dosya okuma modunda açılmamış olabilir. (Örneğin dosya O_WRONLY modunda açılmışsa biz dosyadan okuma yapamayız. Bu durumda da yine errno değişkenine EBADF değeri yerleştirilmektedir.)

- read fonksiyonu yavaş bir aygıttan okuma yapıyorsa ve henüz hiçbir byte okuyamadan bir sinyal oluşmuşsa başarısız olabilir. (Bu durumda errno değişkenine EINTR değeri yüklenmektedir. Yavaş aygırlardan okuma sırasında sinyal oluşumu ile ilgili ayrıntılar "sinyal işlemlerinin" anlatıldığı bölümde ele alınmaktadır.)

- read fonksiyonu gerçek bir fiziksel aygıt sorunu ile karşılaşırsa da başarısız olabilir. (Örneğin "removable" bir aygıt kaldırılmış olabilir ya da diskte fiziksel bir bozukluk oluşmuş olabilir. (Bu durumda errno değişkenine EIO değeri yüklenmektedir.)

- read fonksiyonu ile blokesiz okumlarda (bu durumda dosyanın O_NONBLOCK bayrağı belirtilerek açılmış olması gerekir) okunacak herhangi bir bilgi oluşmamışsa fonksiyon başarısız olabilir. (Bu durumda errno değişkenine EAGAIN değeri yüklenmektedir. Bu konu "boru (pipe) haberleşmelerinin" anlatıldığı bölümde ele alınmaktadır.)

read fonksiyonu aynı zamanda en az 1 byte değer okumuşsa dosyanın son erişim zamanınıda güncellemektedir.

Aşağıda prosesin çalışma dizininde "text.txt" isimli bir dosyadan 100 byte okuyup onu yazısal biçimde ekrana bastıran bir program görüyorsunuz. Programı çalıştırmadan önce böyle bir dosyayı oluşturmayı unutmayınız. Ya da isterseniz buradaki dosya ismini değiştirebilirsiniz:

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

#define SIZE    100

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    char buf[SIZE + 1];
    ssize_t result;

    if ((fd = open("test.txt", O_RDWR)) == -1)
        exit_sys("open");

    if ((result = read(fd, buf, SIZE)) == -1)
        exit_sys("read");

    buf[result] = '\0';
    puts(buf);

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

read fonksiyonu her çağrıldığında proses çekirdek moduna geçerek sistem fonksiyonunu çalıştırdığı için bu fonksiyonun bir döngü içerisinde byte byte okuma amaçlı kullanılmasının kötü bir teknik olduğunu belirtelim. Eğer bir dosyadaki tüm byte'lar üzerinde işlem yapmak istiyorsanız dosyadaki byte'ları birer birer değil blok blok okumalısınız. Bunun için kullanılan klasik kalıp şöyledir:

```
ssize_t result;
char buf[BLOCK_SIZE];
...
while ((result = read(fd, buf, BLOCK_SIZE)) > 0) {
    ...
}

if (result == -1)
    exit_sys("read");
```

Burada döngünün read fonksiyonu 0'dan büyük bir değere geri döndüğü sürece yinelendiğine dikkat ediniz. Bu durumda bu döngüden ancak fonksiyonunun başarısız olması nedeniyle ya da dosya göstericisinin EOF'a gelmesinden dolayı dosyadan okuma yapılamaması nedeniyle çıkılabilir. Döngünün çıkışında read fonksiyonunun başarısızlıkla geri dönüp dönmediğine de bakılmıştır. Aşağıda da benzer bir örnek görüyorsunuz. Burada program komut satırı argümanıyla aldığı dosyanın içeriğini hex sistemde 16'şarlı biçimde yazdırmaktadır:

Örneğin:

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

#define BLOCK_SIZE      4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fd;
    unsigned int i, k;
    unsigned char buf[CHUNK_SIZE];
    ssize_t result;

    if (argc != 2) {
        fprintf(stderr, "%s: Wrong number of arguments! One file path argument muest be
specified\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if ((fd = open("test.txt", O_RDWR)) == -1)
        exit_sys("open");

    k = 0;
    while ((result = read(fd, buf, BLOCK_SIZE)) > 0)
        for (i = 0; i < result; ++i) {
            if (k % 16 == 0)
                printf("%08X ", k);
            printf("%02X%c", buf[i], k % 16 == 15 ? '\n' : ' ');
            ++k;
        }

    if (result == -1)
        exit_sys("read");

    if (k % 16 != 0)
```

```
        putchar('\n');

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

Şimdi aklınıza şöyle bir soru gelebilir: read fonksiyonuyla dosyadan blok blok okuma yaparken blok büyüklüğü ne kadar olmalıdır? Küçük bloklarda read fonksiyonunu çağırma sayısı artacak bu da toplam okuma zamanının artmasına yol açacaktır. Büyük bloklarda da transfer zamanı uzamakta aynı zamanda da büyük miktarda bellek tahsisatı gerekebilmektedir. Bu tür durumlarda transfer edilecek blok büyüklüğünün işletim sisteminin disk tarafında uyguladığı büyük büyüklüğü kadar alınması tavsiye edilmektedir. Bu büyüklük stat ya da fstat onksiyonundan elde edilebilir. stat ve fstat fonksiyonları ilerleyen bölümlerde ele alınmaktadır.

write fonksiyonu bellekteki bir grup byte'ı dosya göstericisinin gösterdiği yerden itibaren dosyaya aktarmak için kullanılır. write fonksiyonu aslında read fonksiyonunun ters yönde işlem yapan biçimi gibi düşünülebilir. Fonksiyonun prototipi aşağıda görülmektedir:

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Fonksiyon ikinci parametresinde belirtilen adresten başlayarak birinci parametresinde belirtilen dosyaya dosya göstericisinin gösterdiği yerden itibaren üçüncü parametresiyle belirtilen miktarda byte değeri yazar. Dosya göstericisini yazılan byte miktarı kadar otomatik olarak artırır. write fonksiyonun ikinci parametresinde belirtilen adresin const olduğuna dikkat ediniz. Fonksiyon bu adrestekileri dosyaya aktarmaktadır, bu adrese bir yazma yapmamaktadır. Fonksiyon normak olarak dosyaya yazılan byte miktarı ile geri dönmektedir. Şüphesiz en normal durum fonksiyonun üçüncü parametresiyle belirtilen miktardaki byte'ı yazarak bu değere geri dönmesidir. Ancak ancak write bazı aygıtlar söz konusu olduğunda üçüncü parametresiyle belirtilen miktarın tamamını yazamadan da geri dönebilir. (Örneğin diskinizin tıka basa dolu olduğunu düşününüz. Bu durumda write sizin talep ettiğiniz miktardaki bilgiyi dosyaya yazamayabilecektir.) write fonksiyonu da tıpkı read fonksiyonunda olduğu gibi çeşitli nedenelerden dolayı başarısız olabilir. Bu durumda -1 değerine geri döner ve errno değişkenine hata değeri yüklenmektedir. Fonksiyon üçüncü parametre için 0 argümanı girilerek çağrılırsa 0 değeri ile geri dönmektedir. Fonksiyon en çok şu nedenlerden dolayı başarısız olabilmektedir:

- write fonksiyonu yavaş bir aygıta yazma yapıyorsa ve henüz hiçbir byte yazılamadan bir sinyal oluşmuşsa başarısız olabilir. (Bu durumda errno değişkenine EINTR değeri yüklenmektedir. Yavaş aygırlara yazma sırasında sinyal oluşumu ile ilgili ayrıntılar "sinyal işlemlerinin" anlatıldığı bölümde ele alınmaktadır.)

- write fonksiyonu gerçek bir fiziksel aygıt sorunu ile karşılaşırsa da başarısız olabilir. (Örneğin "removable" bir aygıt kaldırılmış olabilir ya da diskte fiziksel bir bozukluk oluşmuş olabilir. (Bu durumda errno değişkenine EIO değeri yüklenmektedir.)

- write fonksiyonu ile blokesiz yazmalarda (bu durumda dosyanın O_NONBLOCK bayrağı belirtilerek açılmış olması gerekir) yazma işlemi baaşarısız olabilir. (Bu durumda errno değişkenine EAGAIN değeri yüklenmektedir. Bu konu "boru (pipe) haberleşmelerinin" anlatıldığı bölümde ele alınmaktadır.)

- Yukarıda da belirtildiği gibi write fonksiyonu ile dolu bir diskteki dosyaya yazma yapılmak istendiğinde write yazabildiği kadar byte'ı yazıp yazabildiği byte sayısına geri dönmektedir. Ancak disk tamamen doluysa ve write hiçbir byte'ı bu nedenle diske yazamamışsa bu durumda -1 değerine geri döner ve errno değişkenine ENOSPC değeri yüklenir.

Aşağıda write fonksiyonun kullanımına bir örnek verilmiştir:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    char buf[] = "this is a test\nYes this is a test!";
    ssize_t result;
    size_t len;

    if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1)
        exit_sys("open");

    len = strlen(buf);
    if ((result = write(fd, buf, len)) != len)
        exit_sys("write");

    printf("%lu byte(s) written\n", (unsigned long)len);

    close(fd);

    return 0;
}
```

Şimdi de bir dosyadan blok blok okunanları diğer bir dosyaya blok blok yazarak dosya kopyalaması yapalım. Gerçekten de UNIX türevi sistemlerdeki cp komutu da tipik olarak böyle çalışmaktadır. Aşağıdaki örneği inceleyiniz:

```
/* mycp.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

#define BLOCK_SIZE      4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fds, fdd;
    char buf[BLOCK_SIZE];
    ssize_t result_r, result_w;

    if (argc != 3) {
        fprintf(stderr, "wrong number of aruments!\n");
        fprintf(stderr, "usage: mycp <source path> <destination path>\n");
        exit(EXIT_FAILURE);
    }

    if ((fds = open(argv[1], O_RDONLY)) == -1)
        exit_sys("open");
```

```
    if ((fdd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1)
        exit_sys("open");

    while ((result_r = read(fds, buf, BLOCK_SIZE)) > 0)
        if ((result_w = write(fdd, buf, result_r)) != result_r) {
            if (result_w == -1)
                exit_sys("write");
            fprintf(stderr, "cannot write file!\n");
            exit(EXIT_FAILURE);
        }

    if (result_r == -1)
        exit_sys("read");

    close(fds);
    close(fdd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

Programı aşağıdaki gibi derleyip test edebilirsiniz:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ gcc -o mycp mycp.c
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ./mycp mycp.c test.c
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l mycp.c test.c
-rw-r--r-- 1 kaan study 1152 Şub 18 01:35 mycp.c
-rw-r--r-- 1 kaan study 1152 Şub 18 01:35 test.c
```

Bu örnekte kaynak dosya O_RDONLY modunda, hedef dosyanın ise O_WRONLY|O_CREAT|O_TRUNC modunda açıldığına dikkat ediniz. Bu modda hedef dosya zaten varsa sıfırlanarak açılacaktır. Daha sonra bir döngü içerisinde kaynak dosyadan BLOCK_SIZE kadar byte okunarak hedef dosyaya yazdırılmıştır. Örneğimizde BLOCK_SIZE 4096 olarak alınmıştır. Yukarıda da belirttiğimiz gibi bu tür durumlar için önerilen blok uzunluğu dosya sistemine bağlıdır ve stat ya da fstat fonksiyonuyla elde edilmektedir. Gerçekten de klasik kopyalama işlemi UNIX türevi sistemlerde tamamen yukarıdaki gibi yapılmaktadır. Linux sistemleri kopyalamayı çekirdek modunda daha etkin yapabilmek için "sendfile" ve "copy_file_range" isimli fonksiyonları bulundurmaktadır. Bu fonksiyonlar bire bir aynı isimli sistem fonksiyonlarını çağırmaktadır. sys_sendfile sistem fonksiyonu ilkin soketler için düşünülmüştü. Ancak 2.6 çekirdeklerinde fonksiyon herhangi iki dosya arasında kopyalama yapabilecek hale getirilmiştir. sys_copy_file_range sistem fonksiyonu çok sonraları 4.5 çekirdeği ile sisteme eklenmiştir.  sendfile fonksiyonunun daha genel bir biçimi olarak düşünülebilir.

### pread ve pwrite Fonksiyonları

Dosyalardan okuma yazma yapmak için read ve write fonksiyonlarının yanı sıra pread ve pwrite sismli iki POSIX fonksiyonu da bulunmaktadır. Bu fonksiyonlar UNIX türevi sistemlerde yine aslında birer bir sistem fonksiyonlarını çağırmaktadır. pread ve pwrite fonksiyonları read ve write fonksiyonlarının çok thread'li sistemler için belli bir offset üzerinde atomik işlem yapan biçimleridir. pread fonksiyonunun prototipi şöyledir:

```
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
```

Fonksiyonun ilk üç parametresi read fonksiyonu ile aynıdır. Dördüncü parametre okuma işleminin yapılacağı offset'i belirtmektedir. pread fonksiyonu okumayı dosya göstericisinin gösterdiği yerden değil son parametresiyle belirtilen offset'ten itibaren yapmaktadır. pread fonksiyonu okuma sonrasında dosya betimleyicisine ilişkin dosya göstericisinin değerini değiştirmez. Fonksiyonun diğer tüm davranışı read fonksiyonunda olduğu gibidir.

pwrite fonksiyonu da benzer semantiğe sahiptir. Prototipini inceleyiniz:

```
#include <unistd.h>

ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);
```

pwrite fonksiyonun da write fonksiyonundan tek farkı dosya göstericisinin gösterdiği yere değil dördüncü parametresiyle belirtilen offset'e yazma işlemi yapmasıdır.

Şimdi read ve write fonksiyonları varken nedne pread ve pwrite fonksiyonlarına gereksinim duyulduğunu merak edebilirsiniz. pread ve pwrite özellikle multithread uygulamalarda atomiklik sağlamak için düşünülmüştür. Bir thread lseek fonksiyonu ile dosya göstericisini belli bir offset'e yerleştirdikten sonra oradan okuma yazma yapmak istediğinde başka bir thread araya girerek okuma ve yazma işlemlerinde karışıklığa yol açabilmektedir. Halbukş bu fonksiyonlar kendi içlerinde atomik bir biçimde dosya göstericisini konumlandırarak okuma ve yazma işlemlerini yaparlar.

### lseek Fonksiyonu

Dosya açıldığında dosya göstericisinin 0'ıncı offset'te yani dosyanın başında olduğunu anımsayınız. Bu durumda okuma yazma işlemleri dosyanın başı referans alınarak yapılacaktır değil mi? Pekiyi biz dosyanın istediğimiz bir yerinden okuma yapmak istiyorsak, ya da istediğimiz bir yerine yapmak istiyorsak? İşte lseek fonksiyonu dosya göstericisini istediğimiz offset'e konumlandıran bir POSIX fonksiyonudur. Pek çok UNIX türevi sistemde bu fonksiyon da bire bir ilgili sistem fonksiyonunu çağırmaktadır. lseek fonksiyonun prototipi şöyledir:

```
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

lseek fonksiyonun kullanımı C'nin fseek fonksiyonuna çok benzemektedir. Fonksiyonun birinci parametresi konumlandırılacak dosyaya ilişkin dosya betimleyicisini almaktadır. İkinci parametre konumlandırma offset'ini üçüncü parametre ise konumlandırma orijinini belirtmektedir. Konumlandırma orijin <unistd.h> ve <stdio.h> dosyalarında bildirilmiş olan SEEK_SET, SEEK_CUR ve SEEK_END sembolik sabitlerinden biri biçiminde girilebilmektedir. (Bu sembolik sabitler tipik olarak 0, 1 ve 2 değerleri olarak define edilmiştir. Ancak C standartları ve POSIX standartları bunu garanti etmemektedir.)

Fonksiyonun ikinci parametresi üçüncü parametresindeki konumlandırma offset'ine göre anlam kazanmaktadır. Eğer üçüncü parametredeki konumlandırma orijini SEEK_SET ise bu durum konumlandırmanın dosyanın başından itibaren yapılacağı anlamına gelmektedir. Bu durumda fonksiyonun ikinci parametresi dosyanın başından itibaren offset belirtir. Dolayısıyla bu parametre için girilecek argümanın >= 0 olması gerekmektedir. Örneğin:

```
lseek(fd, 1000, SEEK_SET);
```

Burada dosya göstericisi dosyanın başından itibaren 100'üncü offset'e konumlandırılmaktadır. Eğer üçüncü parametre SEEK_CUR olarak girilirse bu durum konumlandırmanın dosya göstericisinin bulunduğu duruma göre yapılacağını belirtir. Bu durumda birinci parametredeki offset pozitif, negatif ya da 0 olarak girilebilmektedir. Pozitif ileri, negatif geri anlamına gelir. Örneğin:

```
lseek(fd, -1, SEEK_CUR);
```

Burada dosya göstericisi bulunduğu konumdan 1 geriye konumlandırılmaktadır. Fonksiyonun son parametresi SEEK_END olarak girilirse konumlandırma EOF durumuna göre yapılır. Örneğin dosyanın sonuna ekleme yapmak için dosya göstericisini EOF pozisyonuna şöyle konumlandırabilirsiniz:

```
lseek(fd, 0, SEEK_END);
```

Aşağıdaki çağrı ile dosya göstericisi dosyanın son byte'ını gösterecek biçimde konumlandırılmaktadır:

```
lseek(fd, -1, SEEK_END);
```

lseek fonksiyonu ile dosya göstericisi EOF pozisyonundan öteye de konumlandırılabilir. Bu özel durum izleyen başlıkta ele alınmaktadır.

lseek fonksiyonu başarı durumunda dosyanın başından itibaren dosya göstericisinin konumlandırıldığı offset değerine, başarısızlık durumunda -1 değerine geri dönmektedir. Normal koşullarda genellikle fonksiyonun başarısının kontrol edilmesine gereksinim duyulmaz.

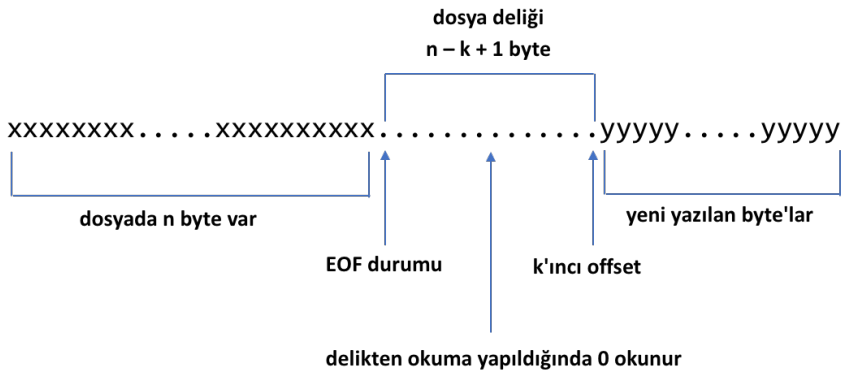
Aşağıda mevcut bir dosyanın sonuna ekleme yapan bir örnek program görüyorsunuz:

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    char buf[] = "this is a test\n";

    if ((fd = open("test.txt", O_WRONLY)) == -1)
        exit_sys("open");

    if (lseek(fd, 0, SEEK_END) == -1)
        exit_sys("lseek");

    if (write(fd, buf, strlen(buf)) == -1)
        exit_sys("write");

    close(fd);

    return 0;
}
```

## Dosya Delikleri

Önceki konuda lseek fonksiyonu ile dosya göstericisinin EOF pozisyonundan ileriye konumlandırılabileceğini belirtmiştir. Bu başlıkta dosya göstericisinin EOF ötesine konumlandırılmasının ne anlam ifade ettiği üzerinde duracağız.

Dosyada n kadar byte olduğunu düşünelim. Bu durumda EOF pozisyonunun da offset değeri n olacaktır. Şimdi biz dosya göstericisini EOF ötesinde k offset'ine konumlandırmış olalım (k > n). İşte bu durumda dosyaya yazma yapıldığında otomatik olarak dosya büyütülmüş olmaktadır. Öyle ki bu işlemden sonra dosyanın n ile k arasındaki kısmı da (yani n - k + 1 kadar byte) dosyaya dahil olur ve n ile k arasında dosyadan okuma yapıldığında 0 değeri okunur. İşte bu n ile k arasındaki bölgeye dosya deliği (file hole) denilmektedir. Dosya deliklerini aşağıdaki şekille görselleştirebiliriz:




Şimdi bir dosya deliği oluşturma örneği verelim. Elimizde örneğin 100 byte uzunluğunda "test.dat" isimli bir dosyamız olsun:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l test.dat
-rw-r--r-- 1 kaan study 100 Şub 19 17:50 test.dat
```

Dosyanın diskte kapladığı alana dikkat ediniz:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ du -B 512 test.dat
8       test.dat
```

"test.dat" dosyası diskte 8 * 512 = 4096 byte yer kaplamaktadır. (Dosyalar diskte onların uzunluğu kadar değil daha fazla yer kaplarlar. Bu konu i-node tabanlı dosya sistemlerinin ele alındığıu bölümde açıklanacaktır.)

Şimdi dosyayı açıp dosya göstericisini 1000000'uncu byte'a konumlandırıp 10 byte yazalım:

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    off_t pos;

    if ((fd = open("test.dat", O_WRONLY)) == -1)
        exit_sys("open");

    pos = lseek(fd, 1000000, SEEK_SET);
    if (write(fd, "0123456789", 10) == -1)
        exit_sys("write");

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

"test.dat" dosyasının yeni durumuna bakınız:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l test.dat
-rw-r--r-- 1 kaan study 1000010 Şub 19 20:19 test.dat
```

Şimdi de dosyanın içerisine bakalım:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ hexdump -C test.dat
00000000  09 11 28 b7 9e 70 a5 36  d2 4b 38 90 3f 90 61 ef  |..(..p.6.K8.?.a.|
00000010  ec 5a 69 ae cf 96 30 cc  b2 eb d3 ca 75 ee b8 de  |.Zi...0.....u...|
00000020  c7 31 5c 5e c2 fe f7 40  99 03 90 5d 25 42 67 04  |.1\^...@...]%Bg.|
00000030  b3 71 de e0 5d e3 9a 69  a6 56 7a d6 31 4e 8c ca  |.q..]..i.Vz.1N..|
00000040  e0 13 27 da 4e 9b 35 5c  e8 32 f5 40 a3 61 d1 0c  |..'.N.5\.2.@.a..|
00000050  a7 b1 df 1d e8 d5 8f e7  56 d8 01 cb 05 3f 7f e3  |........V....?..|
00000060  ba c3 d6 ab 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000f4240  30 31 32 33 34 35 36 37  38 39                    |0123456789|
000f424a
```

Dosya delikleri için işletim sistemi aslında mümkün olduğu kadar yer ayırmamaktadır. Yani yukarıdaki örnekte aslında işletim sistemi 1000010 (bir milyon 10) byte için diskte gerçek anlamda bir yer ayırmayacaktır. Şimdi de delik oluşturduktan sonra dosyanın diskte ne kadar yer kapladığına bakalım:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ du -B 512 test.dat
16      test.dat
```

Görüldüğü gibi dosyanın uzunluğu 1000010 (bir milyon on) byte olduğu halde diskte kapladığı alan 16 * 512 = 8192 byte uzunluktadır. Şüphesiz dosya deliklerinin bulunduğu yere yazma yapılabilir. Bu durumda yazma işlemi ile birlikte yazılan byte'lar için disk tahsisatı da yapılacaktır.

Doya delikleri konusunda son olarak yeniden bir noktaya dikkatinizi çekmek istiyoruz: Delik oluşumu dosya göstericisinin EOF ötesine konumlandırılması oluşturulmamaktadır. Konumlandırma sonrasında write işlemi yapıldığında oluşturulmaktadır. (Yani dosya göstericisini EOF ötesine konumlandırdıktan sonra yazma yapılmazsa delik oluşmayacaktır.) Tabii dosya delikleri tek hamlede pwrite fonksiyonuyla da oluşturulabilir. pwrite fonksiyonunun atomik biçimde tek hamlede dosya göstericisini konumlandırarak yazma yaptığını anımsayınız.

Dosya deliklerinin dosya sistemindeki organizasyonu i-node tabanlı dosya sistemlerinin anlatıldığı bölümde ayrıntılı biçimde ele alınmaktadır.

### Proseslerin umask Değerleri

Bir dosyanın erişim haklarının dosya yaratılırken open, openat ve create fonksiyonlarında dosyayı yaratan kişi tarafından belirlendiğini anımsayınız. Fakat itiraf edelim ki biz bu fonksiyonları anlatırken kasıtlı olarak bir noktayı ihmal ettik. Şimdi bu bölümde onu telafi edeceğiz.

Açıklamalarımıza zemin oluşturmak amacıyla, open fonksiyonu ile dosyanın sahibine, aynı gruptakilere ve diğerlerine "rw" hakkı vererek aşağıdaki gibi bir dosya yaratmak isteyelim:

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
```

```
    if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_EXCL,
                S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)) == -1)
        exit_sys("open");

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

Şimdi yaratılan dosyanın erişim haklarına bakalım:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l test.txt
-rw-r--r-- 1 kaan study 0 Şub 19 23:30 test.txt
```

Burada dikkatnizi çeken bir nokta var mı? Dosyanın erişim haklarının "-rw-rw-rw-" olması gerekiyordu değil mi? Halbuki "-rw-r--r--" biçiminde. Pekiyi ama neden? İşte open, openat ve creat gibi bazı yaratıcı fonksiyonlarda belirtilen erişim hakları aslında nihai erişim hakları değildir. Bu erişim hakları prosesin kontrol bloğunda bulunan ve ismine "umask değeri" denilen bir değerle işleme sokularak nihai erişim hakları belirlenmektedir. Yani yukarıdaki örnekte dosyanın grup ve diğer haklarında 'w' erişiminin olmaması prosesin umask değeri ile ilgilidir.

Proseslerin umask (file mode creation mask) değeri bir grup S_IXXX biçimindeki erişim haklarının "bit veya" işlemine sokulmasıyla oluşturulmaktadır. Tipik olarak herhangi bir prosesin umask değeri S_IWGRP|S_IWOTH biçimindedir. Proseslerin umask değerlerinin "bit düzeyinde değili" open, openat ve creat gibi fonksiyonlarda belirtilen erişim hakları ile "bit and" işlemine sokularak nihai erişim hakları belirlennektedir. Daha açık bir anlatımla, programcının open, openat ve creat gibi fonksiyonlarda girmiş olduğu erişim hakları "mode" isimli değişken ile, prosesin umask değeri de "umask" isimli değişken ile temsil ediliyor olsun. Bu durumda dosyaya yansıtılacak nihai erişim hakları şöyle belirlenecektir:

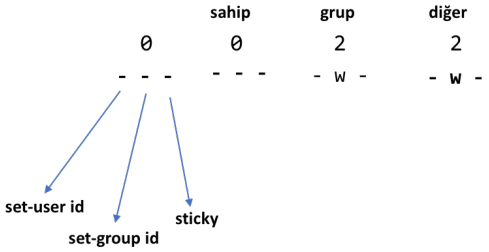
```
mode & ~umask
```

umask etkisi şöyle de ifade edilebilir: Prosesin umask değerini oluşturan erişim hakları aslında maskelenecek (yani open, openat ve create gibi fonksiyonlarda belirtilse bile dosyaya yansıtılmayacak) erişim özelliklerini belirtmektedir. Örneğin S_IWGRP|S_IWOTH umask değeri aslında "programcı dosyayı yaratırken S_IWGRP ve S_IWOTH haklarını belirtse bile bu haklar dosyaya yansıtılmayacak" anlamına gelmektedir. Ayrıca umask etkisi her türlü proses için geçerlidir. Yani örneğin prosesin etkin kullanıcı id'si 0 olsa bile (yani kök proseslerde de) umask değeri yukarıda anlatıldığı biçimde etki göstermektedir.

Pekiyi prosesin umask değeri nasıl oluşturulmaktadır? Yani bu değer nasıl belirlenmektedir? Proseslerin umask değerleri proses yaratılırken üst prosesten aktarılmaktadır. Yani alt prosesin yaratılması sırasında üst prosesin umask değeri neyse alt prosesin de umask değeri öyle olacaktır. Tabii üst prosesin de umask değeri onun üst prosesinden alınmaktadır. Biz kabuk üzerinde bir programı çalıştırırken çalıştırılan programın (yani program için yaratılacak prosesin) üst prosesi kabuk prosesidir. O halde kabuk üzerinde bir prog çalıştırılıyorsa çalıştırılan programın umask değeri kabuğun umask değeridir. Kabuğun umask değeri "umask" isimli komutla öğrenilebilir. Örneğin:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ umask
0022
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$
```

Buradaki 0022 değeri 4 octal digit belirtiyor. Bu digitlerin anlamı şöyledir:




Umask değerlerindeki en soldaki 3 bit "set-user id" , "set-group id" ve "sticky" özellikleridir. Bu özelliklerden ileride bahsedilecektir.

umask komutu ile biz kabuk prosesinin umask değerini de değiştirebiliriz. Örneğin:

```
kaan@kaan-VirtualBox:~$ umask
0001
kaan@kaan-VirtualBox:~$ umask 2
kaan@kaan-VirtualBox:~$
```

Artık kabul üzerinde bir program çalıştırdığımızda yaratılacak prosese kabuğun bu yeni umask değeri aktarılacaktır. Bu yeni umask değerinin 0002 olduğuna ve diğer kişilere 'w' haklarının kaldırıldığına dikkat ediniz. Şimdi kabuğun umask değerini 0 yapıp yukarıdaki programı yeniden çalıştıralım:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ umask 0
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ./sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l test.txt
-rw-rw-rw- 1 kaan study 0 Şub 24 15:16 test.txt
```

Prosesin umask değeri umask isimli POSIX fonksiyonuyal değiştirilebilmektedir. Linux sistemlerinde bu POSIX fonksiyonu doğrudan sys_umask isimli sistem fonksiyonunu çağırmaktadır. umask fonksiyonunun prototipi şöyledir:

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

mode_t umask(mode_t cmask);
```

Fonksiyon parametre olarak yeni umask değerini alır, geri dönüş değeri olarak eski umask değerini verir. Fonksiyon her zaman başarılı olmaktadır. Bu noktada prosesin umask değerini alan ayrı bir fonksiyonun olmadığını da söyleyelim. Eğer amacınız prosesin umask değerini almaksa bunu iki aşamada yapabilrsiniz:

```
mode_t mode;
...
mode = umask(0)
printf("%ld\n", (long)mode)
umask(mode)
```

Pekiyi kabuk prosesinin umask değerini kalıcı olarak nasıl değiştirebiliriz? Aslında bunun için en uygun yöntem

### Dosya Sistemi Üzerinde Yardımcı İşlemler Yapan POSIX Fonksiyonları

Dosyaların açılması, kapatılması, dosyalardan okuma yapılması ve dosyalara yazma yapıması temel dosya işlemlerini oluşturmaktadır. Bu teemel işlemlerin yanı sıra dosyalar üzerinde faydalı işlemler yapan birtakım yardımcı fonksiyonlar da vardır. Bu bölümde bu yardımcı fonksiyonları inceleyeceğiz.

### chmod, fchmod ve fchmodat fonksiyonları

chmod ve fchmod fonksiyonları var olan bir dosyanın erişim haklarını değiştirmek için kullanılmaktadır. Biz bir dosyayı yarattıktan sonra onun erişim haklarını istediğimiz zaman bu fonksiyonlarla değiştirebiliriz. chmod fonksiyonunun prototipi şöyledir:

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

int chmod(const char *path, mode_t mode);
```

Fonksiyonun birinci parametresi erişim hakları değiştirilecek dosyanın yol ifadesini, ikinci parametresi ise dosyaya atanmak istenen yeni erişim haklarını belirtmektedir. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri döner. Bir dosyanın erişim haklarının chmod fonksiyonuyla değiştirilebilmesi için chmod fonksiyonunu çağıran prosesin etkin kullanıcı id'sinin 0 olması (yani root prosesi olması) ya da dosyanın kullanıcı id'si ile aynı olması gerekmektedir. (Başka bir deyişle dosyanın erişim hakları ya root prosesi tarafından ya da dosyanın sahibi tarafından değiştirilebilir. Fakat Linux sistemlerinde root prosesi olmayan ve dosyanın sahibi olmayan prosesler de eğer CAP_FOWNER yeteneğine sahiplerse erişim haklarını değiştirebilmektedir.) chmod ile dosyanın erişim haklarını değiştirmek isteyen prosesin etkin kullanıcı id'si 0 değilse (yani proses root prosesi değilse, ya da Linux sistemlerinde CAP_FOWNER yeteneğine sahip değilse) ve grup id'si ya da herhangi bir ek grup id'si dosyanın grup id'si ile aynı değilse bu durumda dosyanın set-group id bayrağı (S_ISGID) reset edilmektedir. Dosyaların set-user id ve set-group id özellikleri ilerideki bölümlerde ele alınmaktadır

UNIX türevi işletim sistemleri genellikle dosyanın erişim haklarını değiştirmek için bir sistem fonksiyonu bulundurmaktadır. Örneğin Linux sistemlerinde aslında chmod fonksiyonu doğrudan sys_chmod isimli sistem fonksiyonunu çağırmaktadır. Ayrıca chmod, fchmod ya da fchmodat fonksiyonlarının prosesin umask değerini dikkate almadıklarını da belirtmek istiyoruz.

Aşağıdaki dosyaların erişim haklarını değiştiren örnek bir program görüyorsunuz:

```
/* mychmod */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>

void exit_sys(const char *msg);
int is_octal(const char *str);

int main(int argc, char *argv[])
{
    int i;
    long mode;
    mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH,
S_IXOTH};
    mode_t result_mode;

    if (argc < 3) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if (!is_octal(argv[1]) || (mode = (mode_t)strtoul(argv[1], NULL, 8)) > 0x777) {
        fprintf(stderr, "invalid octal digits!..\n");
        exit(EXIT_FAILURE);
    }

    result_mode = 0;
    for (i = 8; i >= 0; --i)
        if (mode >> i & 1)
            result_mode |= modes[8 - i];
```

```
    for (i = 2; i < argc; ++i)
        if (chmod(argv[i], result_mode) == -1)
            fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));

    return 0;
}

int is_octal(const char *str)
{
    int i;

    for (i = 0; str[i] != '\0'; ++i)
        if (str[i] < '0' || str[i] > '7')
            return 0;

    return 1;
}

void exit_sys(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
}
```

Programı aşağıdaki gibi derleyip kullanabilirsiniz:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ gcc -o mychmod mychmod.c
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ./mychmod 666 test.txt
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l test.txt
-rw-rw-rw- 1 kaan study 0 Şub 24 15:16 test.txt
```

fchmod fonksiyonu chmod fonksiyonu gibidir. Yalnızca fchmod yol ifadesini değil betimleyicisini parametre olarak alır. Prototipini inceleyiniz:

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

int fchmod(int fildes, mode_t mode);
```

Fonksiyonun birinci parametresi erişim hakları değiştirilecek dosyaya ilişkin dosya betimleyicisini, ikinci parametresi ise değiştirilmek istenen yeni erişim haklarını belirtmektedir. Eğer dosya zaten açıksa dosya betimleyicisi yoluyla erişim haklarını değiştirmenin yol ifadesi yoluyla değiştirmekten daha hızlı bir yöntem olduğunu söyleyebiliriz. Fonksiyonun diğer bütün davranışları chmod fonksiyonuyla aynıdır.

fchmodat fonksiyonu çok thread'li sistemlerde göreli yol ifadelerinin thread temelinde oluşturulmasını sağlamak için eklenmiştir. Bu anlamda fonksiyonun davranışı openat fonksiyonuna çok benzemektedir. Prototipini inceleyiniz:

```
#include <sys/stat.h>
#include <fcntl.h>

int fchmodat(int fd, const char *path, mode_t mode, int flag);
```

Fonksiyonun prototipi <sys/stat.h> dosyası içerisindedir. Birinci parametede kullanılabilen AT_FDCWD sembolik sabiti ve son parametrede kullanılabilen AT_SYMLINK_NOFOLLOW sembolik sabiti <fcntl.h> dosyası içerisinde bildirilmiştir. Fonksiyonun davranışını maddeler halinde şöyle açıklayabiliriz:

1) Eğer fonksiyonun ikinci parametresi için girilen argüman mutlak yol ifadesiyse birinci parametresi için girilen argüman dikkate alınmaz. Dolayısıyla fonksiyonun davranışı son parametresi dışında chmod fonksiyonuyla aynı olur.

2) Eğer fonksiyonun ikinci parametresindeki yol ifadesi göreli ise bu durumda bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil birinci parametresiyle belirtilen dizinden itibaren yapılmaktadır. Tabii bu durumda birinci parametrenin bir dizine ilişkin dosya betimleyicisi olması gerekir. Fonksiyonun ikinci argümanına göreli yol ifadesi girildiğinde fonksiyonun birinci argümanına ilişkin dizin betimleyicisinin O_SEARCH açış moduyla açılıp açılmadığında da bakılmaktadır. Eğer bu betimleyici O_SEARCH moduyla açılmışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaz. Ancak bu dizin O_SEARCH moduyla açılmamışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaktadır.

3) Fonksiyonun birinci parametresi için AT_FDCWD özel değeri kullanılırsa bu durumda ikinci parametresi için girilen yol ifadesi göreli olsa bile arama yine prosesin çalışma dizininden itibaren yapılmaktadır. Yani bu durumda fonksiyonun davranışı flags parametresi dışında chmod gibidir.

4) Eğer fonksiyonun son parametresi 0 yerine AT_SYMLINK_NOFOLLOW biçiminde girilmişse bu durumda söz konusu hedef dosya sembolik bağlantı dosyasıysa bağlantı dosyasının gösterdiği dosya için değil bağlantı dosyasının kendisi için işlem yapılmaktadır.

fchmodat fonksiyonu da başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri döner.

chmod, fchmod ve fchmodat fononları başarı durumunda dosyanın son durum değişikliği zamanını güncellemektedir.

Dosyanın erişim haklarını değiştirme işlemi kabuk üzerinde chmod POSIX komutuyla da yapılabilmektedir. (Tabii aslında chmod komutu bu işlemi chmod fonksiyonunu çağırarak gerçekleştirir.) Burada kısaca chmod komutundan da biraz bahsetmek istiyoruz. chmod komutunun pek çok kullanım biçimi vardır. Fakat burada biz yalnızca temel kullanımını açıklamak istiyoruz:

- Komut birden fazla dosya üzerinde işlem yapabilmektedir.

- Komutta ikinci komut satırı argümanı olarak "ugoa" harflerinden bir ya da birden fazlası belirtildikten sonra '+', '-' ya da '=' karakterleri, daha sonra da "rwx" karakterleri bunları izler. "u (user/owner)", "g(group)", "o (oher)", "a (all)" anlamına gelmektedir. Örnekleri izleyiniz:

```
chmod u+rw test.txt
```

Burada dosyanın sahipliğine "read" ve "write" hakkı eklenmiştir.

- a (all) dosyanın sahiplik, grup ve diğer haklarına ekleme ya da çıkartma yapmak için kullanılır. Örneğin:

```
chmod a-w test.txt
```

Burada dosyanın sahiplik, grup ve diğer haklarından "write" çıkartılmıştır.

- Komutta eğer "ugoa"dan hiçbiri belirtilmezse varsayılan durumda "a" (yani all) belirtilmiş gibi işlem yapılmaktadır. Örneğin:

```
chmod +rw test.txt
```

Burada dosyanın sahiplik, grup ve diğer haklarına "read" ve "write" eklenmiştir.

- = karakteri ilgili sınıfı tam olarak belirtilen özellikle set eder. Örneğin:

```
chmod g=rx test.txt
```

Burada grup hakkı "r-x" yapılmak istenmiştir.

- Erişim haklarında ',' karakteri devam anlamına gelmektedir. Örneğin:

```
chmod u+rw,g+r,o+r test.txt
```

Burada dosyanın sahibine, grubuna ve diğerlerine "read" hakkı verilişitir.

- Eğer ikinci komut satairı argümanı yukarı belirtilen kalıpta değil de sayısal biçimdeyse bu durumda orada belirtilen octal değere ilişkin olarak haklar belirlenmektedir. Örneğin:

```
chmod 644 test.txt
```

Burada 644 "rw-r—r—" anlamına gelmektedir. Komutla ilgili ayrıntılar için çeşitli kaynaklara başvurabilirsiniz.

### chown, fchown, lchown ve fchownat fonksiyonları

Önceki konularda bir dosyanın kullanıcı id'sinin onu yaratan prosesin etkin kullanıcı id'si olarak, grup id'sinin de onu yaratan prosesin etkin grup id'si ya da dosyanın içinde bulunduğu dizinin grup id'si olarak alındığını belirtmiştik. İşte yaratılmış olan bir dosya ya da dizinin kullanıcı ve grup id'leri daha sonra chown ya da fchown fonksiyonlarıyla değiştirilebilmektedir. chown fonksiyonunun prototipi şöyledir:

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
```

Fonksiyonun birinci parametresi kullanıcı ve grup id'si değiştirilecek dosyanın yol ifadesini alır. İkinci parametresi dosyanın değiştirilecek olan yeni kullanıcı id'sini üçüncü parametresi de değiştirilecek olan yeni grup id'sini belirtir. Fonksiyon başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri dönmektedir.

UNIX türevi sistemlerde ilk başlarda bir prosesin kendi dosyasının ya da dizininin kullanıcı id'sini değiştirmesi normal karşılanıyordu. Ancak sonraları bunun kötüye kullanılabileceği ortaya çıktı. Bu nedenle POSIX standartları bir dosyanın ya da dizinin kullanıcı id'sinin dosyanın sahibi tarafından değiştirilebilmesini işletim sisteminin isteğine bıraktılar. Sıradan proseslerin dosyanın ya da dizinin kullanıcı id'sini değiştirememesi durumuna İngilizce "change own restricted" denilmektedir. UNIX türevi bir sistemde sıradan bir prosesin kendi dosya ve dizininin kullanıcı id'sini değiştirip değiştiremeyeceği derleme zamanında ve çalışma zamanında bazı kontroller sayesinde anlaşılabilmektedir. Bu kontrolleri şöyle açıklayabiliriz:

1) Eğer <unistd.h> dosyası içerisinde _POSIX_CHOWN_RESTRICTED sembolik sabiti tanımlanmamışsa ya da -1 olarak tanımlanmışsa işletim sisteminde "change own restricted" ile ilgili herhangi bir özellik yoktur. Bu durum POSIX standartlarının kısmen desteklendiği sistemler için düşünülmüştür. Bu sistemler bazı özelliklere sahip olmadıkları için bu konu ile ilgili işlem yapan fonksiyonlara da sahip değildir.

2) Eğer <unistd.h> dosyası içerisinde _POSIX_CHOWN_RESTRICTED sembolik sabiti 0 olarak tanımlanmışsa o sistemde "change own restricted" özelliği olabilir ya da olmayabilir. Bunun olup olmadığını anlamak için pathconf ya da fpathconf fonksiyonunu çağırıp geri dönüş değerine bakmak gerekir.

3) Eğer <unistd.h> dosyası içerisinde _POSIX_CHOWN_RESTRICTED sembolik sabiti 0'dan büyük bir biçimde bildirilmişse bu durumda o sistemde "change own restricted" özelliği vardır.

Pekiyi bu yazılanlardan ne sonucu çıkarmalıyız? Onları da özetleyelim:

1) Çalıştığınız sistemde "change own restricted" özelliğinin olabileceğini varsayarak kodunuzu yazabilirsiniz. Yani kodunuzda normal bir proses iseniz kendi dosyalarınızın ve dizinlerinizin kullanıcı id'sini değiştirmeye çalışmamalısınız.

2) Çalıştığınız sistemde "change own restricted" özelliğinin olup olmadığını anlayabilmek için <unistd.h> dosyasını include ederek _POSIX_CHOWN_RESTRICTED sembolik sabitinin değerine bakabilirsiniz. Eğer bu değer 0 ise bu durumda asıl sonucu pathconf ya da fpathconf fonksiyonlarıyla elde etmelisiniz. (Bu fonksiyonlar kitabımızda başka bir bölümde ele alınmaktadır.) Eğer bu sembolik sabit 0'dan büyükse bu durumda sisteminizde kesinlikle "change own restricted" özelliği vardır.

Sisteminizde "change own restricted" özelliğinin dosya sistemine ve onun mount edilme biçimine göre değişebileceğini söyleyelim. Bu nedenle eğer _POSIX_CHOWN_RESTRICTED sembolik sabiti 0 ise pathconf ya da fpathconf fonksyonu ile bu kontrolün dosya temelinde yapılması gerekmektedir. Aşağıdaki örnek böyle bir kontrolün nasıl yapılacağı konusunda size bir fikir verebilir:

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

void exit_sys(const char *msg);

int main(void)
{
    long result;

#ifndef _POSIX_CHOWN_RESTRICTED
    #error Change on restricted not supported
#endif

#if _POSIX_CHOWN_RESTRICTED == 0

    if ((result = pathconf(".", _PC_CHOWN_RESTRICTED)) == -1)
        exit_sys("pathconf");

    printf(result ? "Change own restrict available\n" : "Change own restricted not available\n");
#else
    printf("Change own restricted available\n");
#endif

    return 0;
}

void exit_sys(const char *str)
{
    perror(str);

    exit(EXIT_FAILURE);
}
```

Linux sistemlerinde ve Mac OS X sistemlerinde default durumda "change own restricted" özelliğinin bulunduğunu ayrıca belirtmek istiyoruz.

Pekiyi sistemimizde "change own restricted" özelliği varsa chown fonksiyonu nasıl davranacaktır? Bunu da maddeler halinde açıklamak istiyoruz:

1) Eğer chown fonksiyonunu çağıran prosesin etkin kullanıcı id'si 0 ise fonksiyon dosya ya da dizinin kullanıcı ve grup id'sini istebir engellemeyle karşılaşmadan değiştirecektir. Buradan kök proseslerin her dosyanın kullanıcı ve grup id'sini değiştirebileceği sonucunu çıkartabilirsiniz.

2) Eğer chown fonksiyonunu çağıran prosesin etkin kullanıcı id'si dosyanın ya da dizinin kullanıcı id'si ile aynı ise bu durumda programcı dosya ya da dizinin kullanıcı id'sini değiştiremez ancak grup id'sini kendi prosesinin gerçek grup id'si olarak ya da ek grup id'lerinden biri olarak değiştirebilmeketdir. Ancak bu durumda chown fonksiyonunun ikinci parametresi için -1 özel değeri girilmelidir.

fchown fonksiyonu chown fonksiyonunun dosya betimleycisini parametre olarak alan bir biçimidir. fchown fonksiyonunun prototipi şöyledir:

```
#include <unistd.h>

int fchown(int fd, uid_t owner, gid_t group);
```

Fonksiyonun birinci parametresinin yol ifadesini değil dosya betimleyicisini aldığına dikkat ediniz. Genel olarak daha önce de bahsettiğimiz gibi eğer dosya açıksa doğrudan betimleyici ile işlem yapan fonksiyonlar daha hızlı çalışma eğilimindedir. Birinci parametrenin dışında fonksyonun genel davranışı tamamen chown fonksiyonunda olduğu gibidir.

lchown fonksiyonu ise chown fonksiyonu ise chown fonksiyonunun sembolik bağlantıları (symbolic links) izlemeyen biçimidir. Yani lchown fonksiyonuna sembolik bağlantı dosyası veridliğinde bu fonksiyon sembolik bağlantı dosyasının gösterdiği dosya üzerinde değil sembolik bağlantı dosyasının kendisi üzerinde işlem yapmaktadır. Bunun dışında lchown fonksiyonu ile chown fonksiyonu arasında başkaca bir farklılık yoktur. Aşağıda lchown fonksiyonunun prototipini görüyorsunuz:

```
#include <unistd.h>
int lchown(const char *path, uid_t owner, gid_t group);
```

fchown fonksiyonu ise chown fonksiyonunun dosya betimleyicisi ile işlem yapan biçimidir:

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
```

Fonksiyonun diğer tüm davranışları chown fonksiyonu ile aynıdır. fchownat ise fonksiyonun çok thread'li uygulamalar için düşünülmüş thread'e göreli yol ifadesi oluşturmak için kullanılan biçimidir. Prototipini inceleyiniz:

```
#include <sys/stat.h>
#include <fcntl.h>

int fchownat(int fd, const char *path, uid_t owner, gid_t group, int flag);
```

Fonksiyonun prototipi <sys/stat.h> dosyası içerisindedir. Birinci parametrede kullanılabilen AT_FDCWD sembolik sabiti ve son parametrede kullanılabilen AT_SYMLINK_NOFOLLOW sembolik sabiti <fcntl.h> dosyası içerisindedir. fchownat fonksiyonu genel çalışma biçimi olarak diğer "at"li fonksiyonlar ile benzerdir. Yine 4 durum söz konusudur:

1) Eğer fonksiyonun ikinci parametresi için girilen argüman mutlak yol ifadesiyse birinci parametresi için girilen argüman dikkate alınmaz. Dolayısıyla fonksiyonun davranışı son parametresi dışında chwown fonksiyonuyla aynı olur.

2) Eğer fonksiyonun ikinci parametresindeki yol ifadesi göreli ise bu durumda bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil birinci parametresiyle belirtilen dizinden itibaren yol belirtir. Tabii bu durumda birinci parametrenin bir dizine ilişkin dosya betimleyicisi olması gerekmektedir. Fonksiyonun ikinci argümanına göreli yol ifadesi girildiğinde fonksiyonun birinci argümanına ilişkin dizin betimleyicisinin O_SEARCH açış moduyla açılıp açılmadığında da bakılmaktadır. Eğer bu betimleyici O_SEARCH moduyla açılmışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaz. Ancak bu dizin O_SEARCH moduyla açılmamışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaktadır.

3) Fonksiyonun birinci parametresi için AT_FDCWD özel değeri kullanılırsa bu durumda ikinci parametresi için girilen yol ifadesi göreli olsa bile arama yine prosesin çalışma dizininden itibaren yapılmaktadır. Yani bu durumda fonksiyonun davranışı flags parametresi dışında chown gibidir.

4) Eğer fonksiyonun son parametresi 0 yerine AT_SYMLINK_NOFOLLOW biçiminde girilmişse bu durumda söz konusu hedef dosya sembolik bağlantı dosyasıysa bağlantı dosyasının gösterdiği dosya için değil bağlantı dosyasının kendisi için işlem yapılmaktadır.

chown, fchown ve fchownat fonksiyonları başarı durumunda dosyanın son durum değişiklik zamanını güncellemektedir.

### Dosyalarda Katı Bağlar (Hard Links)

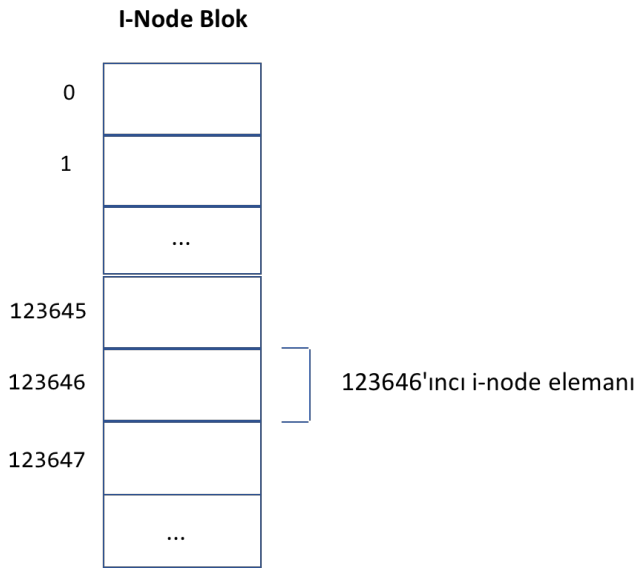
UNIX türevi sistemlerde kullanılan i-node tabanlı dosya sistemlerinde "katı bağ (hard link)" isimli bir özellik bulunmaktadır. Katı bağ aynı dosyaya referans eden farklı dizin girişlerinin oluşturulmasını sağlayan bir organizasyondur. Bu bölümde dosya sistemindeki katı bağ organizasyonu üzerinde duracağız.

I-Node tabanlı dosya sistemlerinde her dosyanın bir katı bağ sayaç değeri vardır. Dosyaların katı bağ sayaç değerlerini ls -l komutu ile görüntüleyebilirsiniz. Örneğin:

```
-rw-rw-r-- 1 kaan study  614 Şub 25 16:27 sample.c
-rw-r--r-- 1 kaan study 1152 Şub 18 01:47 test.c
drwxr-xr-x 2 kaan study 4096 Şub 26 16:49 testdir
```

Bu örnekte "sample.c" ve "test.c" dosyalarının katı bağ sayaç değerlerinin 1, "testdir" isimli dizinin ise 2 olduğunu görüyorsunuz.

I-Node tabanlı dosya sistemlerinde dosyaların (dizinlerin de birer dosya gibi ele alındığını unutmayınız) metadata bilgileri diskte i-node elemanı denilen bir veri yapısında tutulmaktadır. Bir dosyanın ls -l ile elde ettiğiniz bütün bilgileri (dosyanın kullanıcı ve grup id'si, erişim hakları, uzunluğu gibi) aslında dosyaya ilişkin i-node elemanından alınmaktadır. Dosyanın içerisindeki bilgiler diskte çeşitli bloklara yayılmış olarak bulunmaktadır. Dosyanın parçalarının hangi bloklarda bulunduğu bilgisi de i-node elemanın içerisinde saklanmaktadır. Pekiyi dosyalara ilişkin i-node elemanları diskte neredediir? İşte i-node tabanlı dosya sistemlerinde diskteki bütün dosyaların i-node elemanları disk üzerinde ismine "i-node blok" denilen bir bölümde bir dizi biçiminde tutulmaktadır. Her i-node elemanının bu i-node blok denilen dizide bir index numarası vardır. Aşağıdaki şekli inceleyiniz:



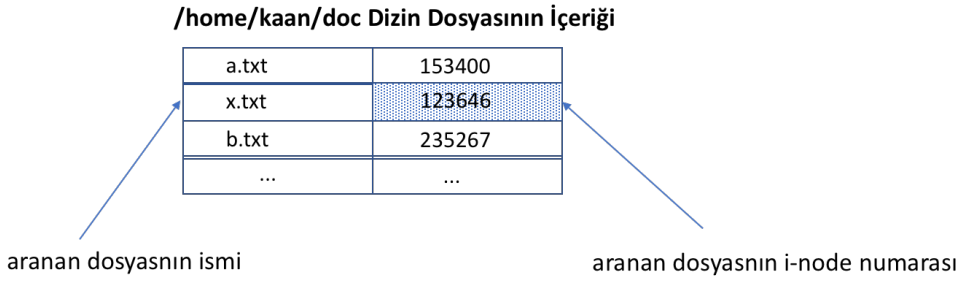

İşletim sistemi dosyanın içerisindeki verilere erişebilmek için önce dosyanın i-node elemanına erişmek zorundadır. Bildiğiniz gibi dosyalar kavramsal olarak dizinlerin içerisindedir. Dizinlerin de birer dosya gibi ele alındığını önceki bölümlerde belirtmiştik. Fakat aslında dizin dosyalarının içerisinde dosya bilgileri değil yalnızca o dizindeki dosyaların isimleri ve i-node numaraları bulunmaktadır. Dizin dosyalarının aşağıdaki gibi kayıtlardan oluştuğunu varsayabilirsiniz:

**Dizin Dosyasının İçeriği**

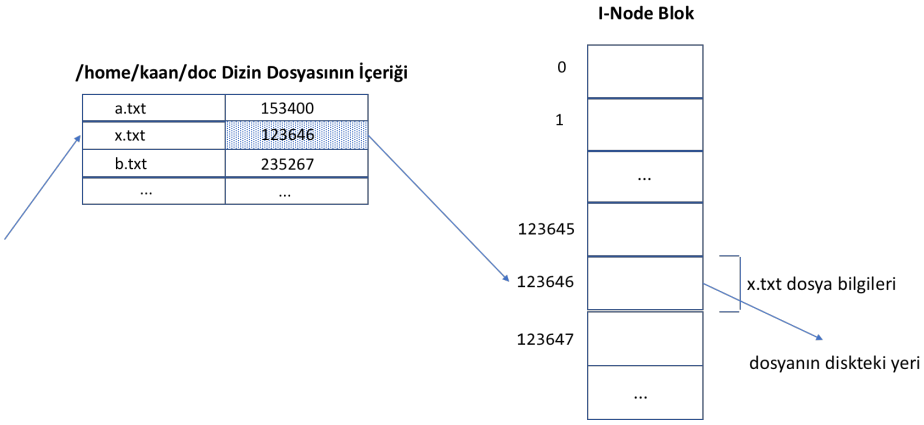
| dosya ismi | i-node numarası |
|---|---|
| dosya ismi | i-node numarası |
| dosya ismi | i-node numarası |
| ... | ... |
| dosya ismi | i-node numarası |
| dosya ismi | i-node numarası |
| dosya ismi | i-node numarası |
| ... | ... |

Dosyaların gerçek metadata bilgileri o dosyalara ilişkin i-node elemanının içerisindedir.

Bir yol ifadesi verildiğinde işletim sistemi dizin dosyasında dizin girişleri içerisinde yol ifadesindeki hedef dosyanın ismini bularak onun i-node numarasını elde eder. Sonra bu i-node numarasını disk üzerindeki i-node bloğa index yaparak dosyaya ilişkin i-node elemanına ulaşır. İşletim sisteminin yol ifadesinden hareketle dosyanın i-node elemanına nasıl eriştiğini bri örnekle açıklayalım. Örneğin siz bir sistem fonksiyonuna "/home/kaan/doc/x.txt" biçiminde bir yol ifadesi vermiş olun. Bu durumda işletim sistemi "doc" dizin dosyasının içerisinde "x.txt" ismini arayacak ve eğer bulursa onun i-node numarasını elde edecektir:

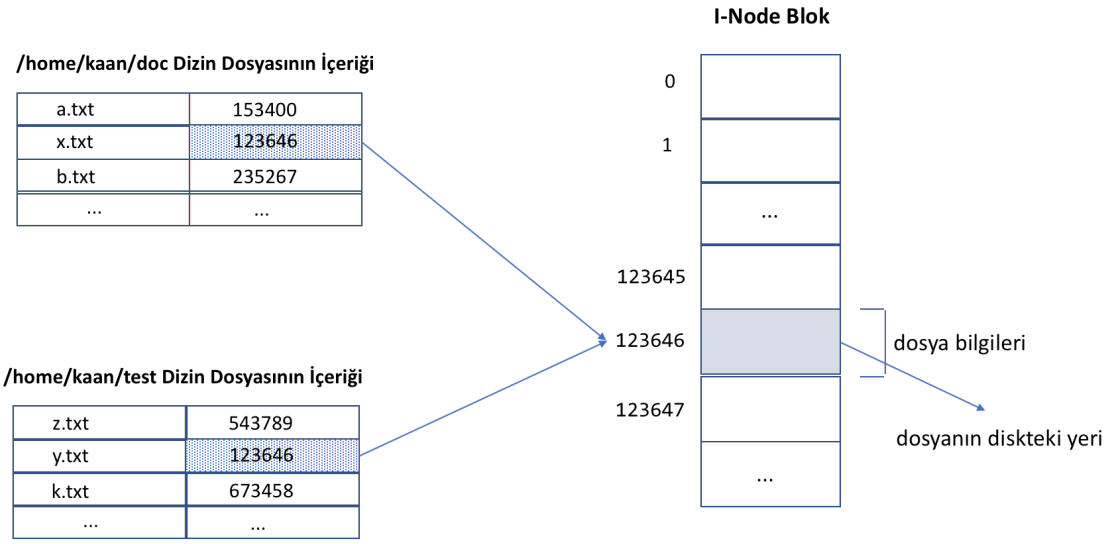



İşletim sistemi dosyanın i-node numarasını elde ettikten sonra o dosyaya ilişkin i-node elemanına erişir. Dosyanın içindeki bilgilerin disk üzerinde nerede olduğu da i-node elemanının içerisindedir.




Yukarıda özetlediğimiz organizasyonun bazı ayrıntıları vardır. Biz burada bu ayrıntıları göz ardı ettik. Bu ayrıntılar i-node tabanlı dosya sistemlerinin anlatıldığı bölümde ele alınmaktadır.

İşte farklı dizin girişlerinin aynı i-node elemanına referans etmesi durumuna "katı link (hard link)" denilmektedir. Bunu anlayabilmek için iki farklı dizin girişinin aynı i-node elemanına referans ettiği bir durumu düşününüz. Örneğin "/home/kaan/doc" dizinindeki "x.txt" dizin girişi ile "home/kaan/test" isimli dizindeki "y.txt" dizin girişinin 123646 biçiminde aynı i-node elemanına referans ettiğini varsayalım:




Bu durumda bizim open fonksiyonunda "/home/kaan/doc/x.txt" yol ifadesini kullanmamızla "home/kaan/test/y.txt" yol ifadesini kullanmamız arasında hiç fark yoktur. open fonksiyonu her iki yol ifadesinde de aslında dizin girişlerinde aynı i-node numarasını bulacak ve dolayısıyla aynı dosyaya erişecektir. Dosyaların bütün önemli metadata bilgilerinin i-node elemanı içerisinde olduğunu anımsayınız. Bu durumda yukarıdaki örnekte iki katı bağdaki dosyaların tüm özellikleri aynı olacaktır. Şimdi yukarıdaki örneğe ilişkin durumu komut satırında oluşturalım. Katı bağ oluşturmak için "ln" isimli POSIX komutu kullanılmaktadır:

```
kaan@kaan-VirtualBox:~$ ls -l doc
toplam 12
-rw-r--r-- 1 kaan study   6 Şub 27 17:53 a.txt
-rw-r--r-- 1 kaan study 571 Şub 27 17:54 b.txt
-rw-r--r-- 1 kaan study 515 Şub 27 17:54 x.txt
kaan@kaan-VirtualBox:~$ ln doc/x.txt test/y.txt
kaan@kaan-VirtualBox:~$ ls -li doc
toplam 12
4986511 -rw-r--r-- 1 kaan study   6 Şub 27 17:53 a.txt
4986528 -rw-r--r-- 1 kaan study 571 Şub 27 17:54 b.txt
4986527 -rw-r--r-- 2 kaan study 515 Şub 27 17:54 x.txt
kaan@kaan-VirtualBox:~$ ls -li test
toplam 12
4986531 -rw-r--r-- 1 kaan study 110 Şub 27 17:56 k.txt
4986527 -rw-r--r-- 2 kaan study 515 Şub 27 17:54 y.txt
4986530 -rw-r--r-- 1 kaan study  82 Şub 27 17:55 z.txt
kaan@kaan-VirtualBox:~$
```

Dosyaların i-node numaralarını elde etmek için ls komutuna -i seçeneğinin de eklenmesi gerekmektedir.

Dizinlerin katı bağ sayaçlarının 1'den büyük olduğu dikkatinizi çekmiş olabilir. Örneğin:

```
kaan@kaan-VirtualBox:~$  mkdir testdir
kaan@kaan-VirtualBox:~$ ls -ld testdir
drwxr-xr-x 2 kaan study 4096 Şub 27 18:06 testdir
kaan@kaan-VirtualBox:~$
```

Bu örnekte dizin'in katı bağ sayacının 2 olduğunu görüyorsunuz. Pekiyi neden yeni yaratılmış bir dizin'in katı bağ sayacağı 2 olmaktadır? İşte UNIX/Linux sistemlerinde (Windows sistemlerinde de böyle) bir dizin yaratıldığında dizin içerisinde "." ve ".." isimli iki dizin girişi otomatik olarak yaratılmaktadır. "." girişi o anda bulunulan dizini, ".." girişi de o anda bulunulan dizinin üst dizinini belirtmektedir. (UNIX/Linux sistemlerinde "." ile başlayan dizin girişlerinin default durumda görüntülenmediğini, bu girişlerin görüntülenmesi için "-a" seçeneğinin kullanılması gerektiğini anımsayınız.) Örneğin:

```
kaan@kaan-VirtualBox:~$ ls -la testdir
toplam 8
drwxr-xr-x  2 kaan study 4096 Şub 27 18:06 .
drwxr-xr-x 25 kaan kaan  4096 Şub 27 18:06 ..
```

O halde aslında yeni yaratılan bir dizin dosyasını gösteren iki dizin girişi bulunmaktadır: Dizinin yaraıldığı dizindeki giriş ve yeni yaratılan dizinin içerisindeki "." isimli giriş:

```
kaan@kaan-VirtualBox:~$ ls -ldi testdir
4986532 drwxr-xr-x 2 kaan study 4096 Şub 27 18:06 testdir
kaan@kaan-VirtualBox:~$ ls -lai testdir
toplam 8
4986532 drwxr-xr-x  2 kaan study 4096 Şub 27 18:06 .
4980738 drwxr-xr-x 25 kaan kaan  4096 Şub 27 18:06 ..
```

Bir dizin içerisinde dizin yarattıkça o dizinin katı bağ sayacı artar. Çünkü dizin içerisinde yaratılan dizinin içerisindeki ".." dizini de üst dizinin i-node elemanını gösteren bir katı bağdır. ƒ

### link ve linkat Fonksiyonları

Bir dosyanın katı bağını oluşturabilmek için link ve linkat isimli POSIX fonksiyonları kullanılmaktadır. link fonksiyonun prototipi şöyledir:

```
#include <unistd.h>

int link(const char *path1, const char *path2);
```

Fonksiyonun birinci parametresi katı bağı oluşturulacak dosyaya ilişkin bir dizin girişini, ikinci parametresi de oluşturulacak katı bağ girişini belirtmektedir. link fonksiyonunun ikinci parametresinde belirtilen yol ifadesinin birinci parametresiyle belirtilen yol ifadesi ile aynı dizine ilişkin olması zorunlu değildir. Biz bir dosyanın katı bağını yazma hakkında sahip olduğumuz herhangi bir dizinde oluşturabiliriz. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri döner. Fonksiyonun kullanımına şöyle bir örnek vermek istiyoruz:

```
/* mylink.c */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if (link(argv[1], argv[2]) == -1)
        exit_sys("link");

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);

    exit(EXIT_FAILURE);
```

```
}
```

Yukarıdaki "mylink" isimli program birinci komut satırı argümanı ile belirtilen dosyanın ikinci komut satırı argümanı ile belirtilen katı bağını oluşturmaktadır.

Dizinlerin katı bağlarının oluşturulması UNIX türevi sistemlerde şüpheyle karşılanmaktadır. Çünkü dizinlerin katı bağları dizin ağacını dolaşan fonksiyonlarda sorunlar oluşturabilmektedir. Bazı UNIX türevi sistemler bu nedenle dizinler için katı bağ oluşturulmasına hiç izin vermezler. Bazıları ise yalnızca kök proseslere bu izni vermektedir. POSIX standartlarında link fonksiyonu proseslerin dizinler için katı bağ oluşturup oluşturamayacağı işletim sistemini yazanların isteğine bırakılmıştır. Örneğin Linux sistemlerinde link fonksiyonu ile normal prosesler de kök prosesler de dizinler için katı bağlar oluşturamamaktadır.

linkat fonksiyonu link fonksiyonun at'li biçimidir. Prototipini inceleyiniz:

```
#include <unistd.h>
#include <fcntl.h>

int linkat(int fd1, const char *path1, int fd2, const char *path2, int flag);
```

Fonksiyonun prototipi <unistd.h> dosyası içerisindedir. Birinci parametrede kullanılabilen AT_FDCWD sembolik sabiti ve son parametrede kullanılabilen AT_SYMLINK_NOFOLLOW sembolik sabiti <fcntl.h> içerisinde bildirilmiştir.

linkat fonksiyonu genel çalışma biçimi olarak diğer "at"li fonksiyonlar ile benzerdir. Yine 4 durum söz konusudur:

1) Eğer fonksiyonun ikinci parametresi için girilen argüman mutlak yol ifadesiyse birinci parametresi için girilen argüman dikkate alınmaz. Benzer biçimde eğer fonksiyonun dördüncü parametresi için girilen argüman mutlak yol ifadesiyse üçüncü parametresi için girilen argüman dikkate alınmaz. Dolayısıyla bu durumda fonksiyonun davranışı son parametresi dışında link fonksiyonuyla aynı olur.

2) Eğer fonksiyonun ikinci parametresindeki yol ifadesi göreli ise bu durumda bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil birinci parametresiyle belirtilen dizinden itibaren yol belirtir. Tabii bu durumda birinci parametrenin bir dizine ilişkin dosya betimleyicisi olması gerekmektedir. Benzer biçimde fonksiyonun dördüncü parametresindeki yol ifadesi göreli ise bu durumda da bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil üçüncü parametresiyle belirtilen dizinden itibaren yol belirtir. Fonksiyonun ikinci ya da dördüncü argümanına göreli yol ifadesi girildiğinde fonksiyonun birinci ve üçüncü argümanına ilişkin dizin betimleyicisinin O_SEARCH açış moduyla açılıp açılmadığında da bakılmaktadır. Eğer bu betimleyici O_SEARCH moduyla açılmışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaz. Ancak bu dizin O_SEARCH moduyla açılmamışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaktadır.

3) Fonksiyonun birinci ve üçüncü parametreleri için AT_FDCWD özel değeri kullanılırsa bu durumda ikinci ve dördüncü parametreler için girilen yol ifadeleri göreli olsa bile arama yine prosesin çalışma dizininden itibaren yapılmaktadır. Yani bu durumda fonksiyonun davranışı flags parametresi dışında link fonksiyonunda olduğu gibidir.

4) Eğer fonksiyonun son parametresi 0 yerine AT_SYMLINK_NOFOLLOW biçiminde girilmişse bu durumda ikinci parametre için girilen argüman sembolik bağlantı dosyasıysa bağlantı dosyasının gösterdiği dosya için değil bağlantı dosyasının kendisi için katı link oluşturulur.

link ve linkat fonksiyonları başarı durumunda söz konusu dosyanın son durum değişikliği zamanını günceller. Aynı zamanda yeni girişin yaratıldığı dizinin ve son güncelleme zamanı ve son durum değişikliği zamanı da güncellenmektedir.

Bir dosyanın katı bağı kabuk üzerinde ln isimli POSIX komutuyla oluşturulabilmektedir. Bu komutu önceki konuda kullanmıştık. Şüphesiz bu komut aslında link isimli POSIX fonksiyonu kullanılarak yazılmıştr. Pek çok sistemde link fonksiyonu doğrudan bu işlemi yapan bir sistem fonksiyonunu çağırır. Örneğin Linux sistemlerinde link fonksiyonu doğrudan sys_link isimli sistem fonksiyonunu çağırmaktadır.

### unlink, remove ve unlinkat Fonksiyonları

Bir dosyayı silmek için remove, unlink ve unlinkat isimli üç fonksiyon kullanılmaktadır. unlink ve remove fonksiyonlarının işlevi tamamen aynıdır. Ancak remove bir standart C fonksiyonudur. Dolayısıyla tüm sistemlerde bulunmaktadır. unlink ise bir POSIX fonksiyonudur. Dolayısıyla yalnızca UNIX türevi sistemlerde bulunmaktadır. unlink fonksiyonunun prototipi şöyledir:

```
#include <unistd.h>

int unlink(const char *path);
```

Fonksiyon silinecek dosyanın yol ifadesini parametre olarak alır. Başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. remove fonksiyonun prototipi de şöyledir:

```
#include <stdio.h>

int remove(const char *path);
```

remove fonksiyonun davranışı tamamen unlink fonksiyonuyla aynıdır.

unlink ve remove fonksiyonları ilgili dosya için dizin girişini silerler. Ancak dosyanın gerçekten silinip silinmeyeceği o dosyanın katı bağ sayacına bağlıdır. Dosyalar ancak o dosyalara ilişkin i-node elemanlarını gösteren hiçbir dizin girişi kalmayınca gerçekten silinmektedir.

Dosyaların katı bağlarını ele aldığımız bölümde her dosya için diskte bir i-node elemanının tutulduğunu belirtmiştik. Anımsarsanız her dosyanın bir katı bağ sayacı da vardı. İşte dosyaların katı bağ sayaçları diskte o dosyaya ilişkin i-node elemanının içerisinde tutulmaktadır. unlink ya da remove fonksiyonuyla bir dosya silinmek istendiğinde önce onun i-node elemanındaki katı bağ sayacı 1 ekseiltilir. Ancak dosyanın katı bağ sayacı 0'a düşerse (yani artık o dosyayı hiçbir dizin girişi göstermiyorsa) dosya gerçekten silinmektedir. (Fonksiyonun isminin neden "unlink" olduğunu da anlamışsınızdır. Çünkü bu fonksiyon bir anlamda "link" fonksiyonunun tersini yapmaktadır.) Ayrıca unlink ve remove fonksiyonlarındaki yol ifadeleleri sembolik bağlantı dosyalarına ilişkin olursa bu durumda sembolik bağlantı dosyalarının kendileri unlink işlemine sokulmaktadır, onların referans ettiği dosyalar değil.

Bir dosyayı silebilmek için prosesin dosyanın kendisine "w" hakkına sahip olması gerekmez. Dosyanın içinde bulunduğu dizine "w" hakkının olması gerekir. Çünkü dosyanın silinmesi sırasında aslında o dosyaya değil, o dosyanın içinde bulunduğu dizine yazma yapılmaktadır. (Dizin girişinin silinmesinin dizin dosyasına yazma yapma anlamına geldiğine dikkat ediniz.)

Pekiyi unlink ve remove fonksiyonlarıyla bir dosya silinmeye çalışılırken o dosya bir proses tarafından açıksa ne olacaktır? İşte UNIX türevi sistemlerde bu durumda dosyaya ilişkin dizin girişi silinmekte ve katı bağ sayacı 1 eksiltilmektedir. Eğer dosyanın katı bağ sayacı 0'a düşmüşse bu durum çekirdek tarafından not alınmakta ve dosyayı son proses de kapattığında gerçek anlamda silme işlemi yapılmaktadır. Yani biz bu fonksiyonlarla dosya açık olsa bile silme işlemini yapabiliriz. Bu durumda dosya artık dizin girişlerinde görülmeyecek ancak gerçek silme gecikmeli olarak yapılacaktır.

unlink ve remove fonksiyonlarıyla dizinlerin silinip silinmeyeceği POSIX standartlarında işletim sistemini yazanların tercihine bırakılmıştır. POSIX standartları işletim sistemi destekliyorsa ve ancak ayrıcalıklı bir proses tarafından (tipik olarak kök prosesi) bu işlemin yapılabileceğini belirtmiştir. Linux ve Mac OS X sistemlerinde unlink ve remove fonksiyonlarıyla dizinlerin silinmesine izin vermemektedir.

unlinkat fonksiyonu unlink fonksiyonunun at'li biçimidir. Bu anlamda fonksiyonun davranışı temel olarak diğer at'li biçimlerde olduğu gibidir. Ancak unlinkat fonksiyonuyla istenirse dizinler de silinebilmektedir. Prototipini inceleyiniz:

```
#include <unistd.h>
#include <fcntl.h>

int unlinkat(int fd, const char *path, int flag);
```

Fonksiyonun prototipi <unistd.h> dosyası içerisindedir. Birinci parametrede kullanılabilen AT_FDCWD sembolik sabiti ve son parametrede kullanılabilen AT_REMOVEDIR sembolik sabiti <fcntl.h> içerisinde bildirilmiştir.

Fonksiyonun davranışını şöyle özetleyebiliriz:

1) Eğer fonksiyonun ikinci parametresi için girilen argüman mutlak yol ifadesiyse birinci parametresi için girilen argüman dikkate alınmaz. Benzer biçimde eğer fonksiyonun dördüncü parametresi için girilen argüman mutlak yol ifadesiyse üçüncü parametresi için girilen argüman dikkate alınmamaktadır. Dolayısıyla bu durumda fonksiyonun davranışı son parametresi dışında unlink fonksiyonuyla aynı olur.

2) Eğer fonksiyonun ikinci parametresindeki yol ifadesi göreli ise bu durumda bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil birinci parametresiyle belirtilen dizinden itibaren yol belirtir. Tabii bu durumda birinci parametrenin bir dizine ilişkin dosya betimleyicisi olması gerekmektedir. Benzer biçimde fonksiyonun dördüncü parametresindeki yol ifadesi göreli ise bu durumda da bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil üçüncü parametresiyle belirtilen dizinden itibaren yol belirtir. Fonksiyonun ikinci ya da dördüncü argümanına göreli yol ifadesi girildiğinde fonksiyonun birinci ve üçüncü argümanına ilişkin dizin betimleyicisinin O_SEARCH açış moduyla açılıp açılmadığında da bakılmaktadır. Eğer bu betimleyici O_SEARCH moduyla açılmışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaz. Ancak bu dizin O_SEARCH moduyla açılmamışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaktadır.

3) Fonksiyonun birinci ve üçüncü parametreleri için AT_FDCWD özel değeri kullanılırsa bu durumda ikinci ve dördüncü parametreler için girilen yol ifadeleri göreli olsa bile arama yine prosesin çalışma dizininden itibaren yapılmaktadır. Yani bu durumda fonksiyonun davranışı flags parametresi dışında unlink fonksiyonunda olduğu gibidir.

4) Eğer fonksiyonun son parametresi 0 yerine AT_REMOVEDIR girilirse bu durumda birinci ya da ikinci parametre için girilen argüman dizine ilişkinse dizin de silinebilmektedir. Yani  son parametre iiçin girilen AT_REMOVEDIR argümanı unlinkat fonksiyonu ile dizinlerin de silinmesini sağlamaktadır. Sonraki başlıkta da ele alındığı gibi bir dizinin silinebilmesi için dizinin içinin boş olması ("." ve ".." dışında dizi,n içerisinde hiçbir girişin bulunmaması) gerekmektedir.

unlink, remove ve unlinkat fonksiyonları başarı durumunda dizinin içinde bulunduğu dosyanın son güncelleme zamanını, eğer dosyanın katı bağ sayacı 0'a düşmemişse dosyanın son durum değişikliği zamanını güncellemektedir.

Bir dizin girişini silme işlemi kabuk üzerinde "rm" isimli POSIX komutuyla yapılmaktadır. Tabii bu "rm" programı aslında silme işlemini unlink fonksiyonunu kullanarak yapar. Pek çok UNIX türevi sistemde unlink ve unlinkat POSIX fonksiyonları doğrudan bu işlemi yapan sistem fonksiyonlarını çağırmaktadır. Örneğin Linux sistemlerinde unlink (dolayısıyla remove da) fonksiyonu sys_unlink sistem fonksiyonunu, unlinkat fonksiyonu da sys_unlinkat sistem fonksiyonunu çağırmaktadır.

### mkdir, mkdirat ve rmdir Fonksiyonları

Normal (regular) dosyaların open, openat ve creat fonksiyonlarıyla yaratıldığını, unlink, remove ve unlinkat fonksiyonlarıyla silindiğini görmüştük. İşte dizinler de mkdir ve mkdirat fonksiyonuyla yaratılıp, rmdir fonksiyonuyla silinmektedir. mkdir fonksiyonunun prototipi şöyledir:

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

int mkdir(const char *path, mode_t mode);
```

Fonksiyonun birinci parametresi yaratılacak dizinin yol ifadesini, ikinci parametresi ise erişim haklarını alır. İkinci parametre ile belirtilen erişim hakları ayrıca prosesin umask değeri ile işleme sokulmaktadır. Fonksiyon başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri döner. Dizin başarılı bir biçimde yaratılırsa dizin dosyasının kullanıcı id'si fonksiyonu çağıran prosesin etkin kullanıcı id'si olarak, grup id'si de fonksiyonu çağıran prosesin etkin grup id'si ya da yaratılan dizinin içinde bulunduru dizinin grup id'si olarak belirlenmektedir. Linux ve Mac OS X sistemlerinde default durumda yeni yaratılan dizinin grup id'si dizini yaratan prosesin etkin grup id'si olarak atandığını da

belirtelim. Ayrıca tıpkı normal (regular) dosyalarda olduğu gibi bir dizinin yaratılabilmesi için yaratma işlemini yapan prosesin dizinin yaratılacağı dizine "w" hakkının olması gerekmektedir.

Önceki konularda da belirttiğimiz gibi UNIX türevi sistemlerde (Windows sistemlerinde de böyle) yeni bir dizin yaratıldığında içerisinde otomatik olarak "." ve ".." isimli iki dizin girişi oluşturulmaktadır. "." dizini dizinin kendi dizinini, ".." dizini de dizinin üst dizinini belirtmektedir. Genel olarak içerisinde yalnızca "." ve ".." olan dizinlere "boş dizin" denilmektedir.

Aşağıdaki komut satırı argümanı ile aldığı yol ifadesine ilişkin dizin yaratan basit bir program görüyorsunuz:

```
/* mymkdir.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int is_octal(const char *str);
void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int i;
    int mode;
    mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH,
S_IXOTH};
    mode_t result_mode;

    if (argc != 3) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if (!is_octal(argv[1]) || (mode = (mode_t)strtoul(argv[1], NULL, 8)) > 0x777) {
        fprintf(stderr, "invalid octal digits!..\n");
        exit(EXIT_FAILURE);
    }

    result_mode = 0;
    for (i = 8; i >= 0; --i)
        if (mode >> i & 1)
            result_mode |= modes[8 - i];

    umask(0);
    if (mkdir(argv[2], result_mode) == -1)
        exit_sys("mkdir");

    return 0;
}

int is_octal(const char *str)
{
    int i;

    for (i = 0; str[i] != '\0'; ++i)
        if (str[i] < '0' || str[i] > '7')
            return 0;

    return 1;
}

void exit_sys(const char *msg)
{
```

```
    perror(msg);

    exit(EXIT_FAILURE);
}
```

Programı aşağıdaki gibi derleyip test edebilirsiniz:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ gcc -o mymkdir mymkdir.c
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ./mymkdir 777 testdir
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -ld testdir
drwxrwxrwx 2 kaan study 4096 Mar  2 13:20 testdir
```

Programda mkdir fonksiyonu çağrılmadan önce prosesin umask değerinin 0 yapıldığına dikkat ediniz. Aksi halde komut satırında verdiğiniz erişim hakları prosesin default umask değerinin tersi ile maskelenecektir.

mkdirat fonksiyonu mkdir fonksiyonunun at'li biçimidir. Bu nedenle genel olarak çalışma biçimi diğer at'li fonksiyonlara benzerdir. Aşağıda mkdirat fonksiyonunun prototipini görüyorsunuz:

```
#include <sys/stat.h>
#include <fcntl.h>

int mkdirat(int fd, const char *path, mode_t mode);
```

Fonksiyonun prototipi <sys/stat.h> dosyası içerisindedir. Birinci parametre için kullanılabilecek AT_FDCWD sembolik sabiti <fcntl.h> içerisinde bildirilmiştir.

Fonksiyonun çalışması şöyledir:

1) Eğer fonksiyonun ikinci parametresi için girilen argüman mutlak yol ifadesiyse birinci parametresi için girilen argüman dikkate alınmaz. Benzer biçimde eğer fonksiyonun dördüncü parametresi için girilen argüman mutlak yol ifadesiyse üçüncü parametresi için girilen argüman dikkate alınmaz. Dolayısıyla bu durumda fonksiyonun davranışı son parametresi dışında mkdir fonksiyonuyla aynı olur.

2) Eğer fonksiyonun ikinci parametresindeki yol ifadesi göreli ise bu durumda bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil birinci parametresiyle belirtilen dizinden itibaren yol belirtir. Tabii bu durumda birinci parametrenin bir dizine ilişkin dosya betimleyicisi olması gerekmektedir. Benzer biçimde fonksiyonun dördüncü parametresindeki yol ifadesi göreli ise bu durumda da bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil üçüncü parametresiyle belirtilen dizinden itibaren yol belirtir. Fonksiyonun ikinci ya da dördüncü argümanına göreli yol ifadesi girildiğinde fonksiyonun birinci ve üçüncü argümanına ilişkin dizin betimleyicisinin O_SEARCH açış moduyla açılıp açılmadığında da bakılmaktadır. Eğer bu betimleyici O_SEARCH moduyla açılmışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaz. Ancak bu dizin O_SEARCH moduyla açılmamışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaktadır.

3) Fonksiyonun birinci ve üçüncü parametreleri için AT_FDCWD özel değeri kullanılırsa bu durumda ikinci ve dördüncü parametreler için girilen yol ifadeleri göreli olsa bile arama yine prosesin çalışma dizininden itibaren yapılmaktadır. Yani bu durumda fonksiyonun davranışı mkdir fonksiyonunda olduğu gibidir. mkdir ve mkdirat fonksiyonları başarı durumunda yeni yaratılan dizinin son erişim zamanını, son güncelleme zamanını ve son durum değişikliği zamanını, aynı zamanda da dizin dosyasının içinde yaratıldığı dizininin de son güncelleme zamanını ve son durum değişikliği zamanını güncellemektedir.

rmdir fonksiyonu boş bir dizini dizini silmek için kullanılmaktadır. Fonksiyonun prototipi şöyledir:

```
#include <unistd.h>

int rmdir(const char *path);
```

Fonksiyonun parametresi silinecek dosyanın yol ifadesini belirtir. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri döner.

rmdir fonksiyonu da tıpkı unlink, remove ve unlinkat fonksiyonlarında olduğu dizine ilişkin üst dizindeki dizin girişini siler ve dizine ilişkin katı bağ sayacını 1 eksiltir. Eğer dizinin katı bağ sayacı 0'a düşmüşse dizini gerçek anlamda silmektedir. rmdir fonksiyonu sembolik bağlantı dosyalarında çalışmaz. Yani bir dizini gösteren sembolik bağlantı dosyasına rmdir uygulandığında fonksiyon başarısız olmaktadır. Tıpkı unlink, remove ve unlinkat fonksiyonlarında olduğu gibi rmdir fonksiyonu ile bir dizini silebilmek için dizinin içinde bulunduğu dizine "w" hakkının olması gerekmektedir. rmdir fonksiyonu çağrıldığında bu dizini açmış olan bir proses varsa dizin girişi silinmekte birlikte gerçek silme işlemi dizini kullanan son proses dizini kapattığında yapılmaktadır.

rmdir fonksiyonu başarı durumunda dizinin içinde bulunduğu dizinin son güncelleme zamanını ve son durum değişikliği zamanını güncellemektedir.

Bir dizini silmek için "rmdir" isimli bir POSIX kabuk komutu da bulunmaktadır. Bu komut rmdir fonksiyonu çağrılarak gerçekleştirilmiştir. Dolayısıyla rmdir fonksiyonundaki kısıtlar bu komu için de geçerlidir. rmdir fonksiyonu pek çok UNIX türevi sistemde doğrudan bu işi yapan bir sistem fonksiyonunu çağırır. Örneğin Linux sistemlerinde rmdir POSIX fonksiyonu doğrudna sys_rmdir isimli sistem fonksiyonunu çağıracak biçimde yazılmıştır.

### rename ve renameat Fonksiyonları

Bir dizin girişinin ismini yaratıldıktan sonra rename ve renameat isimli POSIX fonksiyonlarıyla değiştirebiliriz. rename POSIX fonksiyonu olmanın yanı sıra aynı zamanda zaten standart bir C fonksiyonudur. Prototipini inceleyiniz:

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Fonksiyonun birinci parametresi ismi değiştirilecek eski dizin girişini ikinci parametresi ise yeni ismi belirtmektedir. Eski ve yeni dizin girişi isimlerinin aynı diziin içinde olması gerekmemektedir. Yani biz bir dizinin ismini başka bir dizinde bir isim olarak değiştirebiliriz. Başka bir deyişle UNIX türevi sistemlerde isim değiştirme bir bakıma taşıma (move) anlamına gelmektedir. rename fonksiyonuyla dosyaların değil aynı zamanda dizinlerin de isimlerini değiştirebilirsiniz. Yani isim değiştirme bağlamında dosya ile dizin arasında bir farklılık yoktur. Aşağıda rename fonksiyonunun kullanımına bir örnek görüyorsunuz:

```
/* mymv.c */

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if (rename(argv[1], argv[2]) == -1)
        exit_sys("rename");

    return 0;
}

void exit_sys(const char *msg)
{
    perror(msg);
```

```
    exit(EXIT_FAILURE);
}
```

Bir dizin girişinin ismini değiştirmek o dizine yazma yapmak anlamına geldiğinde göre rename fonksiyonunun başarılı olması için prosesin birinci ve ikinci parametreler için girilen girişlerin içinde bulunduğu dizinlere "w" hakkının bulunuyor olması gerekir.

rename fonksiyonunun işlevine ilişkin birkaç ayrıntıdan da bahsetmek istiyoruz. Eğer rename fonksiyonunda hedef dizin girişi (yani ikinci parametreyle belirtilen dizin girişi) varsa isim değiştirme bu giriş silinip gerçekleştirilmektedir. Bu nedenle fonksiyonu çağırırken dikkat etmelisiniz. Çünkü olan bir dizin girişini yanlışlıkla silebilirsiniz. Eğer fonksiyonun ikinci parametresiyle belirtilen dizin girişi bir dizin olarak varsa birinci parametreye ilişkin dizin girişinin de bir dizin belirtmesi gerekmektedir. Benzer biçimde ikinci parametreyle belirtilen dizin girişi dizin olmayan bir dosya biçiminde varsa birinci parametreye ilişkin dizin girişinin de dizin belirtmeyen bir dosya biçiminde olması gerekmektedir. rename fonksiyonun birinci parametresi sembolik bağlantı dosyası ise fonksiyon sembolik bağlantı dosyasının referans ettiği dosyanın ismini değil sembolik bağlantı dosyasının kendi ismini değiştirir. Benzer biçimde eğer fonksiyonun ikinci parametresi sembolik baağlantı dosyası ise biçiminde mevcut ise fonksiyon sembolik bağlantı dosyasının referans ettiği dosyayı değil sembolik bağlantı dosyasının kendisini silmektedir. Fonksiyonun iki parametresi aynı fiziksel dosyaya ilişkin bir dizin girişi belirtiyorsa fonksiyon bir şey yapmadan başarılı bir biçimde geri dönmektedir.

renameat fonksiyonu rename fonksiyonunun at'li biçimidir. Bu bağlamda fonksiyonun davranışı diğer at'li fonlsiyonlara benzerdir. Fonksiyonun prototipi şöyledir:

```
#include <fcntl.h>

int renameat(int oldfd, const char *old, int newfd, const char *new);
```

renameat fonksiyonunun davranışını şöyledir:

1) Eğer fonksiyonun ikinci parametresi için girilen argüman mutlak yol ifadesiyse birinci parametresi için girilen argüman dikkate alınmaz. Dolayısıyla bu durumda fonksiyonun davranışı son parametresi dışında rename fonksiyonuyla aynı olur.

2) Eğer fonksiyonun ikinci parametresindeki yol ifadesi göreli ise bu durumda bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil birinci parametresiyle belirtilen dizinden itibaren yol belirtir. Tabii bu durumda birinci parametrenin bir dizine ilişkin dosya betimleyicisi olması gerekmektedir. Fonksiyonun ikinci argümanına göreli yol ifadesi girildiğinde fonksiyonun birinci argümanına ilişkin dizin betimleyicisinin O_SEARCH açış moduyla açılıp açılmadığında da bakılmaktadır. Eğer bu betimleyici O_SEARCH moduyla açılmışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaz. Ancak bu dizin O_SEARCH moduyla açılmamışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaktadır.

3) Fonksiyonun birinci parametresi için AT_FDCWD özel değeri kullanılırsa bu durumda ikinci parametresi için girilen yol ifadesi göreli olsa bile arama yine prosesin çalışma dizininden itibaren yapılmaktadır. Yani bu durumda fonksiyonun davranışı flags parametresi dışında rename gibidir.

rename ve renameat fonksiyonları başarı durumunda birinci ve ikinci parametresi ile belirtilen girişlerin bulunduğu dizin dosyalarının son güncelleme ve son durum değişikliği zamanlarını güncellemektedir.

**truncate ve ftruncate fonksiyonları**

Mevcut bir dosyanın uzunluğunu truncate ya da ftruncate isimli POSIX fonksiyonlarıyla değiştirebiliriz. truncate fonksiyonunun prototipi şöyledir:

```
#include <unistd.h>

int truncate(const char *path, off_t length);
```

Fonksiyonun birinci parametresi uzunluğu değiştirilecek dosyanın yol ifadesini, ikinci parametresi ise dosyanın yeni uzunluğunu belirtmektedir. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri döner. truncate fonksiyonunun ikinci parametresi için girilen değer dosyanın o anki uzunluğundan büyük ya da küçük (ya da eşit) olabilir. Eğer bu parametre için değer dosyanın o anki uzunluğundan küçükse dosya kırpılarak belirtilen uzunluktan sonraki kısım yok edilir. Eğer son parametre için girilen değer dosyanın uzunluğundan fazla ise bu durumda dosya deliği oluşmaktadır. Dosya deliklerininin gerçek anlamda bir disk tahsisatına yol açmadığını ve delikten okuma yapıldığında hep 0 okunduğunu anımsayınız.

truncate işlemi dosyaya yazma yapılmasına yol açmaktadır. Bu nedenle prosesin truncate yapılan dosyaya yazma "w" hakkının olması gerekmektedir. Fonksiyon başarı durumunda dosyanın son güncelleme zamanını ve son durum değişikliği zamanını güncellemektedir.

ftruncate fonksiyonu truncate fonksiyonunun dosya betimleyicisi ile çalışan biçimidir. Prototipini inceleyiniz:

```
#include <unistd.h>

int ftruncate(int fildes, off_t length);
```

Fonksiyonun birinci parametresi dosyaya ilişkin dosya betimleyicisini, ikinci parametresi dosyanın yeni uzunluğunu belirtmektedir. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri döner. Fonksiyonun birinci parametre dışındaki davranışı tamamen trunc fonksiyonunda olduğu gibidir.

### Dosya Bilgilerinin Elde Edilmesi: stat, fstat, fstatat ve lstat Fonksiyonları

Önceki bölümlerde bir dosyanın bütün önemli metadata bilgilerinin disk üzerinde i-node blok denilen bölümde bulunan i-node elemanında saklandığını belirtmiştik. Bu bölümde i-node elemanlarındaki dosya bilgilerinin nasıl elde edileceğini göreceğiz.

Elimizde bir dosyaya ilişkin yol ifadesi varsa ya da açılmış bir dosya betimleyicisi varsa stat, fstat, fstatat ve lstat POSIX fonksiyonlarıyla dosyaya ilişkin tüm bilgileri elde edebiliriz. (Örneğin ls -l komutu da dosyaya ilişkin bilgileri aslında bu fonksiyonları çağırarak elde etmektedir.) Bu nedenle stat, fstat, fstatat ve llstat POSIX fonksiyonları en önemli yardımcı dosya fonksiyonlarındandır. UNIX türevi sistemlerde genel olarak stat, fstat, fstatat ve lstat fonksiyonları doğrudan işletim sisteminin bu işlemleri yapan sistem fonksiyonlarını çağırırlar. Örneğin Linux sistemlerinde stat, fstat, fstatat ve lstat fonksiyonları sys_stat, sys_fstat, sys_fstatat ve sys_lstat isimli sistem fonksiyonlarını çağırmaktadır.

stat fonksiyonu dosyanın yol ifadesinden hareketle dosya bilgilerini elde eder. Fonksiyonun prototipini inceleyiniz:

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

int stat(const char *restrict path, struct stat *restrict buf);
```

Fonksiyonun birinci parametresi bilgileri elde edilecek dosyanın yol ifadesini, ikinci parametresi ise dosya bilgilerinin yerleştirileceği struct stat türünden yapı nesnesinin adresini alır. Fonksiyon, ikinci parametresiyle aldığı bu yapı nesnesinin içini doldurmaktadır. (Bu parametrenin const olmayan bir yapı göstericisi olduğuna dikkat ediniz.) stat fonksiyonu başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri döner. struct stat yapısı <sys/stat.h> dosyası içerisinde şöyle bildirilmiştir:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
```

```
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Yapının st_dev elemanı dev_t türündendir, dosyanın içinde bulunduğu aygıta ilişkin aygıt sürücüsünün aygıt numarasını belirtmektedir. Dosyalar diskte (ya da başka bir yerde) bulunurlar. Disk aygıtları da aygıt sürücülerle yönetilmektedir. İşte yapının bu st_dev elemanı dosyanın içinde bulunduğu aygıtı yöneten aygıt sürücüsünün aygıt numarasını belirtir. dev_t türü POSIX standartlarına göre herhangi bir tamsayı türü olarak typedef edilmiş olmak zorundadır.

Yapının st_inode elemanı ino_t türündendir, dosyanın i-node numarasını belirtmektedir. Dosyaların i-node numaralarının i-node elemanlarında değil dizin girişlerinde bulunduğunu anımsayınız. (stat fonksiyonu zaten dosyanın i-node numarasından hareketle i-node elemanını belirler.) ino_t türü POSIX standartlarına göre herhangi bir işaretsiz tamsayı türü olarak typedef edilmiş olmak zorundadır.

Yapının st_mode isimli elemanı mode_t türündendir, dosyanın türünü ve erişim haklarını belirtir. mode_t türü POSIX standartlarına göre işaretli bir tamsayı türü olarak typedef edilmiş olmak zorundadır. Bu eleman hakkında ayrıntılı açıklamayı birkaç paragraf ötede bulacaksınız.

Yapının st_nlink elemanı nlink_t türündendir. Dosyanın katı bağ sayaç değerini belirtmektedir. nlink_t türü POSIX standartlarına göre herhangi bir tamsayı türü olarak typedef edilmiş olmak zorundadır.

Yapının st_uid ve st_gid elemanları sırasıyla uid_t ve gid_t türündendir, dosyanın kullanıcı id'sini ve grup id'sini belirtmektedir. uid_t ve gid_t türleri POSIX standartlarına göre herhangi bir tamsayı türü olarak typedef edilmiş olmak zorundadır. Dosyaların kullanıcı ve grup id'lerinin i-node elemanında sayısal biçimde kodlandığına dikkat ediniz. Bu id'lerin nasıl kullancı ve frup isimlerine dönüştürüleceği başka bir bölümde açıklanmaktadır.

Yapının st_rdev elemanı eğer dosya bir aygıt sürücü dosyası ise onun aygıt numarasını belirtmektedir. Aygıt dosyalarının aygıt numaralarının ne anlama geldiği "aygıt sürücülerin" anlatıldığı bölümde açıklanmaktadır.

Yapının st_size elemanı off_t türündendir, dosyanın byte uzunluğunu belirtmektedir. off_t türü -lseek fonksiyonunu anlattığımız bölümde de belirttiğimiz gibi- POSIX standartlarına göre işaretli bir tamsayı türü olarak typedef edilmiş olmak zorundadır.

Yapının st_blksize elemanı blksize_t türündendir ve dosyanın içinde bulunduğu dosya sistemine ilişkin blok büyüklüğünü belirtmektedir. Dosyaların parçaları disk üzerinde belirli büyüklükteki bloklarda dağıtık olarak bulunabilmektedir. Bu anlamda blok kavramı bir dosyanın parçası olabilecek en küçük disk birimini belirtir. Blok büyüklüğü hemen her zaman 512'nin (bir disk sektörünün uzunluğu 512 byte'tır) katları biçimindedir. Örneğin dosya sistemindeki blok büyüklüğü 4096 byte ise dosyamız 1 byte uzunluğunda olsa bile diskte 4096 byte yer kaplayacaktır. Benzer biçimde örneğin dosyamız 5000 byte büyüklüğünde ise aslında bu dosya diskte 2 * 4096 = 8192 byte yer kaplar. Blok kavramının ne anlam ifade ettiğine ilişkin ayrıntılar kitabımızda i-node tabanlı dosya sistemlerinin ele alındığı bölümde ayrıntılarıyla açıklanmaktadır. Yapının st_blksize elemanı aynı zamanda dosya kopyalama gibi işlemlerde alınacak optimal tampon büyüklüğünü de belirtir. Kopyalama sırasında kaynak dosyadan hedef dosyaya blok blok aktarma yapılırken tek hamlede dosyadan okunacak ya da yazılacak optimum miktar genellikle burada belirtilen uzunluk olmaktadır. blksize_t türü POSIX standartlarına göre işaretli bir tamsayı türü olarak typedef edilmiş olmak zorundadır.

Yapının st_blocks elemanı blkcnt_t türündendir ve dosyanın diskte kaç disk tahsisat birimi kadar yer kapladığını belirtir. Ancak burada belirtilen disk tahsisat birimi sistemden sisteme değişebilmektedir. (Örneğin Linux ve Mac OS X sistemleri için bu değer 1 sektör yani 512 byte'tır.) st_blocks değerinin st_blksize ile doğrudan bir ilgisi yoktur. st_blksize dosya sistemindeki blok büyüklüğünü belirtmektedir. Halbuki st_blocks dosyanın kaç disk birimi kadar yer kapladığını belirtir. Dosyanın diskte na kadar yer kapladığını hesaplamak için st_blocks elemanında yazan değerin o sistemdeki disk birim uzunluğu ile (Linux ve Mac OS X'te 512) çarpılması gerekir. blkcnt_t türü de POSIX standartlarına göre işaretli bir tamsayı türü biçiminde typedef edilmiş olmak zorundadır. (Yapının st_blksize elemanı ile st_blocks elemanın anlamı bazen programcılar tarafından karıştırılabilmektedir. Durumu şöyle açıklığa kavuşturalım: Yapının st_blksize elemanı bir

dosyanın parçalarının dosya sisteminde kaç byte'lık bloklarda tutulacağını belirtir. Yapının st_blocks elemanı ise dosyanın diskte kapladığı toplam alanı belli bir büyüklüğün (tipik olarak 512) katı olarak vermektedir. Örneğin dosyamız 5000 byte olsun. Yapının st_blksize elemanının 4096 ve st_blocks elemanına ilişkin birimin de 512 olduğunu varsayalım. 5000 byte'lık dosya tek bir 4096'lık bloğa sığmaz. Dolayısıyla işletim sistemi bunun için diskte iki blok ayıracaktır. İki bloğun diskte toplam kapladığı alan 4096 * 2 = 8192 byte'tır. Bu durumda st_blocks elemanında 16 değerinin bulunması gerekir. Çünkü 8192 / 512 = 16'dır.)

UNIX sistemlerinde kullanılan dosya sistemlerinde her dosya için üç zamanın tutulduğunu belirtmiştik. İşte yapının st_atime, st_mtime ve st_ctime elamanları time_t türündendir ve sırasıyla dosyanın son okunma zamanını, son günceleme zamanını ve son durum değişilkliği zamanını belirtir. time_t türü POSIX standartlarına göre bir tamsayı biçiminde typedef edilmiş olmak zorundadır. (Bu türün typedef koşulunun C Programlama Dilindekinden farklı olduğuna dikkat ediniz. C Programlama Dilinde time_t türü tamsayı ya da gerçek sayı türü olarak typedef edilmiş olabilir.) Yazılımda zaman ölçmekte kullanılan orijin noktasına "epoch" denilmektedir. POSIX sistemlerinde "epoch" 01/01/1970 00:00 olarak belirlenmiştir. Böylece yapının st_atime, st_mtime ve st_ctime elemanları aslında "01/01/1970 00:00"dan itibaren geçen saniye sayısını tutar. (C Programlama Dilinde time_t türü için açık bir "epoch" belirtilmemiştir. Yani bu türün belirttiği saniye sayısı C Programlama Dilinin standartlarına göre derleyiciden derleyiciye değişebilir.) C Programlama Dilinden de bildiğiniz gibi elimizde time_t türünden bir "epoch" varsa onu "ctime" fonksiyonuyla yazısal biçime dönüştürebiliriz, "localtime" fonksiyonuyla onu bileşenlerine ayırabiliriz ve "strftime" fonksiyonuyla da istediğimiz gibi formatlayabiliriz.

Şimdi stat yapısının bu st_mode elemanı hakkında biraz daha ayrıntılı bilgi verelim. Yapının bu elemanı dosya türünü ve erişim haklarını bit bit kodlamaktadır. Ancak bu elemanın hangi bitlerinin hangi dosya türlerine, hangi bitlerinin hangi erişim haklarına ilişkin olduğu UNIX türevi sistemlerde sistemden sisteme değişebilmektedir. Bu nedenle <sys/stat.h> dosyası içerisinde mode_t türünden bir nesnenin içerisinden dosyanın tür bilgisini ve erişim haklarını elde edebilmek için çeşitli makrolar bulundurulmuştur. Buna göre dosyanın türü aşağıdaki makrolardan biri ile elde edilebilir:

| Tür Belirlemekte Kullanılan Makrolar | Anlamı |
|---|---|
| S_ISREG(m) | Dosya normal (regular) bir dosya mı? |
| S_ISDIR(m) | Dosya bir dizin dosyası mı? |
| S_ISCHR(m) | Dosya bir karakter aygıt sürücü dosyası mı? |
| S_ISBLK(m) | Dosya bir blok aygıt sürücü dosyası mı? |
| S_ISFIFO(m) | Dosya bir boru (pipe) dosyası mı? (Boru dosyalarını "fifo" dosyaları da denilmektedir.) |
| S_ISLNK(m) | Dosya sembolik bağlantı dosyası mı? |
| S_ISSOCK(m) | Dosya bir soket dosyası mı? |

Bu makrolar stat yapının st_mode elemanını parametre olarak alırlar. Eğer yapılan test olumluysa 0 dışı bir değere, olumsuzsa 0 değerine geri dönerler. Örneğin dosyanın bir dizin dosyası olup olmadığı şöyle belirlenebilir:

```
struct stat finfo;
...

if (stat(path, &finfo) == -1)
    exit_sy("stat");

if (S_ISDIR(finfo.st_mode)) {
    ....
}
```

Dosya türünü belirleme işlemi yukarıdaki makroların dışında sembolik sabitler kullanılarak da yapılabilmektedir. Yapının st_mode elemanı aşağıdaki sembolik sabitlerle "bit and" işlemine sokulduğunda 0 dışı bir değer elde ediliyorsa dosya ilgili türden, 0 elde ediliyorsa dosya ilgili türden değildir:

| Tür Belirlemekte Kullanılan Sembolik Sabitler | Anlamı |
|---|---|

| | |
|---|---|
| S_IFREG | Dosya normal (regular) bir dosya mı? |
| S_IFDIR | Dosya bir dizin dosyası mı? |
| S_IFCHR | Dosya bir karakter aygıt sürücü dosyası mı? |
| S_IFBLK | Dosya bir blok aygıt sürücü dosyası mı? |
| S_IFFISO | Dosya bir boru (pipe) dosyası mı? (Boru dosyalarını "fifo" dosyaları da denilmektedir.) |
| S_IFLNK | Dosya sembolik bağlantı dosyası mı? |
| S_IFSOCK | Dosya bir soket dosyası mı? |

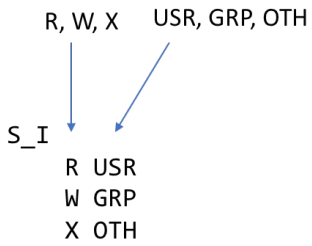
Bu durumda örneğin yukarıdaki dizin testini şöyle de yapabilirdik:

```
struct stat finfo;
...

if (stat(path, &finfo) == -1)
    exit_sy("stat");

if (S_IFDIR & finfo.st_mode) {
    ....
}
```

stat yapısının st_mode elemanında hem dosyanın tür bilgisinin hem de erişim haklarının bulunduğunu belirtmiştik. Bazen programcı bu st_mode elemanından yalnızca erişim haklarını almak isteyebilir. Bunun için S_IFMT sembolik sabitinden faydalanılmaktadır. Bu sembolik sabit dosya tür bilgilerinin bulunduğu bitlerin 1 olduğu diğer bitlerin 0 olduğu bir değere sahiptir. Bu sayede örneğin biz dosya erişim haklarını aşağıdaki gibi elde edebiliriz:

```
printf("%lo\n", (unsigned long)(finfo.st_mode & ~S_IFMT))
```

Dosyanın erişim haklarının S_I ile başlayan öneklerle sembolik sabit olarak nasıl isimlendirildiğini open fonksiyonunda görmüştük. Burada yine de bir tekrar yapmak istiyoruz. Dosyanın erişim hakları için toplam 9 sembolik sabit bildirilmiştir. Bu sembolik sabitlerin her biri dosyanın "rwxrwxrwx" haklarından birinin olup olmadığını belirlemek amacıyla kullanılmaktadır. İşte tipik olarak programcı yapının st_mode elemanını bu 9 sembolik sabitle "bit and" işlemine sokar. Eğer bu işlemden 0 dışı bir değer elde ederse bu erişim hakkının olduğu 0 değerini elde ederse bu erişim hakkının olmadığı sonucunu çıkartır. open fonksiyonundan da anımsayacağınız gibi erişim haklarını belirten bu sembolik sabitler S_I öneki ile başlatılmıştır. Bu öneki sırasıyla R, W ya da X harfleri, bu harfleri de USR, GRP ve OTH sonekleri izlemektedir.




| Sembolik Sabit | Anlamı |
|---|---|
| S_IRUSR | Dosyanın sahibine "r" hakkını temsil eder |
| S_IWUSR | Dosyanın sahibine "w" hakkını temsil eder |
| S_IXUSR | Dosyanın sahibine "x" hakkını temsil eder |
| S_IRGRP | Dosyanın grubuna "r" hakkını temsil eder |
| S_IWGRP | Dosyanın grubuna "w" hakkını temsil eder |
| S_IXUSR | Dosyanın grubuna "x" hakkını temsil eder |
| S_IROTH | Diğerlerine "r" hakkını temsil eder |
| S_IWOTH | Diğerlerine "w" hakkını temsil eder |
| S_IXOTH | Diğerlerine "x" hakkını temsil eder |

| | |
|---|---|
| S_IRWXU | Dosyanın sahibine "r", "w" ve "x" hakkını temsil eder |
| S_IRWXG | Dosyanın grubuna "r", "w" ve "x" hakkını temsil eder |
| S_IRWXO | Diğerlerine "r", "w" ve "x" hakkını temsil eder |
| S_ISUID | Set user id bayrağını temsil eder |
| S_ISGID | Set grup is bayrağını temsil eder |
| S_ISVTX | Sticky bayrağını temsil eder |

Peki bir dosyanın bilgilerini stat fonksiyonuyla alabilmek için özel bir koşulun sağlanması gerekmekte midir? Yanıt hayır. Biz yol ifadesi olarak erilşebildiğimiz tüm dosyaların bilgilerini stat fonksiyonuyla elde edebiliriz.

Aşağıdaki örnekte komut satırı argümanı olarak verilen dosyalara stat fonksiyonu uygulanmış ve elde edilen bilgiler ekrana yazdırılmıştır:

```
/* mystat.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <sys/stat.h>
#include <unistd.h>

const char *lsmode(mode_t mode);

int main(int argc, char *argv[])
{
    struct stat finfo;
    struct tm *pt;
    char buf[50];
    int i;

    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 1; i < argc; ++i) {
         if (i != 1)
            printf("-----------------------------------------\n");

        if (stat(argv[i], &finfo) == -1) {
            fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));
            continue;
        }

        printf("Name: %s\n", argv[i]);
        printf("Inode No: %lu\n", (unsigned long)finfo.st_ino);
        printf("Hard Link: %lu\n", (unsigned long)finfo.st_nlink);
        printf("Access modes: %s\n", lsmode(finfo.st_mode));
        printf("User Id: %ld\n", (unsigned long)finfo.st_uid);
        printf("Group Id: %lu\n", (unsigned long)finfo.st_gid);
        printf("File Size: %ld\n", (long)finfo.st_size);
        printf("File System Block (Cluster) Size: %ld\n", (long)finfo.st_blksize);
        printf("Number Of Sectors (512 Bytes): %ld\n", (long)finfo.st_blocks);

        pt = localtime(&finfo.st_mtime);
        strftime(buf, 50, "%b %d %H:%M", pt);
        printf("Last Update: %s\n", buf);

        pt = localtime(&finfo.st_atime);
```

```
        strftime(buf, 50, "%b %d %H:%M", pt);
        printf("Last Access: %s\n", buf);

        pt = localtime(&finfo.st_atime);
        strftime(buf, 50, "%b %d %H:%M", pt);
        printf("Last Status Change: %s\n", buf);
    }

    return 0;
}

const char *lsmode(mode_t mode)
{
    static char modetxt[11];
    mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP,
                              S_IROTH, S_IWOTH, S_IXOTH};
    int i;

    if (S_ISREG(mode))
        modetxt[0] = '-';
    else if (S_ISDIR(mode))
        modetxt[0] = 'd';
    else if (S_ISCHR(mode))
        modetxt[0] = 'c';
    else if (S_ISBLK(mode))
        modetxt[0] = 'b';
    else if (S_ISFIFO(mode))
        modetxt[0] = 'p';
    else if (S_ISLNK(mode))
        modetxt[0] = 'l';
    else if (S_ISSOCK(mode))
        modetxt[0] = 's';
    else
        modetxt[0] = '?';

    for (i = 0; i < 9; ++i)
        modetxt[1 + i] = (mode & modes[i]) ? "rwx"[i % 3] : '-';

    return modetxt;
}
```

Programı aşağıdaki gibi derleyip çalıştırabilirsiniz:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ gcc -o mystat mystat.c
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ./mystat sample.c test.c
Name: sample.c
Inode No: 4986539
Hard Link: 1
Access modes: -rw-rw-r--
User Id: 1000
Group Id: 1001
File Size: 2397
File System Block (Cluster) Size: 4096
Number Of Blocks (512 Bytes): 8
Last Update: Mar 05 17:11
Last Access: Mar 05 17:11
Last Status Change: Mar 05 17:11
----------------------------------------
Name: test.c
Inode No: 4990241
Hard Link: 1
Access modes: -rw-r--r--
User Id: 1000
Group Id: 1001
File Size: 919
File System Block (Cluster) Size: 4096
Number Of Blocks (512 Bytes): 8
Last Update: Feb 18 14:10
Last Access: Feb 18 14:10
Last Status Change: Feb 18 14:10
```

Programdaki lsmode isimli fonksiyon dosyaya ilişkin erişim haklarını alarak onu "ls -l" komutunda olduğu gibi yazıya dönüştürmektedir. Bu işlemin nasıl yapıldığına dikkat ediniz. Önce dosyanın türü tespit edilip uygun harf kodlanmıştır. Daha sonra bir dizi içerisindeki S_IXXX sembolik sabitleriyle bit and işlemi uygulanarak ilgili özelliklerin olup olmadığına bakılmıştır.

fstat fonksiyonu stat fonksiyonunun dosya betimleyicisi ile çalışan biçimidir. Bir dosyayı zaten açmışsak ve elimizde dosya betimleyicisi varsa onun bilgilerini fstat fonksiyonuyla daha hızlı bir biçimde elde edebiliriz. Fonksiyonun prototipi şöyledir:

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

int fstat(int fildes, struct stat *buf);
```

Fonksiyonun birinci parametre dışındaki bütün davranışı stat fonksiyonu ile aynıdır.

fstatat fonksiyonu da stat fonksiyonunun at'li biçimidir. Prototipini inceleyiniz:

```
#include <sys/stat.h>
#include <fcntl.h>

int fstatat(int fd, const char *restrict path, struct stat *restrict buf, int flag);
```

Fonksiyonun prototipi <sys/stat.h> dosyası içerisindedir. Birinci parametrede kullanılabilen AT_FDCWD sembolik sabiti ve son parametrede kullanılabilen AT_SYMLINK_NOFOLLOW sembolik sabiti <fcntl.h> dosyası içerisinde bildirilmiştir. fstatat fonkdiyonun davranışı şöyledir:

1) Eğer fonksiyonun ikinci parametresi için girilen argüman mutlak yol ifadesiyse birinci parametresi için girilen argüman dikkate alınmaz. Dolayısıyla fonksiyonun davranışı son parametresi dışında stat fonksiyonuyla aynı olur.

2) Eğer fonksiyonun ikinci parametresindeki yol ifadesi göreli ise bu durumda bu göreli yol ifadesi prosesin çalışma dizininden itibaren değil birinci parametresiyle belirtilen dizinden itibaren yapılmaktadır. Tabii bu durumda birinci parametrenin bir dizine ilişkin dosya betimleyicisi olması gerekir. Fonksiyonun ikinci argümanına göreli yol ifadesi

girildiğinde fonksiyonun birinci argümanına ilişkin dizin betimleyicisinin O_SEARCH açış moduyla açılıp açılmadığında da bakılmaktadır. Eğer bu betimleyici O_SEARCH moduyla açılmışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaz. Ancak bu dizin O_SEARCH moduyla açılmamışsa söz konusu dizin üzerinde "x" hakkı kontrolü yapılmaktadır.

3) Fonksiyonun birinci parametresi için AT_FDCWD özel değeri kullanılırsa bu durumda ikinci parametresi için girilen yol ifadesi göreli olsa bile arama yine prosesin çalışma dizininden itibaren yapılmaktadır. Yani bu durumda fonksiyonun davranışı flags parametresi dışında stat fonksiyonu gibidir.

4) Eğer fonksiyonun son parametresi 0 yerine AT_SYMLINK_NOFOLLOW biçiminde girilmişse bu durumda söz konusu hedef dosya sembolik bağlantı dosyasıysa bağlantı dosyasının gösterdiği dosya için değil bağlantı dosyasının kendisi için işlem yapılmaktadır.

lstat fonksiyonu stat fonksiyonunun sembolik bağlantıları izlemeyen biçimidir. Prototipini inceleyeniz:

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

int lstat(const char *restrict path, struct stat *restrict buf);
```

lstat fonksiyonunun birinci parametresi bir sembolik bağlantı dosyasının yol ifadesi ise fonksiyon sembolik bağlantı dosyasının referans ettiği dosyanın bilgilerini değil sembolik bağlantı dosyasının bilgilerini elde eder. Özellikle dizin ağacını dolaşan kodlarda stat yerine lsatat fonksiyonunun kullanılması uygun olmaktadır. Aksi takdirde dizinlere yapılan sembolik bağlantılar bu kodları sonsuz döngüye sokabilir.

<ARADAKİ KONULAR TAMAMLANACAK>

### Dosyaların Set User Id, Set Group Id ve Sticky Erişim Özellikleri

Şimdieye kadar bir dosyanın erişim haklarının üçerli üç gruptan oluştuğunu görmüştük. Bu üç grup dosyanın sahiplik, grupluk ve diğer haklarını belirtiyordu. Örneğin:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample.c
-rw-rw-r-- 1 kaan study 320 Nis 30 15:15 sample.c
```

Aslında dosya erişim haklarında bir üçlük grup daha vardır. Yani dosya erişim hakları üçerli üç gruptan değil üçerli dört gruptan oluşmaktadır. Bu başlıkta ee aldığımız bu yeni grubun özelliklerine sırasıyla "set user id", "set group id" ve "sticky" denilmektedir.

Dosyaların set user id, set group ve sticky erişim özelliklerini chmod POSIX fonksiyonuyla ya da kabuk üzerinden chmod komutuyla değiştirilebiliriz. chmod fonksiyonunda erişim haklarının S_IXXX biçimindeki sembolik sabitlerin bit düzeyinde veya işlemi ile oluşturulduğunu anımsayınız. İşte set user id, set group id ve sticky özellikleri için de aşağıdaki sembolik sabitler kullanılmaktadır:

```
S_ISUID
S_ISGID
S_ISVTX
```

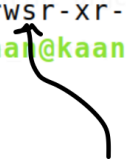
Böylece örneğin "sample" isimli çalıştırılabilir olan bir dosyaya set user id ve set group id özelliklerini aşağıdaki gibi ekleyebiliriz:

```
mode_t mode;
struct stat finfo;

if (stat("sample", &finfo) == -1)
     exit_sys("stat");
```

```
if (chmod("sample", finfo.st_mode & ~S_IFMT | S_ISUID) == -1)
    exit_sys("sample");
```

Dosyaların set user id, set group id ve stick özelliklerini "chmod" kabuk komutuyla da değiştirebiliriz. Bu komutta "u+s" seçeneği set user id özelliğini eklemek için "u-s" seçeneği bu özelliği çıkartmak için, "g+s" seçeneği set group id özelliğini eklemek için "g-s" seçeneği bu özelliği çıkartmak için, "o+t" seçeneği sticky özelliğini eklemek için "o-t" de bu özelliği çıkartmak için kullanılmaktadır. ls -l komutunda set user id, set group id ve stick için ayrı bir üçlük yer ayrılmamıştır. Bu üç özellik dosyanın sahiplik, grup ve diğer haklarındaki 'x' hakkında gizlice kodlanmaktadır. Örneğin "sample" dosyasının set user id özelliğini etkin hale getirelim:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample
-rwxr-xr-x 1 kaan study 8576 May  3 10:29 sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ chmod u+s sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample
-rwsr-xr-x 1 kaan study 8576 May  3 10:29 sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$
```



**Dosyanın sahibinde 'x' hakkı var ve dosya için set user id özelliği etkin hale getirilmiş**

ls -l komutunda dosyanın sahiplik haklarında 'x' hakkı yerinde 's' harfinin bulunması bu dosyanın hem sahiplik özelliğinde 'x' hakkının bulunduğu hem de dosyanın set user id özelliğinin etkin olduğu anlamına gelmektedir. Eğer dosyanın sahiplik haklarındaki 'x' hakkı yerine 'S' harfini görürseniz bu durum dosyanın sahiplik özelliğinde 'x' hakkının olmadığı ancak dosyanın set user id özelliğinin etkin olduğu anlamına gelir. Sonraki paragrafta da göreceğiniz gibi set user id özelliği yalnızca çalıştırılabilen dosyalar için anlamlıdır. Örneğin:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l test.txt
-rw-r--r-- 1 kaan study 8 Mar 21 22:55 test.txt
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ chmod u+s test.txt
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l test.txt
-rwSr--r-- 1 kaan study 8 Mar 21 22:55 test.txt
```

**Dosyanın sahiplik haklarında 'x' hakkı yok ancak dosyanın set user id özelliği etkin (bu durum anlamlı değil)**

Benzer biçimde eğer dosyanın grup erişim haklarındaki 'x' hakkı yerine 's' harfi varsa bu dosyanın hem grup özelliğinde 'x' hakkının oldduğu hem de dosyanın set group id özelliğinin etkin olduğu anlamına gelmektedir. Eğer dosyanın grup erişim haklarındaki 'x' yerine 'S' karakteri bulunuyorsa bu durum da dosyanın gruba erişim haklarında 'x' hakkının bulunmadığı ancak dosyanın set group id özelliğinin etkin hale getirildiği anlamına gelmektedir. Örneğin:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample
-rwxr-xr-x 1 kaan study 8576 May  3 10:29 sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ chmod g+s sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample
-rwxr-sr-x 1 kaan study 8576 May  3 10:29 sample
```

**Dosyanın grup erişim haklarında 'x' hakkı var ve dosya için set group id özelliği etkin**

Dosyanın diğer erişim haklarında 'x' yerine 't' harfi varsa ('s' değil 't' olduğuna dikkat ediniz) bu durum dosyanın diğer erişim haklarında 'x' hakkının olduğu ve dosyanın stick özelliğinin etkin olduğu anlamına gelmektedir. Benzer biçimde eğer dosyanın diğer erişim haklarında 'x' yerine 'T' harfi varsa bu da dosyanın diğer haklarında 'x' hakkının olmadığı ancak dosyanın sticky özelliğinin etkin olduğu anlamına gelmektedir. Örneğin:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample
-rwxr-xr-x 1 kaan study 8576 May  3 10:29 sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ chmod o+t sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample
-rwxr-xr-t 1 kaan study 8576 May  3 10:29 sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$
```

**Dosyanın diğer erişim haklarında 'x' hakkı var ve dosyanın sticky özelliği etkin**

chmod komutunda set user id, set group id ve sticky özellikleri sayısal argümanla da değiştirilebilir. Bunun için en sola dördüncü bir octal basamak eklemek gerekir. Örneğin:

```
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample
-rwxr-xr-x 1 kaan study 8576 May  3 10:29 sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ chmod 6755 sample
kaan@kaan-VirtualBox:~/Study/Unix-Linux-SysProg$ ls -l sample
-rwsr-sr-x 1 kaan study 8576 May  3 10:29 sample
```

**Dosyanın set user id ve set group id özellikleri etkin**

Biz şimdiye kadar dosyaların set user id, set group id ve sticky özelliklerinin nasıl etkin hale getirileceğini gördük. Şimdi de bu özelliklerin ne anlam ifade ettiğini açıklayacağız.

Öncelikle "set user id" özelliğini ele alalım. "Set user id" yalnızca çalıştırılabilir dosyalar için anlamlı olan bir özelliktir. Çalıştırılabilir olmayan dosyaların (script dosyaları da bu anlamda çalıştırılabilir değildir) set user id özelliğine sahip olmasının bir anlamı yoktur. (Yani örneğin siz bir metin dosyasının ya da script dosyasının set user id özelliğini etkin hale getirebilirsiniz ancak bunun bir anlamı yoktur.) Set user id özelliği bulunan bir dosyayı exec fonksiyonlarıyla çalıştırdığımızda exec uygulayan prosesin (yani çalıştırılan programın) etkin kullanıcı id'si exec uygulanan dosyanın kullanıcı id'si olacak biçimde değiştirilmektedir. Yani set user id özelliği etkin hale getirilmiş bir program dosyasını çalıştırdığımızda prosesimizin etkin kullanıcı id'si artık program dosyasının kullanıcı id'si yapılarak program çalıştırılmaktadır. Örneğin prosesimizin etkin kullanıcı id'si "kaan (1000)" olsun. Biz de bu proseste kullanıcı id'si "ali" olan ve set user id özelliği etkin olan "sample" isimli bir programı çalıştırmak isteyelim. Dosyayı çalıştırma hakkına sahip olduğumuzu varsayıyoruz. İşte exec işlemi ile bu "sample" programı çalıştırılırken artık prosesimizin etkin kullanıcı id'si "kaan" değil "ali" olacaktır. Şimdi konuyu bir örnekle açıklayalım. UNIX/Linux sistemlerinde kullanıcının parolalarını değiştirmek için /usr/bin dizininin altında "passwd" isimli bir programdan faydalanılmaktadır. "passwd" program dosyasının özellikler şöyledir: