Unix/Linux Sistem Programlama Ders Notları

CSD 2002

Bu notlar C ve Sistem Programcıları Derneği'nin düzenlediği, Kaan Aslan tarafından açılmış olan Unix/Linux Sistem Programlama kursunda derste tutulan notlardan derlenerek hazırlanmıştır.

C ve Sistem Programcılar Derneği
Adres: **2.Taşocağı Cad. Oğuz Sok. No:5 Mecidiyeköy / İstanbul**
Telefon: **(212) 288 35 20**

18-07-02

UNIX Grubu İşletim Sistemlerinin Tarihsel Gelişimi

UNIX işletim sisteminin tarihi 1968-69 senelerine dayanır. C programlama dili UNIX işletim sisteminin geliştirilmesi süresinde tasarlanmıştır. 1968 yılında AT&T'den K.Thombson, D.Ritchie ve B.Kernighan Multics isimli bir işletim sistemi projesine başladılar. Multics projesi MIT gibi üniversitelerle ortak olarak yürütülüyordu. Ancak Multics projesinin çeşitli problemlerinden dolayı 1969 yılında AT&T projeden ayrılma kararı vermiştir. 1969 yılında ekip yeni bir işletim sistemi yazma projesine başlamıştır. 1970 yılında B.Kernighan bu yeni işletim sistemini Multics'ten kelime oyunu yaparak UNIX ismini vermiştir.

UNIX işletim sistemi 1970 yılında DEC'in PDP-7 makinelerinde yazılmıştır. UNIX'in bu ilk versiyonu tamamen sembolik makine dilinde yazılmıştır. Yalnızca işletim sisteminin çekirdek kısmı değil, tüm yardımcı programlar da sembolik makine dilinde yazılmışlardır. K.Thombson bu yeni işletim sistemi için önceleri Fortran derleyicisi yazmaya niyetlendiyse de daha sonra tamamen yeni bir arayışa geçmiştir. K.Thombson BCPL dilinden esinlenerek alçak seviyeli işlemlerde kullanılabilecek B programlama dilini tasarlamış ve derleyicisini yazmıştır. B her yönü ile tanımlanmış bir programlama dili değildir. Ancak çalışma ekibini sembolik makine dilinden kurtarmıştır. D.Ritchie bu süreç içerisinde B programlama dilini geliştirerek bugün kullandığımız C programlama dilini tasarlamış ve derleyicisini yazmıştır (1971).

1973 yılında ekip DEC'in PDP-11 makineleri için UNIX işletim sistemini tamamen C programlama dilini kullanarak yeniden yazmıştır. Bu özelliği ile UNIX sembolik makine dilinde yazılmayan ilk işletim sistemi olmuştur. Bu durum işletim sisteminin kaynak kodlarının taşınabilmesini sağlamış ve UNIX işletim sisteminin DEC makinelerinin dışında da çalışabilmesine ön ayak olmuştur.

UNIX işletim sisteminin kaynak kodları pek çok araştırma gurubuna önceleri ücret talep edilmeden verilmiştir. Bunun sonucu olarak bu ürünün geliştirilip pek çok farklı UNIX işletim sistemlerinin oluşmasına neden olmuştur. Bunların en ünlüsü Berkeley tarafından geliştirilmiş olan BSD versiyonlarıdır (Berkeley Software Distribution). BSD versiyonlarından üçü serbest olarak dağıtılmıştır (Free BSD, Net BSD, Open BSD). Zamanla AT&T'nin orijinal UNIX sistemleri ile Berkeley'in sistemlerinde rekabet oluşmuşsa da AT&T'nin sistemleri üstünlük kazanmıştır. AT&T, UNIX sistemlerine bir versiyon numarası vermiştir. Ayrıca bu sistemler önce System3 daha sonra da System5 biçiminde isimlendirilmiştir. UNIX grubu sistemlerin temel referansı System5'tir.

AT&T 80'li yıllarda UNIX sistemlerine telif uygulama kararı almıştır. Bu durum üniversitelileri tedirgin etmiştir, çünkü üniversitelerde bilgisayar araştırmalarında hep UNIX'in kaynak kodları kullanılıyordu. Bunun üzerine Hollandalı bir profesör olan Andrew Tanenbaum kendi derslerinde kullanabileceği küçük bir UNIX sistemi yazmıştır. Bu işletim sistemi Minix diye isimlendirilmiştir.

1984 yılında Richard Stallman serbest yazılım fikrini ortaya atarak FSF (Free Software Foundation) diye bilinen birliği kurmuştur. Amacı tamamen serbest ve bedava olacak bir çeşit UNIX sistemi yazmaktı (www.fsf.org www.gnu.org). GNU projesi kapsamında pek çok yardımcı yazılım geliştirilmiştir. Örneğin bu gün Linux sistemlerinde kullanılan gcc derleyicisi ve imacs editörü bu proje kapsamında geliştirilmiştir. Ancak Richard Stallman ve

FSF kurumu yardımcı araçları geliştirdiyse de işletim sisteminin kendisini tamamen geliştirip bitirememiştir. Serbest yazılım (Free Software) bedava yazılımdan ziyade başkasının oluşturduğu bir yazılımı hiç izin almadan geliştirmek anlamındadır. Bu kavramda kişi programını satabilir, ama kaynak kodu üzerinde bir hakka sahip olamaz, yani onu sahiplenemez. Bunun için GNU Public License geliştirilmiştir.

1991 senesine gelindiğinde GNU projesine ilişkin pek çok şey yazılmıştır. Ama işletim sisteminin çekirdeği yazılmamıştır. İşte Linus Torwalds, ismine Linux dediği bir çekirdek yazarak GNU projesini tamamlamıştır. Bugün Linux'un pek çok sürümü vardır, fakat bu sürümlerin hepsi aynı çekirdeği kullanmaktadır. GNU/Linux sistemleri Intel makinelerinin dışında başka donanımlara da aktarılmıştır.

1980 yılların sonlarına doğru UNIX sistemleri kendi aralarında taşınabilir yapılmaya çalışılmıştır. Böylelikle bir UNIX sistemi için yazılmış kaynak kodun başka bir UNIX sisteminde de problemsiz derlenerek çalıştırılabileceği düşünülmüştür. Böyle bir standardizasyon Richard Stallman tarafından önerilmiştir. Standartlaşma ile IEEE ilgilenmiş ve 1003 numaralı standardizasyon ekipleri kurulmuştur. Bu standartlar POSIX (Portable Operating System Interface for UNIX) biçiminde isimlendirilmiştir. Örneğin 1003.1 ya da POSIX.1 UNIX sistemlerindeki sistem fonksiyonlarının neler olması gerektiğini belirler. Diğer alt gruplar başka işlevsel özellikler konusunda belirlemeler oluşturmuştur. Bu gün kullanılan UNIX ve Linux sistemlerinin hemen hepsi burada belirtilen POSIX standartlarına uygundur. POSIX aslında IEEE tarafından yalnızca UNIX sistemlerine özgü bir standart olarak ele alınmamıştır. UNIX gurubu dışındaki işletim sistemleri de isterlerse POSIX standartlarına uyum sağlayabilirler. Örneğin NT gurubu sistemlerin POSIX alt yapısı vardır ve POSIX'i desteklemektedirler.

UNIX Sistemine Giriş ve Temel Kavramlar

Temel olarak UNIX/Linux sistemleri monolitik bir yapıya sahiptir. Monolitik sistemlerde işletim sisteminin çekirdek kısmı büyüktür ve neredeyse tek parça halindedir. Bu tür sistemlerde sisteme ekleme yapmak zordur ve tüm çekirdeğin yeniden derlenmesini gerektirmektedir. Monolitik yapının tersi mikro kernel teknolojisidir. Mikro kernel sistemlerde işletim sisteminin çekirdeği küçük tutulur. İşletim sisteminin pek çok fonksiyonu gerektiğinde sonradan yüklenilebilen modüller halinde tasarlanır. Bu tür sistemlerde işletim sisteminin genişletilmesi başkaları tarafından daha kolay yapılır, ancak tasarımları ayrıntılı ve zordur. Win32 sistemleri tipik olarak mikro kernel sistemi ile tasarlanmıştır.

UNIX sistemleri yüklendiğinde çekirdek tarafından oluşturulmuş bir process login programını çalıştırır. Böylece sisteme girecek kişi bir login ekranıyla karşılaşır. Login ekranı ile karşılaşıldığında işletim sistemi yüklenmiştir ve o anda login programı çalışmaktadır. login programı kullanıcıdan bir kullanıcı ismi ve password ister ve bunun doğruluğunu kontrol eder ve daha önce belirlenen bir programı o kullanıcı için çalıştırır. Sisteme girdikten sonra kullanıcı için çalıştırılacak program genellikle bir komut yorumlayıcı programdır. Böylece sisteme girildikten sonra kullanıcı bir komut yorumlayıcı ile karşılaşılır. UNIX sistemlerinde komut yorumlayıcılar (command interpreter / shell) bir tane değildir. Bir kullanıcı sisteme girdiğinde hangi komut yorumlayıcının çalıştırılacağı ve kullanıcıya ilişkin diğer bilgiler, kullanıcı yaratılırken belirlenir ve korunmuş olan bazı dosyaların içerisinde saklanır. UNIX sistemleri bir süper kullanıcının her türlü işlemi yapabileceği bir sistemdir. Süper kullanıcının

kullanıcı ismi “root” biçimindedir. Sisteme süper kullanıcı olarak giren birisi hiçbir güvenlik engeline takılmadan sistemle ilgili her türlü işlemleri yapabilir. Her türlü dosyaya erişebilir.

UNIX/Linux Sistemlerinde Kullanılan Komut Yorumlayıcı (Shell) Programlar

UNIX/Linux sistemlerinde komut yorumlayıcı programlar işletim sisteminin çekirdek kısmından ayrıdır ve birden fazladır. Login işlemi ile birlikte sisteme girildiğinde hangi komut yorumlayıcı programın (shell) çalıştırılacağı kullanıcı yaratılırken belirtilmektedir. UNIX/Linux sistemlerinde en yaygın olarak kullanılan komut yorumlayıcı programlar şunlardır:

1) C Shell (csh)
2) Bourne Shell (bsh)
3) Bourne Again Shell (bash)
4) Korne Shell (ksh)

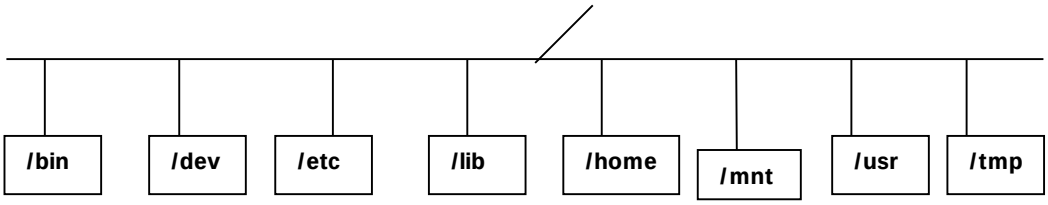
Bu komut yorumlayıcı programlar arasında bazı komut farklılıkları, script dillerinde çeşitli farklılıklar vardır. Bunlardan en yaygın kullanılanı Linux’ta bash’tir. UNIX sistemlerinde C Shell de yaygın olarak kullanılmaktadır. Kuşkusuz komut yorumlayıcılar birer sistem programlarıdır ve yeni komut yorumlayıcılar yazılabilir. Kullanıcı login olduğunda başka kişiler tarafından yazılmış komut yorumlayıcılar ile de çalışabilir. Komut yorumlayıcılar işletim sistemi için sıradan birer process’tir. Shell programından çıkmak için (shell programından çıkıldığında yeniden login programında geri dönülür.) uygun bir shell komutu girmek gerekir. Bu shell komutunun ismi genel olarak logout’tur.

UNIX/Linux sistemlerinde içsel komut kavramı yoktur. Bütün komutlar aslında birer çalışabilen dosyadır. Dolayısı ile her türlü komut yorumlayıcıda bu komutlar çalışmaktadır.

UNIX/Linux Dosya Sistemine İlişkin Temel Dizinler

UNIX/Linux sistemlerinin dizin yapısı DOS’ta gördüğümüz yapıya çok benzerdir. Ancak dizin geçişleri ‘\’ ile değil ‘/’ ile yapılır. Kök dizin tek bir ‘/’ ile temsil edilir. Path sisteminin oluşturulması aynıdır. Ancak UNIX/Linux sistemlerinde sürücü kavramı yoktur. Bu sistemler başka dosya sistemlerini kullanabilmektedir, ama başka dosya sistemleri mount işleminden sonra dizin ağacında bir dizin biçiminde monte edilmektedir. Örneğin içerisinde Windows yüklü olan bir makineye ayrıca Linux kurduğumuzu düşünelim. Linux içerisinde Windows’taki C, D ve E sürücülerine erişmek isteyelim. Bu sürücüler Linux dizin ağacının istenilen bir yerine bir dizin gibi monte edilirler. Sonra biz bu dizinlere geçtiğimizde aslında Windows’taki sürücülerin kök dizinlerine geçmiş oluruz. Başka bir dosya sistemini UNIX/Linux sistemlerine monte etme işlemlerine mount işlemi denir.

Tipik bir UNIX/Linux sistemi kurulduğunda kurulum programları tarafından bir dizin ağacı oluşturulur ve dizinlere uygun programlar yerleştirilir. Tabii bu dizin ağaçları sistemden sisteme değişebilmektedir ve standart değildir. Ancak pek çok sistemde aşağıdaki gibidir:



/bin dizini çalışabilen dosyaları ve sistem komutlarını barındırır.
/dev dizini aygıt dosyaları için ayrılmıştır.
/etc dizini önemli sistem dosyalarını tutar.
/lib dizini kütüphane dosyalarını barındırır.
/home dizini kullanıcılar için default dizinlerin kök dizinidir.
/mnt dizini mount işlemleri için düşünülmüştür.
/usr dizini çeşitli programlar tarafından kullanılan bir dizindir.
/tmp geçici dosyaların yaratıldığı dizin olarak kullanılır.
Tabii kurulum işlemi ile bu ağaç büyümektedir.

UNIX/Linux Sistemlerinin Kurulumu

UNIX/Linux sistemlerinin kurulumu maalesef Windows sistemlerinden daha zordur. Pek çok sistemi kurmadan önce boot ve root disketlerinin oluşturulması gerekir ve kuruluma bu disketlerle başlamak gerekir. Bu disketler Windows ya da DOS ortamında hazırlanabilmektedir. Bu tür sistemlerde makine boot disketi takılarak disketten açılır. Sonra root disketinin takılması istenir ve sonra da bir prompt ile karşılaşılır. Bu aşamada setup programı çalıştırılarak kuruluma başlanır. Ancak UNIX/Linux sistemlerinin çoğunda kurulum işlemleri kolaylaştırılmıştır. Örneğin doğrudan makine CD'den boot edilerek kurulum başlatılabilir. UNIX/Linux sistemleri ayrı bir patition'a kurulmalıdır.

UNIX/Linux sistemlerinin eski versiyonlarının çoğu PC'lere kurulduğunda disk bölümleme tablosunun (partition table) ana dörtlük girişlerinden birini kullanmak ister. Bu sistemler tek başlarına yüklendiğinde genellikle problemsiz bir biçimde kurulur. Ancak genellikle makinede zaten bir Windows sistemi vardır ve bu sistemler ikinci bir işletim sistemi olarak yüklenmek istenir. Bu durumda özellikle extended partition yaratırken, enxtended partition'un diskin geri kalan tüm alanı kullanması yoluna gidilmemelidir. Yani geri kalan ana girişler için yer bırakılmalıdır. Eğer tüm disk extended partition için kullanılmışsa extended partition'ı küçülterek yer açan özel programlar vardır (fibs, partition resizer). Ana girişler için yer açıldıktan sonra yeni bir UNIX/Linux partition'ı yaratılır. Bu partition için UNIX/Linux formatı atıldıktan sonra kurulum problemsiz olarak devam eder.

Ancak bu sistemlerin son versiyonları extended partition'ın logical sürücüleri üzerine de kurulabilmektedir. Yani örneğin diskin tamamını extended partition olarak kullanmış olalım ve extended partition içerisindeki mantıksal sürücüler D, E, F, G olsun. Örneğin G sürücüsü feda edilerek sistem G sürücüsü üzerine de kurulabilir.

İşletim sisteminin kurulmasından sonra en önemli problem makine açıldığında sistemin istenilen işletim sistemi ile boot edilebilmesidir. Yani makineyi açtığımızda bir yazı çıkacak, biz de istediğimiz işletim sistemi ile çalışmaya başlayabileceğiz. Bu işlem için Linux sistemlerindeki Lilo programı kullanılabilir ya da özel boot yöneticisi programlarla bu işlem yapılabilir. Lilo programı zaten kurulumun son aşamasında bu işlemi yapmak üzere devreye

girer. Lilo mbr sektörüne ya da Linux'un boot sektörüne yerleştirilebilir. Lilo mbr'ye kurulduğunda maalesef pek çok durumda buradan silinebilmektedir. Bunun için UNIX/Linux sisteminin bir açılış disketinin saklanması gerekir.

Virtual PC ve WMware Programları

Virtual PC ve WMware programları NT gurubu ve 98 gurubu işletim sistemlerinde çalışan ve sanki ayrı bir makinede çalışıyormuş izlenimi veren programlardır.

Bu programlar bir dosyayı sanal bir makinenin hard disk'i olarak görürler ve tamamen orijinal kodları Windows sistemlerinin çok işlemliliği içerisinde çalıştırırlar. Bu programlar kullanılarak UNIX/Linux sistemleri sanki ayrı bir makineye kuruluyormuş gibi kurulabilir. Bu sistemlerde sanal makinenin harddisk'inde yapılan her şey gerçek makinede bir dosyayı etkilemektedir. Biraz yavaş çalışmalarına karşın deneysel faaliyetler için çok iyi bir ortam oluştururlar.

Virtual PC programında fare o anda çalışmakta olan işletim sistemi penceresinin içerisine click yapılırsa sanki o sistemin faresiymiş gibi ele geçirilir. Fareyi tekrar ele geçirmek için aktif tuşa basmak gerekir. Aktif tuş virtual PC kurulduğunda sağ Alt tuşudur. O andaki makinenin ekranını tüm ekranı kaplar hale getirmek için aktif tuş + Enter yapmak gerekir.

Temel UNIX/Linux Komutları

UNIX/Linux sistemlerinde içsel komut kavramı yoktur, bütün komutlar aslında birer programdır ve temel komutların POSIX standartlarında belirlenmiştir. Bu sistemlerde bir komutun genel yapısı şöyledir:

```
<komut ismi> [seçenekler] [parametreler]
```

Seçenekler genellikle -x formatındadır. x burada bir ya da birden fazla karakteri temsil eder. Bir komut birden fazla seçenek alabilir. Örneğin:

```
-x -y -z
```

Bu tür seçenekler tek bir - içerisinde de yazılabilirler. Örneğin:

```
-xyz
```

ls Komutu

Bu komut Dos'taki dir komutuna karşılık gelir. Geniş bir komuttur ve pek çok seçeneği vardır. ls -l ile dosyalara ilişkin ayrıntılı bilgiler görüntülenebilir.

cat Komutu

Bu komutu Dos'taki type komutu gibidir.

pwd Komutu

Bu komut o anda çalışılan dizinin hangisi olduğunu görüntüler.

rm ve cp Komutları

Bu komutlar DOS'daki del ve copy komutları gibidir.

more Komutu

Sayfa sayfa type etmekte kullanılır.

mkdir ve rmdir Komutları

Dizin yaratıp dizin silmek için kullanılır.

cd Komutu

Dizin değiştirir.

UNIX/Linux Sistemlerinin Dosya Sistemine İlişkin Temel Bilgiler

POSIX dosya sisteminde dosya isimlerinin büyük/küçük harf duyarlılığı vardır. Örneğin, ankara isimli dosyayla Ankara isimli dosya aynı anda bulunabilir ve birbirlerinden farklıdır. Bu dosya sistemi de FAT sisteminde olduğu gibi bir kök dizin ve iç içe geçmiş dizinler topluluğundan oluşur.

POSIX sistemlerinde orta karmaşıklıkta bir koruma mekanizması uygulanmıştır, yani dosyalar ve dizinler bunları yaratanların isteğiyle diğer kullanıcılardan gizlenebilmektedir. Her dosya ve dizine erişim üç durum dikkate alınarak değerlendirilmektedir:

1) Erişimi dosyayı yaratan kişi mi yapmaya çalışmaktadır?
2) Erişim dosyayı yaratan kullanıcının dahil olduğu gruptan herhangi bir kullanıcı tarafından mı yapılmak istenmektedir? Çeşitli kullanıcılar bir grup altında toplanabilir. Yani erişimde dosyayı yaratan kişi olmanın yanı sıra yaratan kişi ile aynı grupta olmanın da önemi vardır.
3) Erişimi ne dosyayı yaratan kişi ne de dosyayı yaratan kişi ile aynı grupta olan bir kişi mi yapmaya çalışmaktadır, yani herhangi bir kullanıcı mı yapmaya çalışmaktadır?

İşte bir dosyaya ya da dizine erişimde bu ölçütlere dikkat edilmektedir. POSIX sisteminde her dosyanın onu yaratan kişi (owner), yaratan kişinin grubu ve herhangi bir kişi için erişim hakları vardır. Yani her dosya için o dosya üzerinde kimin hangi işlemleri yapacağı belirlenmiştir.

Bir dosya üzerinde üç erişim işlemi yapılabilir: okuma, yazma ve çalıştırma. Bir dosyanın erişim bilgilerini görmenin en kolay yolu ls -l komutunu uygulamaktır. Bu komut uygulandığında dosyanın tipik erişim bilgisi aşağıdaki gibi olur:

```
-rw-rw-rw-
```

Bu erişim bilgisindeki en soldaki karakter dosyanın türünü belirtir. Örneğin bu karakter - ise dosya sıradan bir dosyadır, d ise bir dizin, p ise bir pipe vs... Sonraki karakterler üçerli

gruplara ayrılır. İlk grup dosyayı yaratan kişi için, sonraki grup dosyayı yaratan kişinin grubu için, sonraki grup ise herhangi bir kişi için erişim haklarını belirtmektedir. Örneğin,

```
d     rwx         rwx         rwx
      sahibi      grup        herhangi birisi
```

Örneğin bir dosyanın erişim bilgileri şöyle olsun:

```
-rw-r--r--
-,rw-,r--,r--
```

Dosyayı yaratan kişi buradaki dosya üzerinde hem okuma hem yazma yapabilir, ancak dosyayı yaratan kişi ile aynı gruptaki kişiler ve diğer kişiler yalnızca okuma yapabilirler.

Bir dosyanın okunabilmesi demek dosya fonksiyonlarıyla okuma işlemlerinin yapılabilmesi demektir. Bir dosyaya dosya fonksiyonlarıyla yazma, ekleme gibi işlemlerin yapılabilmesi için yazma hakkının olabilmesi gerekir. POSIX sistemlerinde çalıştırılabilen dosyalar uzantılarıyla tespit edilmezler, erişim haklarıyla tespit edilirler. Yani dosyanın x erişim özelliği varsa bu dosya çalıştırılabilir bir dosyadır. İsmi yazıldığında işletim sistemi tarafından yüklenir. Bir dizin için okuma hakkının olması o dizin için dizin listesinin alınabilmesi anlamındadır. Yazma hakkının anlamı o dizinde yeni bir dosya yaratma ya da olan bir dosyayı silme anlamındadır. Çünkü aslında dizinler de birer dosya gibidir. Dizinler için çalıştırılabilme hakkı tamamen farklı bir anlama gelir, bu hak bir path ifadesinde bir dizinin içerisinden başka bir dizine atlanabileceğini belirtir. Örneğin, bir dosya açacak olalım ve dosyanın path bilgisi şöyle olsun:

```
/a/b/c.dat
```

Burada biz c.dat dosyasının içeriğini yalnızca okumak isteyelim, yani fopen() fonksiyonu ile c.dat dosyasını "r+" modunda açmak isteyelim. Burada a dizininin içerisinden geçilmektedir. Yani a'nın çalıştırılabilir hakkına sahip olması gerekir. c.dat dosyasının okuma ve yazma haklarına sahip olması gerekir. b'nin okuma ve çalıştırılabilir hakkına sahip olması gerekir, ancak yazma hakkına sahip olmasına gerek yoktur. Dizinler için böyle bir mekanizma kurulmasının anlamı şudur: Bir kişi bir dizini korumak isteyebilir ama o dizinin altındaki başka bir dizini korumak istemeyebilir, bu durumda korumak istediği dizin için okuma ve yazma hakkı vermez onun alt dizini için okuma ve yazma hakkı verir, ancak üst dizin için geçiş hakkını vermesi gerekir.

Aşağıdaki path ile verilen fonksiyonu "w+" modunda açmak isteyelim.

```
/a/b/c/d.dat
```

Bu durumda a, b, c dizinleri için x hakkının olması gerekir, ancak a ve b için r ve w haklarının olmasına gerek yoktur. d.dat dosyasının okuma ve yazma hakkına sahip olması gerekir. Burada c dizininin hangi haklara sahip olması gerekir? Bir dizin için yazma hakkı dizin içerisinde yeni bir giriş oluşturmak anlamındadır. Yani yukarıdaki örnekte d.dat dosyası daha önce varsa, yani biz truncate işlemi yapıyorsak, c dizininin yazma hakkına sahip olmasına gerek yoktur. Ancak d.dat dosyası yoksa dizin içerisinde yeni bir giriş oluşturulacaktır, bu durumda c'nin yazma hakkına sahip olması gerekir.

Kullanıcı ID Değeri

Bir kullanıcı yaratıldığında bir kullanıcı ismi (user name), bir password ve kullanıcı ismine karşılık gelen bir kullanıcı ID değeri (user ID) belirlenmektedir. Örneğin kullanıcı ismi "kaan" ve ID'si "500" olabilir. Kullanıcı ID değeri 0 - 65535 arasında sabit bir değerdir. Her ne kadar ID değerinin sistem genelinde tek olması zorunlu olmasa da kullanım bakımından sistem genelinde tek olmalıdır. Bir kullanıcı yaratıldığında kullanıcı ismi, ID değeri ve password bilgisi /etc/passwd içine text dosyası olarak yazılır. /etc dizini korunmuş bir dizin değildir, ancak passwd dosyası kurulum programı tarafından kurulum sırasında root kullanıcısı tarafından yaratılmış bir dosya biçiminde oluşturulur. passwd dosyasının herhangi bir kullanıcı için sadece okuma hakkı vardır. Yani passwd dosyası bir editörle incelenebilir, ama içeriği normal bir kullanıcı tarafından değiştirilemez.

Bazı sistemlerde password bilgisi şifrelenmiş olarak passwd dosyası içerisinde yazmaktadır. Ancak bazı sistemlerde güvenliği arttırmak için password bilgisi okuma hakkı da olmayan /etc/shadow dosyasında saklanır. passwd ve shadow dosyalarının içeriği ileride ele alınacaktır. passwd ve shadow dosyaları login işlemi sırasında ve daha pek çok işlem sırasında login programları tarafından okunmaktadır. Örneğin passwd dosyasında bir kullanıcı için oluşturulan satır root olarak girilip silinirse kullanıcı sisteme giriş yapamaz.

root kullanıcısının ID değeri 0'dır. Pek çok sistemde 1 - 100 arasındaki değerler reserve edilmiştir. Kullanıcının ID değeri getuid fonksiyonuyla elde edilebilir ve çeşitli test işlemlerinde kullanılabilir.

Grupların ID Değerleri

Kullanıcılarda olduğu gibi her grup için de bir grup ismi ve ID değeri vardır. Bir grup yaratıldığında grubun ismi, ID değeri ve grup içerisindeki kullanıcıların kimler olduğu /etc/group dosyasında belirtilmektedir. group dosyası da passwd dosyasında olduğu gibi root tarafından yaratılmıştır ve diğer kişiler için yalnızca okuma hakkı verilmiştir. Bir kullanıcının grubuna ilişkin ID değeri getgid fonksiyonuyla elde edilebilir.

UNIX/Linux Process Yapısının Temelleri

UNIX/Linux sistemlerinin process yapısı tamamen hiyerarşik bir yapı göstermektedir. Diğer işletim sistemlerinde olduğu gibi POSIX sistemlerinde de bir process başka bir programı çalıştırabilir yani yeni bir process yaratabilir. Yeni çalıştırılan process (child process) ile onu çalıştıran process (parent process) arasındaki ilişki Windows sistemlerine göre sıkıdır. Klasik olarak POSIX sistemlerinde her process'in bir ID değeri vardır. Bu ID değeri sistemde tektir. POSIX sistemleri yüklendiğinde temel yükleme işlemleri için 0 ID numarasına sahip bir process yaratılır. Bu process init process'ini yaratmaktadır. init process'in ID değeri 1'dir. Sisteme girmek için kullanılan login de bir programdır, yani bir process olarak çalıştırılır. Login tipik olarak init process'inin bir alt process'dir (child process). Kullanıcı username ve password bilgilerini başarılı bir biçimde girdiyse /etc/passwd dosyasında belirtilen shell programı çalıştırılır.

Yani shell process'i tipik olarak login process'inin alt process'i olarak çalışır. Bu durumda tipik bir POSIX sisteminde sisteme login olunduğunda process hiyerarşisi şöyle olacaktır. Bu noktada artık shell üzerinden bir program çalıştırırsak çalışan process shell process'inin bir alt

process'i olarak çalıştırılacaktır. UNIX/Linux sistemlerinde bir process'in yeni bir process'i çalıştırması fork ve exec fonksiyonları ile yapılmaktadır. Bu sistemlerde yaratılan alt process'ler üst process'in pek çok bilgisini doğrudan almaktadır. UNIX/Linux sistemlerinin process yönetimi ileride ayrıca ele alınacaktır.

Process'lerin Erişimlerde Kullandıkları ID Değerleri

Her process'in erişim sisteminde kullandığı beş tür ID değeri vardır.

1) Gerçek kullanıcı ID değeri (real user ID)
2) Gerçek grup ID değeri (real group ID)
3) Etkin kullanıcı ID değeri (effective user ID)
4) Etkin Grup ID değeri (effective group ID)
5) Ek grup ID değerleri (supplementory group ID)

Bu ID değerleri process fork ve exec fonksiyonları ile yaratılırken doğrudan üst process'in process tablosundan alınmaktadır. Bu ID değerleri yetkin process'ler tarafından değiştirilebilmektedir.

Sisteme girildiğinde shell process'i normal olarak login process'i tarafından yaratılmıştır. login process'i username ve password kontrolünü yapıp shell process'ini çalıştırdıktan sonra, bu process'in gerçek kullanıcı ID değerini ve gerçek grup ID değerini passwd dosyasında belirtilen değerler ile oluşturur. Sonuç olarak shell process'inin gerçek kullanıcı ID değeri ve gerçek grup ID değeri kullanıcı yaratılırken belirlenmiş olan ID değerleri olur. Biz shell üzerinde kendi yazdığımız bir programı çalıştırırken aslında shell process'i komut satırında belirtilen programı yine fork ve exec fonksiyonları ile çalıştırır. Bu durumda çalıştırılan programın gerçek kullanıcı ve grup ID değerleri de shell process'inin aynısı olacaktır.

POSIX sistemlerinde erişim kontrolleri gerçek kullanıcı ve grup ID değerleri ile değil etkin kullanıcı ve grup ID değerleri ile yapılır. Normal olarak çalıştırılan process'in etkin kullanıcı ve grup ID değerleri gerçek kullanıcı ve grup ID değerleri ile aynıdır. İşte çalıştırılabilen programların kullanıcı ve grup için iki bayrağı vardır. Eğer bu bayrakları set edilmiş olan bir dosyadan fork ve exec işlemleri ile bir process yaratılmış ise yaratılan process'in etkin kullanıcı ve grup ID değerleri o çalıştırılabilen dosyanın sahibinin ve grubunun ID değeri olur (bu bayrakların set edilmiş olup olmadığı ls –l komutunda x yerine s harfinin görülmesinden anlaşılır). Örneğin diskte passwd isimli bir program olsun. Programın erişim hakları şöyle olsun:

```
-rwxr-xr-x  root  shadow      /usr/bin/passwd
```

şimdi eğer bu dosyanın kullanıcı ya da grup bayrakları set edilmemiş ise bu program çalıştırıldığında yaratılan process'in etkin kullanıcı ve grup ID değerleri bizimkinin aynısı olacaktır ve bu programın içerisinde bazı öncelikli erişimler yapılıyor ise bu erişimler başarısız olacaktır. İşte şimdi eğer bu dosyanın kullanıcı ID bayrağı set edilmiş olsa bu process'in etkin kullanıcı ID değeri dosyanın sahibinin kullanıcı ID değeri yani root olacaktır. Erişim kontrolüne etkin ID değerleri katıldığı için erişimler gerçekleşecektir. Görüldüğü gibi çalıştırılabilen dosyaların bu özel bayrakları sanki bu dosyaları dosyanın sahibi gibi çalıştırılmasını sağlamaktadır. Bu bayrakların set edilebilmesi için erişim hakkına sahip olmak gerekir. Bu bayrakların set edilmesi chmod komutu ile yapılmaktadır.

Özetle biz root olmasak bile root istedikten sonra root tarafından oluşturulmuş programı root erişim hakkı ile çalıştırabiliriz.

UNIX/Linux sistemlerinde normal olarak bir kullanıcı tek bir gruba ilişkin olabilir. Yani process'in gerçek grup ID'si bir tanedir. Ancak bir kullanıcının birden fazla gruba üye olması faydalı bir durum oluşturduğundan "ek grup ID" kavramı ile bir kullanıcı birden fazla gruba aitmiş gibi de gösterilebilmektedir. Her grubun hangi kullanıcılardan oluşturulduğu /etc/group dosyasında belirtilmiştir. Bu dosyaya bakıldığında bir kullanıcının farklı gruplarda bulunabileceği görülmektedir. Ancak kullanıcının gerçek grubu /etc/passwd dosyasında belirtilen gruptur. Bir shell process'i yaratıldığı zaman process'in ek grup ID'leri /etc/group dosyasından alınarak oluşturulmaktadır. Özetle bir kullanıcının tek bir grup ID'si vardır o da /etc/passwd dosyasında belirtilendir. Ancak kullanıcı birden fazla gruba ilişkin olabilir o da /etc/group dosyasında belirtilmektedir. Ek grup ID'leri de erişimde kontrol işlemine sokulmaktadır.

Erişim Kontrolleri

Edinilen bilgilerden sonra POSIX sistemlerinin uyguladığı erişim kontrolleri sırası ile ele alınabilir. Örneğin bir process'in bir dosyaya erişmek istediğini düşünelim. Erişim algoritmasında gerçek kullanıcı ve grup ID'leri değil, etkin kullanıcı ve grup ID'leri ile ek grup ID'leri test işlemine sokulmaktadır.

1) Erişimi gerçekleştirmek isteyen process'in etkin kullanıcı ID değeri 0 (root) ise erişim gerçekleştirilir.
2) Dosyanın sahibinin etkin kullanıcı ID değeri ile belirtilen kullanıcı olup olmadığına bakılır. Eğer dosyanın sahibi process'in etkin kullanıcı ID'siyle belirtilen kullanıcı ise, bu kez dosya sahipliğinin rwx alanlarına bakılarak erişim hakkının olup olmadığı test edilir. Erişim hakkı varsa erişim gerçekleşir, yoksa erişim reddedilir ve bir sonraki aşamaya geçilmez.
3) Process'in etkin grup ID'si ile dosyanın ilişkin olduğu grup karşılaştırılır. Eğer dosya process'in etkin grup ID'si ile belirtilen gruba ilişkin ise dosyanın grup erişim haklarına bakılır. Bu haklar uygunsa erişim gerçekleştirilir, değilse erişim reddedilir ve sonraki aşamaya geçilmez.
4) Process'in ek grup ID'lerinden herhangi birisi dosyanın ilişkin olduğu gruptan mı diye bakılır. Eğer gruptan ise dosyanın grup erişim hakları incelenir. Erişim hakları uygun ise erişim gerçekleştirilir. Uygun değilse erişim reddedilir ve sonraki aşamaya geçilmez.
5) Bu aşamada artık process herhangi biri durumuna gelmiştir. Bu nedenle dosyanın herhangi birine ilişkin erişim haklarına bakılır. Eğer erişim hakkı uygunsa erişim gerçekleştirilir, değilse erişim reddedilir.

Örneğin aşağıda a isimli çalışabilen dosyanın ve b data dosyasının bilgileri verilmiştir.

```
-r-sr-xr-x  Ali   Project1    a
-r--rw-r--  Veli  Project1    b
```

a programının b data dosyasına bir şeyler yazmak istediğini düşünelim. Grubu Project3 olan Mehmet isimli kullanıcı login olarak a programını çalıştırabilir mi? A programı b dosyasına yazma yapabilir mi?

1) Mehmet kullanıcısı herhangi bir kişi olarak a dosyasına erişip onu çalıştırabilir.

2) A programı çalışıp process olduğunda a process'inin etkin kullanıcı ID değeri Ali etkin grup ID değeri Project3'tür. Çünkü a çalışabilen dosyasının yalnızca user ID değeri set edilmiştir.
3) Bundan sonra a process'inin etkin kullanıcı ID'si b dosyasının ID'si ile uyuşmadığından, etkin grup ID'si b dosyasının ID'si ile uyuşmadığından ve b dosyasına herhangi bir kişi yazma işlemi yapamayacağından, sonuç olarak Mehmet kullanıcı a programını çalıştırabilir, ama a programı b dosyasına yazma yapamaz.

Eğer soruda a çalışabilen dosyasının grup ID bayrağı set edilmiş olsaydı, a process'inin etkin grup ID değeri Project1 olacağından yazma işlemi yapılabilecekti.

Dosya ve Dizinlerin Sahiplik Bilgilerinin Oluşumu

Dosya yaratmak için fopen standart C fonksiyonu ya da open sistem fonksiyonu kullanılabilir. Yaratılacak dosyanın sahibi, grubu ya da sıradan kişi için erişim hakları dosya yaratılırken open fonksiyonunun parametresinde belirlenir. Ancak yaratılmış olan dosyanın kullanıcı ID'si ve grup ID'si bu fonksiyonlarda belirlenmez. Bu belirleme otomatik olarak yapılır. Yaratılmış her dosyanın bir kullanıcı ID'si ve grup ID'si vardır. Yaratılan dosyanın kullanıcı ID'si onu yaratan process'in etkin kullanıcı ID'si olarak belirlenir. Ancak yeni yaratılan dosyanın grup ID değeri konusunda UNIX versiyonları arasında farklar vardır. Örneğin AT&T SVR4 modellerinde yeni yaratılan dosyanın grup ID'si dosya yaratan process'in etkin grup ID'si olarak belirlenmektedir. Ancak BSD sistemlerinde yeni yaratılan dosyanın grup ID'si içinde yaratılma işleminin yapıldığı dizinin grup ID'si olarak belirlenmektedir. Ancak BSD sistemlerinde daha sonra bu durum dizinin grup ID bayrağının set edilip edilmemesine göre tespit edilir. POSIX.1 de bu durum isteğe bağlı bir biçimde işletim sistemini yazanlara bırakılmıştır (Linux sistemlerinde default olarak dosyayı yaratan process'in grup ID değeri kullanılmıştır).

UNIX/Linux Sistemlerinde Dosya Sistemine İlişkin İşlemler

POSIX sistemlerinde dosya işlemleri üç fonksiyon grubu ile yapılabilir:

1) Standart C fonksiyonları ile yapılabilir. Standart C fonksiyonlarına tamponlanmış fonksiyonlar (buffered I/O functions) da denilmektedir. Bilindiği gibi standart C fonksiyonları doğrudan işletim sisteminin aşağı seviyeli dosya fonksiyonlarını çağırarak işlemlerini yapar. Ancak bir tamponlama mekanizması kullanırlar. Örneğin standart C fonksiyonları ile bir okuma yapıldığında önce okunan bilgi tamponda mı diye bakılır, tamponda ise alınır, tamponda değilse tazeleme yapılır ve ondan sonra yine aynı tampondan alınır. Böylelikle sistemin daha etkin çalışacağı düşünülmüştür. Tabii standart C fonksiyonları işletim sistemine özgü ayrıntılı dosya işlemleri için yetersizdir.
2) Standart POSIX dosya fonksiyonları ile POSIX.1 de tanımlanmış olan open, close, read, write gibi aşağı seviyeli fonksiyonlar vardır. Bu fonksiyonların yetenekleri daha fazladır. POSIX fonksiyonları bazı sistemlerde doğrudan sistem fonksiyonu konumunda olabilir ya da bu fonksiyonlar da gerçek sistem fonksiyonlarını çağıran arabirim fonksiyonlar olabilir.
3) Gerçek sistem fonksiyonlarının çağırılması ile yapılabilir. İşletim sisteminin gerçek sistem fonksiyonları standart değildir ve pek çok sistemde tasarım tekniği farklı olabilir.

UNIX/Linux programlamada eğer hızlı, ayrıntısı olmayan, fakat çok taşınabilir kodlar yazılmak istenirse o zaman doğrudan standart C fonksiyonlarının kullanılması en iyi seçenektir. Ancak POSIX sistemlerine özgü işlemler yapılacaksa o zaman POSIX fonksiyonları tercih edilmelidir. Doğrudan sistem fonksiyonlarının kullanılması için çok az gerekçe vardır. Standart C kütüphanesi aynı zamanda POSIX standartlarında da geçerli bir kütüphanedir.

Temel dosya işlemlerinin dışında POSIX sistemlerinde dosya sistemine ilişkin bir takım faydalı işlemler için standart POSIX fonksiyonları da vardır. Örneğin dizin dosya silme, dizin değiştirme, dosyanın ismini değiştirme gibi işlemler için standart POSIX fonksiyonları kullanılır.

Dosya Sistemi ile İlgili İşlem Yapan Yardımcı POSIX Fonksiyonları

Bu fonksiyonların bir kaç istisnası dışında hepsinin geri dönüş değeri int türündendir ve başarıyı anlatmaktadır. Fonksiyonlar başarılı olduğu zaman 0 değerine başarısız olduğu zaman –1 değerine geri dönerler. Başarısızlık durumunda kütüphanede bulunan errno isimli global değişken set edilir. errno içerisindeki değer başarısızlığın nedenini belirtmektedir. Bu fonksiyonların çoğunun prototipleri unistd.h dosyası içerisindedir. Fonksiyonların kullandığı çeşitli tür isimleri sys/types.h dosyasında bildirilmiştir. Ancak her fonksiyonun gereksinim duyduğu başlık dosyaları ayrıca belirtilecektir.

errno Değişkeni, strerror ve perror Fonksiyonları

Standart C fonksiyonlarının bazıları ve POSIX fonksiyonlarının çoğu başarısızlık durumunda –1 değerine geri dönerler. Ancak bu fonksiyonlar başarısızlığın nedenini geri dönmeden önce kütüphanedeki errno isimli global bir değişkene yazarlar. errno errno.h dosyası içerisinde extern olarak bildirilmiştir. Bu değişkenin global tanımlaması kütüphane içerisinde yapılmıştır. errno int türden bir değişkendir. Hata oluştuktan sonra errno içerisindeki her bir değer faklı bir hata durumunu anlatmaktadır. Bu sayısal değerler aynı zamanda EXXX biçiminde sembolik sabitler olarak errno.h içerisinde define edilmiştir. Böylece errno değişkeninin karşılaştırılmasında sembolik sabitler kullanılır.

```
if (errno == ENOEND) {
      //...
      //...
}
```

Bir hata oluştuğunda hata olasılıklarının ne olduğu POSIX içerisinde dokümante edilmiştir. Örneğin bir dosyanın açılamadığını düşünelim; dosyanın açılamamasının belirli sayıda nedeni vardır. Ancak programcılar oluşan hata için bir mesaj metni yazdırmak isterler ve bu da yardımcı fonksiyonlar kullanılmadan zor yapılacak bir işlem değildir. strerror fonksiyonu hata numarasını, yani errno değişkeni içerisindeki sayıyı parametre olarak alır ve mesaj metnine geri döner. Prototipi string.h dosyası içerisindedir.

```
char *strerror(int errno);
```

Fonksiyon static yerel bir dizinin adresi ile geri dönmektedir. Örneğin fonksiyon şöyle kullanılabilir:

```
if (open(...) == -1) {
      pMessage = strerror(errno);
      puts(pMessage);
      exit(1);
}
```

perror fonksiyonu doğrudan errno global değişkenine bakarak mesaj metnini stdout dosyasına yazdıran kullanışlı bir fonksiyondur. Prototipi stdio.h dosyası içerisindedir.

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

Fonksiyon önce parametresi ile belirtilen yazıyı yazar, sonra : karakterini yerleştirir, sonra da errno değişkenine karşılık gelen mesaj metni yazdırır, en sonunda '\n' ile aşağı satıra geçiş yapılır. Örneğin;

```
if (open(...) == -1) {
      perror("Error");
      exit(1);
}
```

unlink ve remove Fonksiyonları

Bu fonksiyonlar dosya silmek amacı ile kullanılır ve ikisi tamamen eşdeğerdir. remove fonksiyonunun prototipi stdio.h içerisinde unlink fonksiyonunun unistd.h içerisindedir.

```
int unlink(const char *path);
int remove(const char *path);
```

Bu fonksiyonlar dosyanın bağlantı sayısını bir azaltır. Eğer bağlantı sayısı 0'a düşmüşse dosyayı fiziksel olarak silerler. Dosya bağlantıları ileride ele alınacaktır. Fonksiyonlar başarılı ise 0'a, başarısız ise –1 değerine geri dönerler. Bu fonksiyonlar ile dizinler silinemez. remove fonksiyonu Dos ve Windows sistemlerinde aynı biçimde vardır. Aynı zamanda standart C fonksiyonudur.

rename Fonksiyonu

Bu fonksiyon dosyanın ismini değiştirmek için kullanılır. Aynı zamanda standart C fonksiyonudur. Prototipi stdio.h içerisindedir.

```
int rename(const char *oldname, const char *newname);
```

Fonksiyon başarısızlık durumunda –1 değerine başarılı olma durumunda 0 değerine geri dönerler.

chdir Fonksiyonu

Bu fonksiyon o andaki geçerli dizini değiştirmekte kullanılır. Geçerli dizin (current working directory) göreli path verildiğinde göreli path ifadesinin nereden başlatılacağını belirlemektedir. Fonksiyonun prototipi unistd.h içerisindedir.

```
int chdir(const char *path);
```

Fonksiyon başarısızlık durumunda –1, başarı durumunda 0 değerine geri döner.

getcwd Fonksiyonu

Bu fonksiyon o anda çalışılan dizini path ifadesi olarak elde etmekte kullanılır. Prototipi unistd.h içerisindedir.

```
char *getcwd(char *path, size_t size);
```

Fonksiyonun birinci parametresi path bilgisinin doldurulacağı dizinin adresi, ikinci parametresi dizinin uzunluğudur (Yani fonksiyon en fazla size-1 karakter yerleştirir). Fonksiyon başarı durumunda birinci parametresinde belirtilen adrese, başarısızlık durumunda NULL değerine geri döner.

```
/* getcwd.c */

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

int main()
{
      char path[256];

      if (getcwd(path, 256) == NULL) {
            fprintf(stderr, "Cannot Get Path\n");
            exit(1);
      }

      printf("%s\n", path);

      return 0;
}
```

stat ve fstat Fonksiyonları

Bu fonksiyonlar ismi ya da handle değeri bilinen bir dosyanın bilgilerini elde etmek için kullanılırlar. Bu fonksiyonların prototipleri sys/stat.h dosyası içerisindedir. Ancak bu fonksiyonları kullanırken çeşitli typedef isimlerine de gereksinim duyulur. Bu tür isimleri sys/types.h içerisinde belirtilmiştir. Bu nedenle bu fonksiyonları kullanabilmek için bu iki dosya include edilmelidir.

```
int stat(const char *path, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

Bu iki fonksiyon da tamamen aynı işlemleri yapar. stat fonksiyonu dosyanın path bilgisi ile fstat handle değeri ile çalışır. Yani fstat fonksiyonunu kullanabilmek için önce dosyanın aşağı seviyeli open fonksiyonu ile açılmış olması gerekir. Fonksiyonlar başarı durumunda 0 değerine, başarısızlık durumunda –1 değerine geri dönerler ve errno set edilir. stat isimli yapı

sys/stat.h içerisinde bildirilmiştir. Fonksiyonların ikinci parametresi bu türden bir yapı nesnesinin adresini alır.

Bilgisi elde edilecek dosya bilgiyi elde etmek isteyen process'e erişim bakımından kapalı olsa bile yine dosya bilgileri elde edilir. Yani dosya r ve w özellikleri ile erişime kapalı olsa bile yine de bilgileri elde edilir. Ancak dosya hangi dizinin içerisindeyse o dosyaya giden path ifadesindeki dizinlerin erişecek process için x haklarının olması gerekir. stat yapısının elemanları şu anlamları içerir:

`1) dev_t st_dev:`
dev_t türü genellikle unsigned short int biçimindedir ve dosya bir aygıt dosyası ise onun numarasını belirtir.
`2) ino_t st_ino:`
ino_t unsigned long int türündendir ve dosyanın inode numarasını barındırır.
`3) mode_t st_mode:`
Bu dosyanın bildiğimiz rwx erişim haklarıdır. mode_t unsigned short türündendir. Bu eleman aynı zamanda ilgili dosyanın türünü belirlemek için de kullanılır.
`4) nlink_t st_nlink:`
nlink_t genellikle unsigned short türündendir. Burada dosyanın toplam link sayısı vardır.
`5) uid_t st_uid:`
uid_t unsigned short türündendir. Dosyanın kullanıcı ID'sini belirtir.
`6) gid_t st_gid:`
Dosyanın erişim testinde kullanılan grup ID değerini verir. gid_t unsigned short türündendir.
`7) dev_t st_rdev:`
Özel dosyalar için aygıt numarasını verir.
`8) off_t st_size:`
off_t long türündendir. Bu eleman dosyanın byte uzunluğunu verir.
`9) long st_blksize:`
Bu eleman dosya için tahsis edilecek en iyi blok büyüklüğünü belirtir.
`10)  long st_blocks:`
Bu eleman dosya için tahsis edilen toplam blok sayısını içerir.
`11)  time_t st_atime, st_mtime, st_ctime:`
POSIX dosya sisteminde dosya zamanına ilişkin burada belirtilen üç bilgi ayrı ayrı tutulur. Sırası ile bu zamanlar en son erişim zamanı, en son değiştirme zamanı ve dosya durumunun en son değiştirilme zamanıdır (FAT sisteminde yalnızca en son değiştirme zamanı tutulmaktadır).

```
/* mcopy –t /home/student/stat.c a: */

/* stat.c */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
      struct stat status;

      if (argc != 2) {
            fprintf(stderr, "Wrong number of argument");
            exit(1);
      }

      if (stat(argv[1], &status) == -1) {
```

```
            perror("stat error");
            exit(1);
      }

      printf("%ld \n", status.st_size);
      printf("user ID: %d \n", status.st_uid);
      printf("group ID: %d \n", status.st_gid);

      return 0;
}
```

Dosya Türünün Belirlenmesi

Genel olarak POSIX sistemlerinde bir dosya normal sıradan bir dosya olabilir, bir dizin dosyası olabilir, özel karakter dosyası olabilir, özel blok dosyası olabilir, socket ya da FIFO dosyalar olabilir. Dosyanın erişim bilgileri ve türü stat yapısının unsigned short türden st_mode elemanına bit olarak kodlanmıştır. Ancak bu bit kodlama işlemi sistemden sisteme değiştiği için taşınabilirliği sağlamak amacı ile özel makrolar kullanılmalıdır. st_mode elemanı bu makroya sokulur, makrodan sıfır değeri elde edilirse dosya ilgili türden değil, sıfır dışı bir değer elde edilirse dosya ilgili türdendir. Kullanılan makrolar şunlardır:

```
S_ISREG
S_ISDIR
S_ISCHR
S_ISBLK
S_ISFIFO
```

Örneğin bir dosyanın dizin olup olmadığı aşağıdaki işlemle anlaşılabilir.

```
if (S_ISDIR(status.st_mode)) {
      //...
      //...
}
```

Dosyanın erişim hakları için de & işlemine sokulabilecek aşağıdaki sembolik sabitler tanımlanmıştır:

```
S_IRUSR
S_IWUSR
S_IXUSR
S_IRGRP
S_IWGRP
S_IXGRP
S_IROTH
S_IWOTH
S_IXOTH
```

st_mode değeri buradaki sembolik sabitlerle & işlemine sokularak ilgili özelliğin olup olmadığı anlaşılır. Örneğin:

```
if (status.st_mode & S_IWUSR) {
      //...
}
```

Burada dosyanın sahibine yazma hakkı verilip verilmediği test edilmek istenmiştir.

DOS ve Win32 Sistem Programcıları için Not: stat fonksiyonu DOS'taki findfirst, Win32'deki FindFirstFile fonksiyonlarına işlevsel olarak benzemektedir. Bilindiği gibi bu fonksiyonlar dosya bilgilerinin dizin girişlerinden alıp ffblk ya da WIN32_FIND_DATA türünden yapılar içerisine doldururlar.

UNIX/Linux Sistemlerinin Disk Organizasyonunun Temelleri

AT&T tabanlı UNIX işletim sistemlerinin dosya sistemi s5fs (System 5 File System) ismiyle anılır. Berkeley (BSD) sistemleri klasik UNIX dosya sistemine bazı yenilikler getirmiştir. Berkeley tabanlı bu dosya sistemine FFS/UFS (Fast File System / UNIX File System) denilmektedir. Linux sistemlerinde çeşitli eklentiler yapılmıştır. Şu anda Linux sistemlerinin yaygın olarak kullandığı dosya sistemi ext2 (Second Extention File System) denir.

DOS ve Win32 Sistem Programcıları için Not: Klasik DOS'un dosya sistemi FAT ismiyle ile anılır. Win95 ile birlikte bu sistem uzun dosya isimlerini içerecek biçimde genişletilmiştir. Bu dosya sistemi VFAT biçiminde isimlendirilmiştir. OS/2 sistemleri HPFS isimli farklı bir dosya sistemi kullanırlar. Nihayet Windows NT'lere özgü ayrıntılı bir dosya sistemi de kullanılmaktadır. Bu sistemler NTFS ismi ile anılır.

Dosya sistemlerinin incelenmesi iki aşamada yapılmalıdır.

1) Kullanıcı gözüyle incelenmesi: Bu inceleme dosya sisteminin kullanıcılar tarafından nasıl kullanılacağını anlamak ile ilgilidir. Yani dosya sisteminin dış dünyadan nasıl gözüktüğünün anlaşılmasıdır. İşletim sisteminin dışsal görüntüsünün tamamı resmi olarak sunduğu dosya sistem fonksiyonları kadardır.
2) Dosya sisteminin disk organizasyonunun incelenmesi: Dosya sistemi aslında aşağı seviyeli bir disk organizasyonuna dayanır. Bu inceleme klasik kullanıcı ya da programcılar için önemli değildir. Ancak aşağı seviyeli programlama ile uğraşan sistem programcıları için disk organizasyonunun bilinmesi önemlidir. Disk organizasyonu bilinmeden bazı kavramların anlaşılması da zor olmaktadır.

Bilindiği gibi tipik bir FAT sisteminin disk organizasyonu aşağıdaki gibidir.

Boot Sector
FAT(1)
FAT(2)
ROOTDIR
DATA

FAT dosya sisteminde boot sektör içerisindeki BPB alanı diskin kritik bilgilerini tutmaktadır. Dosya içerikleri DATA bölümünde cluster'larda saklanmaktadır. Bir cluster ardışık n sektör uzunluğundadır n sayısı BPB bloğunda yazmaktadır.

Yukarıda sözü edilen UNIX/Linux dosya sistemlerinin disk organizasyonu birbirlerine çok benzerdir. Tipik UNIX/Linux dosya sistemlerinin dosya organizasyonu diski dört bölüme ayrıdır.

Boot Sector
Super Block
i-node Block
Data Block

Boot sektör işletim sisteminin yüklenmesinde kullanılan sektördür. Boot sektörü hemen süper blok sektörleri izler Süper blok içerisinde disk organizasyonuna ilişkin kritik bilgiler tutulmaktadır. Süper blok görev bakımından tamamen FAT dosya sistemindeki boot sektör BPB bloğu gibidir. UNIX/Linux dosya sistemlerinde de dosyanın parçaları ardışık sektörlerde tutulur. Ancak FAT sistemindeki gibi "cluster" biçiminde değil "block" biçiminde isimlendirilmektedir. i-node blokları i-node elemanlarından oluşur. Her i-node elemanı bir dosya bilgisini tutar.

Yararlı POSIX komutları: df komutu dosya sistemindeki toplam blok sayısını, ne kadar bloğun kullanıldığını ve ne kadar bloğun boş olduğunu görmek amacı ile kullanılır.

Bir i-node elemanı içerisinde hangi bilgiler vardır? Dosyaya ilişkin dosyanın ismi dışındaki bütün bilgiler i-node elemanında saklanır. Örneğin dosyanın erişim bilgileri, zaman bilgileri, hangi kullanıcıya ve gruba ilişkin olduğu, dosyanın datalarının hangi bloklarda olduğu bilgileri burada tutulmaktadır. stat ve fstat fonksiyonları dosya bilgilerini i-node elemanlarından almaktadırlar. i-node elemanlarında dosyanın ismi tutulmaz. Bir dosyanın i-node numarası dosyanın en önemli bilgilerindendir. Bilindiği gibi stat fonksiyonu dosyanın i-node numarasını da vermektedir. POSIX standartlarında dosyanın i-node numarasına kısaca dosya numarası denilmektedir. Dosya seri numarası POSIX fonksiyonlarında kullanılan bir kavramdır. POSIX standartları yalnızca UNIX/Linux türevi olan sistemler için düşünülmemiştir. Herhangi bir sistem bir POSIX ara yüzü sağlayabilir. Bu durumda bu sistemler farklı disk organizasyonları kullanabilecekleri için dosya numarası kavramını nasıl üretirler? İşte bu sistemler sırf POSIX uyumunu sağlamak için dosyalar için birer dosya numarası uydurabilmektedir. i-node bloğunda ilki sıfırdan başlamak üzere her i-node elemanına bir numara verilmiştir. Dosya numarası ya da dosyanın i-node numarası demekle o dosyanın bilgilerinin kaçıncı i-node elemanında saklandığı anlaşılmaktadır.

UNIX/Linux sistemlerinde de tıpkı FAT dosya sisteminde olduğu gibi dizin ile dosya kavramlarının organizasyonları arasında fark yoktur. Normal dosyaların içerisinde bizim oluşturduğumuz bilgiler vardır. Dizin dosyaların içerisinde de dizin içerisindeki dosyaların bilgileri bulunur. Bir dizin dosyasının içeriği iki elemanlı kayıtlar biçimindedir.

i-node no	Dosya ismi

Bu durumda bir dizin dosyasının içerisi tipik olarak şöyle olacaktır:

i-node	**Dosya ismi**
i-node no	Dosya ismi
i-node no	Dosya ismi

Burada her bir kayıt kaç byte uzunluktadır? Bu durum sistemden sisteme değişebilmektedir. i-node numarası için genellikle 4 byte yer ayrılır. Klasik UNIX sistemlerinde dosya ismi 14 karakter uzunluğundaydı. Ancak bu gün pek çok sistemde dosya isimleri 256 karakter olabilmektedir. Bu sistemde bir dosyanın içeriğine erişilebilmesi için önce o dosya dizin girişlerinde aranmalı ve i-node numarası elde edilmeli sonra i-node bloklarına gidilerek i-

node bilgileri ele geçirilmeli. Tabii bu sistemde root dizininin yeri bilinmek zorundadır. Gerçekten de POSIX sistemlerinde root dizininin i-node numarası sıfırdır. Örneğin stat fonksiyonu ile C/D/E/F/g.dat dosyasının bilgilerini alacak olalım. Burada stat fonksiyonu root dizininden hareketle sırasıyla dizinlerin içerisinden geçecek ve en sonunda dosyayı bulacaktır. Tasarımda bu işlem kendi kendini çağıran fonksiyonlarla kolayca yapılabilir.

UNIX/Linux sistemlerindeki en önemli özelliklerden birisi link kavramıdır. Link bir dosyaya farklı bir dizinden farklı bir isimle erişmek anlamına gelir. Şöyle ki, farklı dizinlerdeki farklı isimli iki dosyanın i-node elemanları aynı olursa bu iki dosya aslında aynı dosyalardır. İsimler yalnızca bir semboldür. Bir dosyanın yeni bir linkinden oluşturmak için link sistem fonksiyonu kullanılır.

```
int link(const char *oldname, const char *newname);
```

Fonksiyonun prototipi unistd.h içerisindedir. Başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. Bir dosyanın bir link'i oluşturulduğunda artık orijinali ile linki arasında ayırt edici hiç bir özellik kalmaz. Yani orijinal kopya ve link kavramları yoktur, iki bilgi tamamen eş değer düzeyde tutulmaktadır. Bir dosyanın kaç link'e sahip olduğu dosyaya ilişkin i-node elemanında da tutulmaktadır. Bilindiği gibi bu bilgi stat ya da fstat fonksiyonları ile alınabilir. Dosyanın linklerinin sayısı ls -l shell komutuyla da alınabilir. Bir dosyayı silmek için yalnızca unlink isimli bir fonksiyon vardır. unlink fonksiyonu silinmek istenen dosyanın i-node elemanına bakar, onun link sayısını bir eksiltir. Link sayısı sıfıra düşmedikten sonra unlink fonksiyonu yalnızca dizin girişini silmektedir. Ancak link sayısı sıfıra düştüğünde unlink gerçek silmeyi yapar. Dizinler için link sayısı dizin yaratıldığında ikiden başlar. Çünkü aslında link sayısı ilgili i-node elamanına referans eden dizin girişlerinin sayısıdır. Bir dizin yaratıldığında hem yaratılan dizinin içerisinde referans oluşacak hem de yaratılmış olan dizinde oluşturulan '.' dosyası için bir referans oluşacaktır. Yeni yaratılan dizin içerisinde yaratılan her dizin için dizinin link sayısı bir artar. Çünkü alt dizinlerin '..' isimli dosyaları da üst dizinin i-node elemanına referans etmektedir.

Bir dosyanın toplam link sayısı i-node elemanında yazar. Zaten bu nedenle stat fonksiyonu ile elde edilebilmektedir. Ancak bir dosyanın bütün linklerinin isimleri pratik bir biçimde bulunamaz. Bunun için bütün dizin ağacının dolaşılması gerekir.

UNIX / Linux sistemlerinde Dizin Dolaşma İşlemleri

UNIX/Linux sistemlerinde dizin dolaşma işlemleri için opendir, readdir ve closedir fonksiyonları kullanılmaktadır. Bu fonksiyonlar POSIX standartlarında belirtilmiştir, dolayısıyla pek çok sistem tarafından desteklenmektedir. Bu dosyaların prototipleri dirent.h dosyası içerisindedir.

DOS ve Win32 Sistem programcıları için not: DOS'ta bu işlemi yapan fonksiyonlar findfirst ve findnext, Win32'de FindFirstFile ve FindNextFile'dır.

POSIX sistemlerinde önce opendir fonksiyonu ile bir handle alınır, sonra readdir her çağırıldığında yeni bir dizin girişi elde edilir.

```
DIR *opendir(const char *name);
```

Fonksiyonun parametresi içeriği elde edilecek dizin ismidir. Örneğin "/home/kaan" gibi

olmalıdır. Fonksiyon DIR yapısı türünden bir handle alanını dinamik olarak tahsis eder ve handle değeri ile geri döner. Başarısızlık durumunda fonksiyon NULL değerine geri döner.

```
struct dirent *readdir(DIR *dir);
```

Bu fonksiyon her bulduğu dosyanın bilgilerini kendi içerisindeki static yerel bir struct dirent yapısının içerisine yerleştirir ve bu yapının başlangıç adresiyle geri döner. Fonksiyonun parametresi opendir fonksiyonundan alınan handle değeridir. Geri dönüş değeri static bir alanın adresi olduğu için fonksiyon çok girişli (reentered) değildir. Programcı bu fonksiyonu döngü içerisinde çağırır. Fonksiyon artık dizinde bir dosya bulamadığı zaman NULL değeri ile geri döner. Dizin dolaşma işlemi bittikten sonra handle alanı closedir fonksiyonu ile kapatılmalıdır.

```
int closedir(DIR *dir);
```

dirent yapısı şöyledir:

```
struct dirent {
      long d_ino;
      char d_name[NAME_MAX + 1];
}
```

Görüldüğü gibi dirent yapısı aslında dizin girişindeki bilgileri vermektedir. UNIX/Linux sistemlerinin disk organizasyonu çeşitli değişiklikler göstermektedir. Örneğin bazı sistemlerde her dizin elemanı i-node numarası ve dosya ismi dışında başka bilgiler de içerebilmektedir. Bu sistemlerde dirent yapısının daha fazla elemanı olabilir. Yukarıda açıklanan dirent yapısı POSIX standartlarında tanımlanmış olan en küçük durumdur.

```
/*dizindolas1.c */

#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main(int argc, char *argv[])
{
      DIR *dp;
      struct dirent *dinfo;

      if (argc == 1) {
            printf("Usage mydir <dir name>\n");
            exit(1);
      }

      if (argc >= 3) {
            printf("too many argument!\n");
            exit(2);
      }

      dp = opendir(argv[1]);

      if (dp == NULL) {
            perror("opendir");
            exit(1);
      }
```

```
      while ((dinfo = readdir(dp)) != NULL)
            printf("%s\n", dinfo->d_name);

      closedir(dp);
}
```

/* dizindolas2.c */

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

#define MAX_PATH_SIZE 1024

void DirWalk(const char *path, void (*Proc) (const char *))
{
      char fname[MAX_PATH_SIZE];
      struct dirent *de;
      struct stat status;
      DIR *dir;

      if ((dir = opendir(path)) == NULL) {
            perror("opendir");
            return;
      }
      while ((de = readdir(dir)) != NULL) {
            sprintf(fname, "%s/%s", path, de->d_name);
            Proc(fname);
            if (strcmp(de->d_name, ".") != 0 &&
                                    strcmp(de->d_name, "..") != 0) {
                  if (stat(fname, &status) == -1) {
                        perror("stat");
                        break;
                  }
                  if (S_ISDIR(status.st_mode))
                        DirWalk(fname, Proc);
            }
      }
      closedir(dir);
}

#if 1

void Disp(const char *name)
{
      puts(name);
}

int main(int argc, char *argv[])
{
      char fname[MAX_PATH_SIZE];
      char *plast;
      size_t size;

      if (argc != 2) {
            printf("Usage: dirtree <path>\n");
            exit(1);
```

```
	}

	strcpy(fname, argv[1]);
	plast = strchr(fname, '\0') - 1;

	if (plast == '/')
	*plast = '\0';
	DirWalk(fname, Disp);

	return 0;
}
#endif
```

/* dizindolas3.c */

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

#define MAX_NAME_SIZE 1024

void DirWalk(int indent, const char *name, void (*Proc)(int, const char *))
{
	char cwd[MAX_NAME_SIZE];
	struct dirent *de;
	struct stat status;
	DIR *dir;

	if ((dir = opendir(name)) == NULL) {
		perror("opendir");
		return;
	}

	if (getcwd(cwd, MAX_NAME_SIZE) == NULL) {
		perror("getcwd");
		return;
	}

	if (chdir(name)) {
		perror("chdir");
		return;
	}

	while ((de = readdir(dir)) != NULL) {
		Proc(indent, de->d_name);
		if (strcmp(de->d_name, ".") != 0 &&
					strcmp(de->d_name, "..") != 0) {
			if (stat(de->d_name, &status) == -1) {
				perror("stat");
				break;
			}
			if (S_ISDIR(status.st_mode))
				DirWalk(indent + 1, de->d_name, Proc);
		}
	}
	closedir(dir);
	if (chdir(cwd)) {
```

```
            perror("chdir");
            return;
      }
}

#if 1

void Disp(int indent, const char *name)
{
      int i;

      for (i = 0; i < indent * 2; ++i)
            putchar(' ');
            puts(name);
}

int main(int argc, char *argv[])
{
      char fname[MAX_NAME_SIZE];
      char *pLast;

      if (argc != 2) {
            fprintf(stderr, "Usage: dirtree <path>\n");
            exit(1);
      }
      strcpy(fname, argv[1]);
      pLast = strchr(fname, '\0');
      if (*pLast == '/' && strlen(fname) != 1)
            *pLast = '\0';
      DirWalk(0, fname, Disp);
      return 0;
}

#endif
```

/* dizindolas4.cpp */

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <vector>
#include <algorithm>
#include <cstdio>
#include <dirent.h>

using namespace std;

int main(int argc, char *argv[])
{
      DIR *dp;
      struct dirent *dinfo;
      vector<string> dirlist;

      if (argc == 1) {
            cerr << "Usage mydir <dir name>\n";
            exit(1);
      }
      if (argc > 3) {
            cerr << "too many arguments!\n";
            exit(2);
```

```
        }
        dp = opendir(argv[1]);
        if (dp == NULL) {
                perror("opendir");
                exit(1);
        }
        while ((dinfo = readdir(dp)) != NULL)
                dirlist.push_back(dinfo->d_name);
        sort(dirlist.begin(), dirlist.end());
        copy(dirlist.begin(), dirlist.end(),
        ostream_iterator<string>(cout, "\n"));
        closedir(dp);
}
```

Blok Kavramı

UNIX/Linux sistemlerinde bir dosyanın parçası olabilecek en küçük tahsisat birimine blok denilmektedir. Blok kavramı FAT ve VFAT dosya sistemlerindeki cluster kavramıyla aynı anlamdadır.

DOS veWin32 Sistem Programcıları İçin Not: FAT ve VFAT sistemlerinde bir cluster'ın kaç sektörden oluştuğu boot sektör içerisindeki BPB alanında belirtilmektedir.

Bir bloğun kaç sektör olduğu bazı UNIX sistemlerinde sistemin kurulumu ya da dosya sisteminin oluşturulması sırasında belirlenebilmektedir. Bazı sistemlerde belirleme yapılmaz. UNIX sistemlerinde 1 blok = 2 sektör = 1024 byte'tır. UNIX/Linux sistemlerinde de bütün disk sıfırdan başlayarak bloklara ayrılmıştır. Her bloğun bir numarası vardır.

UNIX/Linux sistemlerinde blok analizi için tipik olarak df ve du komutları kullanılmaktadır. df komutu dosya sistemindeki toplam blok sayısını, tahsis edilmiş ve kullanılabilir blok sayılarını vermektedir. du komutu ise recursive olarak belirlenen dizinden itibaren dibe inerek tüm dizinlerin kaçar blok uzunlukta olduğunu bilgisini vermektedir. du komutu dizin dolaşma fonksiyonları ile yazılmıştır. Dolayısıyla erişim hakkı yetersizliğinden dolayı bir dizine geçilemez ise error mesajını stderr dosyasına yazar.

C ve C++ programcıları için not: C'de stdin klavyeyi, stdout ekranı ve stderr de error dosyalarını temsil eder. C++'ta cout nesnesi stdout dosyası ile, cin nesnesi stdin dosyası ile ve cerr nesnesi stderr dosyası ile ilişkilidir. Default olarak sistemlerde stderr dosyası stdout dosyasına yönlendirilmiş durumdadır. DOS'ta komut satırında > yönlendirme işlemi yalnızca stdout dosyasını yönlendirir. Örneğin DOS'ta den.exe > x biçiminde bir programı çalıştırdığımızda programın stdout dosyasına yazdıkları x dosyasına aktarılır, ancak stderr dosyasına yazdıkları ekrana çıkmaya devam eder. Aynı durum UNIX/Linux sistemleri için de geçerlidir.

UNIX/Linux sistemlerinde shell üzerinden stderr dosyası 2> sembolü ile herhangi bir dosyaya yönlendirilebilir. Bazı UNIX/Linux komutları çeşitli error mesajları oluşturmaktadır. Bu error mesajlarından kurtulmak için stderr dosyası yönlendirilebilir. Bu tür yönlendirmeler için özel /dev/null dosyası tasarlanmıştır. Bu dosya gerçek bir dosya değildir gelen bilgileri siler.

```
du 2> /dev/null
```

mount İşlemi

DOS ve Windows sistemlerinde dosya sistemleri sürücü kavramı altında kullanıcıya sunulmaktadır. Bu sistemler heterojen dosya sistemlerini farklı sürücüler olarak gösterebilmektedir. Ancak UNIX/Linux sistemlerinde sürücü kavramı yoktur. mount işlemi ile bir dosya sisteminin kök dizini önceden yaratılmış olan bir dizin ile çakıştırılarak o noktaya monte edilebilmektedir. mount edebilmek için kesinlikle bir dizinin yaratılmış olması gerekir. mount edilen nokta ve bu noktanın aşağısındaki tüm ağaç mount işleminden sonra erişilebilirlik özelliğini kaybeder. Bu nedenle genellikle mount işlemleri için ayrı dizinler yaratmak gerekir. mount işlemi mount sistem fonksiyonu ile yapılabilir ya da doğrudan çalıştırılabilen bir mount programı da sağlanmıştır. mount işlemi sistemin bütünlüğü bakımından önemli bir işlemdir bu nedenle bu işlemi ancak root kullanıcısı yapabilir. mount komutunun basit biçimi şöyledir:

```
mount <düğüm noktası> <mount noktası>
```

mount işlemi UNIX/Linux sistemlerinde her aygıt ve dosya sistemi /dev dizinindeki sembolik bir dosya ile temsil edilir. Örneğin floppy disket dev/fd0 ile, birinci sabit diskin disk bölmeleri /dev/hda1, /dev/hda2 ile temsil edilmektedir. Örneğin floppy aşağıdaki gibi mount edilebilir.

```
mount /dev/fd0 /usr/floppy
```

mount komutu düğüm noktası ile belirtilen sürücüdeki dosya sistemini otomatik olarak tespit etmeye çalışır. mount komutuna -t switch'i eklenebilir. Bu switch'i dosya sistemine ilişkin tür bilgisi izlemelidir. Örneğin:

```
mount -t vfat /dev/fd0 /usr/floppy
```

Sistem açıldığında gördüğümüz kök dosya sistemi de aslında mount işlemi ile elde edilmiştir. mount işlemi umount işlemi ile kaldırılabilir (mount edilen aygıt üzerinde işlemler devam ediyorsa umount işlemi yapılamaz). Örneğin:

```
umount /usr/floppy
```

UNIX/Linux Sanal Dosya Sistemi (VFS)

UNIX/Linux sistemlerinde donanım aygıtları (sabit disk, disket sürücü, fare, seri port, paralel port ...) birer dosya olarak işlem görür. Bütün bu aygıtlar temel sistem fonksiyonları olan open, close, read ve write fonksiyonları ile yönetilebilirler. UNIX/Linux sistemlerinde üzerinde çalışılan aygıt ne olursa olsun aygıta temel sistem fonksiyonları ile hep aynı biçimde erişilir. Sistemin aygıt bağımsız dosya işlemlerini gerçekleştiren bölümüne sanal dosya sistemi (Virtual File Sytem) denilmektedir. UNIX/Linux sistemleri 80'li yıllardan itibaren nesne yönelimli programlama tekniğinden etkilenmiştir. Sanal dosya sistemi çok biçimli (polimorfik) bir tasarım yapısına sahiptir. Yani programcı open ya da read fonksiyonunu kullanırken bu işlemler gerçekte ilgili aygıt sürücüsünün open ya da read fonksiyonlarını çağırır.

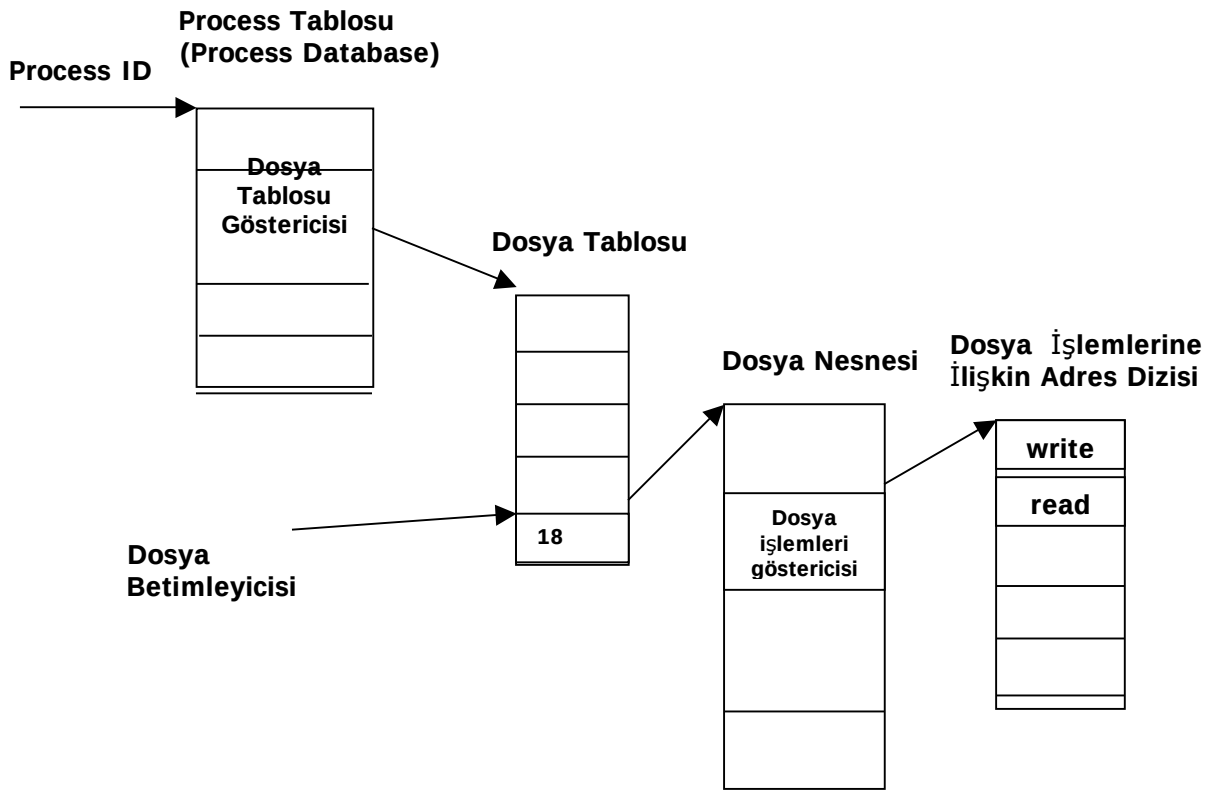
Bilindiği gibi bir process yaratıldığında işletim sistemi process'e ilişkin kritik bilgileri bir process tablosunda saklamaktadır (Win32 sistemlerinde process tablosuna process veritabanı (process database) denilmektedir). Programcı process üzerinde işlemler yapmak isterse

process tablosuna erişmekte kullanılan bir handle değerinden faydalanır. Process'in handle değeri işletim sistemlerinde process'i çalıştıran kişi tarafından doğal olarak elde edilir. Çalışmakta olan başka process'lerin handle değerleri işletim sisteminin sağladığı çeşitli fonksiyonlarla elde edilebilmektedir.

Process'e ilişkin process tablosunda pek çok bilgi tutulmaktadır. Bu konudan process yönetimi kısmında bahsedilecektir. Ancak konu ile ilgili olarak bazı açıklamalar burada yapılacaktır. Process'in kullanmakta olduğu dosyalar bir biçimde işletim sistemlerinde process tablosunda tutulmaktadır.

Popüler işletim sistemlerinde dosyalara erişmek için bir handle değeri kullanılır. Win32 sistemlerinde bu değere "File Handle" UNIX/Linux sistemlerinde dosya betimleyicisi (File Descriptor) denilmektedir. Bir dosya açıldığında işletim sistemi dosya işlemlerini yönetebilmek için bir alan tahsis eder. Dosyaya erişim için gerekli bilgileri o alana yerleştirir. Bu alana genellikle dosya nesnesi (file object) denilmektedir. Programcı dosyayı açtığında handle değeri olarak dosya nesnesinin adresini elde etmez. İşletim sistemi dosya nesnesinin adresini dosya tablosu denen bir tabloya yazar.

Handle değeri olarak bu tablodaki satır numarasını verir. Dosya tabloları process'e özgüdür ve girişi process tablosu içerisindedir. Bu tasarım biçimi popüler pek çok işletim sisteminde aynı biçimde kullanılmaktadır. Örneğin UNIX/Linux sistemleri ile Win32 sistemleri bu bakımdan benzer tasarıma sahiptir. Örneğin UNIX/Linux sistemlerinde open sistem fonksiyonunu çağırarak bir dosya açmış olalım ve bize handle değeri olarak (dosya betimleyicisi olarak) 18 değeri verilmiş olsun. Şöyle bir şekil oluşur.



Görüldüğü gibi, bir dosyayı açtığımızda elde edilen dosya betimleyicisi process'e özgü göreli bir değerdir. Yani biz bu process'te read fonksiyonuna 18 değerini girerek okuma yaparsak, bu dosyadan okuma yaparız, ama başka bir process bu değerle başka dosyadan okuma yapabilir.

Dosya tablosunun uzunluğu genellikle dinamik olarak ayarlanmaz, bu uzunluk baştan belirlenmiştir. Tablonun uzunluğu aynı anda açık olabilecek dosya sayısını belirtmektedir. Örneğin Linux2.X kernel versiyonlarında bu sayı 256 biçimindedir.

Modern UNIX/Linux sistemlerinde VFS dosya sistemi içerisinde tüm aygıtlar üzerinde işlemler read, write gibi klasik POSIX fonksiyonları ile yapılır. Bu fonksiyonlar nesne yönelimli bir yaklaşımla aygıta ilişkin özel read, write gibi fonksiyonları çağırırlar. Bu durum tipik olarak dosya nesnesi içerisinde dosya işlemlerini yapacak olan fonksiyonların adreslerini tutan bir tablo yoluyla yapılır. Görüldüğü gibi sistem fonksiyonlarını bir dosya betimleyicisiyle çağırdığımızda gerçekte hangi fonksiyonların çalıştırılacağı bu betimleyicilerin gösterdiği dosya nesnelerine bağlıdır. Aslında bu işlemin C++'taki çok biçimli karşılığı şöyledir:

```
class FileObject {
public:
      virtual int read(...) = 0;
      virtual int write(...) = 0;
private:
      //...
      //...
};

sysread(FileObject *pFile, ...)
{
      pFile->read(...);
}
```

UNIX/Linux sistemleri pek çok popüler dosya sistemini desteklemektedir. Bu dosya sistemlerinin disk organizasyonları birbirinden farklıdır. Her ne kadar farklı olsa da UNIX/Linux sistemleri bunları bellekte bir super block yapısı içerisinde ifade ederler. super block yapısı her dosya sistemi bilgilerini tutabilecek şekilde organize edilmiştir. Super block yapıları kendi aralarında bir bağlı liste biçiminde tutulur. Aygıta ilişkin super block yapısı mount işlemi sırasında oluşturulmakta, umount işlemi ile de yok edilmektedir. İlgili dosya sistemine erişimde veri yapısı olarak giriş noktası super block yapısıdır.

Neden dosya handle değeri olarak dosya tablosunda bir offset verilmektedir de doğrudan dosya nesnesinin adresi verilmemektedir? Bu tasarımda birincil amaç yönlendirme (redirection) yapılabilmektedir. Örneğin dosya tablosu ile oynanılarak dosya betimleyicilerine ilişkin nesne adresleri değiştirilebilir. Böylelikle programda hiç bir değişiklik yapmadan programın çalıştığı dosyalar değiştirilebilmektedir. İkincil olarak dosya nesnesine doğrudan erişilmemesi daha güvenli bir yöntemdir. Bu dosya sisteminin tasarımı Win32 sistemlerinde de benzer biçimdedir.

POSIX Dosya Fonksiyonları

POSIX fonksiyonları bazı sistemlerde doğrudan sistem fonksiyonu olabilir, bazı sistemlerde ise sarma fonksiyon olabilir. Örneğin Linux sistemlerinde tipik olarak gerçek sistem

fonksiyonları 80h kesmesi biçiminde yazılmıştır. 80h kesmesi tipik olarak kesme kapısı (interrupt gate) içerir. Kesme kapısı yoluyla kod ring3 önceliğinden ring0 önceliğine geçer. Burada ilk çalışan kod kaç numaralı sistem fonksiyonunun çağırıldığını anlayarak bir tablodya bakarak uygun fonksiyonu çağırır. Böylece sistem fonksiyonu ring0 modunda çalıştırılır. Kuşkusuz sistem fonksiyonundan IRET komutu ile geri dönüldüğünde tekrar ring3 düzeyine dönülür. Genel olarak intel tabanlı çalışan işletim sistemlerinde ring0'da çalışan kodlar için "kernel mod", ring3'te çalışan kodlara ise "user mod" denilmektedir.

Anahtar Notlar: Intel işlemcileri korumalı moda geçirildiğinde 4 öncelik derecelendirilmesiyle işlemler yürütülür. En öncelikli derece 0'dır, en az öncelikli derece 3'tür. Çalışmakta olan bir kodun 0-3 arasında bir öncelik derecesi vardır. Bellek bölgesi de benzer biçimde 0-3 arasında öncelik derecelerine atanmıştır. Düşük öncelikli kodlar kod ve data bakımından yüksek öncelikli bellek bölgelerine erişemezler. İşletim sisteminin kritik kodları ve dataları 0 öncelikli bölgelerde bulunur (ring0) böylece sıradan programlar bu dataları bozamazlar. Ancak sistem fonksiyonlarının bellekte her bölgeye erişebilmesi gerekir. Bellekte her bölgeye erişebilmesi için sistem fonksiyonlarının ring0'da çalıştırılabilmesi gerekir. İşte intel işlemcilerinde kapı denilen bir kavram yoluyla işletim sisteminin izin verdiği güvenli kodlar öncelik derecesi yükseltilerek çalıştırılabilmektedir. Böylelikle ring3'te çalışan kodlar işletim sisteminin güvenli kodlarını ring0'da çalıştırabilmektedir.

open Fonksiyonu

open fonksiyonu tipik olarak fopen fonksiyonu tarafından çağırılan POSIX fonksiyonudur.

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

open fonksiyonu iki parametreli ya da üç parametreli olarak kullanılabilir. Fonksiyonun eskiden iki parametresi vardı, daha sonra genişletildi. İki farklı isimli fonksiyon olmasın diye değişken sayıda parametre alan bir görüntüye sokuldu.

open fonksiyonu iki ya da üç parametreli çağırılır. Üç parametreli kullanılırsa üçüncü parametre erişim bilgileridir. Fonksiyonun ikinci parametresi çeşitli sembolik sabitlerin bit or işlemine sokulması ile elde edilir. Bu sembolik sabitler şunlardır:

```
O_RDONLY
O_WRONLY
O_RDWR
O_APPEND
O_CREAT
O_EXCL
O_NONBLOCK
O_TRUNC
```

O_RDONLY, O_WRONLY, O_RDWR belirlemelerinden en az biri kullanılmalıdır. Bu belirlemeler isimlerinden de anlaşılacağı gibi okuma yazma işlemlerinin yapılacağını belirtir. O_CREAT dosyayı yoksa yaratır, dosya varsa bu değerin bir etkisi olmaz. O_TRUNC dosya varsa onu sıfırlar ancak dosya yoksa bir etkisi yoktur. O_EXCL tek başına kullanılmaz mutlaka O_CREAT ile kullanılır. Normalde dosya varsa O_CREAT hiç bir anlam ifade etmez. Ancak bununla birlikte O_EXCL de kullanılmışsa bu durum fonksiyonun başarısız olmasına yol açar. O_APPEND yazma işlemleri sırasında sona eklemeye yol açar. Yani her yazma işlemi sırasında yazma fonksiyonları otomatik olarak dosya göstericisini EOF durumuna çekerek oraya yazar. open fonksiyonu başarılıysa dosya betimleyicisi değerine,

başarısızsa –1 değerine geri döner. open fonksiyonunun prototipi ve açış modunu oluşturmakta kullanılan sembolik sabitler fcntl.h başlık dosyası içerisindedir.

open fonksiyonunun üçüncü parametresi ancak O_CREAT belirlemesi yapıldıysa anlamlıdır. Bu parametre erişim haklarını belirlemekte kullanılır, fonksiyon bu parametreyi O_CREAT belirlemesi yapılmışsa kullanır (eğer yanlışlıkla O_CREAT belirlemesi yapıldığı halde üçüncü parametre girilmemişse bir derleme zamanı hatası oluşmaz, fonksiyon stack'ten rasgele bir değer çeker). Erişim hakları için geleneksel olarak dokuz bitlik sayı kullanılmaktadır. Dokuz bitlik bu sayıda her üç bit yüksek anlamlıdan düşük anlamlıya doğru sırasıyla sahip, grup, diğer erişim bilgilerini belirlemektedir. 0666 -> 110 110 110 sırasıyla wr-wr-wr-. Ancak fonksiyonun üçüncü parametresinin bu biçimde dokuz bit olarak girilmesi POSIX standartlarınca garanti altına alınmamıştır. Bunun için daha önce incelediğimiz sembolik sabitler kullanılır. Bu sembolik sabitler sys/stat.h başlık dosyası içerisindedir.

```
S_I + R + USR
S_I + W + OTH
S_I + X + GRP
```

Örneğin S_IWUSR dosyanın sahibine yazma hakkı verir.

```
fd = open("a.dat",
            O_RDWR | O_CREAT | O_TRUNC,
            S_IUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
```

fopen fonksiyonundaki açış modlarının open fonksiyonundaki bazı karşılıkları şunlardır:

```
"r"   O_RDONLY
"r+"  O_RDWR
"w"   O_WRONLY | O_CREAT | O_TRUNC
"w+"  O_RDWR | O_CREAT | OTRUNC
```

Process'in Maskeleme Değeri

open ve create fonksiyonlarında belirtilen erişim hakları process'in maskeleme değeri denilen bir değer ile işleme sokularak gerçek erişim hakları belirlenir. Her process'in bir maskeleme değeri vardır. Bir process başka bir process'i yarattığında bu değer yeni process'e aktarılır (Yani örneğin shell üzerinden program çalıştırdığımızda çalıştırılacak process'in maskeleme değeri shell'in maskeleme değeri olur). Process maskeleme değerini istediği zaman umask sistem fonksiyonu ile değiştirebilir. Genellikle shell tarafından belirlenmiş default maskeleme değeri 022 biçimindedir. Process'in maskeleme değerindeki 1 olan bitler dosyanın yaratılması sırasında kesinlikle dikkate alınmayacak erişim özelliklerini belirtmektedir. Örneğin 022 (8'lik sistemde) maskeleme değeri şu anlama gelir:

```
0     2     2
000   010   010
```

Burada görüldüğü gibi grup ve diğer kişiler için yazma hakkı tamamen maskelenmiştir. Yani biz bu maskeleme değeri ile dosyayı 666 erişim haklarıyla yaratmak istesek bile dosya wr-r--r--haklarıyla yaratılacaktır. Bu işlemlerin daha algılanabilir karşılığı şöyledir:

```
<open fonksiyonunda belirtilen erişim bitleri> & ~<maskeleme değeri>
```

Sonuçta elde edilen değerin 0 olan bitleri erişimin yasaklandığı 1 olan bitleri erişimin kabul edildiği yerleridir. Örneğin:

```
 022  =     000 010 010
~022  =     111 101 101
  &         110 100 100
            wr- r-- r--
```

umask Fonksiyonu

Process'in default maskeleme değerini değiştirmekte kullanılır.

```
mode_t umask(mode_t mask);
```

Fonksiyon process'in maskeleme değerini parametresiyle belirtilen değer olarak belirler ve eski maskeleme değerini geri dönüş değeri olarak verir. Process'in maskeleme değerini alan başka bir fonksiyon yoktur. Bu işlem şöyle yapılabilir:

```
mode_t val;
val = umask(0);
umask(val);
```

close Fonksiyonu

Açılan her dosya kapatılmalıdır. Kapatılma işlemi bilinçli olarak yapılmaz ise sistem tarafından process bittiğinde açık olan dosyalar kapatılır.

```
int close(int fd);
```

Fonksiyonun prototipi unistd.h başlık dosyası içerisindedir.

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

int main()
{
      int fd;

      umask(0);

      fd = open("a.dat", O_RDWR | O_CREAT, 0666);
      if (fd == -1) {
            fprintf(stderr, "Cannot open file..\n");
            exit(1);
      }
      close(fd);
}
```

read ve write Fonksiyonları

POSIX dosya fonksiyonları çok azdır. Bütün yazma ve okuma işlemleri read ve write sistem fonksiyonları ile yapılır.

```
int read(int fd, void *buf, unsigned int nbyte);
int write(int fd, const void *buf, unsigned int nbyte);
```

Fonksiyonların birinci parametresi dosya betimleyicisi, ikinci parametresi transfer adresi ve üçüncü parametresi transfer edilecek byte sayısıdır. Fonksiyonların geri dönüş değerleri transfer edilen byte sayısıdır. Örnek bir kopyalama programı. Fonksiyonların prototipleri unistd.h başlık dosyası içerisindedir.

UNIX/Linux ve DOS sistemlerinde aşağı seviyeli sektör temelinde kopyalama yapan sistem fonksiyonları yoktur. Kopyalama işlemi iki dosyanın açılıp bilgilerin blok blok taşınması ile yapılır. Blok büyüklüğü için sektör katları tercih edilmelidir.

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

#define BLOCK_SIZE 1024

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

      if (argc != 3) {
            fprintf(stderr, "Wrong number of arguments...!\n");
            exit(1);
      }

      if ((fds = open(argv[1], O_RDONLY)) == -1) {
            fprintf(stderr, "Cannot open source file : %s\n", argv[1]);
            exit(1);
      }

      if ((fdd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0666)) == -1) {
            fprintf(stderr,"Cannot open destination file...!\n", argv[2]);
            close(fds);
            exit(1);
      }
      while (( n = read(fds, buf, BLOCK_SIZE)) > 0)
            write(fdd, buf, n);

      printf("1 File Copied...\n");
      close(fds);
      close(fdd);

      return 0;
}
```

read ve write fonksiyonları başarısızlık durumunda –1 değerine geri dönerler. Bu durumda başarısızlığın nedeni klasik olarak errno değişkeninden alınabilir. Ancak yukarıdaki programda başarısızlık kontrolü yapılmamıştır. Yani okuma hatası olduğunda program

döngüden çıkmaktadır. Bir dosya başarılı bir biçimde açıldıysa normal olarak read ve write fonksiyonlarında hata oluşmaması gerekir. Hata ancak kritik bir biçimde ortaya çıkabilir. Zaten bu tür durumlarda sistemin kendisiside çoğu kez düzgün çalışmayacaktır. read ve write fonksiyonlarının son parametreleri ve geri dönüş değerleri bazı UNIX/Linux sistemlerinde unsigned biçimindedir. Ancak o sistemlerde de –1 değeri özeldir.

creat Fonksiyonu

Eskiden open fonksiyonunun yaratma özelliği yoktu, bu nedenle dosyayı ilk kez yaratmak için creat fonksiyonu kullanılırdı.

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

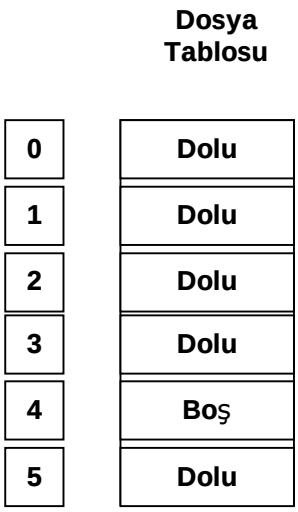
Bu fonksiyon eski programların hatırına hala varlığını sürdürmektedir. Eskiden open fonksiyonu yalnızca var olan dosyaları açabiliyordu, yeni bir dosya yaratmak için bu fonksiyonu kullanmak gerekirdi. Aşağıdaki iki çağırma tamamen birbirine eşdeğerdir.

```
creat(path, mode);
open(path, O_WRONLY | O_CREATE | O_TRUNC, mode);
```

POSIX sistemlerinde creat fonksiyonunun kullanılmasının bir gereği kalmamıştır.

open ve creat Fonksiyonlarının Boş Dosya Betimleyicilerinin Tespit Edilmesi

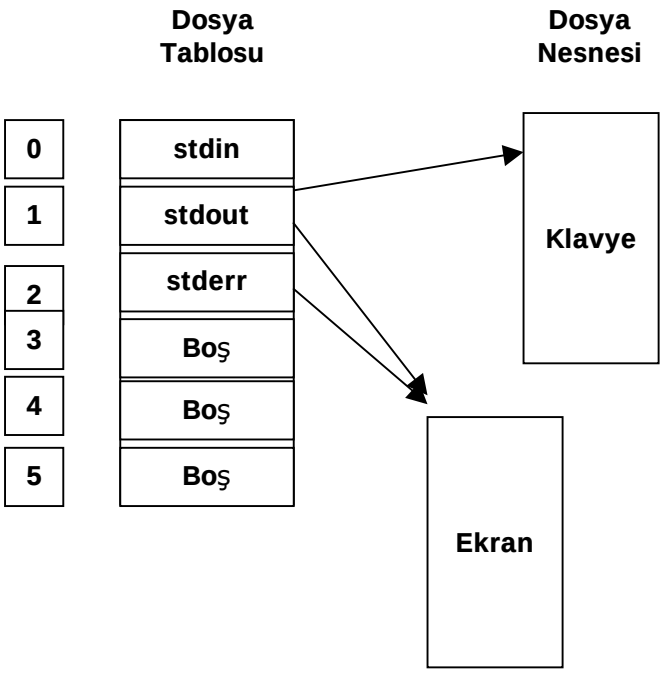
open ve creat fonksiyonları ile yeni bir dosya yaratıldığında dosya betimleyicisi olarak dosya tablosundaki küçükten büyüğe doğru ilk boş slot elde edilir. Örneğin process'in dosya tablosunun durumu şöyle olsun:



Şimdi open ya da creat fonksiyonu ile dosya yaratıldığında dosya betimleyicisi olarak kesinlikle 4 elde edilecektir.

stdin, stdout ve stderr Dosyalarının Yaratılması

Shell programının dosya tablosunda default olarak 3 dosya açılmış bir biçimde bulunur. Bunlar 0 betimleyicisine ilişkin stdin (klavye), 1 betimleyicisine ilişkin stdout (ekran) ve 2 betimleyicisine ilişkin stderr'dir. Bütün programlar shell üzerinden çalıştırıldığına göre ve dosya tablosu da çalıştırılan process'e aktarıldığına göre, yeni bir process oluşturulduğunda o process'in de dosya tablosunun ilk üç slotunda bu dosyalar açılmış bir biçimde bulunacaktır. Yeni yaratılmış bir process'in dosya tablosu şöyledir:



Örneğin:

```
write(1, "Kaan", 4);
```

write fonksiyonu yukarıdaki örnekte nesne yönelimli ve çok biçimli bir biçimde process tablosundan ve dosya tablosundan hareket ederek ekrana ilişkin gerçek write fonksiyonunu çağırır ve bu da yazının ekrana çıkmasını sağlar. Kişinin hangi dosyaya yazacağı ve gerçekte hangi fonksiyonun bu yazma işleminde kullanılacağı dosya nesnesinde belirlenmektedir. İki dosya betimleyicisinin değerleri farklı olsa ama bunlar aynı dosya nesnesini gösterse aynı dosya üzerinde işlem yapacaklardır. Dosya nesnesinde aynı zamanda dosya göstericisinin aktif değeri de tutulmaktadır. Bu durumda farklı dosya betimleyicileri aynı dosya nesnesini görüyorlarsa bunların dosya göstericisi de ortak olur. Örneğin fd1 ve fd2 farklı değerlere ilişkin dosya betimleyicileri olsun, fakat aynı dosya nesnesini görsün. Aşağıdaki örnekte write işlemiyle pos pozisyonuna yazma yapılır.

```
lseek(fd1, pos, SEEK_CON);
write(fd2, ...);
```

Dosya göstericisini konumlandırmakta kullanılan lseek fonksiyonu aslında herhangi bir giriş çıkış işlemine yol açmamaktadır. Bu fonksiyon yalnızca dosya nesnesi içerisindeki aktif

pozisyon bilgisini değiştirmektedir. stdout ve stderr dosyalarının default olarak her ikisinin de ekrana ilişkin olması bu biçimde kolaylıkla anlaşılabilir.

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

int main()
{
      int fd;

      close(1);
      if ((fd = open("a.dat", O_RDWR|O_CREAT|O_TRUNC, 0666)) == -1) {
            fprintf(stderr, "Cannot open file...!\n");
            exit(1);
      }
      printf("Merhaba\n");
      return 0;
}
```

C'nin Standart Dosya Fonksiyonları ve Bu Fonksiyonların POSIX Fonksiyonlarıyla İlişkisi

C'nin standart dosya fonksiyonları hangi sistemde yazılmışlarsa o sistemin gerçek sistem fonksiyonlarını çağırırlar. Örneğin fread fonksiyonu UNIX/Linux sistemlerinde read fonksiyonunu, Win32 sistemlerinde ise ReadFile fonksiyonunu çağırmaktadır. stdio.h içerisinde prototipi olan printf, scanf gibi fonksiyonlar da aslında birer dosya fonksiyonlarıdır. Örneğin printf default olarak stdout dosyasını kullanmaktadır. C'nin standart doysa fonksiyonlarının en önemli özelliği tamponlu (buffered) bir çalışma sunmasıdır.

Aslında modern sistemlerin hemen hepsinde dosya işlemlerine ilişkin sistem fonksiyonları da bir tampon mekanizması ya da bir cache sistemi kullanmaktadır. Ancak terminolojide tamponlamalı dosya fonksiyonları denildiğinde çekirdek tarafından yapılan tamponlama değil kullanıcı alanı içerisinde yapılan tamponlama anlaşılmaktadır. Sistem fonksiyonlarının kullandığı tampon bölgeler çekirdeğin bellek alanı içerisindedir. Halbuki standart C fonksiyonlarının kullandığı tampon bölgeler kullanıcı alanı içerisindedir. Aslında modern sistemlerde standart C fonksiyonları iki kez tamponlamaya yol açmaktadır. Böylece hız kazanmak bir yana, yavaşlamaya yol açmaktadır. Fakat C'nin standart dosya fonksiyonları başından beri bir tampon mekanizması içerisinde tasarlanmıştır. Standartlarda anlatımlar ve fonksiyonlar tamamen tamponlu bir çalışma üzerine kurulmuştur. Özetle C'nin dosya fonksiyonlarının tamponlu olması standartlara yansımış temel özelliklerdendir. C'nin dosya fonksiyonları hemen işletim sisteminin sistem fonksiyonlarını çağırmak zorunda değildir. Bilgileri bir tampon alanda tutup istediği bir zamanda sistem fonksiyonunu çağırır. Dosyanın tazelenmesi (flush edilmesi) yani fflush fonksiyonunun çağırılması tampondaki bilginin sistem fonksiyonları çağırılarak dosyaya aktarılması anlamına gelir. fclose işlemi ile ya da dosya kapatılmamışsa process'in sonlanması ile tazeleme işlemi yapılmaktadır. Tazeleme işleminin dosyanın yalnızca okuma modunda açıldığında hiç bir anlamı yoktur. Çünkü tazeleme tampondan dosyaya doğru yapılan bir işlemdir. stdin dosyası da C'de yalnızca okunabilen bir dosya olarak kabul edilir. Bu nedenle `fflush(stdin);` işleminin de bir

anlamı yoktur, ancak pek çok derleyicide bu işlem stdin dosyasının kullandığı tamponu boşaltma anlamına gelir.

stdout dosyasına bir şey yazdığımız zaman hemen ekranda görünür mü? Standartlara göre stdout dosyasına yazılanların tazelenmesi için iki şey gerekir.

1) Yazının sonuna '\n' koymak
2) `fflush(stdout);` yapmak

lseek Fonksiyonu

lseek fonksiyonu dosya göstericisini konumlandırmakta kullanılır. Tipik olarak fseek standart C fonksiyonu bu fonksiyonu kullanmaktadır (Bu fonksiyonun eski sistemlerdeki ismi seek fonksiyonu idi, sonra parametrenin offset değerinin long olması dolayısıyla lseek biçimine dönüştürülmüştür). Prototipi unistd.h başlık dosyası içerisindedir.

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

Fonksiyonun birinci parametresi dosya betimleyicisi, ikinci parametresi konumlandırılacak offset değeri, son parametresi ise konumlandırma referansıdır. Son parametre şu biçimde olabilir:

```
#define SEEK_SET (0)
#define SEEK_CUR (1)
#define SEEK_END (2)
```

off_t türü tipik olarak sistemlerde long biçimindedir. Fonksiyon başarılıysa yeni konumlandırılan offset değerine, başarısızsa –1 değerine geri döner (POSIX fonksiyonlarının çok büyük çoğunluğu başarısızsa –1 değerine geri dönmektedir. Bu fonksiyonlar başarılıysa herhangi bir negatif değere geri dönmemektedir. Bu nedenle bazı programcılar daha etkin olduğu gerekçesiyle hata karşılaştırmasını '== -1' yerine '< 0' biçiminde yaparlar). Ancak lseek fonksiyonu bazı aygıtlar için negatif konumlandırma da yapabilmektedir. Bu nedenle lseek fonksiyonunun hata testi eğer yapılacaksa '== -1' şeklinde yapılmalıdır.

Dosyalardaki Delikler

lseek fonksiyonu ile bir dosyanın uzunluğunun ötesine konumlandırma yapılabilir. Örneğin dosyanın uzunluğu 1000 olduğu halde biz aşağıdaki gibi lseek fonksiyonu ile 10000'inci offset'e konumlandırma yapabiliriz.

```
lseek(fd, 10000, SEEK_SET);
```

Bu işlem sonrasında dosyanın uzunluğu konumlandırılan offset'e çekilir yani arttırılır. Ancak artan bölüm gerçek anlamda disk üzerinde blok temelinde tahsis edilmez. Bu biçimde büyütülen dosya alanına delik (hole) denir. Delikten okuma yapıldığında sistem fonksiyonları sıfır okur. Delik kısmına yazma yapılabilir. Şüphesiz sistem deliklerin dosya içerisindeki yerlerini tutarak yazma sırasında gerçek blok tahsisatını yapmaktadır. Örneğin gerçekte 1000 uzunluğunda olan dosya delik oluşturularak 10000 byte'a çıkarılmış olsun, ls komutu (dolayısıyla stat fonksiyonu) uzunluk olarak delikli uzunluğu rapor edecektir. Ancak du

komutu uygulandığında dosya için tahsis edilen blok sayısının artmadığı görülecektir (du komutunda bildirilen blok miktarının cluster anlamındaki blok ile bir ilgisi yoktur, Linux sistemlerinin çoğu default olarak cluster anlamındaki bloğu 8 sektör = 4096 olarak alırlar). Delik oluşturabilmek için dosya uzunluğunun ötesine konumlandırdıktan sonra bazı sistemlerde write işlemi yapmak gerekir. Delik oluşturma POSIX standartlarında belirtilmiş standart bir davranış değildir.

```
/* lseek.c */

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

int main()
{
      int fd;

      if ((fd = open("a.dat", O_RDWR)) == -1) {
            perror("open error\n");
            exit(EXIT_FAILURE);
      }
      if (lseek(fd, 10000000, SEEK_SET) == -1) {
            perror("lseek error\n");
            exit(EXIT_FAILURE);
      }
      if (write(fd, "kaan", 4) == -1) {
            perror("write error");
            exit(1);
      }
      close(fd);
      return 0;
}
```

STDIN_FILENO, STDOUT_FILENO ve STDERR_FILENO Sembolik Sabitleri

Bir process çalıştırıldığında dosya tablosunda default olarak üç dosya betimleyicisinin bulunduğunu belirtmiştik. Bunlar stdin, stdout, stderr dosyalarına ilişkin betimleyicilerdir. Bunlara ilişkin betimleyiciler sırasıyla geleneksel olarak 0, 1, 2 değerlerindedir. Ancak bu değerler POSIX standartlarına göre zorunlu değerler değildir. Bu standart dosyalara ilişkin betimleyici değerleri unistd.h başlık dosyası içerisinde STDIN_FILENO, STDOUT_FILENO ve STDERR_FILENO biçiminde belirtilmişlerdir. Fakat 0, 1 ve 2 değerlerini doğrudan kullanan pek çok program kodu mevcuttur. Hemen hemen tüm POSIX sistemlerinde bu sembolik sabitler geleneksel değerlerindedir.

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

#define BUFLEN 1024

int main(int argc, char *argv[])
{
      char buf[BUFLEN];
```

```
        int fd;
        int n;

        if (argc != 2) {
                fprintf(stderr, "Wrong number of arguments...!\n");
                exit(EXIT_FAILURE);
        }

        if ((fd = open(argv[1], O_RDONLY)) == -1) {
                perror("open error");
                exit(1);
        }
        while (( n = read(fd, buf, BUFLEN)) > 0 )
                write(STDOUT_FILENO, buf, n);
        close(fd);
}
```

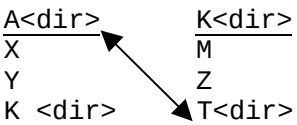
Sembolik Linkler

Bilindiği gibi ln komutu bir dosyanın aynı i-node elemanına ilişkin bir kopyasından oluşturmaktadır. Örneğin,

```
'ln a b'
```

gibi. ln komutu tipik olarak link sistem fonksiyonunu çağırmaktadır. UNIX/Linux terminolojisinde link denildiği zaman aynı i-node elemanını gören dosyalar anlaşılmaktadır. Bazen bu kavrama 'Hard Link' de denilmektedir.

Bir dizin için hard link işlemi yapılabilir mi? Bu işlem çok tehlikeli olabilir, çünkü dizin ağacını dolaşan algoritmalar sonsuz döngüye girebilirler; bu durum dosya sisteminin bozulmasına bile yol açabilmektedir. Örneğin aşağıdaki dizin yapısı içerisinde A ile T dizinleri bir hard link oluşturmuş olsunlar.



Burada A dizini dolaşılmaya çalışılırsa A ile T dizinleri aynı i-node elemanına sahip olduğu için sonsuz döngüsel bir durum oluşacaktır. Bu nedenle pek çok sistemde bir dizinin hard link işlemine sokulması ya tamamen yasaklanmıştır ya da yalnızca root kullanıcılarına serbest bırakılmıştır. İşte bu nedenlerden dolayı sembolik link kavramı da geliştirilmiştir. Sembolik link ln –s komutu ile yaratılır. Örneğin,

```
'ln –s a b'
```

gibi. Dizinler de her hangi bir kullanıcı tarafından sembolik link işlemine sokulabilmektedir. Sembolik link durumu ls –l komutunda açık olarak görülebilmektedir (erişim bilgilerinin en solundaki karakter de 'l' olarak gözükmektedir). Sembolik link dosyaları aslında ayrı bir dosyadır. Ancak başka bir dosyayı referans etmektedir. Sembolik link dosyaları bazı işlemlere sokulduğunda sembolik link dosyasının kendisi değil, onun referans ettiği dosya işlem görmektedir. Örneğin stat fonksiyonu ile bir sembolik link dosyasının bilgileri alınacak olsa sembolik link dosyaların kendi bilgisi değil, onun referans ettiği dosyaların bilgileri

alınmaktadır. Bir dizinin sembolik link'i oluşturulduğunda artık bu dizin sistem için bir dizin görünümünde olmaz, normal bir dosya görünümünde olur. Tabii dizin dolaşma işlemi de stat fonksiyonu ile yapılmamalıdır. Bunun için lstat isminde başka bir fonksiyon bulundurulmaktadır. lstat fonksiyonunun stat fonksiyonundan normal dosyalar için hiç bir farkı yoktur. Ancak sembolik link dosyalarında lstat fonksiyonu sembolik link dosyasının referans ettiği dosyanın bilgilerini değil, sembolik link dosyasının kendi dosya bilgilerini elde eder. lstat fonksiyonu bir sembolik link içeren dizin dosyası ile karşılaştığında bunu bir dizin dosyası olarak görmez. Bütün bu anlatılanların özeti şudur: *Dizin ağacını dolaşan uygulamalarda stat değil, lstat fonksiyonu kullanılmalıdır. Eğer stat fonksiyonu kullanılırsa sembolik link işlemine tutulmuş dizinlerde yine sonsuz döngü oluşmuş olur.* Bazı önemli sistem fonksiyonlarının sembolik link dosyalarına karşı davranışları şöyledir:

Fonksiyon İsmi	**Kendi Dosyası mı? Referans Ettiği Dosya mı?**
lstat	Kendi dosyası
open	referans ettiği dosya
remove	Kendi dosyası
rename	Kendi dosyası
unlink	Kendi dosyası
access	Referans ettiği dosya

Sembolik link oluşturmak için symlink fonksiyonu kullanılır.

```
int symlink(const char *actualpath, const char *sympath);
```

Fonksiyon başarılıysa 0 değerine, başarısızsa –1 değerine geri döner.

Dosyanın Tarih ve Zaman Bilgileri

Bir dosyanın ya da dizinin tarih/zaman bilgisi olarak stat ya da lstat fonksiyonu ile üç tür bilgi çekilmektedir.

```
st_atime
st_mtime
st_ctime
```

st_atime dosyanın data bilgilerine son erişim zamanını belirtir. Yani dosya üzerinde herhangi bir read ya da write işlemi yapıldığında bu değer değişir.
st_mtime dosyanın datalarında yapılan son değişiklik zamanını verir. Yani dosyaya write işlemi uygulandığında bu bilgi güncellenir.
st_ctime dosyanın dataları üzerinde değil, i-node bilgileri üzerinde değişiklik yapıldığında güncellenir. Örneğin dosyanın dataları üzerinde işlem yapmayıp yalnızca i-node ile ilgili işlem yapan `link` fonksiyonu, `utime` fonksiyonu bu zaman bilgisini günceller.
i-node bilgilerine erişim ile ilgili özel bir bilgiye gerek duyulmamıştır.

Dosya Erişim Haklarının Değiştirilmesi

chmod fonksiyonu bir dosyanın erişim haklarını değiştirmek için kullanılır. Ancak dosyanın erişim haklarının değiştirilebilmesi için bu fonksiyonu çağıran process'in etkin kullanıcı ID'sinin root ya da değiştirilecek dosyanın kullanıcı ID'si ile aynı olması gerekir.

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

Fonksiyonun birinci parametresi değiştirilecek dosyanın ismi, ikinci parametresi ise yeni erişim bilgileridir. Fonksiyon başarılı ise 0 değerine, başarısızsa –1 değerine geri döner. Dosyanın erişim bilgilerini değiştirmek için chmod isimli shell komutu da kullanılabilir.

```
chmod 666 a.dat ↵
```

password ve group Dosyaları Üzerinde İşlemler

Önceden de belirtildiği gibi geleneksel olarak UNIX/Linux sistemlerinde yaratılan bir kullanıcıya ilişkin pek çok değerli bilgi bir password dosyasında saklanır. Geleneksel olarak password dosyası /etc/passwd ismi ile bulunur ve geleneksel olarak bu dosya text tabanlı bir dosyadır. POSIX standartlarında password dosyasının genel yapısı belirtilmemiştir. Yalnızca olması gereken minimum bilgilerden bahsedilmiştir. Ancak klasik olarak password dosyası her bir satırın 7 parçadan oluştuğu kayıtlar halindedir. Her satırdaki 7 parça birbirlerinden ':' ile ayrılmıştır. Örneğin "a:b:c:d:e:f:g" gibi. password dosyasının 7 bölümü genellikle şunlardır:

1) *Kullanıcı ismi:* Kullanıcı giriş ismi
2) *Şifrelenmiş password bilgisi:* Yeni kullanıcı yaratılırken bu bölüm boş geçilebilir. Bazı sistemlerde burada x harfi varsa şifrelenmiş password bilgisinin başka bir dosyada olduğu anlamına gelir (geleneksel olarak /etc/shadow).
3) *Kullanıcı ID değeri:* Sistemin kullanıcıları tespit etmek için gerçek kullandığı değer kullanıcı ID değeridir. Kullanıcının ismi okunabilirlik amacıyla düşünülmüştür. Örneğin stat fonksiyonundan elde edilen değer dosyanın kullanıcı ID değeridir. Kullanıcının ID değeri bilindiğinde kullanıcının ismini elde edebilmek için passwd dosyasına başvurmak gerekir. Bu bilgi başka bir biçimde elde edilemez. Tipik olarak ls komutu dizin içerisindeki dosyaların isimlerini readdir fonksiyonu ile alır, bu isimler ile stat fonksiyonunu çağırır, struct stat yapısından kullanıcı ID değerini elde eder, sonra passwd dosyasına başvurarak kullanıcı ismini elde eder ve listeleme işleminde bu ismi kullanır.
4) *Grup ID değeri:* Her kullanıcı bir gruba ilişkin olur, gruplar da sistem tarafından ID değerleri ile kullanılırlar. Bu kısımda kullanıcının ilişkin olduğu grubun ID değeri bulunmaktadır. Grup ID değeri bilindiğinde grubun ismi group dosyasından elde edilir.
5) *Açıklama bilgisi:* Bu bölüm tamamen açıklama bilgisi içerir.
6) *cwd:* Bu bölümde çalıştırılacak olan shell process'inin varsayılan dizini belirtilir. Bir kullanıcı yaratıldığında genellikle buradaki bilgi /home/kullanıcı ismi biçimindedir. Yani geleneksel olarak her kullanıcının bir ismi vardır, bu isim de /home dizininin altında bir dizine karşılık gelir. Aslında teknik olarak buradaki bilgi shell process'inin varsayılan dizin (current working directory) bilgisidir. Aslında varsayılan dizin kavramı kullanıcıya özgü bir kavram değil, process'e özgü bir kavramdır. login programı 7. bölümde belirtilen shell programını çalıştırır ve shell process'inin geçerli dizinini burada belirtilen dizin yapar. Yani kullanıcı shell programına düştüğünde o andaki geçerli dizin bu bölümde belirtilen dizin olacaktır. UNIX/Linux sistemlerinde process'in geçerli dizini fork işlemi sırasında yeni yaratılan process'e doğrudan geçirilmektedir.

7) *shell:* Burada login programının çalıştıracağı shell programının path ismi belirtilmektedir (Sisteme login olunduğunda bir shell programının çalıştırılması zorunlu değildir).

login Programı Ne Yapar?

login programı tipik olarak şunları yapar:

1) Kullanıcı ismi ve password sorma işlemini yapar.
2) /etc/passwd ve /etc/shadow dosyalarına başvurarak doğruluğunu araştırır.
3) password doğruysa passwd dosyasında yedinci bölümde bulunan shell process'ini çalıştırır.
4) shell process'inin kullanıcı ID değerini kullanıcının ID değeri olarak, grup ID değerini kullanıcının ilişkin olduğu grup ID değeri olarak, geçerli dizin bilgisini passwd dosyasında belirtilen 6. bölüm olarak oluşturur.
5) shell üzerinden bir program çalıştırıldığında bu bilgiler çalıştırılan programa aktarıldığından çalıştırılan process'in kullanıcı ID değeri, grup ID değeri ve geçerli dizini aynı olacaktır.

Process'in Geçerli Dizini

Geçerli dizin process'e özgü bir bilgidir ve process tablosunda tutulur. Bu bilgi default olarak fork işlemi sırasında, yani bir process'in çalıştırılması işleminde process'i çalıştıran process'ten default olarak aktarılmaktadır. Yani shell'den bir program çalıştırdığımızda o programın geçerli dizini default olarak shell process'ininki olacaktır, shell'inki de passwd dosyasından alınarak belirlenmektedir. Aslında bir dizinde olmak gibi bir kavram yoktur. shell programları process'in geçerli dizinini sanki bir dizin içerisindeymiş gibi göstermektedir. shell üzerinden cd işlemi uygulandığında aslında shell programı chdir sistem fonksiyonu ile process'in geçerli dizinini değiştirir, sonra bunu bir prompt eşliğinde kullanıcıya gösterir.

passwd Dosyası Üzerinde İşlemler

password dosyalarının genel formatı sistemden sisteme değişebilmektedir. Programcı password dosyası içerisinden bilgi almak isteyebilir. Örneğin bir dosyanın kullanıcı ID'si bellidir (örneğin lstat fonksiyonu ile alınmıştır). Programcı kullanıcı ID'si yerine dosyanın kullanıcı ismini yazdırmak isteyebilir. Bu durumda password dosyasına başvurmak zorundadır. Pek çok UNIX/Linux sisteminde password dosyaları yukarıda belirtilen text formatına sahiptir. Ancak POSIX standartlarında bunun bir garantisi yoktur, bu nedenle password dosyası ile ilgili çeşitli işlemleri yapan taşınabilir bir grup fonksiyon bulundurulmuştur.

getpwuid ve getpwnam Fonksiyonları

```
struct passwd *getpwuid(uid_t uid);       //sys/types.h
struct passwd *getpwnam(const char *name);
```

Bu fonksiyonlar kullanıcı ID veya kullanıcı ismine göre password dosyasında arama yapar ve kişinin password bilgilerini struct passwd isimli bir yapı içerisine doldurur. Geri verilen adres static yerel bir nesnenin adresidir, yani fonksiyon her zaman aynı adresi geri vermektedir.

```
struct passwd {
      char *pw_name;
      char *pw_passwd;
      uid_t pw_uid;
      gid_t pw_gid;
      char *pw_gecos;
      char *pw_dir;
      char *pw_shell;
};
```

Fonksiyonların prototipleri ve struct passwd yapısı pwd.h başlık dosyası içerisinde bildirilmiştir.

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
      struct passwd *passi;

      if (argc != 2) {
            fprintf(stderr, "wrong number f arguments!...\n");
            exit(1);
      }

      passi = getpwnam(argv[1]);

      if (passi == NULL) {
            fprintf(stderr, "Cannot get password info\n");
            exit(1);
      }

      printf("username: %s\n", passi->pw_name);
      printf("user id: %d\n", passi->pw_uid);
      printf("Group id: %d\n", passi->pw_gid);
      printf("Home directory: %s\n", passi->pw_dir);
      printf("Shell prog: %s\n", passi->pw_shell);

      return 0;
}
```

password Dosyasındaki Tüm Kayıtların Elde Edilmesi

password dosyasında tıpkı dizinlerde olduğu gibi gezinilerek tüm kayıtlar elde edilebilir. Bunun için aşağıdaki 3 POSIX fonksiyonu kullanılır.

```
void setpwent(void);
struct passwd *getpwent(void);
void endpwent(void);
```

Programcı bir kere setpwent fonksiyonu ile başlangıç işlemini yapar, sonra döngü içerisinde getpwent fonksiyonunu çağırır. En sonunda da endpwent fonksiyonu ile işlemi bitirir. Örneğin getpwnam fonksiyonu tipik olarak bu fonksiyonlar kullanılarak yazılmıştır. Bu fonksiyon aşağıdaki gibi yazılabilir.

```
struct passwd *getpwnam(const char *name)
{
      struct passwd *passi;

      setpwent();
      while((passi = getpwent()) != NULL) {
            if (!strcmp(name, passi->pw_name))
                  break;
      }
      endpwent();

      return passi;
}
```

group Dosyası Üzerinde İşlemler

Normal olarak her kullanıcının gerçek tek bir grubu vardır. Bir kullanıcının hangi gruba ilişkin olduğu passwd dosyasında grubun ID değeri ile bulunmaktadır. Zaten struct passwd yapısında bu bilgi de bulunmaktadır. Gruplar hakkındaki bilgiler (örneğin grupların isimleri, gruplar içerisinde hangi kullanıcıların bulunduğu, grubun password bilgisi gibi) tipik olarak bir grup dosyasında tutulur. Tipik olarak UNIX/Linux sistemlerinde bu dosya /etc/group dosyasıdır. Grup dosyası tipik olarak grup ismi, ID değeri, grup password'ü ve o grupta hangi kullanıcıların olduğunu belirten bilgilerden oluşur. Bu dosya da tipik olarak her bir kayıt bir satırda olacak biçimde text formatındadır. Grup dosyasında tek tek satırlar incelenerek bir kişinin ek olarak hangi gruplarda bulunduğu da tespit edilebilir. Bir kişinin gerçek grubu bir tanedir ve password dosyasında belirtilir.

Tıpkı password dosyasında olduğu gibi grup dosyası üzerinde de arama bulma işlemleri yapan fonksiyonlar vardır.

```
struct group *getgrgid(gdi_t gid);
struct group *getgrnam(const char *name);
```

Fonksiyonlar "grp.h" başlık dosyası içerisinde bildirilmiştir. Bunların yanı sıra grup dosyasını dolaşan aşağıdaki fonksiyonlar da vardır.

```
void setgrent(void);
struct *getgrent(void);
void endgrent(void);
```

Kullanıcıların Yaratılması

Yeni bir kullanıcı yaratabilmek için değişik sistemlerde değişik yararlı programlar vardır. Örneğin, `adduser` veya `makeuser` gibi özel programlar sistem yöneticisine çeşitli soruları sorarak passwd ve group dosyalarını da güncelleyerek yeni bir kullanıcı yaratırlar. Ancak sistem yöneticisi password ve group dosyalarının formatını biliyorsa doğrudan da manuel olarak kullanıcı yaratabilir. Manuel oluşturma işleminde password alanı boş geçilebilir, daha

sonra passwd programı ile kullanıcı için şifre atanabilir. passwd programı iki biçimde kullanılabilir:

1) passwd ↵
2) passwd <kullanıcı_ismi> ↵

Çalışabilen Dosyanın Bölümleri ve UNIX/Linux Sistemlerinde Process'in Bellek Alanı

Tipik bir çalışabilen dosya temelde 5 bölümden oluşur.

Header
.text // Kod
.data // Data (ilk değer verilmiş)
.bss // Data (ilk değer verilmemiş)
.stack // Stack

Bu bölümler şüphesiz çalışabilen dosyanın formatı ile ilgilidir. Örneğin MZ, ELF, a.out formatlarında tipik olarak bölümler yukarıdaki gibidir. Programın kod bölümünde fonksiyonların makine komutları tutulur (bu bölüm genellikle .text biçiminde isimlendirilmektedir).

.data bölümü static ömürlü nesnelerin tutulduğu bölümdür. .data bölümü genellikle ikiye ayrılır; .data ismi ile belirtilen bölüm ilk değer verilmiş static nesneleri, .bss biçiminde isimlendirilen alan ise ilk değer verilmemiş static nesneleri içerir. Yerel değişkenler tipik olarak stack bölümünde tutulurlar. İlk değer verilmemiş static ömürlü nesnelerin ayrı bir bölümde tutulmasının nedeni çalışabilen dosyada gereksiz yer kaplamasını engellemektir. Örneğin,

```
int g_a[1000000];
```

bu global dizi .data bölümünde tutulsaydı, çalışabilen dosya en azından 4MB olurdu. Halbuki .bss bölümünde bu dizi bilgisi "burada şu uzunlukta dizi var" biçiminde yazılmaktadır. Şüphesiz .bss bölümü açılıp yüklendiğinde yükleyici tarafından bu bölüm sıfırlar ile doldurulmaktadır. .bss bölümü bazı sistemlerde yükleyici tarafından değil, derleyicinin başlangıç kodu tarafından sıfırlanmaktadır.

UNIX/Linux sistemelrinde tipik olarak process'in bellek alanı şöyledir:

Header
.text
.data
.bss
Heap
Stack
System

Görüldüğü gibi stack bölümü boşuna büyürse heap bölgesiyle çakışır ve probleme yol açar. Bazı dosya formatlarında ve sistemlerde stack bölgesinin en fazla ne kadar büyütüleceği belirlenmektedir. Örneğin Win32 sistemlerinde stack uzunluğu PE formatında belirlenir ve

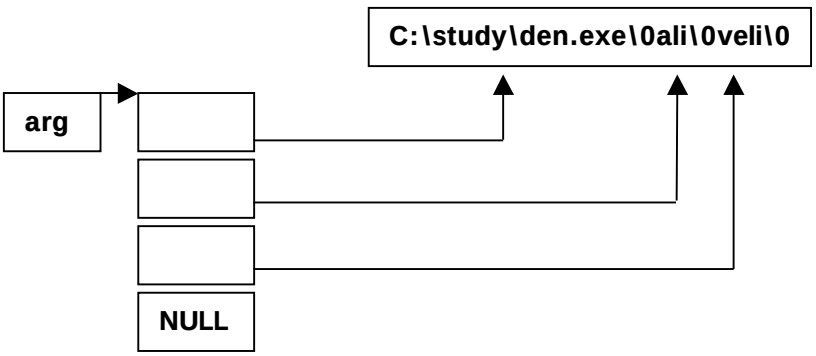
stack bu uzunluğu çalışma zamanında aşamaz. Eğer aşarsa koruma mekanizmasından faydalanılarak program sonlandırılır. Bazı sistemlerde (örneğin DOS ve UNIX sistemlerinde) stack sınırlanmaz, ancak programın çalışma zamanı sırasında heap bölgesiyle çakışma sonucunda program çökebilir. Tabii işletim sistemleri koruma mekanizmasından faydalanılarak stack ile heap bölgesinin çakışmasını kolaylıkla tespit ederler. Yani stack bölgesinin serbest büyüyebildiği sistemlerde stack çakışması yine tespit edilebilmektedir.

Komut Satırı Argümanları ve Çevre Değişkenleri

Bir program çalıştırılırken program isminin yanına yazılan yazılara komut satırı argümanları denir. İşletim sistemi komut satırı argümanlarını yaratılan process'in bellek alanına taşır. Örneğin DOS'ta komut satırı argümanları PSP'nin 128'inci byte'ından itibaren komut satırındaki haliyle yerleştirilir. Genellikle işletim sistemlerinde komut satırı argümanları C'deki argv'de olduğu gibi geçirilmez. C'deki main fonksiyonuna geçirilen argv derleyicilerin giriş kodu tarafından düzenlenmektedir. Tipik olarak derleyicilerin giriş kodu komut satırı argümanlarını işletim sisteminin yerleştirdiği adresten alarak her argümanın sonuna NULL karakter yerleştirmek suretiyle başka bir alana taşırlar, NULL karakterlerle ayrılmış yazıların başlangıç adreslerini bir gösterici dizisine yerleştirirler. Bu gösterici dizisinin adresini de argv olarak main fonksiyonuna geçirirler. Örneğin

```
C:\study> den ali veli↵
```

işlemiyle derleyicinin giriş kodu argv değerini aşağıdaki gibi bir düzenleme ile bulunur.



C ve C++ standartlarına göre main fonksiyonunun parametrik yapısı aşağıdakilerden biri olmak zorundadır:

```
int main(void);
int main(int argc, char *argv[]);
```

Görüldüğü gibi aslında main fonksiyonunun geri dönüş değeri standartlara göre void olmamalıdır. Ancak void olmasında yaygın derleyicilerin hiçbirisi için bir sakınca yoktur. Zaten bu derleyicilerin kendi örneklerinde bile main fonksiyonunun geri dönüş değerinin void alındığı görülmektedir. main fonksiyonunun geri dönüş değeri void alındığında derleyicilerin uyarı vermemesi için belirli bir değer ile geri dönülmesi gerekir. Ayrıca argv[argc]'nin NULL gösterici olması da garanti edilmiştir.

Anahtar Notlar: C'de NULL karakter '\0' ile temsil edilir. Hangi karakter tablosu kullanılırsa kullanılsın, sayısal değerinin sıfır olması garanti edilmiştir. Ancak NULL gösterici standartlara göre sıfır adresi olmak zorunda değildir. Standartlara göre NULL gösterici değeri stdio.h dosyası içerisinde NULL sembolik sabiti ile define edilmelidir. Bu durumda NULL sembolik sabiti güvenli olarak NULL gösterici olarak kullanılabilir. Pek çok derleyicide NULL sembolik sabiti

```
#define NULL ((void *) 0)
```

*biçiminde ifade edilmiştir. Yine standartlara göre NULL gösterici türünden sabit oluşturmak için yalnızca 0 ya da ((void *) 0) kullanılabilir. Bir göstericiye düz bir 0 atamak, ona NULL gösterici atamak anlamına gelir, eğer NULL gösterici değeri 0'ın dışında bir değerse derleyici göstericiye o değeri atamak zorundadır. Yani aslında NULL göstericinin adres değeri 0'ın dışında olabilmektedir, ama programda 0 olarak kabul edilmektedir. C standartlarına göre*

```
if (p) {
      //...
}
```

gibi bir karşılaştırma p'nin içinde NULL gösterici olup olmadığını anlamak için taşınabilirdir. Ancak NULL göstericinin 0 dışında bir değere sahip olabilmesi standartlarda esneklik amacı ile düşünülmüştür. Yaygın bütün derleyicilerde NULL gösterici 0 değeridir. Aşağıdaki işlemlerin hepsi standartlara göre taşınabilirdir.

```
if (p != NULL) {
}
if (p != 0) {
}
if (p != (void *) 0) {
}
```

Çevre değişkenleri belirli bir değişken ismi ile bir yazıyı eşleştiren bir mekanizma oluşturur. Aslında çevre değişkenleri process'lerin handle alanlarında saklanır. Yani process'e özgü bir bilgidir. Process'in çevre değişkenleri UNIX/Linux sistemlerinde fork ve exec işlemi sırasında yeni yaratılan process'e aktarılmaktadır. Örneğin shell üzerinden bir program çalıştırıldığında shell programının çevre değişkenleri çalıştırılan programa geçer. En ünlü çevre değişkeni path değişkenidir. UNIX/Linux shell programlarında shell üzerinde `env` komutu ile shell'e ilişkin çevre değişkenleri görülebilir. DOS'ta `env` komutu yerine `set` komutu kullanılmaktadır. Shell üzerinde belirli bir çevre değişkeninin değerini görmek için

```
echo $<çevre_değişkeni_ismi>
```

yapılır. Aynı işlem DOS'ta

```
set <çevre_değişkeni>
```

biçiminde kullanılır. Programcı çevre değişkenlerinden çeşitli biçimlerde faydalanabilir. En yaygın durum belirli bir dosyanın belirli bir dizinde arandığı durumlardır. Programcı belli bir çevre değişkeninin değerine bakar ve dosyayı orada arar. Örneğin DOS'ta çalışan C

derleyicilerinin hemen hepsi default include dizinini tespit edebilmek için çevre değişkenleri kullanırlar.

getenv Fonksiyonu

Bu fonksiyon standart bir C fonksiyonudur, belirli bir çevre değişkeninin değerini almak için kullanılır.

```
char *getenv(const char *name);
```

Fonksiyon parametre olarak çevre değişkeninin ismini alır, geri dönüş değeri olarak çevre değişkeninin değerinin bulunduğu static alanın adresini verir. Eğer ilgili çevre değişkeni tanımlanmamışsa fonksiyon NULL değeri ile geri döner. Fonksiyonun prototipi stdlib.h içerisindedir. UNIX/Linux sistemlerinde çevre değişkenlerinin büyük harf, küçük harf duyarlılığı vardır, DOS'ta yoktur. UNIX/Linux sistemlerinde PATH çevre değişkeninin içeriği `dizin1:dizin2:dizin3...` biçiminde, DOS ve Windows sistemlerinde `dizin1;dizin2;dizin3...` biçimindedir.

```
/* enviroment.c */

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

int main(void)
{
      char *p;

      p = getenv("PATH");

      if (p == NULL) {
            fprintf(stderr, "Böyle bir değişken yok\n");
            exit(1);
      }

      puts(p);
}
```

UNIX/Linux sistemlerinde standart bazı çevre değişkenleri vardır. Örneğin HOME kullanıcının dizinini, USER kullanıcı ismini belirtmektedir. Belirli bir çevre değişkeninin değerini değiştirmek ya da yeni bir çevre değişkeni eklemek için C standartlarında ve POSIX standartlarında bir fonksiyon yoktur. Ancak `putenv` isimli fonksiyon popüler UNIX ve Linux sistemlerinde desteklenmektedir.

putenv Fonksiyonu

Bu fonksiyon belirli bir çevre değişkenini eklemek için ya da değerini değiştirmek için kullanılır.

```
int putenv(const char *name);
```

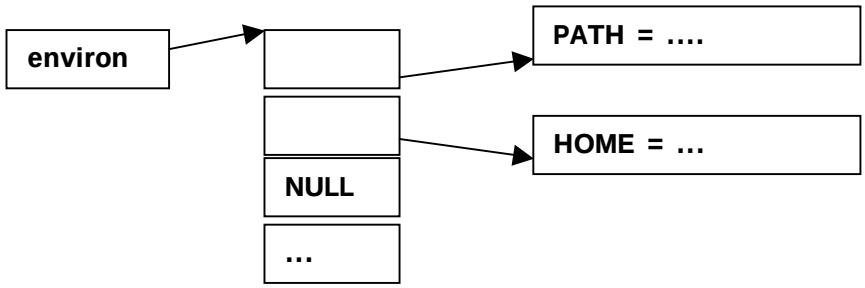
Fonksiyon parametre olarak `"isim=değer"` yazısını alır. Başarılı ise 0 değerine, başarılı değilse 0 dışı bir değere geri döner.

Tüm Çevre Değişkenlerinin Elde Edilmesi

getenv fonksiyonu yalnızca bir tek çevre değişkeninin değerini verir. Ancak programcı bazen process'in tüm çevre değişkenlerini elde etmek isteyebilir. Bunun için POSIX'de

```
extern char **environ;
```

isimli global bir değişken bulundurulmuştur. Şüphesiz environ değişkeninin gördüğü gösterici dizisi derleyicinin başlangıç kodu tarafından oluşturulmaktadır. Programcı extern bildirimi ile bu değişkeni kullanabilir.




```
for (i = 0; environ[i] != NULL; ++i)
      puts(environ[i]);
```

Şüphesiz `putenv` fonksiyonu environ göstericisinin gösterdiği yerdeki diziyi de güncellemektedir.

Standart C Fonksiyonlarının Kullandığı Tamponlama Teknikleri

Standart C fonksiyonları bilindiği gibi process düzeyinde (kullanıcı düzeyinde) tamponlama yapmaktadır. Standart C fonksiyonları ve standart C'nin dosyalama mekanizması üç tamponlama mekanizmasını kullanabilecek biçimde tasarlanmıştır.

1) Tam tamponlama (Full buffering)
2) Satır tamponlama (Line buffering)
3) Tamponlama yok (No buffering)

Tam tamponlama normal bilinen tamponlama sistemidir. Yani belirli bir büyüklükte tampon alınır. Yazma fonksiyonları önce tampona yazar. Tampon dolmuşsa işletim sisteminin sistem fonksiyonları çağırılarak yazma gerçekleştirilir. Okuma sırasında da önce okunacak bilginin tamponda olup olmadığına bakılır. Tampondaysa doğrudan tampondan alınır, değilse işletim sisteminin sistem fonksiyonları çağırılarak tampon tazelenir. Örneğin `fflush` fonksiyonu tipik olarak tampondaki bilgiyi diske yazmaktadır. Kütüphaneyi yazanların kullandığı default tampon uzunluğu stdio.h içerisinde `BUFSIZ` sembolik sabiti ile temsil edilir.

*Satır tamponlama*da tamponlama satır temelinde yapılır. Yani '\n' karakteri görülene kadar bilgi tamponda tutulur, '\n' karakteri görüldüğünde tazeleme yapılır. Nihayet standart C fonksiyonları hiç tamponlama kullanmayabilir.

Peki stdin, stdout ve stderr gibi önemli dosyalar için tamponlama mekanizması default olarak nasıldır? Standartlara göre stderr hiç bir zaman tam tamponlamaya sahip olamaz. Ancak stdin ve stdout eğer terminal ve klavye gibi karşılıklı etkileşimli bir aygıta yönlendirilmişse tam tamponlamaya sahip olamaz. Aynı zamanda bu dosyalar tam tamponlama içeriyorlarsa klavye ve ekran gibi karşılıklı etkileşimli aygıtlarla ilişkili olamaz. Özetle standartlara göre stdin ve stdout başlangıçta tam tamponlamaya sahip olamaz. Yani satır tamponlamasına sahip olabilir ya da hiç tamponlama kullanmayabilir. Böylece örneğin printf'in içine '\n' koyulursa ekrana çıkması garantidir. Ancak uygulamada derleyicileri yazanlar genellikle stdin için satır tamponlaması kullanırlar, stdout için ise tamponlama kullanmazlar. Böylece hemen her derleyicide aşağıdaki yazının ekrana çıkması sağlanmaktadır.

```
printf("Deneme");
```

Ancak stdin, stdout ve stderr daha sonra başka aygıtlara yönlendirilebilir. Bu durumda yönlendirilen aygıt örneğin bir dosya ise etkileşimli bir aygıt olmadığından tam tamponlama kullanılacaktır. Popüler derleyicilerin hemen hepsi stdin için default olarak satır tamponlaması kullanmaktadır. Yani örneğin getchar ile bir karakter almaya çalıştığımızda enter tuşuna basılana kadar tüm satır alınır ve tamponlanır (Malesef C'de getch, kbhit gibi standart fonksiyonlar yoktur. Ancak derleyicilerin hemen hep ek fonksiyonlar olarak çeşitli isimlerle bulundurmaktadır. Bu fonksiyonların standart C fonksiyonlarının kullandığı tampon fonksiyonları ile bir alakası yoktur).

Default Durum

sdtin **à** Satır tamponlaması ya da tamponlama yok (tam tamponlama olamaz)
stdout **à** Satır tamponlaması ya da tamponlama yok (tam tamponlama olamaz)
stderr **à** tamponlama olamaz

Yaygın durum

stdin **à** Satır tamponlama
stdout **à** Tamponlama yok
stderr **à** Tamponlama yok

Eğer stdin ve stdout karşılıklı etkileşimli bir aygıta yönlendirilmişse (yani dosyaya) o zaman tam tamponlamaya sahip olmak zorundadır. stderr nereye yönlendirilirse yönlendirilsin hiç bir zaman tam tamponlamaya sahip olmaz.

Standart Dosya Fonksiyonlarının Yazılmasına İlişin Tasarımsal Bilgiler

Bilindiği gibi dosya bilgi göstericisi olarak kullanılan FILE yapısı içerik bakımdan standart değildir. Tipik olarak bu yapının içerisinde tamponlama işlemlerini sağlamak için bilgiler tutulmaktadır. Örneğin tipik olarak bu yapı en az şu elemanları içermelidir:

```
typedef struct {
      int fdes;
      size_t bufsize;
      void *pBuf;
      size_t pos;
      long filepos;
      /* Diğerleri ... */
```

```
} FILE;
```

Şimdi temel fonksiyonları inceleyelim.

1) *fopen Fonksiyonu:* fopen işletim sisteminin sistem fonksiyonunu kullanarak dosyayı parametreleri ile belirtilen modda açar, sonra FILE türünden bir yapı alanı tahsis ederek içini gerekli bilgilerle doldurur. fopen fonksiyonunun geri verdiği adres static bir alana mı, dinamik bir alana mı sahiptir? Genellikle tasarımcılar FILE türünden static bir yapı dizisi alıp onun bir elemanının adresiyle geri dönerler. Böylece temel bilgiler boş heap alanına gereksinim duymazlar.
2) *fgetc Fonksiyonu:* Bu fonksiyon tampondan bir karakter alır. En taban fonksiyondur. Eğer bilgi tamponda yoksa yeni bir bilgi grubu tampona çekilecektir. Genellikle derleyiciler tampon ile aygıt arasında alış verişi sağlayan bir fonksiyon bulundururlar, eğer bilgi tamponda yoksa bu fonksiyonu çağırırlar.
3) *fgets, fread gibi Fonksiyonlar:* Bu fonksiyonlar fgetc fonksiyonunu kullanarak bilgiyi tek tek kullanıcının belirttiği adrese aktarabilirler. Eğer daha hızlı bir çalışma isteniyorsa fgetc fonksiyonu yerine transfer doğrudan yapılabilir.
4) *fclose Fonksiyonu:* Bu fonksiyon işletim sisteminin close fonksiyonunu çağırarak dosyayı kapatır. Dosya kapatıldıktan sonra artık FILE yapısındaki bilgiler geçersiz olur.
5) *fseek Fonksiyonu:* Bu fonksiyon tampon tazelemesine yol açabilir.
6) *feof Fonksiyonu:* Bu fonksiyon dosyanın sonuna gelip gelmediğimizi çeşitli mekanizmalar ile anlayabilir. Bazı tasarımlarda FILE yapısı içerisinde bir flag tutulmuştur. feof fonksiyonu da bir makro olabilir.

Standart C Fonksiyonlarının Tamponlama Biçimlerinin Değiştirilmesi

Tamponlamadaki default durumlar daha sonra değiştirilebilmektedir. Bunun için setbuf ve setvbuf fonksiyonları kullanılır.

```
void setbuf(FILE *f, char *buf);
```

Bu fonksiyon dosya için kullanılacak tamponu set eder. Bu fonksiyonda programcının daha önceki tampon uzunluğu kadar (default `BUFSIZ` kadar) bir alanı tahsis etmiş olması gerekir. Bu fonksiyon dosya açıldıktan sonra hiç bir işlem yapılmadan uygulanmalıdır. Yani bu fonksiyonla yalnızca kullanılan tamponun bölgesini değiştirebiliriz. Şüphesiz fclose işlemi sırasında standart kütüphane fonksiyonu kullanıcının oluşturduğu alanı free hale getirmez.

```
f = fopen(“a.dat”, “r”);
//...
char *pMyBuf = malloc(BUFSIZ);
setbuf(f, pMyBuf);
```

fclose işlemi sırasında sistem kendi tamponunu kullanıyorsa ve bu tampon heap üzerinde yaratılmışsa free hale getirme işlemi yapılacaktır. setbuf fonksiyonunda tampon NULL olarak girilirse tamponlama kaldırılır.

```
int setvbuf(FILE *f, char *buf, int mode, size_t size);
```

Bu fonksiyon bütün tamponlama mekanizmasını değiştirebilmektedir. Fonksiyonun buf parametresi yeni tamponun adresini, size parametresi ise uzunluğunu belirtmektedir. Bu sayede istenilen uzunlukta tamponun kullanılması mümkün olabilmektedir. mode parametresi

```
_IOFBF
_IOLBF
_IONBF
```

değerlerinden birisi olabilir. Eğer `_IONBF` seçilmişse buf ve size parametreleri dikkate alınmamaktadır. Bu fonksiyon herhangi bir zaman çağırılabilir.

UNIX/Linux sistemlerinde Process'ler

Her işletim sisteminde bir process yaratıldığında sistem bu process'i izleyebilmek ve process işlemlerini yapabilmek için bir handle alanı oluşturur. Bu handle alanı şüphesiz koruma mekanizması tarafından korunmuş olan sistem alanında tahsis edilir. Process yaratan fonksiyonlar geri dönüş değeri olarak yeni yaratılmış process alanının handle değerini verirler. Process'in handle değeri bir tamsayı formatında olabildiği gibi (void *) biçiminde de olabilir. Win32 sistemlerinde process'in handle alanına "process database" denilmektedir. UNIX/Linux sistemlerinde bu alan için çeşitli isimler kullanılmaktadır. Şüphesiz sistem tüm process handle alanlarını bir veri yapısı içinde tutmalıdır. Örneğin tipik olarak sistemler şöyle stratejiler kullanırlar:

1) Process yaratıldığında process'in handle alanı kernel içerisinde organize edilen bir heap alanında tahsis edilebilir ve bu alanlar bağlı liste biçiminde tutulabilir. Dinamik tahsisat hız bakımından dezavantajlı olabilmektedir.
2) Kernel başlangıçta static düzeyde bir yapı dizisi biçiminde handle alanları için genel bir tahsisat yapmış olabilir. Bu durumda process yaratıldığında bu dizi üzerinde boş slot aranır ve böylece hızı yavaşlatan heap tahsisatının önüne geçilmiş olur. Tabii bu yöntemde de static alan üzerinde yine bağlı liste kullanılmalıdır.
3) Bu yöntem ilk iki yöntemin bir karışımı gibidir. Bu yöntemde process handle alanı process'in kernel stack'i üzerinde bulunur (kernel stack process bir sistem fonksiyonunu çağırdığında kapı yoluyla ring0'a geçildiğinde kullanılan stack bölgesidir). Bu sistem özellikle Linux 2.4 tarafından benimsenmiştir. Bu yöntemde iki fayda düşünülmüştür.

a) Process handle alanları sınırlı sayıda değildir.
b) Sistem fonksiyonları stack değişimi yapıldığı için ESP yazmacı üzerinde basit bir işlemle process handle alanına hızlı bir erişim yapabilir.

Process handle alanının oluşturulmasında temel performans ölçütleri:

1) Toplam process handle alanlarının sınırlılığı veya sınırsızlığı
2) Process handle alanına erişim kolaylığı

fork İşlemi

fork sistem fonksiyonu çalışmakta olan bir process'in bir kaç istisna durum dışında tamamen bir kopyasını oluşturur. fork sistem fonksiyonu ile yeni bir process oluşturulduğunda yeni oluşturulan process'e alt process (child process), bu process'i oluşturan process'e ise üst

process (parent process) denir. Bir process'in kullandığı bellek alanına ilişkin tüm bilgiler process handle alanında tutulur. Örneğin process'in bölümlerini oluşturan .text, .data, .bss gibi bellek alanları process handle alanında tutulmaktadır. Process'in sanal bellekteki yerinden başka process'in handle alanında aşağıdaki temel bilgiler tutulmaktadır.

- Çevre değişkenleri
- user id, group id, etkin user id, etkin group id
- O andaki geçerli dizin
- Ek grup id'leri
- Dosya tablosu
- ve diğerleri

fork işlemi sırasında özdeş bir process'in oluşması demek;

1) Process handle alanının bir kopyasından oluşturulması
2) Process'in bellek alanının bir kopyasından oluşturulması

demektir. Zaten bu iki bilgi neredeyse process kavramının tüm bilgilerini oluşturmaktadır. fork işlemi sırasında özdeş kopya çıkartılırken asıl önemli zaman kaybı üst process'in bellek alanının kopyalanması işlemine ilişkindir. Neyseki "copy on write" denilen sanal bellek yönteminde yalnızca üst process'in bellek alanının bir sayfası değiştirildiğinde gerçek bir kopyalama yapılır. "Copy on write" sistemi pek çok nedenden dolayı eski bazı sistemlerde yoktu. "Copy on write" özelliği gerçek kopyanın yalnızca gerektiğinde çıkartması anlamına gelir. Bu sistem Win32'de de aynı programdan ikinci kez çalıştırıldığında kullanılmaktadır.

fork işlemi sırasında yeni bir process yaratıldığına göre, yani yeni bir process handle alanı oluşturulduğuna göre yaratılan alt process'in de bir process handle değeri oluşmaktadır.

fork fonksiyonu yeni process yaratıldığında üst ve alt process'ler için ayrı değerler ile geri döner. Üst process'te fork yeni yaratılan alt process'in handle değeri ile geri döner. Alt process'te ise 0 ile geri döner. Akış fork içerisinde ikiye ayrılmaktadır. Hem üst process, hem de alt process fork işleminden sonra program kodunun aynı yerinden çalışmayı sürdürür. Yani her iki process'in akışı da fork fonksiyonundan çıkar. Ancak kod bir tanedir, fakat akış ikiye ayrılır. fork fonksiyonunu çağıran programcı fonksiyonun geri dönüş değerine bakarak hangi process'in hangi işlemleri yapacağına karar verir. fork fonksiyonunun sembolik biçimi şöyledir.

```
int fork(void)
{
      //    1) Process handle alanının kopyasından oluştur
      //    2) Bellek alanının kopyasından oluştur
      //    3) Yeni bir çizelge elemanı oluştur
      //       ve bunun başlangıcını NEWEXIT yap.

      return (yeni process'in handle değeri);

NEWEXIT:
      return 0;
}
```

fork fonksiyonu başarısız olabilir. Örneğin sistemin izin verdiği en yüksek process sayısı aşıldığında fork başarısız olacaktır. fork başarısız olduğunda –1 değerine geri döner. Bu

durumda fork 3 değere geri dönebilir. ‘–1’, ‘0’ ya da ‘> 0’ . Programcı bu üç değeri de kontrol ederek akışını yönlendirir.

Tipik Bir POSIX Sisteminde Process Sıralaması

UNIX ve Linux sistemleri arasında kendine özgü farklılıkları olsa da temel olarak bu türden bir sistem açıldığında kernel tarafından ilk yaratılan process `swapper` ya da `sched` isimli process’tir. Bu process boot işlemi sırasında yaratılır ve handle değeri (process ID değeri) 0’dır. Yine klasik bir UNIX sisteminde swapper tarafından yaratılan bir `init` process’i vardır. Bu process’in de ID değeri 1’dir. init process’i de sistem kapanana kadar faaliyet göstermektedir. Son olarak, tipik POSIX sistemlerinde özellikle sayfalama amacı ile kullanılan `pagedaemon` ya da `pageout` isimli bir process daha vardır. Bu process’in de ID değeri genellikle 2’dir. init process’i bundan sonra sistemde yaratılacak bütün process’lere ana process görevi yapmaktadır. Tipik olarak bir terminal yaratıldığında oluşturulan `getty` gibi bir process init process’i tarafından yaratılır. getty process’i terminal yaratıldığında genellikle `login` process’ini yaratır, login process’i de daha önce ele alındığı gibi `shell` process’ini oluşturur. Bu durumda tipik bir POSIX sistemindeki shell process’ine kadar olan process’lerin yaratılma süreci şöyledir:

POSIX sistemlerinde process’in handle değeri olarak process’in ID değeri kullanılır. Process’in ID değeri pid_t türü ile temsil edilir ve normal olarak int türündendir. fork fonksiyonun prototipi unistd.h başlık dosyası içerisinde aşağıdaki gibi bildirilmiştir.

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

pid_t türü diğer türlerde olduğu gibi sys/types.h dosyası içerisinde bildirilmiştir.

fork fonksiyonu tipik olarak yeni yaratılan process’in ID değerini elde etmek için process tablosunda önce boş bir giriş arar. Process tablosu process handle alanlarınından oluşan bir yapı dizisi gibi düşünülebilir. fork bu dizide ilk boş elemanı bulur ve yeni process’in ID değerini bu biçimde oluşturur. Tasarımda genellikle process handle alanları static bir dizi biçiminde tutulur, son tahsis edilen slot kernel içerisinde ayrı bir sayaçta saklanır. Boş slot arama işlemleri bu sayaçtan itibaren yapılır, process tablosu bittiğinde yeniden başa dönülür. Tabii POSIX sistemleri arasında process tablosunun organizasyonu bakımından önemli farklılıklar olabilir.

fork Fonksiyonunun Kullanımı

fork fonksiyonunun kullanımında iki kalıp söz konusudur. Birinci kalıp:

```
pid_t pid;

switch (pid = fork()) {
      case –1:
            fprintf(stderr, “Cannot fork..\n”);
            exit(1);
      case 0:
            // <alt process>
            break;
      default:
            // <üst process>
            break;
}
```

İkinci kalıp:

```
pid_t pid;

if ((pid = fork()) > 0) {
      // üst process
}
else if (pid == 0) {
      // alt process
}
else {
      fprintf(stderr, “Cannot fork...\n”);
      exit(1);
}
```

Bazı programcılar okunabilirliği arttırmak için ikinci kalıpta önce hata kontrolüne yönelirler.

```
pid_t pid;

if ((pid = fork()) == -1) {
      perror(“Cannot fork...\n”);
      exit(1);
}
if (pid == 0) {
      // <alt process>
}
else {
      // <üst process>
}
```

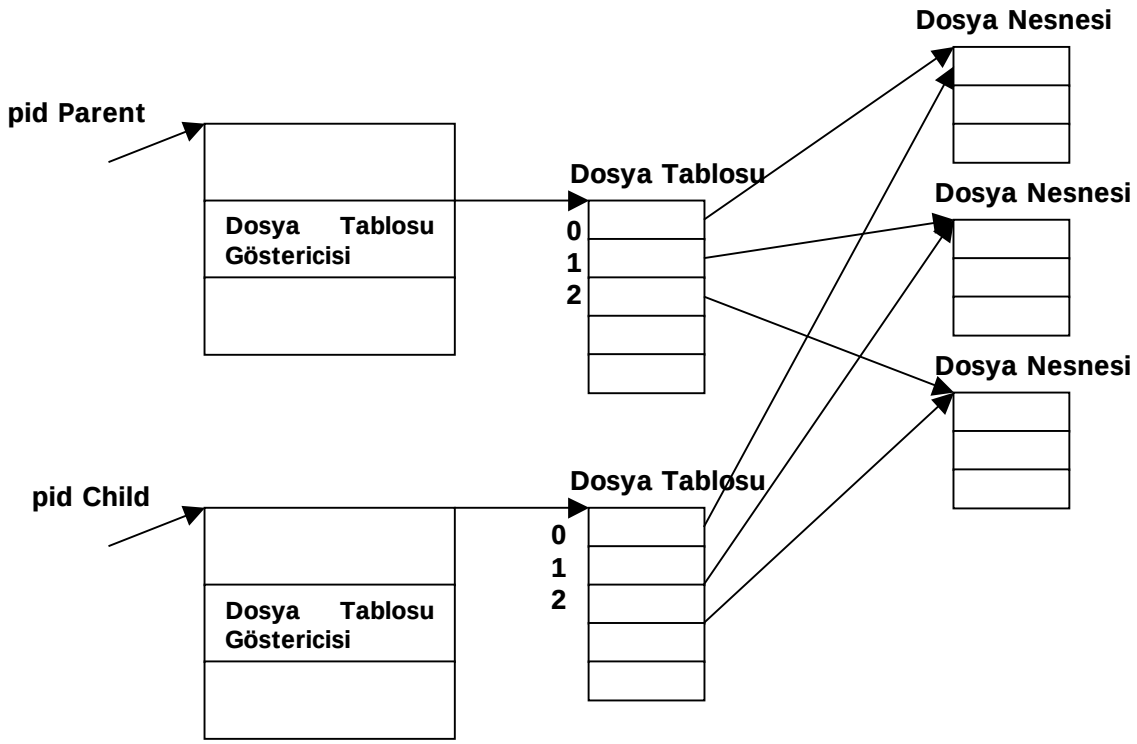
Aşağıdaki programda "process..." yazısı toplam 8 kez yazılır ve toplam 7 tane child process yaratılır.

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

      for (i = 0; i < 3; ++i)
            fork();

      printf("process...\n");

      return 0;
}
```

fork mekanizması eski sistemlerde kötüye kullanıma açıktı. Örneğin, normal bir kullanıcının process'i sürekli fork işlemleriyle yeni processler yaratıp sistemin process tablosunu doldurabiliyor, böylece diğer processler fork yapamaz hale geliyordu. Bu tür sistemlerde çökmenin asıl nedeni kernel'in kendisinin de doğal çalışma içerisinde fork yapamamasıdır. Modern sistemlerde bir process'in fork sayısı sınırlandırılmıştır. Process tablosunda yer tahsis eden fonksiyonlar (yani boş slot arayan fonksiyonlar) belirli bir process sayısını kernel için ayırırlar. Örneğin, process tablosunun toplam uzunluğu 32762 ise bunun 100 tanesi kernel için ayrılmış olabilir.

fork İşleminde Dosya Tablosunun Kopyalanması

fork işlemi sırasında process handle alanının alt process'e kopyalanmasıyla process'in kullandığı dosya tablosu da alt process'e kopyalanmaktadır. Bu durumda fork işlemi sırasında dosya tablolarının durumu şöyle olacaktır:



Görüldüğü gibi burada üst process ile alt process aynı dosya nesnelerini görür hale gelmiştir. Bu tür mekanizmalarda nesne sayacına gereksinim duyulur. Örneğin dosya nesnesinin içerisinde bir sayaç vardır, bir process her fork yaptığında sayaç 1 arttırılır, böylece alt process dosyayı kapattığında dosya nesnesi silinmez, sayacı 1 eksiltilir, sayaç 0'a düşmüşse dosya nesnesi silinir. Şüphesiz `dup` ve `dup2` gibi fonksiyonlar da dosya nesne sayacını arttırmaktadırlar. Dosya nesne sayacı close ile dosya kapatıldığında ya da program sonlandığında eksiltilmektedir.

Linux Kernel Kodlarına İlişkin Açıklamalar

Linux sistemlerinde fork, vfork, clone gibi sistem fonksiyonları kernel moda geçerek sys_fork, sys_vfork, sys_clone gibi fonksiyonları çağırırlar, bu fonksiyonlar da gerçek fork işlemini yapan do_fork fonksiyonunu çağırmaktadır. do_fork fonksiyonu linux/kernel/fork.c dosyası içerisindedir.

do_fork fonksiyonunda önce `alloc_task_struct` fonksiyonuyla yeni bir process handle alanı tahsis edilmiştir, `*p = *current;` işlemi ile üst process'in process handle alanı yeni yaratılan process handle alanına kopyalanmıştır. Daha sonra yeni yaratılan alt process'in process durumu oluşturulmuştur. Daha sonra yaratılan alt process için handle değeri tespit edilmeye çalışılmıştır.

```
p->pid = get_pid(...);
```

Artık bu işlemlerden sonra içerik kopyalaması işlemine geçilir. task_struct yapısı, yani process handle alanı çeşitli göstericiler içermektedir. Bu göstericiler tahsis edilmiş olan başka alanları gösterir, oysa *p = *current işleminden sonra bu göstericiler de aynı alanları gösterir hale gelirler. Bu durumda bu göstericilerin gösterdiği yerler de ayrıca tahsis edilip o alanların da kopyalanması gerekir (bu işlem tamamen C++'taki atama operatör fonksiyonunun yazım gerekçesinde olduğu gibidir). Örneğin `copy_files` fonksiyonu dosya tablosu için yeni bir alan tahsis eder ve üst process'teki tabloyu bu alana kopyalar. `copy_fs` fonksiyonu process'in geçerli dizini gibi alt process'e aktarılan bu tür bilgileri kopyalar. Buradaki en önemli fonksiyon belki de `copy_mm` fonksiyonudur. Bu fonksiyon process'in bellek alanını kopyalar.

Dosya tablosunun alt process'e kopyalanmasının en önemli sonucu alt process'in aynı dosyaları açık olarak görmesidir. Dosya göstericisinin aktif pozisyonu dosya nesnesinde saklandığı için process'lerden biri dosya göstericisini konumlandırdığında diğer process de bunu konumlanmış olarak görür. Şüphesiz dosya nesnesine erişim atomik düzeyde yapılmaktadır. Fakat dosyaya yazım ve dosyadan okuma işlemleri atomik düzeyde yapılmaz.

Standart C Fonksiyonlarının Tamponlaması ve Bunun fork İşlemine Yansıması

Bilindiği gibi standart C fonksiyonları kütüphane içerisinde ayrı bir tamponlama uygularlar. Standart dosya fonksiyonları kullanıldığında bu durum göz önünde bulundurulmalıdır, çünkü örneğin fork işleminden önce standart C fonksiyonlarıyla dosyaya bir şeyler yazılmış olabilir, ancak henüz tampon dolmadığı için bunlar dosyaya write fonksiyonuyla aktarılmamış olabilir. Bu noktada fork yapıldığında tüm bellek alanının kopyası oluşturulduğu için bu tampon bölgelerin de kopyası alt process için oluşturulmuş olacaktır. Böylece üst ve alt process'lerde fork işleminden önce tamponda bulunanlar ilk tazeleme işlemi ile iki kez hedef aygıta aktarılacaktır. Bu tür durumlarda iyi bir kontrol uygulanmazsa pek çok potansiyel problem çıkabilir. Bu nedenle fork işleminden önce standart dosya fonksiyonları kullanılacaksa şu önlemler alınmalıdır:

1) fork işleminden önce dosya tamamen kapatılabilir.
2) fork işleminden önce dosya fflush fonksiyonuyla flush edilebilir.
3) En başta tamponlamasız moda geçilerek işlemler yapılabilir.

Default olarak stdout standart C fonksiyonları için ya tamponlamasız çalışır ya da satır tamponlaması yapmaktadır (standartlara göre başlangıçta default olarak stdout tam tamponlama kullanamaz). Standart C kütüphanelerinin çoğu stdout dosyasını default olarak tamponlamasız modda kullanmaktadır. Bazı kütüphaneler (GNU libc) satır tamponlamalı olarak kullanır, ama stdin gerektiren dosya işlemlerinde stdout dosyasını da flush eder. Bu durumda aşağıdaki kod problemli olacaktır:

```
int main(void)
{
      printf("bir");
      fork();
      printf("iki\n");

      return 0;
}
```

Burada default stdout için satır tamponlaması yapan standart C kütüphanelerinde ekrana

```
biriki
biriki
```

yazısı çıkar.

Alt Process'lerin Beklenmesi ve Zombie Process Kavramı

Bir process sonlandığında process'in sonlanma kodu (exit code) denilen bir sayı bir bilgi olarak onu çağıran process'e iletilir. Process'in sonlanma kodu exit fonksiyonunun parametresi ile ya da main fonksiyonunun geri dönüş değeri ile tespit edilir. Process'in sonlanma kodlarının genellikle sistem için bir anlamı yoktur. Daha çok üst process ile alt process'in anlaşması için bu kod kullanılır.

Process'in sonlanma kodu process bittiğinde nerede saklanır? Sistemlerin çoğu bu çıkış kodunu process handle alanında saklarlar, üst process bu çıkış kodunu alana kadar process handle alanını yok etmez. Örneğin, Win32 sistemlerinde CreateProcess ile yaratılan process sonlandığında process'in bellek alanı boşaltılır, ama process handle alanı boşaltılmaz. Programcı çıkış kodunu GetProcessExitCode fonksiyonu ile alır, daha sonra üst process CloseHandle API fonksiyonu ile process handle alanını siler.

POSIX sistemlerinde çıkış kodunun alınması ve process handle alanının boşaltılması iki ayrı fonksiyonla değil, tek fonksiyonla yapılmaktadır. Bu işi yapan fonksiyonlar `wait` ve `waitpid` isimli fonksiyonlardır. UNIX/Linux sistemlerinde bir alt process sonlandığında onu yaratan process wait ya da waitpid fonksiyonlarıyla çıkış kodunu almazsa process handle alanı wait işlemi yapılana kadar bellekte kalır, bu tür process'lere zombie processler denilmektedir. Zombie processler `ps` komutu ile process listesi alındığında `'z'` harfi ile gösterilir.

Process'lerin çıkış kodları ve genel çıkış bilgileri process handle alanında tutulmak zorunda değildir. Örneğin bu bilgiler process sonlandıktan sonra daha küçük bir alana taşınabilir ve process handle alanı serbest bırakılabilir. wait fonksiyonunun prototipi şöyledir:

```
pid_t wait(int *pStat);
```

Bu fonksiyon sırada bekleyen alt process'in çıkış kodunu alır ve bunu parametresiyle belirtilen adrese yerleştirir. Geri dönüş değeri çıkış kodu alınan process'in handle değeridir. Fonksiyon başarısızlık durumunda –1 değerine geri dönmektedir. Bu fonksiyon eğer alt process henüz sonlanmamışsa alt process sonlanana kadar üst process'i bloke eder. Yani wait fonksiyonu çağırıldığında alt process bitene kadar wait fonksiyonu geri dönmeyecektir. Üst process birden fazla alt process yaratmış ise bu fonksiyon sıradaki ilk alt process'e ilişkin çıkış bilgisini alır. Bu tür durumlarda wait fonksiyonu tarafından alınan çıkış kodunun hangi alt process'e ilişkin olduğunun standart bir belirlemesi yoktur. Bu tür durumlarda fonksiyonun geri dönüş değeri hangi alt process'în çıkış kodunu belirlemek için kullanılır? Programcı alt process'i beklemek istediği halde çıkış kodu ile ilgilenmeyebilir. Bu durumda wait fonksiyonuna NULL değeri parametre olarak geçilir. Fonksiyon çıkış kodunu bu durumda yerleştirmez. Fonksiyonun prototipi sys/wait.h dosyası içerisindedir. Aşağıdaki programda bir alt process yaratılmış ve üst process alt process'i wait fonksiyonu ile beklemiştir. Böylece önce alt process'in sonlanacağı garanti altına alınmıştır. Ekrana sırasıyla

```
I am child..
end..
I am parent..
end..
```

yazıları çıkar.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
      pid_t pid;

      if ((pid = fork()) == -1) {
            perror("fork");
            exit(1);
      }

      if (pid == 0) {
            printf("I am child..\n");
      }
      else {
            wait(NULL);
            printf("I am parent..\n");
      }

      printf("end..\n");

      return 0;
}
```

Şüphesiz wait fonksiyonu eğer beklenecek hiç bir alt process yoksa başarısız olur ve bloke olmadan –1 ile geri döner. Başarısızlık nedeni için errno değişkenine bakılmalıdır.

Process fork ile yaratıldıktan sonra çizelgeleme işleminin üst processten mi yoksa alt processten mi devam edeceği standart olarak belirlenmemiştir, kullanılan çizelgeleme algoritmasına bağlıdır.

wait fonksiyonu ile elde edilen int değer programcıya iki bilgi verir.

1) Alt process'in çıkış kodu
2) Alt process'in sonlanma biçimi

Genellikle çıkış kodu düşük anlamlı byte değerindedir. Ancak bu durum POSIX standartlarında garanti altına alınmadığı için bu değerler standart makrolar ile elde edilmelidir. Örneğin `WEXITSTATUS` makrosu çıkış kodunu, `WIFEXITED` makrosu ise normal sonlanmayı tespit eder.

Sınıf çalışması: fork fonksiyonu ile bir alt process yaratınız. Alt process'te exit fonksiyonunda 10 değeri ile geri dönünüz. Üst process'te wait fonksiyonu ile alt process'i bekleyiniz ve çıkış kodunu alarak yazdırınız.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
      pid_t pid;
      int Status;

      if ((pid = fork()) == -1) {
            perror("fork");
            exit(1);
      }

      if (pid == 0) {
            printf("I am child..\n");
            exit(10);
      }
      else {
            wait(&Status);
            printf("Exit code: %d\n", WEXITSTATUS(Status));
      }

      printf("end..\n");

      return 0;
}
```

Alt process'lerin sonlandırılmasında iki önemli durum oluşabilir.

1) Alt process üst process'ten daha önce sonlanmıştır
2) Üst process alt process'ten önce sonlanmıştır

Alt process üst process'ten önce sonlanmışsa, fakat üst process wait fonksiyonu ile alt process'i beklememişse alt process zombi duruma düşer. Üst process daha önce sonlanmışsa

alt process'in üst process'i otomatik olarak sistem tarafından init process'i yapılacaktır. init process'i tüm alt process'ler için wait fonksiyonu uygulamaktadır. Yani üst process alt process'ten daha önce sonlanmışsa alt process zombi duruma düşmez. Çünkü sistem o process'in üst process'ini init process'i yapacaktır, init process'i de wait ile çıkış kodunu alacaktır. Görüldüğü gibi zombi durumu üst process'in de çalışmaya devam ettiği durumlarda oluşmaktadır.

Üst process sonlandığı anda zombi durumu ortadan kalkmaktadır. Şüphesiz alt process daha önce sonlanmış olabilir ve üst process herhangi bir zaman wait fonksiyonunu uygulayabilir. Bu durumda wait fonksiyonu bloke olmadan başarılı bir biçimde çıkış kodunu alacaktır ve alt process'i zombi durumundan kurtaracaktır.

Win32 Programcıları İçin Not: Win32'de bir program başka bir programı CreateProcess fonksiyonu ile çalıştırdığında alt process sonlanana kadar beklenmesi WaitForSingleObject fonksiyonu ile process handle değeri kullanılarak gerçekleştirilir.

waitpid Fonksiyonu

wait fonksiyonu herhangi bir process'i bekleyebilmektedir. Yani pek çok alt process yaratılmışsa wait fonksiyonu bitmiş olan herhangi bir alt process'in çıkış kodunu alır. Oysa bazı uygulamalarda programcı belirli bir alt process'in çıkış kodunu almak isteyebilir. waitpid fonksiyonu bunun için kullanılmaktadır.

```
pid_t waitpid(pid_t pid, int *stat, int options);
```

Fonksiyonun birinci parametresi beklenecek olan alt process'in handle değeridir. Bu değerin normal olarak sıfırdan büyük olması beklenir. Ancak aşağıdaki özel durumlarda aşağıda belirtilen işlemler oluşmaktadır.

Seçenek	Olay
`pid > 0`	pid ile belirtilen alt process sonlanana kadar beklenir.
`pid == -1`	Herhangi bir alt process için bekleme yapılır. Bunun da wait fonksiyonundan farkı kalmaz.
`pid == 0`	Process grup ID'si üst process ile aynı olan herhangi bir alt process beklenir.
`pid < -1`	Process grup ID'si |pid| ile aynı olan herhangi bir alt process beklenir.

Fonksiyonun ikinci parametresi çıkış kodu bilgilerini yerleştirmek için kullanılır. Son parametre ek belirlemeleri belirtir. Genellikle bu parametre 0 olarak girilir. 0 olarak girilmezse aşağıdaki değerlerden biri olarak girilmelidir.

```
0
WHOHANG
WUNTRACEN
```

Değişken Parametre Alan Fonksiyonlar

Derleyicinin parametre sayısı üzerinde kontrol yapmadığı printf ve scanf gibi özel fonksiyonlara değişken sayıda parametre alan fonksiyonlar denir. Değişken sayıda parametre

alan fonksiyonların parametrelerinde bu durum `...` ile belirtilir. Değişken sayıda parametre alan fonksiyonlar en az bir parametreye sahip olmak zorundadır. Fonksiyon çağırılırken zorunlu olarak yazılması gereken parametreler birden fazla olabilir. Şüphesiz `...` parametre listesinin sonunda olmak zorundadır. Örneğin,

```
void Func(int a, int b, ...);       geçerli
void falan(...);                    geçersiz
void Foo(int a, ..., int b);        geçersiz
```

Değişken sayıda parametre alan fonksiyonlar stdarg.h dosyasındaki makrolar kullanılarak yazılırlar. Bu makrolar;

```
VA_START
VA_ARG
VA_END
```

makrolarıdır. Değişken sayıda parametre alan fonksiyonları anlamak için fonksiyon parametre aktarımına ilişkin temel bilgiler gerekir.

C'de parametrelerin aktarım sırası standart olarak belirlenmemiştir. PC'lerde cdecl denilen biçim kullanılır ve bu biçime göre parameteler sağdan sola fonksiyona aktarılır. Bu sisteme göre parametreler stack'e kopyalandığında düşük adreste en soldaki parametre olacak biçimde ve bir dizi gibi yapı söz konusu olur. Yani, en soldaki parametre değişkeninin adresini ve parametrelerin türlerini bilirsek bütün parametrelere erişebiliriz. Örneğin aşağıdaki kodda bir assert hatası oluşmayacaktır:

```
void Func(int a, int b)
{
      assert(&a + 1 == &b);
}
```

Ancak parametrelerin sağdan sola fonksiyona geçirileceği her sistemde garanti değildir, zaten özel makroların kullanılması taşınabilirliği sağlamak için düşünülmüştür.

Değişken sayıda parametre alan fonksiyonlarda fonksiyonun kaç parametreyle çağırıldığı ve parametrelerin türlerinin neler olduğu bilinmelidir. Bu yüzden birinci parametreden bu bilgilerin anlaşılması gerekir. Örneğin birinci parametrede belirtilen sayıda int türden parametreleri toplayan bir fonksiyon yazacak olalım.

```
void Add(int n, ...)
{
      int total = 0;
      int *p = &n + 1;
      for (i = 0; i < n; i++) {
            total += *p;
            ++p;
      }
      return total;
}

int main(void)
{
      printf("%d\n", Add(5, 10, 20, 30, 40, 50));

      return 0;
```

```
}
```

C'de derleyici bir fonksiyonun prototipini ya da tanımlamasını görmemişse ya da değişken sayıda parametre alan bir fonksiyon çağırılıyorsa default argüman dönüştürmesi uygulanır. Default argüman dönüştürmesinde char ve short türleri int türüne, float türü ise double türüne dönüştürülür.

Yukarıdaki Add fonksiyonunun kodu taşınabilir değildir, taşınabilir hale getirmek için stdarg.h içerisindeki makrolar kullanılmalıdır.

va_list Türü:

va_list parametre çekme işlemlerinde kullanılan bir türdür. Genellikle bu tür char türden bir gösterici olarak ele alınır.

va_start Makrosu:

İşlemlere başlamadan önce bu makro bir kez çağırılmalıdır.

```
void va_start(va_list ar, lastParam);
```

Bu makro ilk parametrenin adresinden faydalanarak ilk parametreden sonraki ilk belirtilmeyen argümanın adresini hesaplar ve ar'nin içerisine yerleştirir. Burada lastParam `...`'dan bir önceki parametreyi temsil etmektedir.

va_arg Makrosu:

Bu makro her çağırıldığında geri dönüş değeri olarak bir sonraki parametreyi elde eder.

```
type va_arg(va_list ar, type);
```

Burada type programcının belirttiği türdür. Bu makro ilk kez çağırıldığında yazılmayan ilk parametrenin bilgisi alınır ve ar'yi günceller. va_arg makrosunda elde edilen değer sağ taraf değeridir.

va_end Makrosu:

Bu makro aslında pekçok sistem için gereksizdir, ancak taşınabilirliği sağlamak için işlem sonunda kullanılmalıdır.

```
void va_end(va_list ar);
```

Pekçok derleyicide bu makro koda hiçbir şey açmamaktadır.

```
void Add(int n, ...)
{
      int i;
      total = 0;

      va_start(ar, n);
      for (i = 0; i < n; ++i)
            total += va_arg(ar, int);
```

```
	va_end(ar);
}
```

Sınıf Çalışması: %d, %c ve %f işlemlerine duyarlı myprintf() fonksiyonunu yazınız. Bu fonksiyonu yazarken stdarg.h makrolarını kullanınız.

```
void myprintf(const char *format, ...);
```

Açıklamalar:

Format karakterinde % karakteri görünene kadar karakterler yazdırılarak ilerlenir. % karakteri görüldüğünde yanındaki karaktere bakılır ve işlemler yapılır. int ve double sayıları yazdırmak için printf kullanılabilir.

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

void myprintf(const char *format, ...)
{
	va_list ar;
	va_start(ar, format);

	while(*format != '\0') {
		if (*format == '%') {
			switch (*(format + 1)) {
				case 'd':
					printf("%d", va_arg(ar, int));
					break;
				case 'c':
					putchar(va_arg(ar, int));
					break;
				case 'f':
					printf("%f", va_arg(ar, double));
					break;
				default:
					putchar(*(format + 1));
			}
			format += 2;
		}
		else {
			putchar(*format);
			format++;
		}
	}
}

void main(void)
{
	myprintf("float = %f int = %d char = %c", 2.34, 5, 'g');
}
```

vprintf, vsprintf ve vfprintf Fonksiyonları

Bu fonksiyonlar printf, sprintf ve fprintf fonksiyonlarının değişken parametre alan biçimleridir.

```
int vprintf(const char *format, va_list ar);
int vsprintf(char *buf, const char *format, va_list ar);
int vprintf(FILE *f, const char *format, va_list ar);
```

Özellikle vsprintf, printf fonksiyonunun işlevini yaratmak için yaygın olarak kullanılmaktadır. Bu fonksiyonlar programcıdan format stringini alırlar, ancak diğer parametreleri almazlar. Diğer parametreler yerine onların va_list türünden adreslerini alırlar. Böylece yalnızca yazı yazdıran bir fonksiyon vsprintf kullanılarak printf haline getirilmiş olur.

```
int MessagePrintf(const char *format, ...)
{
      char buf[100];
      va_list ar;
      va_start(ar, format);
      vsprintf(buf, format, ar);
      MessageBox(NULL, buf, "Message", MB_OK);
      va_end(ar);
}

MessagePrintf("a = %d b = %d\n", a, b);
```

Not: Standart C fonksiyonlarında bir tampon güvenliği problemi vardır ve bu durum çok eleştirilmektedir. Örneğin gets fonksiyonu güvensizdir, çünkü klavyeden ne kadar karakter girileceği belirli değildir. gets fonksiyonu tüm karakterleri belirtilen adrese yerleştirir, '\n' karakterini tampondan siler, ama diziye yerleştirmez. Halbuki fgets en fazla SIZE - 1 karakteri diziye yerleştirir, ancak bu fonksiyon da '\n' karakterini dizinin sonuna yerleştirmektedir. fgets fonksiyonunu gets gibi kullanabilmek için aşağıdaki gibi bir algoritma önerilebilir.

```
char *p;

if (fgets(buf, SIZE, stdin) != NULL)
      if ((p = strchr(buf, '\n')) != NULL)
            *p = '\0';
```

Burada fgets fonksiyonu:

1) abc Ctrl+Z girişi yapıldığında tampona abc karakterlerini ve null karakterini yerleştirir ve buf adresine geri döner.
2) Ctrl+Z girişi yapıldığında tampona bir şey yerleştirmez ve NULL değerine geri döner.
3) abc Enter girişi yapıldığında tampona abc\n ve null karakterlerini yerleştirir ve buf adresiyle geri döner.

Tabii bunun yerine aşağıdaki gibi daha güvenli bir gets yazılabilir:

```
char *mygets(char *buf, size_t size)
{
      size_t  i;
      int ch;

      for (i = 0; (ch = getchar()) != EOF && ch != '\n'; i++) {
            if (i >= size - 1)
                  break;
            buf[i] = ch;
      }
      buf[i] = '\0';
```

```
	return buf;
}
```

exec Fonksiyonları

exec fonksiyonları başka bir programı çalıştırmak için kullanılan genel fonksiyonlardır. Bilindiği gibi bir programın .text, .data, .bss ve .stack gibi bölümleri, yani process'in bellek alanı process handle alanında tutulmaktadır. Böylece işletim sistemi başka bir programı çalıştırmak üzere process'ler arası geçiş yaptığında geçiş yapılan process'in bellek alanı process handle alanından hareketle elde edilmektedir. exec fonksiyonları o anda çalışmakta olan process'in bellek alanını boşaltır, çalıştırılacak programı belleğe yükler, process'in yeni bellek alanını çalıştırılacak programın belek alanı yapar. Yani exec fonksiyonları uygulandığında artık programın bellek alanı ortadan kaldırılır. Yani program başka bir program olarak çalışmaya devam eder. Aslında exec isimli bir fonksiyon yoktur. İsimleri benzer bir grup fonksiyon vardır. Ancak exec sözcüğü bütün bu grubu anlatan bir fonksiyon kavramı olarak kullanılmaktadır. Programda exec fonksiyonu uygulandıktan sonra artık o programa özgü hiç bir kod çalışmayacaktır. Örneğin aşağıdaki kodda hiç bir zaman unreachable code yazısı gözükmeyecektir.

```
int main()
{
	exec(...);
	printf("unreachable code..\n");
}
```

Görüldüğü gibi exec fonksiyonu sonrasında artık bu fonksiyonu uygulayan programın hiç bir varlığı kalmaz. Bir process'in iki önemli bilgisi process handle alanı ve process'in bellek alanıdır. fork işleminde hem handle alanının hem de bellek alanının birer kopyası çıkarılır. exec işleminde ise başka bir program yüklenerek bellek alanı silinerek başka bir process yüklenmektedir. Ancak process'in bellek alanı aynen kalmaktadır. UNIX/Linux sistemlerinde başka bir biçimde program çalıştırmanın yolu yoktur.

Win32 Sistem Programcıları İçin Not: Win32 sistemlerinde exec benzeri bir işlem yoktur. Bu sistemlerde başka bir programın çalıştırılması CreateProcess API fonksiyonu ile yapılır. Bu fonksiyon yeni bir handle alanı oluşturup bir programı çalıştırır.

UNIX/Linux programcıları eski programın devam etmesini sağlayarak yeni bir program çalıştırmak için önce bir kez fork yaparlar, böylece process'in bir kopyası yaratılır. Yaratılan yeni alt process'te exec uygularlar. POSIX sistemlerinde kullanılan exec fonksiyonları şunlardır:

```
execl
execv
execle
execve
execlp
execvp
```

Görüldüğü gibi exec sözcüğünden sonra bunu 'l' ya da 'v' izlemektedir. Bu harfleri de 'e' ya da 'p' izler. 'l' harfi list sözcüğünden 'v' harfi ise vector sözcüğünden kısaltmadır. 'e' environment 'p' ise path sözcüklerinden gelir. Bu fonksiyonların hepsi birinci parametre

olarak çalıştırılacak programın diskteki ismini alırlar. l'li versiyonlar programın komut satırı argümanlarını ayrık parametre olarak, v'li versiyonları da komut satırı argümanlarını bir gösterici dizisi olarak alır. l'li versiyonlar değişken sayıda parametre alan fonksiyon biçiminde yazılmışlardır. Normal olarak bir process'in çevre değişkenleri fork işlemi ile alt process'e tamamen aktarılır. Eğer exec işlemi sırasında exec fonksiyonlarının e'li versiyonları kullanılırsa exec işlemi sırasında process'in çevre değişkenleri kümesi de tamamen değiştirilebilmektedir.

exec fonksiyonlarının p'li ve p'siz versiyonları PATH çevre değişkenine bakılıp bakılmayacağını belirtir. p'siz versiyonlarda çalıştırılacak dosya ismi yalnızca dosya ismi yazılırken belirtilen path'te aranır. Bilindiği gibi path ifadesi / ile başlatılmışsa bu ifadeye mutlak path ifadesi denilir ve yol root dizininden itibaren belirtilir. Eğer path ifadesi / ile başlatılmamışsa buna göreli path ifadesi denir. Göreli path ifadesi bulunulan dizinden itibaren yer belirtir. Path ifadesi . ile başlatılırsa POSIX sistemlerinde mutlak path ifadesi belirtir. exec fonksiyonlarının p'li versiyonlarında dosya yalnızca PATH ile belirtilen dizinde aranır. DOS ve Windows sistemlerinde olduğu gibi bulunulan dizinde aranmaz. Dosya isminin bulunulan dizinde de aranması için bulunulan dizine ilişkin bir PATH ifadesinin eklenmiş olması gerekir.

```
PATH = /home/kaan
```

p'li versiyonlarda eğer dosya ismi / ile başlatılmamışsa arama verilen göreli path ifadesinin tek tek PATH'te belirtilen dizinlerin sonuna eklenmesi ile yapılır. Örneğin p'li versiyonlarla dosya ismi "a/b" biçiminde belirtilmişse PATH'te belirtilen tüm dizinlerin altındaki a dizininde b aranır. p'li versiyonlarda dosya ismi mutlak path ifadesi ile belirtilirse PATH çevre değişkenine baş vurulmaz. Böylece p'li veriyonun bir anlamı kalmaz.

execl Fonksiyonu

```
int execl(const char *path, const char *argv, ...);
```

Fonksiyonun birinci parametresi çalıştırılacak programın path ifadesi, ikinci parametresi ise ilk komut satırı argümanıdır. Bundan sonra istenildiği kadar komut satırı argümanı yazılabilir, ancak NULL ile bitirilmesi gerekir. exec fonksiyonlarının geri dönüş değerleri başarısız ise –1 değeridir. unistd.h başlık dosyası içerisindedir. Son parametre olan NULL gösterici düz sıfır olarak girilirse derleyici bunu null gösterici sabiti olarak yorumlamaz, int olarak yorumlar. Göstericilerle int türünün farklı olduğu sistemlerde bu durum probleme yol açabilir. Bu durumda sıfır'ın (char *) türüne dönüştürülmesi uygundur.

*Sıfır Sabitine İlişkin Not: Standartlara göre null gösterici sabiti 0 ya da (void *) 0'dır. 0 sayısı bir göstericiye atandığında ya da bir gösterici ile != ve == operatörü ile karşılaştırıldığında derleyici tarafından null gösterici olarak değerlendirilir. Ancak 0 sayısı normal işlemlerde int türünden sabit olarak değerlendirilir. Örneğin*

```
Func(x, 0);
```

Derleyici burada 0'ı nasıl yorumlayacaktır? İşte derleyici sıfır'a karşı gelen parametrenin gösterici olup olmadığına bakar. Gösterici ise 0'ın null gösterici olduğu anlamını çıkartır, bu parametreye karşı gelen parametre değişkenleri doğal türlere ilişkinse derleyici 0'ı int olarak değerlendirir. Ancak 0 parametresine karşılık gelen parametre değişkeni prototip

yokluğundan bilinemiyorsa ya da değişken sayıda parametre alan bir fonksiyon söz konusu ise derleyici 0'ı yine int türden kabul eder.

Görüldüğü gibi aslında C'de kullandığımız argv[0] program ismini içermek zorunda değildir. Yani çalışabilen dosyanın ismini alabilmek için argv[0]'a bakmak taşınabilir bir yöntem değildir. Çünkü exec fonksiyonlarında çalıştırılacak dosyanın ismi ile argv[0] istenirse farklı olarak verilebilir. Tabii geleneksel olarak argv[0] program ismine ilişkin bir parametredir. Benzer durum Win32 sistemlerinde de bu biçimdedir. Bu sistemlerde çalışabilen dosyanın path ifadesini GetModuleFileName fonksiyonu ile almak gerekir.

```
int main(void)
{
      printf("Başla\n");
      if (execl("/bin/ls", "ls", "-l", (char *) NULL) == -1) {
            fprintf(stderr, "Cannot exec...\n");
            exit(1);
      }
      printf("Son..\n");
}
```

execlp Fonksiyonu

Bu fonksiyon execl fonksiyonunun PATH çevre değişkenine bakan biçimidir.

```
int execlp(const char *file, const char *argv0, ...);
```

Bu fonksiyon execl fonksiyonundan farklı olarak eğer path ifadesi / ile başlatılmamışsa tek tek path çevre değişkeniyle belirtilen dizinlerde arama yapar. Eğer path / ile başlatılmışsa bu fonksiyonun execl'den bir farkı kalmaz.

execle Fonksiyonu

Normal olarak fork işlemi ile çevre değişkenlerinin hepsi yeni yaratılan process'e aktarılmaktadır (çevre değişkenlerinin process handle alanında tutulduğu varsayılabilir). Yine normal olarak exec işlemi sonrasında aynı çevre değişkenleri etkinliğini sürdürür. Ancak execle ile exec işlemi sırasında yeni çalıştırılacak program çalıştırılmadan önce eski çevre değişkenleri atılıp yeni çevre değişken takımı set edilebilir.

```
int execle(const char *path, const char *argv0, ...,
       /* (char *)0, char *const envp[] */);
```

Görüldüğü gibi komut satırı argümanlarını sonlandırmak için NULL gösterici parametresi girilir, NULL göstericisinden sonra da çevre değişkenleri dizisi girilmektedir. Çevre değişkenlerinin bulunduğu gösterici dizisinin sonu NULL gösterici ile bitirilmelidir. Örneğin;

```
char *myenv[] = {"ali = 100", "veli = 200", NULL};
//...
execle("myprog", "myprog", (char *)0, myenv);
```

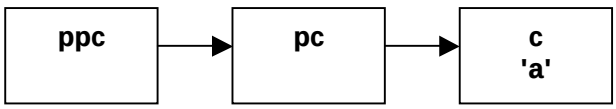
execle fonksiyonu PATH çevre değişkenine bakmaz.

Anahtar Notlar: Göstericiyi gösteren göstericilerde const olma durumu biraz karmaşıktır. Göstericiyi gösteren göstericilerde const anahtar sözcüğü üç yere getirilebilir:

```
1) const char **ppc;
2) char *const *ppc;
3) char **const ppc;
```

Bu ifadelerde sırasıyla üç değişik nesne const yapılmıştır.

```
char c = 'a';
char *pc = &c;
char **ppc = &pc;
```



Bilindiği gibi const nesnenin adresi const bir adrestir ve gösterdiği yer const olan bir göstericiye atanabilir. Aşağıdaki gibi bir atama güvensizdir:

```
char *pc;
const char **ppc;
ppc = &pc;        /* güvensiz */
```

*Buradaki atamanın güvenli olduğu sanılabilir, çünkü const olmayan bir nesnenin const bir nesneye atanması normal izlenimi vermektedir. const char **ppc ifadesinde **ppc ifadesi const'tur, ancak *ppc ifadesi kendisi const olmayan ama gösterdiği yer const olan bir göstericidir. Bu durum *ppc kullanılarak const olmayan bir göstericiye const olan bir adresi gizlice atamaya olanak sağlar. Örneğin:*

```
const char c = 'a';
char *pc;
const char **ppc;
ppc = &pc;
*ppc = &c;
*pc = 'b';
```

*Burada *ppc = &c işlemiyle aslında pc'ye c'nin adresi atanmaktadır, yani artık *pc = 'b' gibi bir işlemle c bozulabilir. Ancak aşağıdaki atama güvenlidir:*

```
const char c;
char *pc;
char *const *ppc;
ppc = &pc;        /* güvenli */
```

Çünkü artık aşağıdaki işlem error'a yol açacaktır:

```
*ppc = &c;        /* error */
```

*Çünkü artık *ppc kendisi const olan bir göstericidir. Aşağıdaki atama da güvensizdir:*

```
const char c;
const char *pc;
char *const *ppc;
ppc = &pc;        /* güvensiz */
```

*Bu durumun güvenli olabilmesi için const char *const *ppc; bildirimi yapılmalıdır.*

Şimdi bir fonksiyonun parametresinin aşağıdaki biçimde olduğunu düşünelim:

```
void Func(char *const *ppc);
```

Bu fonksiyona geçirilen hangi parametreler güvenlidir?

```
1) char *abc[10];
Func(abc);        /* güvenli */

2) const char *abc[10];
Func(abc);        /* güvensiz */
```

Aşağıdaki fonksiyon için inceleme yapalım:

```
void Func(const char **ppc);

1) char *abc[10];
Func(abc);        /* güvensiz */

2) const char *abc[10];
Func(abc);        /* güvenli */
```

Gösterici dizilerinde de const anahtar sözcüğü iki yere yerleştirilebilir:

```
1) const char *p[10];
```
Bu tanımlama p'nin bir gösterici dizisi olduğu, bu diziye gösterdiği yer const olan adreslerin yerleştirileceği anlamına gelir.

```
2) char *const p[10] = {.....};
```
Burada dizinin kendisi const biçimindedir, yani diziye ilk değer verdikten sonra bir daha elemanlara değer atayamayız. İlk değer vermekte kullandığımız adreslerin gösterdiği yerlerin const olması gerekmez. Benzer biçimde const anahtar sözcüğü hem başa hem de dizi isminin önüne getirilebilir.

execv Fonksiyonu

Bu versiyon da PATH çevre değişkenine bakmaz. execl fonksiyonundan farkı komut satırı argümanları gösterici dizisi olarak girilir. Gösterici dizisinin sonu NULL ile bitirilmelidir.

```
int execv(const char *path, char *const argv[]);
```

Fonksiyonun ikinci parametresindeki const yerleşimi kendisi const olan bir gösterici dizisinin adresini geçebilmemize olanak sağlar. Bu fonksiyon tipik olarak aşağıdaki gibi kullanılabilir:

```
int main(int argc, char *argv[])
{
      // ...
      if (execv(argv1, &argv[1]) == -1) {
            perror("exec");
            exit(1);
      }
      return 0;
}
```

Burada başka bir programı çalıştıran program söz konusudur. Buradaki program execute.c olsun, myprog ise başka bir program olsun, execute programını şöyle çalıştırmış olalım:

```
$ execute myprog arg1 arg2
```

Burada myprog programının argv[0] parametresi myprog, argv[1] parametresi arg1'dir.

execvp Fonksiyonu

Bu fonksiyonun execv fonksiyonundan tek farkı PATH çevre değişkenine bakmasıdır.

```
int execvp(const char *file, char *const argv[]);
```

execve Fonksiyonu

Bu versiyon PATH çevre değişkenine bakmaz, ama çevre değişken takımını değiştirir.

```
int execve(const char *file, char *const argv[], char *const env[]);
```

fork ve exec Fonksiyonlarının Birarada Kullanılması

Bilindiği gibi normal olarak exec fonksiyonları process'i başka bir program olarak devam ettirmektedir. Halbuki pek çok uygulamada başka bir programı çalıştıran programın da devam etmesi istenir. Bunun için önce bir kere fork işlemi yapılır, alt process'te exec fonksiyonu uygulanır. Bu işlem tipik olarak aşağıdaki gibi yapılabilir:

```
pid_t pid;

if ((pid = fork()) == -1) {
      perror("fork");
      exit(1);
}
if (pid == 0) {
      if (execl(.....) < 0) {
            perror("exec");
            exit(1);
      }
}
wait(NULL);
// ...
```

Buradaki işlem Win32'de CreateProcess uygulayıp WaitForSingleObject fonksiyonuyla beklemeye karşılık gelir.

Sınıf Çalışması: gcc dereyicisiyle komut satırı argümanı olarak verilen bir dosyayı derleyiniz, sonra çalışabilen dosyayı ls -l ile görüntüleyiniz.

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
{
      pid_t pid;

      if (argc != 2) {
            fprintf(stderr, "Wrong number of argument!..\n");
            exit(1);
      }

      if ((pid = fork()) == -1) {
            perror("Fork");
            exit(EXIT_FAILURE);
      }

      if (pid == 0) {
            printf(argv[1]);
            printf("\n");
            if (execlp("gcc", "gcc", "-g", argv[1], (char *) 0) < 0) {
                  perror("child1 exec");
                  exit(EXIT_FAILURE);
            }
      }
      wait(NULL);

      if ((pid = fork()) == -1) {
            perror("Fork");
            exit(EXIT_FAILURE);
      }

      if (pid == 0) {
            if (execlp("ls", "ls", "-l", (char *) NULL) < 0) {
                  perror("child2 exec");
                  exit(EXIT_FAILURE);
            }
      }
      wait(NULL);

      return 0;
}
```

Örnek Bir Shell Programı

Tipik bir shell programı bir döngü içerisinde prompt çıkartır, sonra gets ya da fgets gibi bir fonksiyonla klavyeden komutu alır, komutu parçalarına ayrıştırır, parçaları bir gösterici dizisine yerleştirir, fork ve execv gibi bir fonksiyonla komutu çalıştırır. Girilen yazının parçalara ayrılması için klasik algoritma kullanılabilir ya da strtok fonksiyonundan faydalanılabilir.

```
char *strtok(char *s, const char *delim);
```

strtok ilkin atomlarına ayrılacak yazının adresiyle çağırılır, daha sonra NULL değeri ile çağırılarak atomlar sırasıyla elde edilir. strtok fonksiyonunun verdiği adres birinci parametresiyle belirtilen dizi içerisindeki adrestir. Fonksiyonun ikinci parametresi ayrım karakterlerinin bulunduğu dizinin adresidir. Fonksiyon ilk çağırıldığında ilk atomun adresiyle, sonraki çağırmalarda sonraki atomların adresleriyle geri döner.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <stdarg.h>
#include <sys/types.h>
#include <unistd.h>

#define ARGV_SIZE       20
#define CMD_SIZE        256
#define PROMPT_SIZE     20
#define DEF_PROMPT      "csdsh>"

int runprog(char **argv)
{
      pid_t pid;
      if ((pid = fork()) == -1)
            return -1;

      if (pid == 0 && execvp(argv[0], argv) == -1)
            return -1;
      return pid;
}

int makeargv(char *str, char **argv, int size)
{
      int argc = 0, i;

      for (i = 0; i < size - 1; ++i) {
            while (isspace(*str))
                  ++str;
            if (*str != '\0')
                  argv[argc++] = str;
            else {
                  argv[argc] = NULL;
                  break;
            }

            while (!isspace(*str) && *str != '\0')
                  ++str;

            if (*str != '\0' && i < size -1)
                  *str++ = '\0';
      }

      return argc;
}

int main(void)
{
      char prompt[PROMPT_SIZE] = DEF_PROMPT;
      char cmd[CMD_SIZE];
      char *argv[ARGV_SIZE];
      char *pstr;

      for(;;) {
            printf(prompt);
            fgets(cmd, CMD_SIZE, stdin);
            if ((pstr = strchr(cmd, '\n')) != NULL)
                  *pstr = '\0';
```

```
            if (makeargv(cmd, argv, ARGV_SIZE) == 0)
                  continue;

            if (strcmp(argv[0], "quit") == 0)
                  break;

            if (runprog(argv) == -1) {
                  if (errno == ENOENT)
                        fprintf(stderr, "Command Not Found\n");
                  else
                        perror("internal error");
                  exit(EXIT_FAILURE);
            }
            else
                  wait(NULL);
      }

      return 0;
}
```

_exit Fonksiyonu

Bilindiği gibi exit fonksiyonu bir standart C fonksiyonudur. Bu fonksiyon tüm dosyaları fflush işlemiyle tazeler. UNIX/Linux sistemlerinde process'i sonlandıran fonksiyon _exit fonksiyonudur. Biz POSIX sistemlerinde exit yerine _exit fonksiyonunu çağırırsak process yine sonlanır ama dosyalar flush edilmemiş olur.

Dosyaların kapatılması gerçek anlamda işletim sisteminin sistem fonksiyonları tarafından yapılmaktadır. exit fonksiyonu standart C fonksiyonlarının oluşturduğu tamponlama mekanizmasını tazelemektedir. Özetle:

1) Standart C fonksiyonları user seviyesinde bir tamponlama yapar.
2) Bütün dosya işlemleri eninde sonunda işletim sisteminin sistem fonksiyonları çağırılarak gerçekleştirilir. Standart C'nin dosya fonksiyonları bu sistem fonksiyonlarının üzerine tamponlama mekanizmasını kurmuşlardır.
3) exit fonksiyonu önce standart C fonksiyonlarının kullandığı tamponları tazeler, sonra bütün dosyaları tek tek kapatır. Bu kapatma sırasında işletim sisteminin sistem fonksiyonları çağırılacaktır.
4) Standart C fonksiyonları kullanılarak dosyalar açılmış olsun. Process _exit fonksiyonuyla sonlandırılsın. Standart C fonksiyonlarının açtığı dosyalar kernel tarafından açık olarak gözükmektedir. _exit fonksiyonu bu dosyaları işletim sistemi seviyesinde kapatır, ancak bu durumda standart C fonksiyonlarının tamponları tazelenmemiş olur bu da bilgi kaybına yol açabilir.

Bilindiği gibi stdout ve stdin dosyaları default olarak terminale ilişkin olduğunda satır tamponlamaya sahip olabilir ya da tamponlamasız çalışabilir, ancak tam tamponlamalı olamaz. Aşağıdaki programda eğer satır tabanlı tamponlama uygulanıyorsa ekrana iki defa "merhabaveli" yazısı çıkar.

```
int main(void)
{
      printf("merhaba");
      fork();
```

```
      printf("veli\n");
      return 0;
}
```

Tabii "merhabaveli" yazılarının düzenli çıkmama olasılığı vardır. Ancak aşağıdaki programda stdout dosyasının default satır tabanlıysa bir tane "merhaba" yazısı yazılacaktır.

```
int main(void)
{
      printf("merhaba");
      if (fork() == 0)
            _exit(0);
      exit(0);
}
```

Çünkü _exit ile çıkıldığında standart C fonksiyonlarının tamponları flush edilmeyecektir.

Standart C fonksiyonları ve fork işlemi ile bazen tamponların iki kere flush edilmesi gibi istenmeyen durumlarla karşılaşılabilir. Örneğin üst ve alt process'lerin her ikisi de exit fonksiyonu ile çıkarsa ve tamponlarda flush edilecek birtakım bilgiler kalmışsa iki kere flush işlemi yapılmış olur, bu da dosyadaki verilerin bozulması sonucunu doğurabilir. Şüphesiz fork işleminden önce tamponların tazelendiğinden eminsek bu durumda hem üst hem de alt process exit ile sonlandırılabilir. Genellikle önerilen yöntem alt process'lerin _exit ile sonlandırılması, üst process'lerin ise exit ile sonlandırılmasıdır.

exec İşleminde Dosya Tablosunun Durumu

Bilindiği gibi fork işlemiyle dosya nesnelerini gösteren dosya betimleyici dizisi de kopyalanmaktadır. Normal olarak exec işlemi sırasında üst process'in açmış olduğu dosyalar açık olarak kalmaktadır. Programcı isterse exec işlemi ile birlikte dosyaların kapanmasını sağlayabilir. Bir dosyanın exec işlemi sırasında kapatılması için `fcntl` fonksiyonu ile "close on exec" bayrağının set edilmesi gerekir.

system Fonksiyonu

Shell programları normal olarak bir prompt eşliğinde komut alıp yorumlama yapar. Ancak –c seçeneği ile kullanıldıklarında tek bir komutu işleyip çalışmayı sonlandırabilirler. İşte system fonksiyonu shell programını –c seçeneği ile çalıştırmaktadır. Örneğin shell aşağıdaki gibi çalıştırılırsa belirtilen komut çalıştırılıp işlem sonlandırılır.

```
/bin/sh –c <komut>  ßEnter
```

/bin/sh sistemde default kullanılan shell'i belirtir. Örneğin Linux sistemlerinde bu dosya bash shell'ine bir sembolik link biçiminde olabilir. system fonksiyonu <stdlib.h> dosyası içerisindedir.

```
int system(const char *command);
```

Bu fonksiyon aynı zamanda standart C fonksiyonudur. command parametresi çalıştırılacak komutu belirtir. Fonksiyon Unix sistemlerinde şüphesiz fork ve exec işlemleriyle komutu

çalıştırmaktadır. Eğer system fonksiyonu fork ya da waitpid fonksiyonlarının başarısızlığı yüzünden başarısız olmuşsa –1 değerine, eğer fork başarılı fakat exec başarısız olmuşsa 127 değerine ve eğer tam başarı elde edilmişse shell programının çıkış koduna geri dönmektedir.

/* system.c */

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

int main(void)
{
      char s[256] = {0}, *pstr;

      for (;;) {
            fgets(s, 256, stdin);
            if ((pstr = strchr(s, '\n')) != NULL)
                  *pstr = '\0';
            if (!strcmp(s, "quit"))
                  break;
            if (system(s) < 0)
                  perror("system");
      }
      return 0;
}
```

system fonksiyonu tipik olarak şöyle yazılmıştır:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int mysystem(const char *command)
{
      pid_t pid;
      int status;

      if (command == NULL)
            return 1;
      if ((pid = fork()) < 0)
            return -1;
      if (pid == 0) {
            execl(“bin/sh”, “sh”, “-c”, command, (char *) 0);
            _exit(127);
      }
      if (waitpid(pid, &status, 0) < 0)
            if (errno != EINTR)
                  return -1;
      return status;
}
```

/* mysystem.c */

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
```

```
#include <unistd.h>

int mysystem(const char *command)
{
      pid_t pid;
      int status;

      if (command == NULL)
            return 1;
      if ((pid = fork()) < 0)
            return -1;
      if (pid == 0) {
            execl("/bin/sh", "sh", "-c", command, (char *) 0);
            _exit(127);
      }
      if (waitpid(pid, &status, 0) < 0)
            return -1;
      return status;
}

int main(void)
{
      char s[256] = {0}, *pstr;

      for (;;) {
            fgets(s, 256, stdin);
            if ((pstr = strchr(s, '\n')) != NULL)
                  *pstr = '\0';
            if (!strcmp(s, "quit"))
                  break;
            if (mysystem(s) < 0)
                  perror("mysystem");
      }
}
```

Gerçek Kullanıcı ID'si, Etkin Kullanıcı ID'si ve Saklanmış Kullanıcı ID'si (Saved Set User ID)

Bilindiği gibi bir dosyaya erişimde process'in etkin kullanıcı ID'si ve etkin grup ID'si test işlemlerine sokulmaktadır. Normalde process'in gerçek kullanıcı ID'si ve grup ID'si passwd ve group dosyalarından bakılarak login programı tarafından set edilmektedir. Normal olarak etkin kullanıcı ID'si gerçek kullanıcı ID'sine eşittir. Ancak exec işlemi sırasında çalıştırılan programın "set user ID" bayrağı set edilmişse process'in etkin kullanıcı ID'si çalıştırılan dosyanın user ID'si olur. Her ne kadar konunun başlığı kullanıcı ID'sine yönelikse de burada anlatılanların hepsi gerçek grup ID'si ve etkin grup ID'si için de geçerlidir. Şüphesiz exec ile çalıştırılan programların "set group ID" bayrakları da vardır.

Bilindiği gibi super user önceliğine sahip olmak demek etkin kullanıcı ID'sinin root olmasıdır. Process'in grup ID'sinin ya da etkin grup ID'sinin root olması yalnızca root dosyalarına erişimde bir ayrıcalık sağlar.

POSIX güvenlik sisteminde ilginç bir kusuru örtmek için saved set user ID denilen ilginç bir ID kavramı da tanımlanmıştır. saved set user ID exec sonrasında etkin kullanıcı ID'si olarak set edilir. Örneğin etkin kullanıcı ID'si A olan bir process fork ve exec yaparak B

kullanıcısına ait X dosyasını çalıştırsın. X'in set user ID bayrağı set edilmişse exec sonrasında alt process'in etkin user ID'si B olur ve saved set user ID'si de B olur.

setuid ve setgid Fonksiyonları

Bu fonksiyonlar basit gibi görünmesine karşın nispeten karmaşık bir kontrol ve set işlemi yaparlar.

```
int setuid(uid_t uid);
int setgid(gid_t gid);
```

Bu fonksiyonlar,

1) Eğer process'in etkin kullanıcı ID'si root ise process'in kullanıcı ID'sini, etkin kullanıcı ID'sini ve saved set user ID'sini set eder. Benzer biçimde setgid fonksiyonu da eğer etkin kullanıcı ID'si root ise process'in grup ID'sini, etkin grup ID'sini parametresiyle belirtilen ID değeri olarak set eder.
2) Eğer process root hakkına sahip değilse, parametreyle belirtilen ID değeri kullanıcı ID değeri ya da saved set user ID değerine eşitse setuid fonksiyonu process'in etkin kullanıcı ID'sini parametresiyle belirtilen ID değeri olarak set eder. Benzer biçimde setgid fonksiyonu da bu koşullarda process'in etkin grup ID'sini set etmektedir.
3) Fonksiyonlar başarısız olursa –1 değerine geri dönerler ve bir erişim nedeniyle başarısız olmuşlarsa errno EPERM değerine set edilir.

Görüldüğü gibi process root hakkına sahip değilse bu fonksiyonlar yalnızca etkin kullanıcı ID'sini ve etkin grup ID'sini set ederler. Ancak bunun için parametresinin kullanıcı ID değerine ya da saved set user ID değerine eşit olması gerekir. Örneğin process'in kullanıcı ID'si A, fakat etkin kullanıcı ID'si B olsun. setuid fonksiyonu ile etkin kullanıcı ID'sini A yapabilir.

Saved set user ID neden tanımlanmıştır?

Saved set user ID tipik olarak set user ID bayrağı set edilmiş bir program exec yapıldığında exec yapılan process'in

```
setuid(getuid());
```

ile eski etkin kullanıcı ID'sine geri dönüp sonra yeniden exec yapıldığı durumdaki yeni etkin kullanıcı ID'sine geçmek için düşünülmüştür. Örneğin kullanıcı ID'si A olan process set user ID bayrağı set edilmiş bir programı çalıştırarak etkin kullanıcı ID'si B olarak devam etsin. Process'in saved set user ID'si de B olmuştur. Şimdi process A'nın dosyalarına erişmek istese erişemez. Ancak etkin kullanıcı ID'sini A yaparak bu erişimi gerçekleştirir.

```
id = geteuid();
setuid(getuid());
...
...
```

Şimdi process'in kullanıcı ID'si A, etkin kullanıcı ID'si A fakat saved set user ID'si B'dir. Şimdi process A'nın dosyalarına erişebilir. Bu işlemden sonra process tekrar etkin kullanıcı ID'sini B yaparak devam edebilir.

```
setuid(id);
```

Saved set user ID kavramı POSIX standartlarına girmiştir. Bazı sistemler (BSD4.3 ve sonrası ve SystemV-R4) saved set user ID kavramı yerine etkin kullanıcı ID'si ile kullanıcı ID'sini yer değiştiren ve etkin grup ID'si ile grup ID'sini yer değiştiren `setreuid` ve `setregid` fonksiyonlarına sahiptir.

Process ID'leri İle İlgili POSIX Fonksiyonları

Bu bölümde process ID'leri ile ilgili fonksiyonların listesi incelenecektir.

getpid Fonksiyonu

Yaratılan alt process'in ID değeri fork fonksiyonunun geri dönüş değeri olarak üst process'e verilmektedir. Bir process istediği zaman kendi ID değerini getpid fonksiyonu ile alabilir.

```
pid_t getpid(void);
```

getppid Fonksiyonu

Bu fonksiyon üst process'in ID değerine geri döner.

```
pid_t getppid(void);
```

Maalesef POSIX sistemlerinde sistemdeki tüm process'leri elde edebilecek taşınabilir sistem fonksiyonları yoktur. ps programı çekirdek ile bütünleşmiş bir program olduğundan bunu yapabilmektedir.

Kullanıcı ve Grup ID'leri İle İlgili POSIX Fonksiyonları

Normal, etkin kullanıcı ve grup ID'leri aşağıdaki fonksiyonlarla alınabilir.

```
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

Bu fonksiyonların dışında set etme işlemi için aşağıdaki fonksiyonlar vardır.

```
int setuid(uid_t uid);
int setgid(gid_t gid);
```

dup ve dup2 Fonksiyonları

Aynı dosya iki kez open fonksiyonuyla açıldığında iki ayrı dosya betimleyicisi ve dosya nesnesi oluşturulur. Bilindiği gibi dosya göstericisinin konumu, open fonksiyonunda belirtilen açma modu dosya nesnesi içerisinde tutulmaktadır. Ancak bir dosyayı iki kez açmak yerine eğer dosyaya ilişkin iki tane dosya betimleyicisi isteniyorsa aynı dosya nesnesini gören farklı betimleyiciler kullanılır. Bu işlem dup fonksiyonuyla yapılır.

```
int dup(int oldfd);
```

Fonksiyonun parametresi çoğaltılacak betimleyici değeridir. Fonksiyon geri dönüş değeri olarak aynı dosya nesnesine ilişkin yeni bir betimleyici verir. Fonksiyonun ilk boş betimleyiciyi verdiği garanti altına alınmıştır. Örneğin;

```
int fd;
fd = open(....);
close(1);
dup(fd);
```

Burada önce stdout dosyasına ilişkin 1 numaralı betimleyici kapatılmıştır. Sonra açılan dosya dup fonksiyonu ile 1 numaralı slota yazdırılmıştır.

Burada stdout başka bir dosyaya yönlendirilmiştir. dup fonksiyonunun geri dönüş değerine bakılmamıştır. Gerçi dup fonksiyonu başka nedenlerden başarısız olabilir. dup fonksiyonu başarılıysa 1 değerine geri dönecektir. Bu işlemden geri dönüş de mümkündür. Yani programcı 1 numaralı betimleyicinin yine ekrana ilişkin olmasını sağlayabilir. Örneğin;

```
close(1);
dup(2);
```

gibi bir işlemle stderr'den faydalanılarak yeniden eski duruma dönüş yapılabilir.

dup eğer dosya betimleyici tablosu doluysa başarısız olur ve –1 değerine geri döner. Fonksiyonun prototipi <unistd.h> dosyası içerisindedir.

Sınıf çalışması: close, dup, fork ve exec fonksiyonlarını kullanarak başka bir programı çalıştırınız, ama çalıştırılan programdaki stdout çalıştıran programda belirlenen dosyaya yönlensin. Program aşağıdaki gibi çalıştırılacaktır.

```
direct<çalıştırılcak_dosya_ismi><stdout_yönlendirmesi_yapılacak_dosya_ismi>
```

Yönlendirilecek dosya önce açılır. Daha sonra,

```
close(1);
dup(fd);
```

uygulanır. Daha sonra fork ve exec yapılır. Sonra çalıştırılan programın sonlanması beklenir.

Anahtar notlar: *Bir dosya fopen standart C fonksiyonuyla açılmış olsun. Eğer programcı aşağı seviyeli çalışmaya geçecekse fileno fonksiyonuyla dosya betimleyici değerini alabilir.*

```
int fileno(FILE *fp);
```

Şüphesiz betimleyici değeri (handle değeri) FILE yapısı içerisinde tutulmaktadır. Standart C fonksiyonlarıyla çalışırken eğer tamponlamasız modda değilsek fork ve exec işlemlerinden önce fflush yapılması gerekmektedir.

/* direct.c */

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char **argv)
{
      int fd;
      FILE *fp;
      pid_t pid;

      fp = fopen("den.dat", "wb");
      if (fp == NULL) {
            fprintf(stderr, "Cannot open file...\n");
            exit(0);
      }
      fd = fileno(fp);

      close(1);
      dup(fd);

      if ((pid = fork()) == -1) {
            perror("fork");
            exit(0);
      }
      if (pid == 0) {
            if (execlp(argv[1], argv[1]) < 0) {
                  perror("exec");
                  exit(0);
            }
      }
      wait(NULL);

      return 0;
}
```

close ve dup fonksiyonları yerine önce close, sonra open fonksiyonları kullanılabilir. Çünkü open fonksiyonu da boş en küçük dosya betimleyici değerine döner.

dup fonksiyonu öncesinde close kullanılırken işlemlerin atomik olmamasından dolayı bazı uygulamalarda signal problemi oluşabilir. Yani close ile dup arasında oluşan signal mekanizmayı bozabilir. dup2 fonksiyonu close ve dup işlemini yapan ve atomik olan bir fonksiyondur.

```
int dup2(int oldfd, int newfd);
```

Bu fonksiyon ikinci parametresiyle belirtilen dosyayı kapatarak birinci parametresiyle belirtilen dosya betimleyicisinin kopyasından çıkartır. Elde edilen betimleyicinin ikinci parametresiyle belirtilen betimleyici olduğu garanti altına alınmıştır. Örneğin;

```
dup2(fd1, fd2);
```

Burada önce fd2 kapatılır. Sonra fd1'in kopyası fd2 slotu kullanılarak çıkartılır. Fonksiyon başarılıysa fd2 değeriyle, başarısızsa –1 değeriyle geri döner. fd2'nin kapatılması ve yeni betimleyicinin oluşturulması atomik işlemlerdir. Yani herhangi bir kesilme oluşmaz.

Unix/Linux Sistemlerinde Process Hiyerarşisi (Yaratılma Aşamaları)

Tipik bir Unix türevi sistemde process'ler belirli bir sıraya göre yaratılmaktadır. Kullanıcının login olması işlemine kadar bir dizi process yaratılır.

Çekirdek yüklenirken ismine `swapper` ya da `sched` denilen ve temel process'ler arası geçiş mekanizması için kullanılan 0 ID'ye sahip bir process yaratılmaktadır.

swapper process'i fork ve exec ile `init` process'ini yaratır. init process'inin ID değeri 1'dir. init process'i zombie durumda olan process'lere üst process görevi yapar ve onları sonlandırır. init terminal işlemleri için genellikle `getty` diye isimlendirilen (çeşitli sistemlerde buna benzer çeşitli isimler kullanılmaktadır) terminal oluşturan programı fork ve exec fonksiyonlarıyla çalıştırır. getty programı genellikle sistemde terminal özelliklerinin bulunduğu bir dosyaya başvurur. Yani bu program komut satırı argümanı olarak yaratılacak terminale ilişkin bilgilerin bulunduğu dosyanın ismini almaktadır. Terminal bilgileri genellikle /etc/ttys gibi bir dizin altında bulunmaktadır. Terminal yalnıza monitör anlamına gelen bir kavram değildir, monitör ve klavyeden oluşan bir kavramdır. Bundan sonra getty `open` fonksiyonuyla /dev dizini altında bulunan ilgili terminal dosyasını önce read modunda, sonra write modunda açar. Sonra 1 numaralı betimleyiciye dup işlemini uygular. Böylece getty programı stdin, stdout ve stderr dosyalarına ilişkin betimleyicileri oluşturmuş olur. Terminal Unix sistemlerinde bir dosya gibi işlem görür. Örneğin programcı isterse /dev dizininin altında bir terminali getty'nin yaptığı gibi açabilir.

Bu işlemden sonra getty `setenv` fonksiyonuyla TERM gibi terminallere ilişkin çeşitli çevre değişkenlerini oluşturur.

Artık sıra login işlemine gelmiştir. getty programı bu aşamada username yazısını çıkartarak kullanıcı ismini sorar. Bundan sonra getty username bilgisini komut satırı argümanı yaparak fork ve exec fonksiyonuyla `login` programını çalıştırır. Artık login programı çalışmaktadır. login programı kullanıcıdan password ister, password'ü encrypt fonksiyonuyla şifreler. login `getpwnam` fonksiyonu ile kullanıcı ismine karşılık passwd dosyasından kullanıcı bilgilerini çeker. passwd dosyasındaki password şifrelenmiş olduğuna göre şifre uyuşumunu kontrol edebilir. Eğer şifre uyuşumu sağlanmamışsa login kendisini exit(1) ile sonlandırır.

Bu sırada getty sonlanmıştır. init aynı işlemleri yeniden yaparak aynı noktaya getirir. Aslında pek çok Unix sisteminde getty programının login yerine başka bir programı çalıştırması sağlanabilir. Örneğin /etc/gettytab gibi bir dosyadan pek çok sistemde bunu değiştirmek mümkündür.

login password uyuşumunun sağlandığını görmüş olsun. Bu noktada gerçek ve etkin kullanıcı ID'si root'tur. login setuid ve setgid fonksiyonlarını kullanarak passwd dosyasında belirtilen kullanıcı ve grup ID'lerini set eder. Bu noktada process artık root değil sıradan bir process haline gelmiştir.

Bundan sonra login programı HOME, SHELL, USER, LOGNAME, PATH gibi çevre değişkenlerini set eder. Sonra chdir fonksiyonu ile passwd dosyasında belirtilmiş olan home dizinine geçer. Artık login fork ve exec fonksiyonlarını kullanarak passwd dosyasında belirtilen shell programını çalıştırır.

Bundan sonra shell çeşitli başlangıç script dosyalarını (Dos'ta autoexec.bat gibi) çalıştırır. Örneğin tipik bir Unix sisteminde bu dosyalar bsh için .login, bash için .login_bash biçimindedir. shell çalıştırıldığında artık login ve getty programları sonlanmıştır. logout komutuyla shell sonlanır ve init programı terminal için yeniden aynı işlemleri yapar.

Pek çok sistemde burada anlatılanlarda küçük değişiklikler olabilmektedir. Örneğin shell'in çalışmış olduğu durumda getty sonlanmış, ama login sonlanmamış olabilir. Böylece shell'de logout yapıldığında dönüş yeri init değil login olabilir.

Yorumlayıcıyla Çalışan Dosyalar

Belirli bir dilde yazılmış olan kaynak programı hiç çalışabilir kod üretmeden satır satır çalıştıran programlara yorumlayıcı denir. Yorumlayıcılar çalışabilen programlardır. Genellikle komut satırı argümanı olarak kaynak dilde yazılmış olan programı alırlar. Onları ifadelere ayırarak yorumlayıp çalıştırırlar. Unix sistemlerinde exec fonksiyonları yalnızca makine diline dönüştürülmüş ELF ya da a.out dosyalarını değil, belli bir formatta düzenlenmiş text dosyalarını da çalıştırabilirler. exec fonksiyonlarının bir text dosyayı çalıştırabilmeleri için dosyanın ilk satırının şu biçimde olması gerekir:

```
#! <çalışabilen dosyanın full path bilgisi>
```

exec fonksiyonları ilk satırı yukarıdaki gibi başlatılmış olan bir text dosyayı nasıl çalıştırır. Aslında bu text dosyayı değil, ilk satırında belirtilmiş olan gerçek çalıştırılabilir dosyayı çalıştırır. Bu dosyayı o programa komut satırı argümanı yapar. Peki bunun tam tersi olması daha doğal değil midir? Yani çalışabilen programı çalıştırıp kaynak kodun bulunduğu dosyayı komut satırı argümanı yapmak. Sistemin diğer biçimde tasarlanmasının iki nedeni vardır:

1) Eğer diğer yöntem benimsenseydi, önce gerçek exec işlemi yapılacaktı. Fakat komut satırı argümanı ile belirtilen kaynak dosyada problem olduğunda yorumlama işlemi yapılmayacaktı ve bu durum vakit kaybına yol açacaktı. Halbuki bu biçimde text dosyanın ilk satırından daha gerçek exec işlemi yapılmadan error durumu anlaşılabilir.
2) Bu biçimde kaynak kodun birinci satırını yeniden düzenleyerek yorumlayıcı dosyanın başka bir yorumlayıcı program tarafından yorumlanması sağlanabilir. Awk gibi klasik shell script yorumlayıcıları bu teknikle çalışırlar.

Tabii bu biçimdeki kaynak dosyaların exec ile çalıştırılabilmesi için text dosyanın x özelliğine sahip olması gerekir.

Yorumlama Dosyalarının exec İle Çalıştırılması

Yorumlama dosyasının ismi x olsun ve bu dosyanın ilk satırında

```
#! /home/kaan/y str3 str4
```

olsun. x dosyası aşağıdaki gibi exec yapılmış olsun.

```
execl("/home/kaan/x", "str1", "str2", NULL);
```

Bu işlem sonrasında /home/kaan/y çalıştırılır. Y programın komut satırı argümanları şöyle olur:

```
argv[0]: /home/kaan/y
argv[1]: str3 str4
argv[2]: /home/kaan/x
argv[3]: str1
argv[4]: str2
```

Görüldüğü gibi çalışabilen y programına argv[1] olarak yorumlama dosyasının ilk satırındaki argümanların hepsi birlikte geçirilmiştir.

Sınıf çalışması: Bir x text dosyası oluşturunuz. Bunun birinci satırına

```
#! /home/.../y    [l] ya da [s]
```

yazınız. Sonra bu dosyanın içerisine her satırına bir yazı yazınız. Birinci satırdaki seçenek l ise satırlardaki bilgiyi satır satır yazdırınız, s ise yan yana yazdırınız. x dosyasını chmod komutuyla çalışabilir hale getiriniz. x'i shell üzerinde çalıştırınız.

signal Kavramı

signal Unix/Linux programlamanın en önemli konularından biridir. signal bir çeşit yazılım kesmesidir. signal bir takım olaylar oluştuğunda işletim sisteminin process'in belirlenmiş olan bir fonksiyonunu çağırması durumudur. signal mekanizması içerisinde belirlenen fonksiyon asenkron bir biçimde çağırılır. Yani örneğin işletim sistemi A process'ini çalıştırıyor olsun. O anda B process'ine ilişkin bir signal durumu bildirilsin. Muhtemelen işletim sistemi çalışmakta olan A process'ine ara verecek, B process'ine geçecek, ancak B process'inde kalınan yerden devam etmeyecek, yalnızca belirtilen fonksiyonu çağıracaktır. Bundan sonra işletim sistemi kendisinin uygun gördüğü çizelgeleme algoritmasıyla devam edecektir. Aslında bir process çalıştırılırken başka bir process için signal oluştuğunda işletim sisteminin nasıl davranacağı standart olarak belirtilmemiştir. Ancak sistemlerin çoğu signal işlemlerini bir an önce gerçekleştirmek için process'ler arası geçiş yaparlar.

signal Oluşturan Durumlar

signal temel olarak üç durum karşısında oluşturulur.

1) *Klavye yoluyla:* Unix sistemlerinde terminallerdeki bazı özel tuşlar bazı signal'lerin oluşmasına yol açar. Örneğin tipik olarak Ctrl+C tuşu SIGINT signal'inin oluşmasına neden olur. Benzer biçiminde genellikle Ctrl+Z tuşu TSTP (task stop) signal'ini, Ctrl+\ tuşu ABRT signal'ini oluşturmaktadır. Bu signal'leri oluşturan tuşlar Unix'te sistemden sisteme değişebilir. En iyi yöntem `stty –a` komutuyla kullanılan terminal için bu tuş kombinasyonları görmektir. İşletim sistemlerinin çoğunda klavyeden hareketle signal benzeri bir çağırma yapan mekanizmalar bulunur. Örneğin, DOS sisteminde klavyeden okuma yapan

bazı DOS fonksiyonları Ctrl+C'ye basıldığında Control Break denilen bir kesmeyi çağırmaktadır.
2) *Shell üzerinden signal yollamak:* Shell üzerinden kill komutu uygulanarak signal yollanabilir. Kill komutunun genel biçimi şöyledir:

```
kill –signal <process id>
```

Her signal'in bir sembolik ismi vardır. Örneğin,

```
kill –ABRT 1003
```

Komutu process ID'si 1003 olan process'e ABRT signal'ini gönderir. – belirleyicisinden sonra sembolik isim yerine doğrudan signal numarası da kullanılabilir. Eğer signal kısmı hiç yazılmazsa default olarak process'e TERM signal'i gönderilir. Bu da process'i sonlandırır.

Sınıf çalışması: Bir console'dan getchar ile bekleme yapan basit bir C programı yazınız. Başka bir console açarak o process'i kill komutuyla yok ediniz.

Bazı shell programlarında `fg` gibi bir komut suspend edilmiş bir process'i CONT signal'ini göndererek devam ettirmektedir.

3) *Sistem fonksiyonları kullanarak signal göndermek:* Signal göndermekte kullanılan en önemli iki sistem fonksiyonu kill ve raise fonksiyonlarıdır. kill fonksiyonu herhangi bir process'e signal gönderebilirken, raise kendi process'ine signal gönderebilmektedir. Zaten shell üzerinde kill komutunu uyguladığımızda kill fonksiyonunu çağırılmaktadır.

kill ve raise Fonksiyonları

kill fonksiyonu process ID'si bilinen bir process'e mesaj göndermekte kullanılır. Prototipi <signal.h> içerisindedir.

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

Fonksiyonun birinci parametresi signal gönderilecek process'in ID'si, ikinci parametresi gönderilecek signal'in numarasıdır. Signal numaraları <signal.h> dosyası içerisinde SIGXXX biçimindeki sembolik sabitler olarak define edilmiştir. Fonksiyon başarılıysa 0 değerine, başarısızsa –1 değerine geri döner. root her process'e signal gönderebilir. Normal bir kullanıcı, kullanıcı ID'si ya da etkin kullanıcı ID'si aynı olan process'lere mesaj gönderebilir. Örneğin process'in kullanıcı ID'si 10 etkin kullanıcı ID'si 20 olsun. Biz bu process içerisinde kill fonksiyonu ile kullanıcı ID'si 10 ya da 20, etkin kullanıcı ID'si 10 ya da 20 olan process'lere mesaj gönderebiliriz. Örneğin bir process'imiz kilitlenmiş olsun ve onu sonlandırmak isteyelim. Başka bir console'dan girerek kill komutuyla process'imizi yok edebiliriz.

raise fonksiyonu şöyledir:

```
int raise(int signal);
```

raise kendi process'ine ilgili signal'i gönderir.

```
raise(signal);
kill(getpid(), signal);
```

ifadeleri benzerdir, ama aynı değildir. kill fonksiyonundaki pid değeri dört seçenekten biri olabilir.

1) `pid > 0 :` Bu durumda signal belirtilen ID'ye ilişkin process'e gönderirilir.
2) `pid == 0 :` Bu durumda signal process grup ID'si fonksiyonu çağıranın process ID'sine eşit olan tüm process'lere gönderilir. Yani bu biçimde process grup lideri bütün üyelerine signal gönderebilmektedir.
3) `pid < 0 :` Bu durumda signal process grup ID'si |pid|'ye eşit olan process'lere gönderilir.
4) `pid == -1 :` Bu POSIX sistemlerinde belirsiz bırakılmıştır.

signal Gönderildiğinde Neler Olur?

Bir signal oluştuğunda prensip olarak sistem mümkün olduğu kadar çabuk signal fonksiyonunu çağırmaya çalışır. signal oluştuğunda sistemin davranışı standart olarak belirlenmemiştir. Ancak bu prensip altında bir signal oluştuğunda sistem o anda çalıştırılmakta olan process'e ara verir, signal gönderilen process'e geçilerek signal fonksiyonunu çağırır. Ancak signal konusunda önemli bir problem vardır. Signal gönderilen process yavaş bir sistem fonksiyonu içerisindeyse (örneğin read/write gibi fonksiyonlar) ne olacaktır? Sistem fonksiyonlarının yarıda kesildiği durumda signal fonksiyonu çağrılırsa, signal fonksiyonunda da benzer işlemlerin yapılması durumunda sistem çökebilir. Bu durumda iki seçenek söz konusudur.

1) Sistem fonksiyonuna ara verilir, ama sistem fonksiyonunun başarısız olduğu kabul edilerek sistem fonksiyonundan çıkılır. Bu durumda signal hızlı bir biçimde işlem görmüş olur, ama sistem fonksiyonu başarısız olur. Eski AT&T SystemV bu biçimde çalışıyordu. Bu sistemlerde yavaş sistem fonksiyonları başarısız olduğunda errno değişkenine bakılarak başarısızlığın signal dolayısıyla olup olmadığı tespit edilmeye çalışılır. Eğer başarısızlık signal dolayısıyla olmuşsa sistem fonksiyonu programcı tarafından yeniden çağrılmalıdır.
2) Sistem, sistem fonksiyonu içerisindeyken signal geldiğinde işlemine ara verir, signal işlemini ele alır, ama signal işlemi bittikten sonra sistem fonksiyonunu yeniden kendisi çağırır. BSD sistemleri bu yöntemi kullanmaktadır. Unix sistemleri genel olarak sistem fonksiyonu içerisinde signal'i yasaklayan bir tasarımı benimsememişlerdir.

Klasik AT&T SystemV'in diğer bir problemi bir signal oluştuğunda sistemin o signal'in durumunu default'a çekmesidir. Yani bir signal oluştuğunda çağırılacak fonksiyon Func olsun. Func fonksiyonunun çalıştırıldığı noktada sistem signal'ı default değere çeker. O noktada tekrar aynı signal'in oluşması bu signal için default işlem yapılmasına, yani process'in sonlanmasına neden olur. Bu tür durumlarda iç içelik sağlamak için signal fonksiyonunun başında yeniden set işlemi yapsak bile maalesef öyle bir zaman aralığı oluşur ki akış signal geldiğinde çağrılacak fonksiyona girmiştir, ama set işlemini yapacak fonksiyona henüz girmemiştir. Böyle zaman aralığına giren durumlarda signal default değere çekilebileceğinden process sonlanabilmektedir. Görüldüğü gibi eski SystemV'lerde bir signal geldiğinde çalıştırılacak fonksiyon içerisinde kötü bir tesadüf ile aynı signal'den oluştuğunda process'in sonlanması gibi bir komplikasyon oluşur.

SystemV'in signal mekanizması maalesef güvenli değildir ve problemlidir. Daha sonra bu problemi çözmek üzere çeşitli gelişmeler olmuştur. Berkeley sistemleri (BSD) problemin çözümü için signal oluştuğunda signal'i default değere çekmeyi kaldırmıştır ve yavaş sistem fonksiyonları içerisinde signal oluştuğunda bu sistem fonksiyonlarını kendi çağrılmasını otomatikleştirmiştir. SystemV daha sonraki sürümlerinde yeni güvenli sistem fonksiyonları tanımlayarak bu işlemi güveli hale getirmiştir. Ancak eski signal fonksiyonları da çalışmaya devam etmiştir. POSIX standartları da yeni güvenli signal fonksiyonları tanımlamıştır. Özetle signal işlemlerinde kullanılan fonksiyonlar iki kısma ayrılabilir:

1) Güvensiz fonksiyonlar. Bu güvensiz fonksiyonlar BSD sistemlerinde güvenli hale getirilmiştir. Ancak SystemV sistemlerinde hala güvensizdir.
2) POSIX standartlarında tanımlanmış güvenli fonksiyonlar.

signal Oluştuğunda Karşılaşılan Üç Durum

Bir signal oluştuğunda üç şey olabilir.

1) signal dikkate alınmaz, yani hiçbir şey yapılmaz.
2) signal oluştuğunda sistem tarafından default bir fonksiyon çağrılır. Hangi signal karşılığında default olarak ne yapılacağı daha önceden belirlenmiştir.
3) signal oluştuğunda bizim belirlediğimiz bir fonksiyon çağrılır.

Hangi signal'lara karşı hangi durumun belirlendiği process handle tablosunda tutulmaktadır. Bu bilgi fork işlemi sırasında alt process'e aktarılır. Bu durumda shell üzerindeki belirleme yani shell process'inin belirlemesi bizim için önemli olmaktadır. Shell process'inin birkaç istisna dışında signal durumları default değerdedir. Bilindiği gibi exec işlemi sırasında process'in bellek alanı değiştirilmektedir. Bu durumda görmemezlikten gelinen ya da default'a çekilen signal'ler için problemli bir durum oluşmaz. Ancak belirli bir fonksiyona set edilmiş signal'lerde problem oluşabilir. İşte exec fonksiyonu belirli bir fonksiyon ile set edilmiş signal'leri default değere çeker.

signal Fonksiyonu

Bir signal oluştuğunda sistemin ne yapacağı signal fonksiyonuyla belirlenir. signal fonksiyonu AT&T'nin eski bir fonksiyonudur. Yani SystemV ve türevlerinde bu fonksiyon ile set işlemi yapıldığında yukarıda açıklanmış olan komplikasyonlar oluşabilir. signal fonksiyonları BSD sistemlerinde güvenlidir. Linux sistemlerinde signal fonksiyonu SystemV'te olduğu gibi güvensizdir.

signal fonksiyonu POSIX standartlarına sokulmamıştır. signal fonksiyonu yerine POSIX standartlarında bir grup farklı signal işlemlerinde kullanılan yeni fonksiyon tanımlanmıştır. signal fonksiyonunun prototipi şöyledir:

```
void (*signal(int signum, void (*sighandler)(int)))(int);
```

signal fonksiyonunun birinci parametresi set edilecek signal'ın numarasıdır. POSIX sistemlerinde 31 signal vardır. Her signal SIGXXX biçiminde sembolik sabitle belirtilmiştir. Fonksiyonun ikinci parametresi signal fonksiyonu olarak set edilecek fonksiyonun başlangıç adresidir. Signal geldiğinde çağırılacak fonksiyonun (signal handler) geri dönüş değeri void,

parametresi int olmak zorundadır. Fonksiyon daha önceki signal fonksiyonunun başlangıç adresine geri döner. Fonksiyon başarısızsa SIG_ERR değerine geri döner. Bu değer tipik olarak şöyle define edilmiştir:

```
#define SIG_ERR ((void(*)(int))-1)
```

Anahtar Notlar: 70'li ve 80'li yılların ortalarına kadar prototip kavramı C'de yoktu. O devirlerde fonksiyon parametreleri eski biçimde tanımlanıyordu, yani parametre parantezinin içerisine bir tür bilgisi yazılmıyordu. Fonksiyon bildirimleri parametre parantezinin içi boş bırakılarak yapılıyordu. Örneğin:

```
long Func();
```

Bu biçimde bildirilen fonksiyonlar herhangi bir parametrik yapıyla çağırılabiliyordu. 80'li yılların ortalarına doğru prototip kavramı C'ye sokuldu. Yeni biçimdeki parametre bildirimi ile prototip kavramı beraber C'ye girmiştir. Ancak derleyiciler eski biçimdeki parametre bildirimlerini ve fonksiyon bildirimlerini kabul etmeye devam ettiler (C90 standartlarında bu durum deprecated yapılmıştır). Özetle C90'da fonksiyon parametre parantezinin içi boş bırakıldığında parametre kontrolü yapılmayacağı anlamına gelmektedir. C99'da ve C++'ta eski biçimdeki parametre bildirimi tamamen kaldırılmıştır. Bu dillerde parametre parantezinin içi boş bırakıldığında void kabul edilir. Ayrıca C90'da fonksiyon göstericileri tanımlanırken parametre parantezinin içinin boş bırakılması da bu göstericiye geri dönüş değeri uygun olmak şartıyla parametrik yapısı herhangi bir biçimde olan fonksiyonun adresinin atanabileceği anlamına gelir.

signal fonksiyonunun ikinci parametresi özel olarak SIG_DFL ya da SIG_IGN olabilir. Bu sembolik sabitler tipik olarak aşağıdaki gibi tanımlanmıştır:

```
#define SIG_DFL (void (*)(int)) 0)
#define SIG_IGN (void (*)())1)
```

SIG_DFL signal durumunu default'a çekmek için, SIG_IGN signal'ı görmemek için kullanılır.

Signal oluştuğunda çağırılacak fonksiyonun parametresi int türdendir. Bu parametreye oluşan signal'ın numarası geçirilir.

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

void SignalHandler(int sno)
{
      printf("signal!..\n");
}

int main(void)
{
      if (signal(SIGINT, SignalHandler) == SIG_ERR) {
            perror("signal");
            exit(1);
      }

      pause();
      printf("signal sonrası\n");
```

```
	return 0;
}
```

pause Fonksiyonu

pause fonksiyonu herhangi bir signal oluşana kadar process'i bloke eder.

```
int pause(void);
```

Fonksiyon başarısızlık durumunda -1 değerine geri döner. Başarı durumunda herhangi bir değer öngörülmemiştir. Örneğin bir process'te aşağıdaki gibi bir işlem yapılsın:

```
for(;;)
	pause();
```

Bu durumda process yalnızca signal oluşumunda aktif hale gelecektir. Şüphesiz process'in sonlanması da signal oluştuğunda çağırılacak fonksiyon tarafından yapılacaktır.

alarm Fonksiyonu ve SIGALRM Signal'ı

alarm fonksiyonu belirli bir saniye sonrasında signal üreten temel bir fonksiyondur.

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

Fonksiyon belirlenen saniye dolduktan sonra SIGALRM signal'ını oluşturmaktadır. Programcı tipik olarak SIGALRM numaralı signal'ı set eder. Signal oluştuğunda çağırılacak fonksiyon içerisinde yeniden alarm fonksiyonunu çağırır. Ana programda da genellikle döngü içerisinde pause ile bekleme yapar.

Sınıf Çalışması: UNIX'in ünlü `at` komutuna benzer bir programı yazınız. Program aşağıdaki gibi çalıştırılacaktır:

```
myat saat:dakika:saniye prog
```

Açıklamalar: Program SIGALRM signal'ını set eder, sonra alarm fonksiyonu ile 1 saniyelik aralıklarla signal oluşmasını sağlar. Ana programda döngü içerisinde pause ile beklenebilir. at standart bir POSIX komutudur. Şüphesiz bu program başka biçimlerde daha etkin düzenlenebilir.

Process'in toplam bir tane alarm zamanlayıcısı vardır. Örneğin alarm fonksiyonu önce 5, sonra 10 parametresiyle çağırılırsa zamanlayıcı 10'a set edilir. Örnek:

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

void OnTime(int sno)
{
	static int i = 0;
```

```
	printf("%d\n", i);
	if (i == 20)
		exit(1);
	++i;
	alarm(1);
}

int main(void)
{
	if (signal(SIGALRM, OnTime) == SIG_ERR) {
		perror("signal");
		exit(1);
	}
	alarm(1);
	for (;;)
		pause();

	return 0;
}
```

alarm fonksiyonuna parametre olarak 0 girilirse daha önce set edilmiş olan alarm iptal edilir. Ancak signal oluştuğunda çağırılacak fonksiyon çağırılmaz.

SIGUSR1 ve SIGUSR2 Signal'ları

Normal olarak signal'lar belirli olaylar gerçekleştiğinde oluşur. Örneğin klavyede Ctrl + C tuşuna basıldığında ya da alarm zamanı tükendiğinde oluşur. İstisna olarak SIGUSR1 ve SIGUSR*2* numaralı signal'lar herhangi bir yazılım ya da donanım olayına bağlı değildir. Bu signal'lar programlama yoluyla, yani kill fonksiyonu ile oluşturulur. Bu signal'lar sayesinde bir program çalışırken dışarıdan müdahale ile programın işleyişinde değişiklikler yapılması mümkün olabilir.

Sınıf Çalışması: Bir alt process yaratınız (fork ile), alt process'te SIGUSR1 signal'ını set ediniz. Üst process'ten kill fonksiyonu ile alt process'e SIGUSR1 signal'ını göndererek alt process'in bunu işlemesini sağlayınız. Alt process'e SIGUSR1 signal'ı gönderildiğinde alt process kendini sonlandırmalıdır.

```
/* signal2.c */

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

void OnExit(int sNo);

int main()
{
	pid_t pid;

	if ((pid = fork()) < 0) {
		perror("fork");
		exit(EXIT_FAILURE);
	}
```

```
      if (pid == 0) {
            if (signal(SIGUSR1, OnExit) == SIG_ERR) {
                  perror("signal");
                  exit(EXIT_FAILURE);
            }
            pause();
      }
      sleep(1);   // child process'in signal fonksiyonunu
                  // işleyebilmesi için süre

      if (kill(pid, SIGUSR1) == -1)
            printf("kill basarisiz oldu\n");

      if (wait(NULL) == -1)
            printf("wait basarisiz oldu\n");

      printf("Parent prog finised\n");

      return 0;
}

void OnExit(int sNo)
{
      printf("Child Prog finised\n");
      exit(10);
}
```

sleep Fonksiyonu

Bu fonksiyon parametresiyle belirtilen saniye kadar ya da herhangi bir signal oluşana kadar process'i çizelge dışı bırakır.

```
unsigned int sleep(unsigned int seconds);
```

Fonksiyonun parametresi beklenecek saniye miktarıdır. Fonksiyon eğer bir signal nedeniyle process uyandırılmışsa kalan zaman miktarına, zaman dolması nedeniyle uyandırılmışsa 0 değerine geri döner. Prototipi <unistd.h> dosyası içerisindedir.

sleep fonksiyonun bazı sistemlerde alarm ve SIGALRM signal'leri kullanılarak yazılmış olabilir. Ancak bu tür yazımların çeşitli problemleri vardır. signal fonksiyonunun içsel olarak yazılması daha anlamlıdır.

Alarm signal ve pause fonksiyonları kullanılarak bir sleep fonksiyonu kusurlu bir biçimde aşağıdaki gibi yazılabilir.

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

void sig_alrm(int n);

unsigned int mysleep(unsignde seconds)
{
      if (signal(SIGALRM, sig_alrm) == SIG_ERR)
            return seconds;
      alarm(seconds);
```

```
	/*
	 * task switch bu noktada oluşursa ve alarm fonksiyonuna
	 * parametre olarak verilen seconds süresinden daha sonra
	 * process'e tekrar sıra geldiğinde signal gelmiş olur,
	 * fakat signal'in oluştuğu anda henüz pause fonksiyonu
	 * çağrılmamış olduğu için signal'i kaçırmış oluruz.
	 */
	pause();
	return alarm(0);
}

void sig_alrm(int n)
{
	return;
}
```

Buradaki en önemli kusur alarm ile pause fonksiyonlarının arasında process'ler arası geçiş oluşması ve sistemin yüklü olduğu bir durumda alarm ile belirtilen zamanın pause fonksiyonundan önce tükenmesidir. Görüldüğü gibi sleep fonksiyonunu pause fonksiyonunun süreli versiyonudur.

longjmp İşlemleri

C'de goto deyiminin etiketi fonksiyon faaliyet alanında olmalıdır. Yani goto ile başka bir fonksiyona dallanılamaz. Bunun için setjmp ve longjmp fonksiyonları kullanılır. Bu fonksiyonlar standart C fonksiyonlarıdır.

Başka bir fonksiyona dallanmanın en önemli problemi stack dengelemesinin yapılması ve bazı yazmaçların saklanmasıdır. Aslında daha önceki bir noktaya dallanmak demek o noktadaki CPU yazmaçlarının değerlerinin yeniden yüklenmesi demektir. Tipik setjmp ve longjmp fonksiyonları şöyle tasarlanır:

1) longjmp ile ancak daha önce geçilmiş olan bir bölgeye dönülebilir. İşte setjmp fonksiyonu o andaki CPU yazmaçlarının değerlerini parametresiyle belirlenen bir yapıya yerleştirir.
2) longjmp yapıldığında bu fonksiyon CPU yazmaçlarının değerlerini yükleyerek setjmp yapılan noktaya geçilmesini sağlar.

Bu fonksiyonların prototipleri <setjmp.h> dosyası içerisindedir.

```
int setjmp(jmp_buf pos);
void longjmp(jmp_buf pos, int val);
```

longjmp yapıldığında akış setjmp fonksiyonunun içerisine döner. Eğer setjmp longjmp ile geri dönüyorsa longjmp'nin ikinci parametresiyle, longjmp nedeniyle geri dönmüyorsa 0 ile geri döner. Böylece programcı setjmp noktasında fonksiyonun geri dönüş değerine bakarak akışın ilk geçiş nedeniyle mi yoksa longjmp nedeniyle mi setjmp'den döndüğünü tespit edebilir.

```
jmp_buf g_pos;

if (setjmp(g_pos) == 0) {
	/* setjmp ilk geçiş nedeniyle geri dönüyor */
	...
}
else {
```

```
      /* setjmp longjmp nedeniyle geri dönüyor */
      ...
}

int setjmp(jmp_buf pos)
{
1)    CPU yazmaçlarını pos'a yaz.
2)    pos.retval = 0 yap
RETURN:
3)    return pos.retval = 0 yap
}

void longjmp(jmp_buf pos, int n)
{
      pos.retval = n;
      // ...
}


/* Örnek */

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

int func(void);

jmp_buf g_pos;

int main()
{
      printf("begins...\n");
      if (setjmp(g_pos) != 0) {
            printf("After jump...\n");
            exit(1);
      }
      func();
      printf("normal finish...\n");
}

int func(void)
{
      printf("Func...,\n");
      if (getchar() == 'j')
            longjmp(g_pos, 1);
}

/*
 * typedef struct _jmp_buf jmp_buf[1];
 */
```

signal oluştuğunda çağrılan fonksiyon içerisinde longjmp yapmak

Bilindiği gibi bir process çalışırken process'ler arası geçiş oluştuğunda işletim sistemi o anda çalışmakta olan process'in yazmaç bilgilerini process handle alanında bir bölgeye yazar. Böylelikle process yeniden kalınan yerden çalışmaya devam edebilir. Bir signal oluştuğunda işletim sistemi kod adresini (yani CS:EIP) saklayarak signal fonksiyonuna aynı yazmaç

takımıyla dallanır. signal fonksiyonu normal sonlandırılırsa akış işletim sistemine döner. İşletim sistemi bu noktada saklamış olduğu orijinal kod adresini (CS:EIP) process handle alanına geri yazar. Eğer signal fonksiyonu geri dönmezse örneğin longjmp yapılmışsa işletim sisteminin eski kod adresini yükleme gibi bir olanağı kalmaz. Böylece process sanki normal çalışmasındaymış gibi o noktadan çalışmaya devam eder. Sonuç olarak signal fonksiyonu içersinde longjmp yapmak bir probleme yol açmaz.

sleep fonksiyonu daha güvenli aşağıdaki gibi yazılabilir.

```
jmp_buf pos;

void sig_alrm(int signo)
{
      longjmp(pos, 1);
}

unsigned mysleep(unsigned seconds)
{
      if (signal(SIGALRM, sig_alrm) == SIG_ERR)
            return seconds;
      if (setjmp(pos) == 0) {
            alarm(seconds);
            pause();
      }
      return alarm(0);
}
```

POSIX signal fonksiyonları

Önceden de bahsedildiği gibi signal fonksiyonu sistemden sisteme değişebilen bir davranışa sahiptir. Yani signal fonksiyonu POSIX standartlarına göre deprecated durumdadır. Signal fonksiyonundaki davranış farklılığı ve bozuklukları şunlardır:

1) signal fonksiyonuna dallanıldığında signal durumunun default'a çekilip çekilmeyeceği sistemden sisteme değişmektedir. Örneğin AT&T sisteminde default'a çekilirken BSD sistemlerinde çekilmemektedir.

2) Yavaş sistem fonksiyonlarının signal fonksiyonuyla kesilmesi durumunda fonksiyonun yeniden çağrılmasının işletim sistemi tarafından mı yoksa programcı tarafından mı yapılacağı sistemden sisteme değişmektedir. AT&T Unix fonksiyonu programcının çağırmasını isterken BSD bunu otomatik yapmaktadır.

3) Signal fonksiyonu çalıştırılırken aynı signal'den bir daha oluştuğu durumdaki davranış sistemden sisteme değişebilmektedir. AT&T Unix'lerde iç içeliğe izin verilmiştir, BSD sistemlerinde izin verilmemiştir.

POSIX standartları bu uyumsuzluğu kaldırmak için yeni çok geniş olanaklara sahip olan signal fonksiyonları tanımlamıştır. POSIX'teki yenilikler ve genişletmeler şunlar olmuştur:

1) POSIX signal sistemini anlamak için üç önemli terimi açıklamak gerekir.
a) *signal'in iletilmesi (deliverance):* İşletim sisteminin signal fonksiyonunu çağırmasına signal'in iletilmesi (deliver) denir.

b) *signal'in askıda olması (pending):* signal'in işletim sistemine bildirilmesiyle signal fonksiyonunun çağrılması arasındaki zamana signal'in askıda olması (pending) denir.
c) *Signal'in bloke olması:* signal işletim sistemine bildirildiği halde programcının isteği doğrultusunda askıda bırakılarak signal fonksiyonunun çağrılmaması durumuna signal'in bloke olması denir.
POSIX signal fonksiyonları istenilen signal'ların bloke olmasını sağlayabilmektedir. Yani işletim sistemine "falanca signal gelirse bunu şimdilik işleme sokma, ben blokeyi çözünce blokeyi çöz" denilmektedir.

2) Bir signal oluştuğunda istenilen signal'lar bloke edilebilmektedir. Bloke çözüldüğünde o signal'lar iletilmektedir. Signal'ların biriktirilmesi söz konusu değildir.

3) İstenilen signal'lar herhangi bir zaman bloke edilebilir.

4) Signal oluştuğunda oluşan signal her zaman otomatik olarak bloke edilir. Yani iç içe signal oluşmaz. Ancak signal'ın tekrar default duruma çekilip çekilmeyeceği ayrıca belirlenebilmektedir.

Bloke Edilmiş Signal'ların İfade Edilmesi

POSIX signal fonksiyonları hangi signal'ların bloke edileceğini belirlemek için sigset_t isimli bir tür kullanırlar. Eğer POSIX sisteminde toplam 32 tane signal var ise sigset_t şöyle typedef edilmiş olabilir:

```
typedef unsigned long sigset_t;
```

Bu durumda her bit ilgili signal'ın bloke edilip edilmeyeceğini belirtebilir. Signal sayısı POSIX standartlarında tam olarak belirlenmemiştir, bu nedenle bloke edilmiş signal'ların nasıl belirleneceği de signal sistemini tasarlayanlara bırakılmıştır. sigset_t türü bir dizi ya da yapı olabilir. sigset_t madem ki soyut bir türdür, o halde hangi signal'ların bloke edileceği yardımcı fonksiyonlarla belirlenmelidir. POSIX sistemlerinde bunun için beş fonksiyon bulunmaktadır:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

sigemptyset fonksiyonu tüm signal'ları bloke edilmemiş biçimde belirler.
sigfillset fonksiyonu bütün signal'ların bloke olduğu belirlemesini yapar.
sigaddset fonksiyonu belirli bir signal'ı bloke kümesine katar.
sigdelset fonksiyonu belirli bir signal'ı bloke kümesinden çıkartır.
sigismember fonksiyonu ilgili signal'ın bloke edilip edilmediği bilgisini verir. İlk dört fonksiyon başarılı ise 0 değerine, başarısız ise –1 değerine geri dönerler. sigismember fonksiyonunun geri dönüş değeri eğer signal bloke edilmiş ise 1, edilmemiş ise 0 değeridir.
Şüphesiz bu fonksiyonlar aslında bir bloke belirlemesi yapmazlar, yalnızca sigset_t türünden bir nesnenin içerisinde set işlemleri yaparlar.

sigprocmask Fonksiyonu

Bu fonksiyon belirlenen signal'ları bloke eder ya da onların blokesini çözer.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Fonksiyonun birinci parametresi SIG_BLOCK, SIG_UNBLOCK ya da SIG_SETMASK olabilir. Fonksiyonun üçüncü parametresi her zaman eski bloke durumunu vermektedir. SIG_SETMASK ikinci parametresi ile belirtilen signal bloke kümesindeki signal'ları bloke eder. Örneğin geçici süre bütün signal'ları bloke edip bunları eski haline getirmek isteyelim:

```
sigset_t nset, oset;
sigfillset(&nset);
sigprocmask(SIG_SETMASK, &nset, &oset);
// ...
sigprocmask(SIG_SETMASK, &oset, NULL);
```

Görüldüğü gibi fonksiyonun ikinci ve üçüncü parametreleri istenirse NULL geçilebilir. NULL geçmek o işlemin yapılmayacağını belirtir. Örneğin ikinci parametre NULL geçilirse ve üçüncü parametre geçilmezse o andaki signal bloke durumu elde edilir.

Birinci parametre SIG_BLOCK olarak girilirse ikinci parametrede belirtilen bloke kümesi o andaki bloke edilmiş signal'lar kümesine katılır. Örneğin o andaki bloke kümesine hiç dokunmadan 17 numaralı signal'ı blokeye eklemek için şu yapılabilir:

```
sigset_t sset;
sigemptyset(&sset);
sigaddset(&sset, 17);
sigprocmask(SIG_BLOCK, &sset, NULL);
```

Nihayet birinci parametre SIG_UNBLOCK ise ikinci parametrede kümeye katılmış olan signal'lar bloke olmaktan çıkartılır. Örneğin 17 numaralı signal'ı bloke olmaktan çıkartmak isteyelim:

```
sigset_t sset;
sigemptyset(&sset);
sigaddset(&sset, 17);
sigprocmask(SIG_UNBLOCK, &sset, NULL);
```

Sınıf Çalışması: Bir program yazınız. Programın başında SIGINT signal'ını bloke ediniz, sonra belirli miktar sleep işlemi uygulayınız. Sonra tekrar bu signal'ı açınız ve durumu inceleyiniz. SIGINT signal'ı için bir signal fonksiyonu yazılmalıdır.

İstenilen signal'ların bloke edilmesi kavramı klasik UNIX System V'lerde yoktu. Signal'ların bloke edilmesi özellikle signal fonksiyonunun kullandığı bazı veri yapılarının seri hale getirilmesinde faydalı olmaktadır.

sigaction Fonksiyonu

Güvenilir olmayan signal fonksiyonlarında signal oluştuğunda çağırılacak fonksiyon signal fonksiyonu ile set edilmektedir. Güvenilir POSIX signal mekanizmasında bu işlem sigaction fonksiyonu ile yapılmaktadır.

```
int sigaction(int signo,
                  const struct sigaction *act,
                  struct sigaction *oact);
```

Fonksiyonun birinci parametresi set edilecek signal numarasıdır. sigaction yapısı şöyledir:

```
struct sigaction {
      void (*sa_handler)(int);
      sigset_t sa_mask;
      int sa_flags;
};
```

Yapının sa_handler elemanı signal oluştuğunda çağırılacak fonksiyonu belirtir. sa_mask elemanı signal oluştuğunda çağırılacak fonksiyonun çalışması sırasında hangi signal'ların bloke edileceğini belirtir. Bir signal oluştuğunda normal olarak signal fonksiyonu süresince o signal'ın kendisi otomatik bloke edilmektedir. Burada buna ek olarak başka hangi signal'ların da bloke edileceği belirtilmektedir. Yapının son elemanı ek birtakım özellikleri belirlemek için kullanılır. Bu eleman 0 olarak geçilirse bu ek belirlemelerden vazgeçilmiş olunur. Örneğin buradaki belirlemelerden biri yavaş sistem fonksiyonları signal ile kesildiğinde bu sistem fonksiyonlarının otomatik olarak yeniden çağırılıp çağırılmayacağı ile ilgilidir (SA_RESTART). Default olarak çağırılmamaktadır. Örneğin SA_RESETHAND tıpkı eski sistemlerde olduğu gibi signal oluştuğunda signal'ın SIG_DFL'ye çekilmesini sağlar. Yapının birinci elemanı şüphesiz SIG_DFL ve SIG_IGN değerlerini alabilir.

sigaction fonksiyonunun üçüncü parametresi önceki sigaction yapısını vermektedir. Bu fonksiyon da başarılı durumda 0 değerine, başarısız durumda –1 değerine geri döner. Örneğin, SIGUSR1 signal'ını set etmek isteyelim, signal fonksiyonu çalışırken SIGINT signal'ını da bloke edelim.

```
struct sigaction sa;
sa.sa_handler = MyHandler;
sigemptyset(&sa.sa_mask);
sigaddset(&sa.sa_mask, SIGINT);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGUSR1, &sa, NULL) == -1) {
      fprintf(stderr, "Cannot set sigaction...\n");
      exit(1);
}
```

sigaction fonksiyonunda bloke edilmesi istenen signal'lar yalnızca signal oluştuğunda çağırılacak fonksiyon çalıştırılırken etkilidir. Signal bittiğinde eski duruma otomatik dönülür. sigprocmask fonksiyonunda belirlenen signal'lar bloke kaldırılmadığı sürece etkili olurlar.

sigsuspend Fonksiyonu

Bazen sigprocmask fonksiyonu ile belirli signal'lar bloke edilip sonra pause işlemi ile bekleme yapılır. Ancak bu iki işlem atomik olarak yapılmadığından bu arada başka bir signal gelerek pause fonksiyonunun sonsuza kadar beklemesine yol açabilir. sigsuspend fonksiyonu sigprocmask fonksiyonundan sonra pause fonksiyonunun çağırıldığı durumlarda bu işlemleri atomik olarak yapar.

sigsuspend fonksiyonu hangi durumlarda gereklidir? sigprocmask ile bir signal'ın blokesinin kaldırıldığını ve pause ile beklendiğini düşünelim. sigprocmask ile pause çağırmaları arasında bu signal oluşursa pause sonsuza kadar bekleyebilir. İşte sigsuspend bu iki işlemi atomik olarak yapmaktadır. Bu durumda,

```
sigprocmask(SIG_UNBLOCK, &sset, NULL);
pause();
```

ile

```
sigsuspend(&sset);
```

atomiklik dışında eşdeğerdir.

```
int sigsuspend(const sigset_t *sset);
```

Fonksiyon parametresiyle belirtilen signal kümesini bloklama bakımından set eder ve atomik bir biçimde pause işlemi yapar. Örneğin;

```
sigset_t oset, nset;
// ...
sigprocmask(SIG_SETMASK, &nset, &oset);
// ...
sigsuspend(&oset);
```

Burada muhtemelen programcı bazı signal'leri bloke etmiş ve gerekli birtakım işlemler yapmıştır. En sonunda da blokeyi çözerek signal oluşana kadar process'i çizelge dışına çıkartmıştır.

Güvenilir signal Fonksiyonlarıyla setjmp ve longjmp İşlemleri

Signal fonksiyonu içerisinde longjmp yapmak bilindiği gibi çok kullanılan bir yöntemdir. Böylelikle programcı sanki process devam ediyormuş gibi işlemlerini sürdürebilir. Ancak güvenilir signal fonksiyonlarında signal fonksiyonu çağrıldığı zaman signal fonksiyonunun çağrılmasına yol açan ve sigaction fonksiyonunda belirtilen signal'lar bloke edilmektedir. Bu durumda signal fonksiyonu içerisinde longjmp yapılırsa bu signal'lar bloke edilmiş bir biçimde kalır. Halbuki asıl istenen setjmp yapıldığı noktadaki signal durumunun aktif hale gelmesidir. setjmp yapıldığı durumdaki signal durumuna geri dönmek için sigsetjmp ve siglongjmp fonksiyonları kullanılır. Bu fonksiyonların kullanımı tamamen setjmp ve longjmp fonksiyonlarında olduğu gibidir. Tek farklılık kullanılan türün jmp_buf değil, sigjmp_buf olmasıdır.

```
void sig_handler(int sno)
{
      ...
      siglongjmp(...);
}
```

abort Fonksiyonu

abort fonksiyonu normal olmayan panik halinde yapılan sonlandırmalar için kullanılır. abort bir standart C fonksiyonudur. Prototipi <stdlib.h> dosyası içerisindedir.

```
void abort(void);
```

abort fonksiyonu SIGABRT isimli signal'i yollamaktadır. İşletim sistemi bu signal'i gönderdikten sonra eğer signal fonksiyonu set edilmemişse default davranış olarak processi sonlandırır. Signal fonksiyonu set edilmişse, fakat signal fonksiyonu içerisinde longjmp ya da exit yapılmamışsa yine işletim sistemi signal fonksiyonunu çağırdıktan sonra process'i sonlandırmaktadır. Bu signal'da processin sonlandırılması ancak signal fonksiyonu içerisinde longjmp yaparak mümkündür. abort fonksiyonunun stdio.h tamponları flush edip etmeyeceği standartlara göre derleyicileri yazanlara bırakılmıştır. Ancak derleyicilerin çoğu abort işleminde stdio.h tamponlarını flush etmemektedir.

Sınıf çalışması: SIGABRT signal'ini set ederek main içerisinde abort fonksiyonunu çağırınız.

```
/* sig.c */

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

void myHandler(int n)
{
      printf("myHandler\n");
      exit();
}

int main()
{
      struct sigaction act = {myHandler, 0, 0};

      if (sigaction(SIGABRT, &act, NULL) == -1) {
            fprintf(stderr, "sigaction error...\n");
            exit(1);
      }
      abort();
      return 0;
}
```

Önemli signal'ler Üzerinde Açıklamalar

1) SIGKILL ve SIGSTOP signal'ları için signal fonksiyonu set edilemez. Bu iki signal da processi sonlandırmaktadır. SIGSTOP, SIGKILL signal'ından farklı olarak core dosyası oluşturmaktadır. Bu signal'lar ignore da edilemezler. Dolayısıyla bir processi sonlandırmanın en etkin yolu SIGKILL ya da SIGSTOP kullanmaktır. (Shell üzerinde kill komutunun default gönderdiği signal SIGKILL değil, SIGTERM signal'ıdır.)

2) Processi sonlandırmak için kullanılan diğer signal'lar SIGTERM ve SIGQUIT signal'larıdır. Bu signal'lara ilişkin signal fonksiyonları set edilebilir. Aralarındaki fark SIGQUIT signal'ının signal fonksiyonu set edilmemişse core dosyası oluşturmasıdır.

3) SIGTSPT signal'ı klavyeden stop tuşu olarak tespit edilen tuşa basıldığında processe gönderilir. Process çizelge dışı bırakılır ve askıda bırakılır. Bu tuş genellikle Ctrl+Z tuşudur. Bu biçimde askıda bırakılan processler SIGCONT signal'ı gönderilerek tekrar çizelgeye alınırlar. Programcı SIGTSPT signal'ını işleyebilir.

4) SIGSEGV signal'i sayfalama hatası yani tahsis edilmemiş bellek bölgesine erişildiğinde çağrılır. Bu signal'a ilişkin signal fonksiyonu yazılabilir. Ancak yazılsa bile signal fonksiyonu içerisinde exit ya da longjmp yapılmadıktan sonra process sonlandırılır. Aslında bu mesajın işlenmesinden sonra processin sonlandırılıp sonlandırılmayacağı sisteme bağlıdır.

Sınıf çalışması: SIGSEGV signal'ini set ederek bir gösterici hatası yapınız.

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

void myHandler(int n)
{
      printf("myHandler\n");
      exit();
}

int main()
{
      int *p = NULL;
      struct sigaction act = {myHandler, 0, 0};
      if (sigaction(SIGSEGV, &act, NULL) == -1) {
            fprintf(stderr, "sigaction error...\n");
            exit(1);
      }
      *p = 10;
      return 0;
}
```

5) Bir alt process sonlandırıldığında üst processe SIGCHLD signal'i gönderilir. Bu signal'ın default durumunda hiçbir şey yapılmaz. Tipik olarak programcı bu signal'ı set ederek wait fonksiyonu ile alt processi zombie olmaktan kurtarır. Şüphesiz bu signal fonksiyonunun içerisinde wait kullanmak yalnızca zombilik durumunu ortadan kaldırır. Yoksa wait fonksiyonunun asıl işlevi alt processin sonlanmasını beklemektir. (Zaten üst process daha önce sonlandığında init processinin alt processi sonlandırması bu şekilde olmaktadır.)

Processler Arası Haberleşmeye Giriş

Modern sistemlerde processlerin bellek alanları birbirinden izole edilmiştir. Bu sistemlerde bir processin başka bir processe bilgi gönderebilmesi için işletim sistemi tarafından sağlanmış özel mekanizmalara ihtiyaç duyulur. Bu mekanizmalara processler arası haberleşme (Interprocess Communication - IPC) denir. IPC yöntemleri genel olarak aynı makinedeki processler arasında ve network altında farklı makinelerdeki processler arasında haberleşme olmak üzere iki gruba ayrılır. Bu iki grup yöntem birbirinden farklıdır.

Pipe Kullanımı

Pipe, Unix sistemlerindeki en ilkel fakat, etkili haberleşme yöntemlerindendir. Pipe yöntemi Win32 sistemlerinde de çok benzer bir biçimde kullanılmaktadır. Pipe yöntemi isimsiz ve isimli olmak üzere ikiye ayrılır. İsimli pipe'lara Unix dünyasında FIFO da denilmektedir.

Pipe aslında kernel içerisinde tahsis edilmiş olan bir FIFO kuyruk sistemidir. Pipe, Unix sistemlerinde dosya sistemi ile bütünleştirilmiştir. Pipe yönteminde temel olarak bir process pipe'a yazma yaparken bir process de okuma yapar. Okuma işlemi FIFO sistemine göre yapılır. Pipe kullanımında senkronizasyon bloke yöntemiyle sağlanabilir. Okuyan process pipe boşsa bloke olur. Pipe'ın bir uzunluğu vardır. Yazan taraf pipe doluysa pipe'ta yer boşalana kadar bloke olur. Pipe'tan okuma yapıldığında yer açılır. İşlem yazan tarafın pipe'ı kapatmasıyla sonlandırılır. Bu işlemi okuyan taraf sanki dosyanın sonuna gelinmiş de 0 byte okunmuş gibi algılar. Pipe'a okuma va yazma yapmak için yine read ve write fonksiyonları kullanılır.

Komut satırında pipe işlemi

Shell üzerinde a ve b iki program olmak üzere,

```
a | b
```

işlemi a programının stdout dosyasını yaratılmış olan bir pipe'a, b programının stdin dosyasını da aynı pipe'a yönlendirir. Böylece a programının stdout dosyasına yazdığı her şey pipe'a yazılır. b programı stdin dosyasından okuma yaptığında pipe'tan okur. Şüphesiz pipe burada bir eşzamanlılık problemini çözmektedir. a programı stdout dosyasını kapattığında ya da sonlandığında b programı sanki pipe'ın sonuna gelmiş gibi 0 byte okur.

```
/* a.c */

#include <stdio.h>

int main(void)
{
      int i;

      for (i = 0; i < 10; ++i)
            printf("%d\n", i);

      return 0;
}
```

```
/* b.c */

#include <stdio.h>

int main(void)
{
      char buf[80];
      int n;

      for (;;) {
            if (scanf("%d", &n) == EOF)
                  break;
            printf("%d\n", n);
      }
      return 0;
}
```

Anahtar notlar: scanf fonksiyonu önce boşluk karakterlerini atar, sonra format karakteriyle belirtilen uygun giriş varsa alır, parametrelere yerleştirdiği giriş sayısıyla geri döner. scanf

parametresi ile belirtilen adrese bir şey yerleştirememiş olduğu durumda dosya sonunu görürse EOF değeri ile geri döner. scanf fonksiyonu eğer format karakterine uygun bir giriş bulamazsa (örneğin "%d" ile sayı okunmak istensin fakat biz "ali" gibi bir yazı girmiş olalım) beğenmediği karakteri tampona geri yazarak (ungetch gibi bir fonksiyonla) okuyabildiği parça sayısıyla geri döner. gets ve fgets fonksiyonları '\n' görene kadar okuma yaparlar. Eğer hiçbir karakter okuyamazlarsa NULL göstericiyle geri dönerler.

Bir dosya üzerinde işlem yapan Unix programlarının çoğu komut satırı argümanı olarak dosya ismi girilmediğinde stdin dosyasından okuma yaparlar. Böylelikle kullanıcı bu programlarla shell üzerinde pipe işlemi uygulayabilir. Örneğin;

```
ls | wc
```

Burada wc komut satırı argümanı almadığına göre stdin'den okuma yapacaktır. ls programı çıkışı stdout dosyasına yazdığına göre pipe işlemi gerçekleşecektir.

Benzer biçimde örneğin more programı da komut satırı argümanı olarak dosya ismi almazsa stdin'den okuma yapar.

popen ve pclose Fonksiyonları

Bu fonksiyonlar yüksek seviyeli pipe işlemi yapan fonksiyonlardır. Kullanımları çok basittir.

```
FILE *popen(const char *cmd, const char *type);
int pclose(FILE *f);
```

Bu fonksiyonlar pek çok sistem tarafından desteklenmelerine karşın bir POSIX fonksiyonu değildirler, ancak POSIX.2 içerisinde bu fonksiyonlara değinilmiştir. Fonksiyonların prototipleri <stdio.h> içerisindedir. popen fonksiyonunun ikinci parametresi "w" ya da "r" olabilir. Bir pipe yaratıldığında iki dosya betimleyicisi oluşur. Bu dosya betimleyicilerinden biri dosyaya yazmakta, diğeri okumakta kullanılır.

popen fonksiyonu şöyle çalışmaktadır: Kullanılan modun "r" olduğunu düşünelim. Fonksiyon önce pipe'ı yaratır ve iki betimleyiciyi oluşturur. Okumakta kullanılan betimleyiciyi FILE yapısıyla ilişkilendirerek geri dönüş değeri olarak verir. Bundan sonra popen shell programını çalıştırmak için bir kez fork yapar, fork işleminden sonra pipe'a yazma yapmakta kullanılan betimleyiciyi dup2 fonksiyonuyla stdout yerine yerleştirir. Bundan sonra exec ile birinci parametresiyle belirtilen programı çalıştırır. Sonuç olarak şöyle bir durum oluşur: popen fonksiyonunun geri döndürdüğü dosyadan okuma yapıldığında aslında pipe'tan okuma yapılmış olur. Fonksiyonunun birinci parametresinde belirtilen komutun çıktıları (yani buradaki programın stdout'u) da pipe'a yönlendirilir. "w" modunda tam tersi bir işlem oluşur. Yani programcı yazdığında pipe'a yazar, yazılanlar pipe aracılığıyla birinci parametresiyle belirtilen programın stdin'ine yönlendirilir. pclose işleminde pipe kapatılır.

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

int main(void)
{
      int ch;
      FILE *f;
```

```
      if ((f = popen("ls", "r")) == NULL) {
            fprintf(stderr, "Cannot create pipe...\n");
            exit(1);
      }

      while ((ch = fgetc(f)) != EOF)
            putchar(ch);
      pclose(f);
      return 0;
}
```

Örneğin program içerisinde elde edilen bir takım bilgilerin more programı yardımıyla ekrana yazdıracak olalım. Bunun için bilgileri önce dosyaya yazıp daha sonra more programını çalıştırmak yerine değerleri buldukça pipe yoluyla more programına aktarmak daha pratik bir yöntemdir.

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

int main(void)
{
      int i, ch;
      FILE *f;
      if ((f = popen("more", "w")) == NULL) {
            fprintf(stderr, "Cannot create pipe...\n");
            exit(1);
      }
      for (i = 0; i < 100; ++i)
            fprintf(f, "%d\n", i);
      pclose(f);
      return 0;
}
```

Shell'deki pipe işlemini yapan benzer bir program ekte verilmiştir. Bu programda popen fonksiyonuyla iki ayrı pipe yaratılmıştır. Pipe'lardan biri "w" diğeri "r" modunda açılmıştır. "r" modunda açılan pipe okunduğunda aslında birinci programın stdout'a gönderdikleri elde edilir. Okunanlar "w" modunda açılmış pipe'a yazılırlar. İkinci program stdin dosyasını okuduğunda bu pipe'ı okuyacaktır.

a **à** stdout **à** pipe1 ("r" modu) **à** fin **à** buf **à** fout **à** pipe2 ("w" modu) **à** stdin **à** b

Aşağı Seviyeli pipe İşlemleri

popen, pclose fonksiyonları kendi içlerinde aşağı seviyeli pipe fonksiyonlarını kullanırlar. Aşağı seviyeli pipe açmak için `pipe` fonksiyonu kullanılır. Bu fonksiyonun prototipi unistd.h dosyası içerisindedir.

```
int pipe(int filedes[2]);
```

Anahtar notlar: C'de prototipte ya da tanımlama sırasında parametre parantezi içerisinde yazılan aşağıdaki ifadeler eş değerdir.

```
1) void Func(int a[size]);
      void Func(int *a);
```

Burada [] içerisi boş bırakılabilir.

```
2) void Func(int a[RowSize][ColSize]);
      void Func(int (*a) [ColSize]);

3) void Func(int f(void));
      void Func(int (*f)(void));

4) void Func(int *a[10]);
      void Func(int **a);

5) void Func(int a[][])
      Geçersiz
```

pipe fonksiyonu iki elemanlı bir dizinin başlangıç adresini alır ve bir pipe oluşturarak okuma ve yazma için gerekli olan dosya betimleyicilerini bu diziye yerleştirir. Fonksiyon başarılıysa 0, başarısızsa –1 değerine geri döner. UNIX sistemlerinde pipe'lar tıpkı dosyalar gibi işlem görür. pipe fonksiyonundan iki dosya betimleyicisi elde edilir. Biri pipe'ı okumak için diğeri yazmak için kullanılır. *Dizinin ilk elemanına yerleştirilen betimleyici okumak için ikinci elemanına yerleştirilen betimleyici yazmak için kullanılır.* Pipe'lar POSIX standartlarında tek yönlüdür. Yani aslında modern sistemlerin çoğunda her iki betimleyiciyle de okuma ve yazma yapılabilmektedir. Bir pipe'a birden fazla process yazma yapabilir ve birden fazla process pipe'tan okuma yapabilir. Pipe açıldıktan sonra klasik olarak read ve write fonksiyonlarıyla okuma ve yazma yapılabilir.

Normal olarak blokeli modda pipe'ı okuyan process'ler aşağıdaki gibi bir döngü içerisinde okuma işlemlerini yaparlar.

```
while (read(...) > 0) {
      // ...
}
```

Burada eğer pipe'a yazma yapan hiçbir process kalmamışsa read fonksiyonu 0 ile geri döner ve döngüden çıkılır. Örneğin, pipe'ta halen 5 byte bulunuyor olsun. pipe'a yazma yapan process pipe'ı kapatmış olsun. Bir process'in bu pipe'dan 10 byte okumak istediğini düşünelim. Okuyan process önce 5 byte'ı okur ve read 5 değeri ile geri döner, bir sonraki çağırmada okuyan process bilgi okuyamaz ve 0 ile geri döner. Görüldüğü gibi pipe'a yazan process kalmaması durumu adeta EOF etkisi yaratmaktadır.

En normal durum önce pipe'a yazma yapan process'in pipe'ı kapatması durumudur. Ancak bunun tersi olursa, yani önce okuma yapan process pipe'ı kapatırsa write işlemi sırasında SIGPIPE signal'i oluşur. Bu signal'in default davranışı process'in sonlanması biçimindedir.

pipe fonksiyonu ile yaratılan pipe'lara isimsiz pipe'lar denir. İsimsiz pipe'lar şöyle bir senaryo ile kullanılabilirler: Program önce pipe fonksiyonu ile pipe'ı yaratır. Sonra fork yapar. Böylece haberleşme üst ve alt process'ler arasında yapılır. Klasik olarak okuma yapacak process yazma yapılan betimleyiciyi kapatır, yazma yapacak process ise okuma için gerekli olan betimleyiciyi kapatır.

```
/* pipe2.c */
```

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

int main(int argc, char *argv[])
{
      int pipedes[2];
      pid_t pid;
      char s[100];

      if (pipe(pipedes) < 0) {
            fprintf(stderr, "Cannot create pipe\n");
            exit(1);
      }

      if ((pid = fork()) == -1) {
            fprintf(stderr, "Cannot create pipe\n");
            exit(1);
      }

      if (pid > 0) {
            close(pipedes[0]);
            write(pipedes[1], "Merhaba pipe", 13);
            close(pipedes[1]);
            wait(NULL);
            exit(0);
      }
      else {
            close(pipedes[1]);
            read(pipedes[0], s, 13);
            puts(s);
            close(pipedes[0]);
            exit(0);
      }
      return 0;
}
```

Sınıf çalışması: Bir pipe açınız, sonra fork işlemi yapınız. fork işleminden sonra aşağıdaki gibi alt process'in stdin'ini pipe'a yönlendiriniz.

```
dup2(filedes[0], STDIN_FILENO);
```

Bundan sonra alt process'te exec uygulayarak wc programını çalıştırınız. Üst process'te pipe'a bir takım yazılar yazınız. Bundan sonra üst process pipe'ı kapatarak wait fonksiyonu ile alt process'i beklemelidir.

/*pipe3.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
      int pipedes[2];
      pid_t pid;
      char s[100];

      if (pipe(pipedes) < 0) {
            fprintf(stderr, "Cannot create pipe\n");
```

```
            exit(1);
      }
      if ((pid = fork()) == -1) {
            fprintf(stderr, "Cannot create pipe\n");
            exit(1);
      }

      if (pid > 0) {
            close(pipedes[0]);
            write(pipedes[1], "Selamlar", 10);
            close(pipedes[1]);
            wait(NULL);
            exit(0);
      }
      else {
            close(pipedes[1]);
            dup2(pipedes[0], STDIN_FILENO);
            execlp("wc", "wc", (char *) 0);
      }
      return 0;
}
```

/* pipe4.c */

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

int main(int argc, char *argv[])
{
      FILE *f;
      int i;

      if ((f = popen("wc", "w")) == NULL) {
            fprintf(stderr, "Cannot open pipe!...\n");
            exit(1);
      }

      for (i = 0; i < 100; ++i)
            fprintf(f, "sayi = %d\n", i);
      pclose(f);

      return 0;
}
```

İsimli pipe Kullanımı

İsimsiz pipe'lar yalnızca üst ve alt process'ler arasındaki haberleşmede kullanılabilirler. İsimsiz pipe'larda iki process'in aynı pipe'ı görebilmesi fork işlemi ile birlikte dosya tablosunun kopyalanması işlemi ile sağlanmaktadır. Oysa isimli pipe'larda üst alt ilişkisi altında bulunmayan herhangi iki process bir isim altında anlaşarak aynı pipe üzerinde anlaşabilmektedir. İsimli pipe'lara FIFO dosyaları da denilmektedir. İsimli pipe'lar dizin girişinde bir dosya gibi görüntülenirler. İsimli pipe'lar shell üzerinden `mkfifo` komutuyla da bir dosya gibi yaratılabilirler.

İsimli pipe'lar üzerinde çalışabilmek için önce pipe'ı yaratmak gerekir. Pipe yaratıldıktan sonra aslında geri kalan işlem tamamen dosya işlemleri gibi yapılır. Yani pipe open

fonksiyonuyla açılır, read ve write fonksiyonlarıyla okuma yazma yapılır ve close fonksiyonuyla kapatılır. Aslında pipe yaratma işlemi komut satırından da mkfifo shell komutuyla yapılabilir.

Not: Ctrl+Z tuşuna basıldığında signal o terminaldeki tüm process'lere gider.

mkfifo Fonksiyonu

Bu fonksiyon isimli pipe'ı yaratmak için kullanılır. mkfifo POSIX standartlarında olan bir fonksiyondur. Aslında sistemlerin çoğunda `mknod` isimli daha yetenekli bir fonksiyon vardır. mknod fonksiyonu yalnızca isimli pipe değil, başka özel dosyaları yaratmakta da kullanılır. mknod bir POSIX fonksiyonu değildir. Sistemlerin çoğunda mkfifo mknod fonksiyonunu çağıracak biçimde yazılmıştır. Özetle mkfifo fonksiyonu mknod fonksiyonunun özel bir kullanımıdır.

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

Fonksiyonun birinci parametresi isimli pipe'ın ismi, ikinci parametresi yaratılacak dosyanın erişim bilgileridir. Fonksiyon başarı durumunda 0 değerine, başarısızlık durumunda –1 değerine döner. Fonksiyonun prototipi <sys/stat.h> dosyası içerisindedir.

Anahtar notlar: printf ile sprintf arasındaki ilişki scanf ile sscanf fonksiyonu arasındaki ilişki ile benzerdir. sprintf ekran yerine bir diziye bilgileri yazar. sscanf ise klavye yerine bir diziden bilgileri alır.

```
/* mymkfifo.c */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
      mode_t mode;

      if (argc != 3) {
            fprintf(stderr, "Wrong number of arguments !...\n");
            exit(1);
      }
      sscanf(argv[2], "%o", &mode);
      if (mkfifo(argv[1], mode) == -1) {
            perror("mkfifo");
            exit(1);
      }
      return 0;
}
```

Şüphesiz mkfifo fonksiyonu processin umask değeriyle yine işleme girmektedir. mkfifo ile yalnızca isimli pipe yaratılır. İsimli pipe'larda pipe'ı yaratarak açan bir mekanizma yoktur. Pipe önce mkfifo ile yaratılır. Sonra open fonksiyonu ile açılır. Pipe close işlemi ile otomatik olarak silinmez. Silmek için sanki bir dosyaymış gibi `unlink` fonksiyonunu kullanmak gerekir. İsimli pipe bir dosya gibi dizin girişinde gözükür. Fakat pipe'a yazma yapıldığında pipe'ın uzunluğu dizin girişinde değişmez hep 0 olarak gözükür.

İsimli Pipe'larda Okuma Yazma İşlemleri

İsimli pipe'larda okuma yazma tamamen isimsiz pipe'larda olduğu gibidir. İsimli pipe'tan okuma sırasında sistem pipe'a yazma potansiyelinde olan process'lerin sayısını izler. Eğer pipe'a yazmak için pipe'ı açmış olan hiçbir process kalmamışsa read fonksiyonu 0 byte okuyarak işlemini sonlandırır. İsimsiz pipe'larda olduğu gibi pipe'ı okuma potansiyelinde hiçbir process kalmamışsa yazma sırasında SIG_PIPE signal'i oluşur (broken pipe), bu signal da default olarak processi sonlandırır. İsimli pipe'ların da bir maximum uzunluğu vardır. Bu uzunluk PIPE_BUF sembolik sabitiyle belirlenen uzunluktur. Pipe doluysa yazma yapan process pipe'ta yer açılana kadar bloke edilir.

İsimli ve İsimsiz pipe'lar Arasındaki Fark

İsimsiz pipe'larda pipe'tan okuma ve yazma yapmak için kullanılan betimleyiciler başka bir processe ancak fork ve/veya exec yoluyla aktarılırlar. Oysa isimli pipe'larda bir isim üzerinde anlaşıldığı için haberleşme işlemi tamamen farklı process'ler arasında pipe yoluyla yapılabilmektedir.

Sınıf çalışması: İsimli bir pipe'ı shell üzerinden yaratınız. İki program yazınız. Birinci program 1'den 1000'e kadar sayıları bir döngü içerisinde pipe'a yazsın. İkinci program aynı pipe'tan okuma yaparak bu sayıları elde etsin. Birinci program pipe'ı O_WRONLY, ikinci program ise O_RDONLY modunda açacaktır.

Açıklamalar: Yazma ve okuma işlemleri işlemi aşağıdaki gibi yapılabilir.

```
for (int i = 0; i < 1000; ++i)
      write(fd, &i, sizeof(int));

while (read(fd, &i, sizeof(int)) > 0)
      printf("%d\n", i);
```

```
/* first.c */

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

int main()
{
      pid_t pid;
      int i;

      pid = open("npipe", O_WRONLY);

      for (i = 0; i < 1000; ++i)
            write(pid, &i, sizeof(int));

      return 0;
}
```

```
/* second.c */
```

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

int main()
{
      pid_t pid;
      int i;

      pid = open("npipe", O_RDONLY);

      while (read(pid, &i, sizeof(int)) > 0)
      printf("%d\n", i);

      return 0;
}
```

Pipe open fonksiyonuyla açılırken işlemle ilgili en az bir process olup olmadığına bakılır. Örneğin pipe read modunda açılacak olsun. Eğer pipe'ı write modunda açmış bir process yoksa bloke open fonksiyonu içerisinde gerçekleşir ve bir process pipe'ı write modunda açtığı zaman read modunda açmaya çalışan processin de blokesi çözülür. Benzer biçimde pipe'ı write modunda açmaya çalışan bir process olsun, ama aynı pipe'ı read modunda açmış olan hiçbir process olmasın. Bu durumda write modunda açmaya çalışan process read modunda pipe açılmaya çalışıldığı sürece bloke olacaktır.

Pipe ile Client-Server Uygulamalar

Client-server program mimarisinde gerçek işlemleri yapan programa server denir. Server programı genellikle bir tanedir. Client programlar process'ler arası haberleşme yöntemlerinden birisiyle server programa bilgi göndererek server'ın kendileri için bir şeyler yapmasını isterler. Server program client programdan gelen işlem isteklerini yerine getirir ve sonucu yine process'ler arası haberleşme yöntemiyle client programa iletir. Genellikle bir server programa birden fazla client program bağlanabilir. Client programların birbirleriyle genellikle bir ilişkisi olmaz. Bunlar server program yoluyla birbirleriyle ilişki kurarlar. Basit bir chat programında tipik olarak client sisteme girdiği zaman server'a mesaj göndererek sisteme girdiğini belirtir. Herhangi bir client mesaj yazdığında bu mesaj server'a iletilir. Server da bu mesajı tüm client'lara dağıtır. ICQ gibi çok fazla kişi tarafından kullanılan mesajlaşma programlarında client'lar birbirleri arasında yine client-server olarak haberleşmektedir. Client ve server arasındaki haberleşmelerde genellikle bir yapı kullanılır. Kullanılan yapının değişken uzunlukta olması daha etkin bir kullanımdır. Haberleşme için kullanılan tipik bir yapı şöyle olabilir.

```
typedef struct _MSG {
      WORD wType;
      WORD wSize;
      char buf[1];      // C++'ta char buf[] şeklinde olabilir.
                        // Bu durumda yapının uzunluğuna
                        // etki etmez.
} MSG;
```

Burada wType elemanı yapılacak işlemin türünü belirtir. Her mesajın ne amaçlı olarak gönderildiğini belirten bir tür bilgisi ve bir data bilgisi vardır. wSize data bilgisinin uzunluğunu tutar. Mesajı okuyan taraf önce

```
2 * sizeof(WORD)
```

kadar bilgiyi okuduktan sonra size kısmının uzunluğuna göre mesajın datalarını okur.

İsimli pipe'larla client-server uygulamalarda, client'lar server'a tek bir pipe üzerinden mesajlarını gönderebilirler. (Tabi gönderilecek mesajın uzunluğunun PIPE_BUF değerini aşmaması gerekir). Ancak server client'larla tek bir pipe üzerinden haberleşemez. Çünkü bir client'ın yalnızca kendisine gönderilen bilgiyi alması gerekir. Bu durumda bir client server'a bağlandığında server'ın o client için yeni bir pipe yaratması gerekir. Mesaj kuyruğu (Message Queue) denilen process'ler arası haberleşme yönteminin bloke mekanizması tamamen pipe'a benzemektedir. Ancak bu sistemde her mesajın bir ID değeri vardır ve process yalnızca belli bir ID'ye sahip mesajları alabilir.

Sınıf çalışması: Aşağıda belirtilen basit client server uygulamayı isimli pipe kullanarak yazınız.

Açıklamalar:

1) Mesajın genel formatı şöyledir:

```
typedef struct _MSG {
      int type;
      int length;
      char buf[1];
}MSG;
```

`type` değişkeni `enum { MSG1, MSG2, MSG3, MSG4, MSG5 };` sabitlerinden biri olabilir.

2) Server önce mesajın türü ve uzunluğunu okur. Uzunluk buf kısmının uzunluğudur. Server mesajı pipe'dan alarak aşağıdaki biçimde display eder:

`Message No N Received:` "`buf` içerisindeki yazı"

3) Client program bir döngü içerisinde programcıdan bir yazı ve mesaj numarası isteyecek. Yazıyı buf kısmına yerleştirecek ve server'a pipe aracılığıyla bu yazıyı gönderecektir.

4) Sınıf çalışması kolay yapılsın diye server client'a mesaj göndermeyecektir. Client klavyeden özel bir yazı girildiğinde sonlandırılacaktır.

5) Uygulamada pipe manuel açılıp manuel silinebilir. pipe ismi client ve server programları tarafından ortak bir biçimde belirlenmiş olmalıdır.

```
/* pipeserver.c – server */

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

```
#include <fcntl.h>

typedef struct _MSG {
      int type;
      int length;
      char buf[1];
} MSG;

int main(void)
{
      char temp[1024];
      int cs, n;
      int length;
      char *pMSG;

      while (1) {
            cs = open("cs.pip", O_RDONLY);
            if (read(cs, temp, sizeof(int) * 2) == 0)
                  exit(1);
            n = *((int *) temp);
            length = *(((int *) temp) + 1);
            read(cs, temp + sizeof(int) * 2, length);
            temp[sizeof(int) * 2 + length] = '\0';
            printf("MSG no: %d\n", n);
            printf("MSG: %s\n", temp + sizeof(int) * 2);
      }
      return 0;
}
```

/* pipeclient.c – client */

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

typedef struct _MSG {
      int type;
      int length;
      char buf[1];
} MSG;

int main(int argc, char *argv[])
{
      char temp[1024];
      int cs, n;
      char *pMSG;

      while (1) {
            printf("MSG no:");
            scanf("%d", &n);
            getchar();
            if (n == -1)
                  exit(1);
            printf("MSG: ");
            gets(temp);
            pMSG = (char *) malloc(sizeof(MSG) + strlen(temp) + 1);
            *((int *) pMSG) = n;
            *(((int *) pMSG) + 1) = sizeof(temp);
            strcpy(pMSG + sizeof(int) * 2, temp);
            cs = open("cs.pip", O_WRONLY);
```

```
            if (cs < 0) {
                  fprintf(stderr, "cannot open pipe...\n");
                  exit(1);
            }
            write(cs, pMSG, sizeof(MSG) + strlen(temp));
      }
      return 0;
}
```

UNIX/Linux Sistemlerinde Process'ler Arası Haberleşme (IPC) Mekanizması

Her ne kadar pipe yöntemi process'ler arası bir haberleşme yöntemiyse de UNIX dünyasında process'ler arası haberleşme başlığı altında şu 3 konu ele alınmaktadır:

1) Mesaj kuyrukları
2) Semaforlar
3) Paylaşılmış bellek alanları

Şüphesiz pipe'lar ve socket'ler de aslında önemli process'ler arası haberleşme yöntemlerindendir. Yukarıdaki üç yöntemin kullanım mekanizmaları birbirlerine benzer olduğu için IPC mekanizmaları dendiğinde bu yöntemler anlaşılmaktadır.

IPC Mekanizmalarındaki Ortak Bazı Özellikler

Yukarıdaki 3 IPC mekanizması bir dosya biçiminde kullanılmamaktadır. Bu nedenle IPC nesnelerinin birer ismi yoktur. Win32 sistemlerinde process'ler arası haberleşme yöntemlerindeki nesneler bir isim ile ifade edilirler. Win32'de tipik olarak bir process bir isim altında IPC nesnesini oluşturur, diğeri de aynı ismi vererek onu kullanır. Verilen ismin tesadüfen kullanılmakta olan bir IPC nesne ismi ile çakışması durumu üzerinde durulmayacak biçimde zayıf bir olasılıktır. Halbuki UNIX sistemlerinde iki process'in aynı IPC nesnelerini görmeleri isim yoluyla değil, anahtar denilen bir sayı yoluyla yapılmaktadır. UNIX sistemlerinde tipik olarak iki process aynı anahtar sayı üzerinde anlaşırlar, böylece iki process de IPC nesnesini yaratır. Bu nesne aynı nesne olur. Şüphesiz UNIX sistemlerinde de anahtar sayı bakımından bir çakışma söz konusu olabilir. Bu tür bir çakışma önemsenmemelidir.

Üç IPC mekanizması için IPC nesnesi elde etmeye yönelik üç get fonksiyonu vardır. Fonksiyonlarda *`msg` ön eki mesaj kuyruğu için, `sem` ön eki semafor için ve `shm` ön eki paylaşılan bellek alanları için kullanılmaktadır.* Örneğin IPC nesnesi elde eden fonksiyonlar `msgget`, `semget` ve `shmget` biçimindedir. get fonksiyonlarının genel yapısı şöyledir:

```
int xxxget(key_t key, ......);
```

Üç get fonksiyonunun da birinci parametreleri key_t türündendir. key_t türü tipik olarak int ya da long biçimindedir. Bu fonksiyonların geri dönüş değerleri yaratılan ya da açılan IPC nesnesine ilişkin handle değeridir (UNIX dünyasında handle yerine ID değeri denilmesi daha yaygındır). Örneğin iki process'ler ortak bir mesaj kuyruğu yoluyla haberleşecek olsunlar. Ortak bir anahtar değerinde anlaşırlar. İki process de msgget fonksiyonunu bu anahtar değer ile çağırır. Böylece iki process de aynı nesneye ilişkin bir ID değeri elde ederler. Diğer fonksiyonlar da bu ID değerini kullanarak haberleşmeyi sağlarlar. İki process'in ID değerleri farklı olabilir, ama aynı anahtar 'x' sayıdan hareketle nesneyi yarattıklarından aynı nesneyi görürler. UNIX sistemlerinde de tıpkı Win32 sistemlerinde olduğu gibi nesneler türlerine göre

birbirlerinden ayrılmaktadır. Yani, İki farklı türden IPC nesnesi aynı anahtarı kullanabilir. Örneğin process'lerden biri 18362 anahtar değeri ile bir mesaj kuyruğu yaratıyor olsun, başka bir process aynı anahtar değeri ile bir semafor yaratabilir. Bunlar farklı nesneler olduğu için anahtar değerler karışmaz.

UNIX/Linux IPC nesneleri process'lerden bağımsız olarak yaşarlar. İsimli pipe'larda olduğu gibi bir IPC nesnesini kullanan hiç bir process kalmazsa IPC nesnesi de otomatik olarak silinmez. IPC nesnelerinin yaratılıp silinmesi ayrıca yapılması gereken bir işlemdir. IPC nesneleri de tamamen dosyalar gibi OWNER-GROUP-OTHER erişim haklarına sahiptir. IPC nesneleri bu bakımdan dosyalara da benzemektedir.

Bir IPC nesnesi yaratıldıktan sonra ID değeri belirtilerek kullanılır. get fonksiyonlarının bir parametreleri daha ortaktır. Bu ortak parametre flag ismi ile belirtilir. flag değeri tamamen dosyalardaki open fonksiyonunun açış modu parametresiyle benzer biçimde kullanılır. Bu parametre de üç oktal digit erişim bilgisi ile IPC_CREAT, IPC_EXCL değerleri or'lanır. Örneğin flag değeri şöyle oluşturulabilir:

```
IPC_CREAT | 0666
```

flag değeri ile anahtar değeri arasındaki ilişki şöyledir:

1) Anahtar değer olarak IPC_PRIVATE girilirse flag parametresi default IPC_CREAT anlamına gelir ve bu durumda fonksiyon kullanılmayan bir anahtar değeri ile IPC nesnesi yaratılır.

2) IPC_CREAT ile IPC_EXCL aynı anda kullanılabilir. Bu durum tıpkı dosyalarda olduğu gibi yoksa yarat, ama varsa "açma error ver geridön" anlamına gelir. Programcı yeni bir IPC nesnesi yarattığından böyle emin olabilir.

ftok Fonksiyonu

Bu fonksiyon yeni bir anahtar numarası elde etmek için bazen kullanılmaktadır.

```
key_t ftok(char *pathname, char proj);
```

Eğer herkes ftok fonksiyonunu kullanırsa anahtar değerinin tek olarak elde edileceği garanti edilebilir. Fonksiyonun birinci parametresi bir dosya ismidir, ikinci parametresi ise bir karakterdir. Fonksiyon birinci parametresiyle verilen dosyanın i-node numarasından faydalanarak ikinci parametresiyle belirtilen ascii kodunu bu sayıya ekleyerek bir sayı oluşturur. Şüphesiz sistemde herkes ftok fonksiyonunu kullanırsa çakışma olasılığı neredeyse kalmaz. Ancak ftok fonksiyonunun şöyle bir avantajı vardır; örneğin iki process (client ve server process'leri olsun) aynı bir dosya isminde anlaşıp aynı sayıyı elde edebilirler. Eğer bir çakışma olursa hiç programa dokunmadan o dosyanın silinip yeniden yaratılması gibi bir yöntem ile i-node numarası değiştirilerek başka bir sayıyı deneyebilirler.

Anahtar Değeri Elde Etmek İçin Yöntemler

Sistem genelinde tek bir anahtar değeri elde etmek için (çakışma problemini engelleyerek) üç yöntem kullanılır.

1) Process'lerden biri IPC nesnesini IPC_PRIVATE ile yaratır, bu durumda sistem olmayan bir anahtar numarası kullanacaktır.
2) ftok fonksiyonu kullanılarak anlaşma sağlanabilir.
3) İki process'in de anlaştığı ortak bir değer kullanılabilir.

IPC Nesnelerinin Yok Edilmesi

IPC nesneleri ortak bellek (shared memory) dışında otomatik olarak yok edilmezler. Yani, IPC nesnelerini bir process'in bilinçli olarak yaratması ve silmesi gerekir. Bu durum tıpkı isimli pipe'larda olduğu gibidir. Tipik olarak UNIX/Linux sistemlerinde `IPCS` komutu ile o an da sistem genelinde yaratılmış olan IPC nesneleri hakkında bilgi edilinebilir. Yine istenirse manuel olarak `ipcrm` shell komutu ile istenilen bir ipc nesnesi yok edilebilir. Şüphesiz ipcrm ile IPC nesnesini silmek için erişim hakkının yeterli olması gerekir. IPC nesneleri xxxctl komutu ile silinebilirler.

Ortak Bellek Alanlarının Kullanımı

Farklı process'lerin farklı doğrusal adreslerde aynı fiziksel RAM bloğunu görmelerine ortak bellek alanları (shared memory) denilmektedir. Bu konu Win32 sistemlerinde "bellek tabanlı dosyalar (memory maped file)" ismiyle çok benzer biçimde bulunmaktadır. Sanal bellek kullanabilen bir işlemcinin tipik olarak fiziksel RAM'e erişmesi iki aşamada yapılmaktadır.

1) Program içerisinde kullanılan bütün adresler doğrusal adreslerdir (linear address). Doğrusal adresler process'e özgüdür, her process'in doğrusal adresleri diğerlerinden ayrılmıştır.
2) Doğrusal adresler işlemci tarafından sayfa tablosu (page table) denilen bir tabloya bakılarak fiziksel adrese dönüştürülmektedir.

Sayfa tablosu tipik olarak şöyle bir görünümdedir:

Doğrusal Adres	***Fiziksel Adres***
0	41
1	53
2	56
3	-
4	-
5	93
...	...

Burada –'ler ile gösterilen alanlar commit edilmiş ancak fiziksel RAM'de yer bulamamış sayfalardır. Bir alanın commit edilmiş olması demek, orası için swap dosyasında yer ayrılmış ama fiziksel RAM'de yer ayrılmamış olması demektir. Commit edilmiş sayfalara erişildiğinde sayfa fiziksel RAM'de yoksa işlemci tarafından kesme çağırılır işletim sistemi de erişilen bölgenin commit edilmiş olup olmadığına bakar, onun için fiziksel RAM'de bir sayfa ayarlar. Sonra sayfa tablosunda güncellemeyi yapar ve çalışma kesiksiz olarak devam eder. Eğer

erişilen bölge commit edilmemişse bu durumda Win32 sistemlerinde process sonlandırılır, UNIX sistemlerinde önce SIGSEGV signal'ı gönderilir, sonra process sonlandırırlır.

shmget Fonksiyonu

Bu fonksiyon ortak bellek alanı yaratmakta kullanılır.

```
int shmget(key_t key, int size, int flag);
```

Fonksiyonun birinci parametresi IPC anahtar numarasıdır. İkinci parametre ortak bellek alanının byte cinsinden büyüklüğünü belirtir. Daha önce yaratılmış ortak bellek alanını kullanmak istiyorsak size belirtilmeyebilir, yani 0 geçilebilir. Aslında ortak bellek alanı daha önce yaratılmışsa fonksiyonun bu parametresi dikkate alınmamaktadır. flag parametresi diğer IPC nesnelerinde olduğu gibi IPC_CREAT ve IPC_EXCL kombinasyonu birleşiminden oluşabilir. Ayrıca nesne yaratılırken erişim modu da or'lanarak flag'de belirtilmelidir. Fonksiyon başarılı ise IPC ID değerine başarısızsa –1 değerine geri dönmektedir.

Ortak bellek alanına ilişkin ID elde edildikten sonra artık bunu gerçek anlamda bellekte oluşturmak gerekir. Bu işlem shmat fonksiyonu ile yapılır.

shmat Fonksiyonu

```
void *shmat(int shmid, void *addr, int flag);
```

Fonksiyonun birinci parametresi shmget fonksiyonundan elde edilen ID değeridir. Fonksiyonun ikinci parametresi ortak bellek alanının process'in bellek alnındaki doğrusal başlangıç adresini belirlemekte kullanılır. Bu adres NULL olarak geçilirse doğrusal adreste nerenin tahsis edileceği sisteme bırakılır. Eğer NULL geçilmeyip bir adres değeri geçilirse sistem tahsisatı o doğrusal adresten başlayarak yapmaya çalışır. Ancak bu mümkün olmayabilir, o zaman fonksiyon başarısız olur. Fonksiyonun flag parametresi SHM_RND ya da SHM_RDONLY olabilir. Ya da hiç bir belirleme yapılmayacaksa bu parametre 0 olarak geçilebilir. Fonksiyonun ikinci parametresine bir adres girilmiş ve üçüncü parametresi SHM_RND biçimindeyse bu durumda belirtilen adres sayfa büyüklüğüne aşağıya doğru yuvarlanır. Fonksiyonun geri dönüş değeri başarılı ise tahsis edilen doğrusal adres alanının başlangıç adresi, başarısız ise ((void *) –1) değeridir. (Bu değer aynı zamanda Windows'taki INVALID_HANDLE_VALUE değeridir.)

shmdt Fonksiyonu

Ortak bellek alnına ilişkin IPC nesnesi yaratılıp doğrusal adres alanına bağlandıktan sonra istenirse bu fonksiyonla doğrusal adres alanındaki bağlantı yeniden koparılabilir.

```
int shmdt(void *add);
```

Fonksiyon doğrusal adrese bağlanan ortak bellek alanının başlangıç adresini parametre olarak alır ve bu bağlantıyı koparır. Bağlantılar koparılsa da IPC nesnesi yaşamaya devam eder. Process sonlanırken zaten bu koparılma işlemi sistem tarafından gerçekleştirilmektedir. Ortak bellek alanlarını kullanabilmek için sys/types.h, sys/ipc.h, sys/shm.h dosyaları include edilmelidir.

Örnek uygulama: Bu uygulamada shm1.c programı bir ortak bellek alanı yaratacak ve buraya bir bilgi yerleştirecektir. shm2.c process'i de bu bilgiyi ortak bellek alanından alacaktır.

```
/* shm1.c */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY_VALUE   1234567

int main(void)
{
      char *padr;
      int shmid;

      if ((shmid = shmget(SHM_KEY_VALUE, 1024, IPC_CREAT|0666)) < 0) {
            perror("shmget");
            exit(1);
      }

      if ((padr = (char *) shmat(shmid, NULL, 0)) == (void *) -1) {
            perror("shmat");
            exit(1);
      }

      strcpy(padr, "shared memory");

      for (;;) {
            if (*padr == 'q')
                  break;
            sleep(1);
      }

      shmdt(padr);
      shmctl(shmid, IPC_RMID, NULL);

      return 0;
}
```

```
/* shm2.c */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY_VALUE   1234567

int main(void)
{
      char *padr;
      int shmid;

      if ((shmid = shmget(SHM_KEY_VALUE, 0, 0666)) < 0) {
```

```
            perror("shmget");
            exit(1);
      }

      if ((padr = (char *) shmat(shmid, NULL, 0)) == (void *) -1) {
            perror("shmat");
            exit(1);
      }

      puts(padr);
      *padr = 'q';
      shmdt(padr);

      return 0;
}
```

Ortak Bellek Alanının Silinmesi ve shmctl Fonksiyonu

Daha önceden de ele alındığı gibi ortak bellek alanı shmget fonksiyonuyla yaratılmakta ve shmat fonksiyonuyla gerçek tahsisat yapılmaktadır. Bu IPC nesnesinin yaratılmasıyla belleğin tahsis edilmesi farklı işlemlerdir. Eğer process shmdt ile detach işlemi yapmadıysa process sonlandığında otomatik detach işlemi yapılır, ancak IPC nesnesi silinmez. IPC nesnesinin silinmesi shell üzerinden manual olarak ya da xxxctl fonksiyonlarıyla yapılmaktadır.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Fonksiyonun üçüncü parametresi shmid_ds türünden bir yapının adresidir. İkinci parametre IPC_STAT ise ortak bellek alanı hakkındaki bilgiler üçüncü parametresiyle belirtilen yapıya yerleştirilir. shmid_ds yapısında değerli pek çok bilgi vardır. İkinci parametre IPC_SET ise tam ters bir işlem yapılır. Yani yapı içerisindeki bilgiler IPC nesnesine yerleştirilir. Ancak bu yapının yalnızca shm_perm elemanı değiştirilebilmektedir. İkinci parametre IPC_RMID ise IPC nesnesi silinir. Bu durumda üçüncü parametrenin bir önemi kalmaz 0 geçilebilir.

Sınıf çalışması: Aşağıda belirtilen ortak bellek alanı uygulamasını yazınız.

1) İki program söz konusudur. A programı ortak bellek alanını yaratır ve B programı aynı anahtarı kullanarak bu bellek alanını açar.

2) 50 byte'lık bir ortak bellek alanı yaratılacaktır. A programı döngü içerisinde klavyeden bir yazı isteyecek ve bunu ortak bellek alanına yerleştirecektir. Bu sırada B programı

```
while (1)
      pause();
```

biçiminde bekler. A programı ortak bellek alanına klavyeden alınan yazıyı yazdıktan sonra SIGUSR1 oluşturacak, B programı da bu signal'i ele alarak ortak bellek alanındaki yazıyı ekrana yazdıracaktır.

3) A programı "quit" girildiğinde quit yazısını da ortak bellek alanına yerleştirir ve döngüden çıkar. IPC detach yapıp nesnesini de siler. B programı ise quit yazısını aldıktan sonra signal fonksiyonu içerisinde programı sonlandırır.

```
/* shm_a.c */
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY         123456

void del_shm(int shmid);

int main(void)
{
      int shmid, fifofd;
      pid_t  pid;
      char *padr;

      if ((fifofd = open("pid", O_RDWR)) < 0) {
            perror("open");
            exit(EXIT_FAILURE);
      }

      if (read(fifofd, &pid, sizeof(pid_t)) < 0) {
            perror("read");
            exit(EXIT_FAILURE);
      }
      close(fifofd);

      shmid = shmget(SHM_KEY, 50, IPC_CREAT|0666);
      if (shmid < 0) {
            perror("shmget");
            exit(EXIT_FAILURE);
      }
      padr = (char *) shmat(shmid, NULL, 0);
      if (padr < 0) {
            perror("shmat");
            del_shm(shmid);
            exit(EXIT_FAILURE);
      }
      for (;;) {
            printf("Text: ");
            gets(padr);
            kill(pid, SIGUSR1);
            if (!strcmp(padr, "quit"))
                  break;
      }
      del_shm(shmid);
      return 0;
}

void del_shm(int shmid)
{
      if (shmctl(shmid, IPC_RMID, NULL) < 0) {
            perror("shmctl");
            exit(EXIT_FAILURE);
      }
}
```

```
/* shm_b.c */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY         123456

void sigusr1_handler(int sno);

char *g_padr;

int main(void)
{
      int shmid, fifofd;
      pid_t  pid;

      if ((fifofd = open("pid", O_RDWR)) < 0) {
            perror("open");
            exit(EXIT_FAILURE);
      }
      pid = getpid();
      if (write(fifofd, &pid, sizeof(pid_t)) < 0) {
            perror("read");
            exit(EXIT_FAILURE);
      }
      close(fifofd);

      shmid = shmget(SHM_KEY, 0, 0);
      if (shmid < 0) {
            perror("shmget");
            exit(EXIT_FAILURE);
      }
      g_padr = (char *) shmat(shmid, NULL, 0);
      if (g_padr < 0) {
            perror("shmat");
            exit(EXIT_FAILURE);
      }
      if (signal(SIGUSR1, sigusr1_handler) < 0) {
            perror("signal");
            exit(EXIT_FAILURE);
      }
      while (1)
            pause();

      return 0;
}

void sigusr1_handler(int sno)
{
      if (!strcmp(g_padr, "quit"))
            exit(EXIT_SUCCESS);
      puts(g_padr);
}
```

Mesaj Kuyrukları

Mesaj kuyrukları kullanımı kolay, client-server haberleşmeye olanak sağlayan IPC mekanizmasıdır. Mesaj kuyruğu mekanizması Windows programlamadaki PostMessage-GetMessage çalışma biçimine çok benzer. Yani gönderen taraf kuyruğa mesajı bırakır. Alıcı taraf kuyrukta mesaj olmadığı sürece bloke olarak bekler, sonra işlemler böyle devam eder. Mesaj kuyruğu msgget fonksiyonuyla yaratılır.

```
int msgget(key_t key, int flag);
```

flag IPC_CREAT ve IPC_EXCL bileşimlerinden oluşabilir ya da 0 geçilebilir. flag ayrıca erişim bilgileriyle kombine edilebilir.

msgsnd Fonksiyonu

Bu fonksiyon mesaj kuyruğuna mesaj göndermek için kullanılır.

```
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Fonksiyonun birinci parametresi mesaj kuyruğunun ID değeri, ikinci parametresi gönderilecek mesajın adresidir. Üçüncü parametre ikinci parametrede belirtilen adresten itibaren kaç byte'lık bir mesaj gönderileceğini belirtir. Son parametre IPC_NOWAIT ya da 0 biçiminde olabilir.

Normalde mesaj kuyruğu sınıra dayandığı için dolu olabilir ve bu durumda yer açılana kadar msgsnd bloke olur. Eğer flag 0 yerine IPC_NOWAIT olarak girilirse fonksiyon bloke olmaz, başarısızlıkla geri döner. İkinci parametrede belirtilen mesajın yapısı önce bir long ID değeri içerecek biçimde olmalıdır. Yani gönderilecek mesajın ilk sizeof(long) kadar kısmı mesajda kesinlikle olmak zorundadır, yapının diğer elemanları programcı tarafından serbest olarak oluşturulabilir. Fonksiyonun üçüncü parametresinde belirtilen uzunluk bu ilk long elemanını kapsamamaktadır. Fonksiyon başarılıysa 0, başarısızsa –1 değerine geri döner. Programcı mesajlarına çeşitli numaralar atayabilir. Yapının ilk long elemanı bu amaçla kullanılmaktadır. Böylece alıcı taraf istediği mesajları filtreleyebilir ya da mesajın türüne göre işlemlerini düzenleyebilir.

msgrcv Fonksiyonu

Bu fonksiyon mesaj kuyruğundan istenilen ID'ye sahip olan mesajı almakta kullanılır.

```
int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

Fonksiyonun birinci parametresi mesaj kuyruğunun ID değeri, ikinci parametresi mesajın yerleştirileceği adrestir. Üçüncü parametre mesajdan okunacak byte sayısıdır. type parametresi üç seçenekten biri olabilir:

1) `type == 0`, bu durumda kuyruktaki ilk mesaj alınır.
2) `type > 0`, bu durumda ID'si type olan ilk mesaj alınır.
3) `type < 0`, bu durumda kuyruktaki `abs(type) >= ID` olan en küçük mesaj alınır. Kuyruktaki mesajlardan abs(type) değerinden küçük ya da eşit olan en küçük ID'li mesaj alınır. `type == -6` ise 3 ID'li mesaj alınacaktır.

Örneğin kuyrukta 3, 5, 8 ID'lerine sahip 3 mesaj bulunsun (3 ilk mesaj, 8 son gönderilen mesajdır). type == 0 ise ilk mesaj olan 3 numaralı mesaj alınır. type == 8 ise 8 diğer mesajlar atlanarak ID'si 8 olan mesaj alınır.

Sınıf çalışması: Mesaj kuyruğu yaratan aşağıdaki programı yazınız.

Açıklamalar:

1) Programın ismi msgcreate olacaktır ve bir komut satırı argümanı alacaktır. Bu argüman bir dosya ismi olacaktır.
2) Komut argv[1]'den alınan dosya ismi ftok fonksiyonuna sokulacak ve bir anahtar değer elde edilecektir. ftok fonksiyonunun ikinci parametresi 1 alınacaktır.
3) ftok fonksiyonundan alınan anahtar değer ile msgget fonksiyonu çağırılacak msgget fonksiyonunda flag olarak IPC_CREAT | 0666 verilecektir.

```
/* msgcreate.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
      key_t key;
      int msgReturn;

      if (argc != 2) {
            fprintf(stderr, "argc failed!..\n");
            exit(1);
      }

      key = ftok(argv[1], 1);
      if (key == -1) {
            perror("ftok");
            exit(1);
      }

      msgReturn = msgget(key, IPC_CREAT | 0666);
      if (msgReturn == -1) {
            perror("msgget");
            exit(1);
      }

      return 0;
}
```

Sınıf Çalışması: Belirtilen bir kuyruğa mesaj gönderen ve mesaj alan programları ayrı ayrı yazınız.

Açıklamalar:

1) Mesaj gönderen program msgsnd isminde olacak ve üç komut satırı argümanı alacaktır.

```
msgsnd <dosya_ismi> <mesaj id'si> <"mesaj metni">
```

Program birinci parametresiyle belirtilen dosya ismini kullanarak ftok fonksiyonu ile bir anahtar elde edecek, iki ve üçüncü parametrelerle belirtilen mesajı kuyruğa msgsnd fonksiyonu ile yollayacaktır.

2) msgget fonksiyonunda flag değeri 0 geçilebilir.

3) Mesaj almak için kullanılan programın ismi msgrcv olacaktır. Bu program iki komut satırı argümanı alacaktır.

```
msgrcv <dosya_ismi> <mesaj id'si>
```

Program yine msgget ile mesaj kuyruğunu yaratacak ve belirtilen ID'ye sahip mesajı alacaktır. Bu ID doğrudan msgrcv fonksiyonunda kullanılacağından dolayı pozitif, 0 ya da negatif olabilir. msgrcv fonksiyonunda mesaj uzunluğunda büyük bir uzunluk girilebilir. msgrcv bu mesajı alarak ekrana yazdıracaktır.

```
/* msgrcv.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdlib.h>
#include <string.h>

typedef struct tagMSG{
      long id;
      char buf[100];
} MSG;

int main(int argc, char *argv[])
{
      key_t key;
      int msgReturn;
      int msgReceive;
      MSG msg;

      if (argc != 3) {
            fprintf(stderr, "argc failed!..\n");
            exit(1);
      }

      key = ftok(argv[1], 1);
      if (key == -1) {
            perror("ftok");
            exit(1);
      }

      msgReturn = msgget(key, 0);
      if (msgReturn == -1) {
            perror("msgget");
            exit(1);
      }

      msg.id = atoi(argv[2]);
```

```
      msgReceive = msgrcv(msgReturn, &msg, sizeof(MSG), msg.id, 0);

      if (msgReceive == -1) {
            perror("msgsnd");
            exit(1);
      }

      printf("id = %d msg = %s\n", msg.id, msg.buf);

      return 0;
}
```

/* msgsnd.c */

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdlib.h>
#include <string.h>

typedef struct tagMSG{
      long id;
      char buf[100];
} MSG;

int main(int argc, char *argv[])
{
      key_t key;
      int msgReturn;
      int msgSend;
      MSG msg;

      if (argc != 4) {
            fprintf(stderr, "argc failed!..\n");
            exit(1);
      }

      key = ftok(argv[1], 1);
      if (key == -1) {
            perror("ftok");
            exit(1);
      }

      msgReturn = msgget(key, 0);
      if (msgReturn == -1) {
            perror("msgget");
            exit(1);
      }

      msg.id = atoi(argv[2]);
      strcpy(msg.buf, argv[3]);

      msgSend = msgsnd(msgReturn, &msg, sizeof(MSG), 0);
      if (msgSend == -1) {
            perror("msgsnd");
            exit(1);
      }

      return 0;
```

```
}
```

Mesaj Kuyruğunda Bloke İşlemleri

msgsnd ve msgrcv fonksiyonlarının son parametresi flag parametreleridir. flag olarak ya 0 geçilir ya da IPC_NOWAIT geçilir. msgsnd fonksiyonunda bu parametre 0 geçilirse mesaj kuyruğu dolduğunda msgsnd bloke olur. Şu olaylar blokenin açılmasını sağlar:

1) Kuyruktan mesaj alınarak kuyrukta yer açılması.
2) Mesaj kuyruğunun silinmesi, bu durumda msgsnd başarısızlıkla geri döner ve errno EIDRM olur.
3) Bir signal oluşursa msgsnd başarısızlıkla geri döner ve errno EINTR ile set edilir.

Mesaj kuyruğu üzerinde üç limit söz konusudur.

1) Yaratılmış olan mesaj kuyruğundaki toplam byte sayısı
2) Mesaj kuyruğundaki toplam mesaj sayısı
3) Process'in yaratabileceği mesaj kuyruğu sayısı

msgsnd fonksiyonunda flag değeri olarak IPC_NOWAIT geçilirse bu durumda mesaj kuyruğu dolu olsa bile fonksiyon bloke olmaz. Başarısızlıkla yani –1 ile geri döner. errno EAGAIN biçiminde set edilir.

msgrcv fonksiyonundaki flag parametresi de benzerdir. Flag 0 olarak girilirse kuyruk boş olduğu sürece ya da ID ile belirtilen değere ilişkin mesaj yoksa fonksiyon bloke olur. Fonksiyonun blokesi şu durumlarda çözülür:

1) İstenen ID'ye uygun mesaj kuyruğa yerleştirildiğinde
2) Mesaj kuyruğu silinmiştir. Bu durumda msgrcv başarısız olur ve errno EIDRM olarak set edilir.
3) Bir signal oluşursa msgrcv başarısızlıkla geri döner ve errno EINTR olarak set edilir.

msgrcv fonksiyonunda flag olarak 0 yerine IPC_NOWAIT girilirse process bloke olmaz, fonksiyon başarısız olur. errno ENOMSG biçiminde set edilir.

Mesaj Kuyrukları İle Client Server Programlama

Client server programlar mesaj kuyrukları kullanılarak kolay bir biçimde oluşturulabilirler. Bunun için iki mesaj kuyruğu yeterlidir (Aslında tek mesaj kuyruğu bile kullanılabilir ama sistem gereksiz yere karmaşık olur). Mesaj kuyruklarından biri server programın okuma yaptığı, client programların yazma yaptığı mesaj kuyruğudur. Diğer mesaj kuyruğu server'ın yazma yaptığı, client'ın okuma yaptığı mesaj kuyruğudur. Bu durumda server programını organizasyonu şöyle olur:

```
for (;;) {
1)      Client kuyruğunda mesajı bekle
2)      Mesajı al ve işlemi yap
3)      Sonucu server kuyruğuna yolla
}
```

Client programların organizasyonu da şöyle olur:

```
for  (;;) {
1)       Server kuyruğunda kendine ait ID'li mesajı bekle
2)       Mesajı al ve işlemi yap
}
```

Şüphesiz client mesajları yollanırken ID olarak process ID kullanmak uygun bir yöntemdir.

Sınıf çalışması: Aşağıda belirtilen client/server programını yazınız.

Açıklamalar: Client programlar bir mesaj olarak dört işlem için iki operandı ve yapılacak işlemin ne olduğunu gönderirler. Server işlemini yaparak sonucunu client'e gönderir. Çalışmada iki kuyruk kullanılacaktır, biri client kuyruğu diğeri ise server kuyruğudur.

UNIX/Linux Sistemlerinde Static ve Dinamik Kütüphaneler

Bilindiği gibi DOS ve Windows sistemlerinde static kütüphanelerin uzantıları `.lib` biçimindedir. Unix/Linux sistemlerinde bu dosyalar `.a` uzantılıdır. DOS'ta dinamik kütüphane kullanımı yoktur. Windows'ta dinamik kütüphanelerin uzantısı `.dll` biçimindedir. Unix/Linux dünyasında dinamik kütüphanelere `"shared library"` denilmektedir. Bu sistemlerde dinamik kütüphaneler `.sa` ya da `.so` uzantılıdır.

Static Kütüphanelerin Yaratılması ve Kullanılması

Bilindiği gibi DOS'ta ve Windows'ta static kütüphaneler `lib.exe` ya da `tlib.exe` gibi özel programlarla yaratılmaktadır. Aslında static kütüphane dosyalarının formatları `.obj` modülleri tutan bir biçimdedir. Yani static kütüphaneler aslında .obj modüllerden oluşmaktadır. Unix/Linux sistemlerinde de durum tamamen DOS ve Windows'ta olduğu gibidir. Bu sistemlerde static kütüphaneler geleneksel olarak `ar` programı tarafından yaratılıp organize edilirler. ar programının genel biçimi şöyledir:

```
ar <-seçenek> <.a dosya ismi> <.o dosya ismi>
```

Örneğin mylib.c içerisindeki fonksiyonları static kütüphaneye ekleyecek olalım. Önce dosya –c switch'i ile derlenerek mylib.o oluşturulur. Sonra,

```
ar –r mylib.a mylib.o
```

biçiminde ekleme yapılır.

static kütüphaneler make dosyaları içerisinde ya da komut satırında doğrudan link aşamasına katılırlar.

Anahtar notlar: cc ya da gcc derleyicileri default olarak derleme işleminden sonra ld linker'ını çalıştırmaktadır. Yani gcc ya da cc switch'lerine ld linker switch'leri de eklenmiştir. Aslında cc ya da gcc komut satırına .c, .cpp, .o, .a gibi dosyalar girilebilir. gcc ya da cc .o ve .a uzantılı dosyaları linker'a iletmektedir.

Bir static kütüphanenin kullanılması iki biçimde yapılabilir.

1) static kütüphane dosyası komut satırı içerisinde açıkça belirtilir. Örneğin:

```
gcc –o use use.c mylib.a
```

Burada use.c içerisindeki add ve multiply fonksiyonları gerçekte mylib.a kütüphanesindedir.

2) –l switch'i ile .a dosyalarının önceden belirtilen bazı dizinlerde de aranması sağlanabilir. Bunun için aranma dizini –L ile ayrıca belirtilebilir. gcc otomatik olarak bazı dizinlere bakmaktadır. Örneğin, -l switch'ini izleyen kütüphane ismi şu biçimde belirlenir: Başına bir lib eklenir ve .a uzantısı sona eklenir. Örneğin kütüphanenin ismi libc.a olsun. Biz bu kütüphaneyi kullanmak için –lc yazmalıyız. –l ile kütüphane ismi bitişik yazılmalıdır. Örneğin;

```
gcc –o use use.c –L. –lmylib
```

Burada `use.c` dosyası derlenecek, link aşamasında `libmylib.a` kütüphanesine bakılacaktır. Ayrıca bulunulan dizin de arama işlemine dahil edilecektir.

Sınıf çalışması: Bir kütüphane dosyası hazırlayınız. Bir .c dosyasından bu kütüphane içerisindeki fonksiyonları çağırınız ve link işlemine kütüphaneyi de dahil ediniz.

Anahtar notlar: `file` isimli program ile bir dosyanın formatı hakkında bilgi edinilebilir.

```
/* staticlib.c */

int add(int a, int b)
{
      return a + b;
}


ar –r staticlib.a staticlib.o


/* usestaticlib.c */

#include <stdio.h>

int main()
{
      printf("%d", add(3, 5));

      return 0;
}

gcc –o usestaticlib usestaticlib.c staticlib.a
```

ar programının başka faydalı switch'leri de vardır. Örneğin `–t` switch'i ile kütüphane içerisindeki amaç dosyalar görüntülenebilir.

Dinamik Kütüphanelerin Oluşturulması

Dinamik kütüphaneler `ld` linker'ı ya da cc/gcc derleyicileri ile `–shared` switch'i kullanılarak oluşturulur. Örneğin;

```
gcc –o mylib.sa –shared mylib.o
```

Burada `mylib.o` dosyasından `mylib.sa` isimli bir dinamik kütüphane oluşturulmuştur. İşlem şöyle de yapılabilirdi.

```
gcc –o mylib.sa –shared mylib.c
```

Burada mylib.c derlenerek mylib.o elde edilmiştir ve bu da dinamik kütüphaneye yerleştirilmiştir.

Dinamik Kütüphanelerin Kullanılması

Tıpkı Windows sistemlerinde olduğu gibi dinamik kütüphanelerin yükleyici tarafından program çalışırken yüklenmesi ve daha sonra yüklenmesi seçenekleri Unix/Linux sistemlerinde de söz konusudur. Ancak Unix/Linux sistemlerinde import kütüphanesi kavramı yoktur. Kütüphanenin kendisi yine link işleminde kullanılmaktadır. Örneğin bu işlem şöyle yapılabilir.

```
gcc –o use use.c mylib.sa
```

Ancak yine de mylib.sa dinamik kütüphanesinin özel bir dizinde bulunması gerekir. Örneğin `/usr/lib` dizini tüm sistemlerde özel dizin olarak kullanılır.

Sınıf çalışması: Bir dinamik kütüphane oluşturunuz. Daha sonra bu dinamik kütüphaneyi bir program yoluyla kullanınız.

ldd Programı

Bir programın hangi dinamik kütüphaneleri kullandığı `ldd` programı ile tespit edilebilir.

Sınıf çalışması: Bir dinamik kütüphane yaratınız, başka bir programda o dinamik kütüphaneden çağırma yapınız. Sonra ldd programı ile rapor alınız.

Dinamik Kütüphanelerin Dinamik Yüklenmesi

Dinamik kütüphanelerin yüklenme ayrıntıları ileride ele alınacaktır. UNIX/Linux sistemlerinde de tıpkı Windows sistemlerinde olduğu gibi bir dinamik kütüphane programcının istediği bir zaman yüklenebilir.

Win32 Sistem Programcısı İçin Notlar: Win32'de bir dinamik kütüphane programın çalışma zamanı sırasında LoadLibrary API fonksiyonu ile yüklenebilir. Daha sonra dinamik kütüphaneden istenilen bir fonksiyonun adresi GetProcAddress fonksiyonu ile elde

edilebilmektedir. Dinamik kütüphanelerin boşaltılması FreeLibrary fonksiyonu ile yapılmaktadır.

UNIX/Linux sistemlerinde de tamamen benzer bir mekanizma kullanılmaktadır. Bu sistemlerde tipik olarak `dlopen` LoadLibrary gibi, `dlsym` GetProcAddress gibi ve `dlclose` FreeLibrary fonksiyonu gibi işlem görmektedir.

dlopen Fonksiyonu

Dinamik kütüphane dlopen fonksiyonu ile yüklenerek kullanıma hazır hale getirilebilir.

```
void *dlopen(const char *filename, int flag);
```

Fonksiyonun birinci parametresi dinamik kütüphane dosyasının ismidir. Birinci parametredeki path / ile başlatılmamışsa sistem dosyayı belirlenen bazı dizinlerde arar. Eğer path / ile başlatılmışsa sistem dosyayı yalnızca belirtilen yerde arar. Sistem ile belirlenen dizinler UNIX sistemleri arasında farklılıklar gösterebilmektedir. Genellikle otomatik bakılan dizinler şunlardır:

1) LD_LIBRARY_PATH çevre değişkeni ile belirtilen dizinler.
2) /usr/lib ve /lib dizinleri

Dinamik yüklemede kullanılan fonksiyonlar lib.c (standart C) kütüphanesi içerisinde değildir. Pek çok UNIX ve Linux sistemlerinde bu fonksiyonlar `libdl.a` kütüphanesinin içerisindedir. Bu nedenle derleme yapılırken bu kütüphanenin de link aşamasında eklenmesi gerekir. Örneğin;

```
gcc –o den –ldl den.c
```

ve ya

```
gcc –o den den.c /usr/lib/libdl.a
```

Dinamik kütüphane kullanımına örnek:

```
/* libmylib.c */

#include <stdio.h>

int add(int a, int b)
{
      return a + b;
}
```

```
gcc –o mylib.so -shared mylib.o
```

```
/* usedynamiclib.c */

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

typedef int (*PFUNC) (int, int);
```

```
int main()
{
      void *handle;
      PFUNC pAdd;

      handle = dlopen("/home/student/libmylib.so", RTLD_NOW);
      if (handle == NULL) {
            fprintf(stderr, dlerror());
            exit(EXIT_FAILURE);
      }

      pAdd = (PFUNC) dlsym(handle, "add");
      if (pAdd == NULL) {
            fprintf(stderr, dlerror());
            exit(EXIT_FAILURE);
      }
      printf("%d", pAdd(100, 200));

      dlclose(handle);

      return 0;
}
```

POSIX Thread İşlemleri

Klasik UNIX sistemleri bir thread mekanizmasına sahip değildi. Ancak modern UNIX sistemlerine thread mekanizması dahil edilmiştir. Thread fonksiyonları POSIX standartlarına da eklenmiştir. POSIX thread fonksiyonları yalnızca arabirim fonksiyonlardır. Yani thread mekanizmasının kurulması ve oluşturulması sistemler arasında aşağı seviyeli olarak farklılıklar gösterebilmektedir. POSIX thread mekanizması Win32 thread mekanizmasına mantıksal olarak oldukça benzemektedir. UNIX sistemlerinde değişik thread kütüphaneleri de kullanılabilmektedir. POSIX thread fonksiyonları pthread ismi ile başlar ve bu kütüphaneye pthread kütüphanesi denilmektedir.

pthread_create Fonksiyonu

Bu fonksiyon thread'i yaratmakta kullanılmaktadır.

```
int pthread_create(pthread_t *thread,
                        pthread_attr_t *attr,
                        void *(*start_routine)(void *),
                        void *arg);
```

pthread_t türü sistemden sisteme değişebilmekle beraber tipik olarak unsigned long biçimindedir. Birinci parametre thread'in ID değerinin yerleştirileceği pthread_t türünden değişkenin adresini alır (Win32 sistemlerinde hem thread ID'si hem de thread handle değeri olmak üzere iki kavram vardır. Halbuki POSIX thread fonksiyonlarında yalnızca ID kavramı vardır). Fonksiyonun ikinci parametresi pthread_attr_t türünden bir adres alır. Bu tür genellikle bir yapı biçimindedir. Bu parametre thread'e ilişkin çeşitli özellikleri belirlemekte kullanılır. NULL geçilebilir. NULL geçildiğinde default özellikler kullanılır. Fonksiyonun üçüncü parametresi geri dönüş değeri (void *), parametresi de (void *) olan bir fonksiyon adresidir. Yaratılan thread akışı bu adresten başlar. Fonksiyonun son parametresi thread fonksiyonuna aktarılacak parametredir. Thread fonksiyonlarının hepsinin prototipleri

`pthread.h` dosyası içerisindedir. Aslında fonksiyonda thread ID'si yerine NULL da geçilebilir. Tabii bu iyi bir teknik değildir. Fonksiyon başarı durumunda 0, başarısızlık durumunda sıfır dışı bir değere geri döner. Thread fonksiyonları libc.a kütüphanesinin içerisinde değildir. Bu fonksiyonlar `libpthread.a` kütüphanesinin içerisindedir.

```
/* thread.c */

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

#define SIZE 10

void *ThreadFunc(void *parg)
{
      for (;;) {
            fputc((int) parg, stderr);
      }
      return 0; // 0 göstericiye atandığında
                  // NULL anlamına gelir.
}

int main(void)
{
      pthread_t tid[SIZE];
      int i;

      for (i = 0; i < SIZE; ++i) {
            if (pthread_create(&tid[i], NULL, ThreadFunc,
                                                (void *) (i + 'a'))) {
                  fprintf(stderr, "Cannot create thread!...\n");
                  exit(1);
            }
      }
      getchar();

      return 0;
}
```

pthread_exit Fonksiyonu

Bu fonksiyon bir thread akışı içerisinde kendi thread akışını sonlandırmak için kullanılır. Thread fonksiyonu sonlandığında zaten sistem tarafından bu fonksiyon çağırılmaktadır.

```
void pthread_exit(void *retval);
```

Fonksiyonun parametresi thread fonksiyonunun geri dönüş değeridir. Yani thread fonksiyonu içerisinde return yapmak ile bu fonksiyonun çağırılması aynı anlamdadır.

Thread'lerin Sonlandırılması

Thread'lerin sonlandırılması ile process'lerin sonlandırılması işletim sistemleri için de benzer mantık içerisindedir. Örneğin process'in sonlandırılması exit ya da _exit fonksiyonu ile yapılırken, thread'lerin sonlandırılması pthread_exit fonksiyonu ile yapılmaktadır. Şüphesiz

en normal sonlanma biçimi process'in ya da thread'in kendi kendini sonlandırmasıdır. Thread'lerin de `pthread_cancel` fonksiyonu ile başka bir thread tarafından zorla sonlandırılması mümkündür (Win32 sistemlerinde benzer işlem TerminateThread fonksiyonu ile yapılmaktadır). Bilindiği gibi UNIX sistemlerinde process sonlandıktan sonra handle alanının boşaltılması iki biçimde gerçekleştirmektedir.

1) Üst process'in wait ya da waitpid fonksiyonu ile beklemesi.
2) Üst process'in sonlanması ile alt process'in üst process'inin init olması ve init tarafından alt process handle alanının yok edilmesi.

Aynı durum POSIX thread'leri için de söz konusudur. Yani bir thread sonlandığında thread'in exit kodu thread'in handle alanına yazılır ve exit kod alınana kadar handle alanı korunur. Thread'in exit kodu `pthread_join` fonksiyonu ile alınır. Yani pthread_join mantıksal bakımdan wait fonksiyonu gibi çalışmaktadır. Bu fonksiyon da wait fonksiyonunda olduğu gibi thread sonlanana kadar bekleme sağlar. Ancak pthread_join fonksiyonunu herhangi bir thread uygulayabilir (halbuki wait fonksiyonunu ancak üst process uygulayabilmektedir).

```
int pthread_join(pthread_t thID, void **thread_return);
```

Fonksiyonun birinci parametresi thread'in ID değeri, ikinci parametresi geri dönüş değerinin yerleştirileceği göstericinin adresidir. İkinci parametre NULL geçilebilir.

Sınıf çalışması: Bir thread yaratınız, thread içerisinde 1000 kere a harfini ekrana bastırınız, ana thread'i pthread_join ile bekletiniz.

```
/* pthread_join.c */

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

#define     SIZE  10

void *ThreadFunc(void *parg)
{
      int i = 0;
      static int count = 0;

      for (; i < 10; ++i) {
            fputc((int) parg, stderr);
      }

      return (void *) count++;
}

int main(void)
{
      pthread_t tid[SIZE];
      int i;
      void *pReturn;

      for (i = 0; i < SIZE; ++i) {
            if (pthread_create(&tid[i], NULL, ThreadFunc,
                                          (void *) i + 'A')) {
                  fprintf(stderr, "Cannot create thread\n");
                  exit(1);
```

```
            }
      }
      for (i = 0; i < SIZE; ++i) {
            pthread_join(tid[i], &pReturn);
            printf("%d\n", (int) pReturn);
      }
      return 0;
}
```

```
gcc –o pthread_join pthread_join.c -lpthread
```

Thread Fonksiyonlarının Geri Dönüş Değeri

Thread fonksiyonları POSIX 1.c'de eklendiği için başarısızlık durumunda errno değişkenini set etmemektedir. perror fonksiyonu errno'ya bakarak mesaj yazdığından hatanın nedeni doğrudan bu fonksiyon ile elde edilemez. Ancak hata mesajı `strerror` fonksiyonu ile yazdırılabilir. Tüm thread fonksiyonlarının geri dönüş değeri başarı durumunda 0, başarısızlık durumunda başarısızlığın nedenini anlatan hata kodudur. Yani bu fonksiyonların geri dönüş değerleri alınıp strerror fonksiyonuna parametre yapılabilir.

Detached Thread'ler

Normal olarak bir thread pthread_join yapılmadıkça handle alanını korur (UNIX sistemlerinde bir process'in açabileceği maximum thread sayısı belirlenmiştir. Thread'in handle alanı yok edilmedikçe maalesef thread açık kabul edilmektedir). Thread'ler istenirse pthread_detach fonksiyonu ile detached moda sokulabilirler. Detached modda olmayan thread'lere joinable thread'ler denir. Detached thread'ler sonlandığında otomatik olarak handle alanı yok edilir. Detached thread pthread_join fonksiyonu ile beklenemez. Bu durumda thread sonlanana kadar bekleme mekanizması başka biçimde kurulmalıdır. Fonksiyonun prototipi pthread.h dosyası içerisindedir.

```
int pthread_detach(pthread_t th);
```

pthread_self Fonksiyonu

O anda çalışmakta olan thread'in ID değerini elde etmekte kullanılır.

```
pthread_t pthread_self(void);
```

Ana Thread'in Sonlandırılması ve exit Fonksiyonu

UNIX/Linux sistemlerinde ana thread pthread_exit fonksiyonu ile sonlandırılabilir (aynı biçimde Win32 sistemlerinde de ana thread Exit_Thread fonksiyonu ile sonlandırılabilmektedir). UNIX ve Win32 sistemlerinde son thread de yok edildiğinde process otomatik olarak çekirdek tarafından sonlandırılır. exit ya da _exit fonksiyonları process'i sonlandırır, process sonlandığında da tüm thread'ler çekirdek tarafından otomatik

sonlandırılmaktadır (main fonksiyonu bittiğinde başlangıç kodu tarafından zaten exit çağırılmaktadır).

Thread ID'lerinin Karşılaştırılması

POSIX sistemlerinde thread'lerin ID değerlerini temsil eden pthread_t türü herhangi bir tür olabilir (pek çok sistemde long türündendir). Bu nedenle iki thread'in ID'sini karşılaştıran ayrı bir fonksiyon yazılmıştır.

```
int pthread_equal(pthread_t thread1, pthread_t thread2);
```

Fonksiyon iki ID birbirine eşitse sıfır dışı bir değere, eşit değilse 0 değerine geri döner.

```
/* pthread_equal.c */

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

#define SIZE      10

pthread_t g_mthread;

void *ThreadFunc(void *parg)
{
      if (pthread_equal(pthread_self(), g_mthread))
            printf("main thread is here!..\n");
      else
            printf("another thread is here!..\n");
      printf("end...\n");

      return NULL;
}

int main(void)
{
      pthread_t tid;
      int result;

      g_mthread = pthread_self();

      if ((result = pthread_create(&tid, NULL,
                                          ThreadFunc, NULL )) != 0) {
            fprintf(stderr, "%s\n", strerror(result));
            exit(1);
      }

      ThreadFunc(NULL);
      pthread_join(tid, NULL);


      return 0;
}
```

Dinamik Tahsisat İşlemlerinin Çok Thread'li Sistemlerde Güvenilirliği

Global bir bağlı listenin birden fazla thread tarafından asenkron biçimde kullanıldığını düşünelim. Özel bir seri hale getirme mekanizması uygulanmazsa thread'ler arası geçiş sırasında bağlı liste bozulabilir. Dinamik tahsis eden fonksiyonlar da kendi içinde bağlı liste kullanmaktadır. Peki bu durumda iki farklı thread aynı anda malloc fonksiyonunu kullanırsa problem oluşur mu? İşte genellikle dinamik tahsisat fonksiyonları kendi içerisinde bir seri hale getirme mekanizması ile çalışmaktadır. POSIX sistemlerindeki malloc fonksiyonu bu biçimde thread'ler arası güvenliği olan bir fonksiyondur. Win32 sistemlerinde seri hale getirme işlemi programcının isteğine bırakılmıştır. C++'ın STL kütüphanesindeki nesne tutan sınıflar tek thread'li sistemler için tasarlanmıştır. Bu nedenle bu sınıflar kullanılırken seri hale getirme mekanizması programcı tarafından sağlanmalıdır.

pthread_once Fonksiyonu

Bazen bir kod parçasının yalnızca bir kez, yani yalnızca bir thread tarafından çalıştırılması istenir. Bu tür durumlarda global değişken ile oluşturulmuş bir bayrak mekanizması ile bunun karşılanmaya çalışılması probleme yol açar. Şüphesiz bu tür işlemler senkronizasyon nesneleri ile yapılmalıdır. pthread_once fonksiyonu bunu sağlamaya yöneliktir. pthread_once fonksiyonu bir fonksiyon göstericisi parametresi alır ve yalnızca bir thread için belirtilen fonksiyonu çağırır. Bu durumda tüm thread'ler pthread_once fonksiyonunu çağırırlar, ancak bu fonksiyon içinde belirtilen kod yalnızca bir kere çalıştırılır.

```
int pthread_once(pthread_once_t *once_control, void (*init_func)(void));
```

Fonksiyon şüphesiz kendi içerisinde bir flag mekanizması kurmaktadır. Bu flag değişkenini birinci parametre olarak programcı girmelidir. Bu değişken static ömürlü olmalıdır ve işin başında PTHREAD_ONCE_INIT ile ilk değer almalıdır. Yani örnek bir kullanım şöyle olabilir:

```
pthread_once_t g_init = PTHREAD_ONCE_INIT;

void Func(void)
{

}

pthread_once(&g_init, Func);
```

fork İşlemi ve Thread'li Çalışma

Çok thread'li uygulamalarda thread'lerden biri fork işlemi yaptığında yeni yaratılan alt process tek bir threadle çalışmaya başlar. O thread fork işlemini yapan thread'dir.

```
/* forkthread.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <pthread.h>

#define SIZE      10
```

```
pid_t  g_pidChild;

void *ThreadFunc1(void *parg)
{
      int i;

      g_pidChild = fork();
      if (g_pidChild == -1) {
            perror("fork");
            exit(1);
      }
      if (g_pidChild == 0) {
            g_pidChild = getpid();
            for (i = 0; i < SIZE; ++i) {
                  printf("Child process: %d\n", getpid());
                  sleep(1);
            }
            exit(1);
      }
      wait(NULL);

      return NULL;
}

void *ThreadFunc2(void *parg)
{
      int i;
      for(i = 0;i < SIZE; ++i) {
            if (getpid() == g_pidChild)
                  printf("child process thread 2\n");
            else
                  printf("parent  process thread 2\n");
            sleep(1);
      }
      return NULL;
}

int main(void)
{
      pthread_t tid1, tid2;
      int result;

      if ((result = pthread_create(&tid1, NULL,
                                   ThreadFunc1, NULL )) != 0) {
            fprintf(stderr, "%s\n", strerror(result));
            exit(1);
      }
      if ((result = pthread_create(&tid2, NULL,
                                   ThreadFunc2, NULL )) != 0) {
            fprintf(stderr, "%s\n", strerror(result));
            exit(1);
      }

      pthread_join(tid1, NULL);
      pthread_join(tid2, NULL);


      return 0;
}
```

Birden çok thread'le çalışırken fork yapıldığında senkronizasyon nesneleri yüzünden kilitlenme (deadlock) durumu oluşabilir. Bu nedenle fork işleminden sonra böylesi kilitlenme durumlarını çözüp bazı geri alma işlemlerini sağlamak için pthread_atfork fonksiyonu düşünülmüştür.

```
int pthread_atfork(void (*prepare)(void),
                   void (*parent)(void), void (*child)(void));
```

Programcı isterse fork işleminden önce bu fonksiyonu çağırarak fork işlemi yapıldığında sistemden burada belirtilen üç fonksiyonun çağrılmasını isteyebilir. prepare fonksiyonu sistem tarafından fork yapılmadan önce üst process için çağırılır. parent fonksiyonu fork işleminden sonra henüz normal akışa geçmeden üst process için çağırılır. child fonksiyonu ise fork işleminden sonra henüz normal akışa geçmeden önce alt process için çağırılır. Fonksiyon göstericileri NULL olarak geçilebilir; bu durumda NULL geçilen fonksiyonlar için çağırma yapılmaz. Fonksiyon başarılıysa 0 değerine, başarısızsa 0 dışı bir değere geri döner.

/* atfork.c */

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <pthread.h>

#define SIZE      10

void *g_ptr;

void child(void)
{
      printf("child after fork!..\n");
      free(g_ptr);
}

void *ThreadFunc1(void *parg)
{
      return NULL;
}

void *ThreadFunc2(void *parg)
{
      g_ptr = malloc(10000);
      sleep(10);

      return NULL;
}

int main(void)
{
      pthread_t tid1, tid2;
      int result;
      pid_t pidChild;

      if ((result = pthread_create(&tid1, NULL,
                                   ThreadFunc1, NULL )) != 0) {
            fprintf(stderr, "%s\n", strerror(result));
            exit(1);
      }
```

```
        if ((result = pthread_create(&tid2, NULL,
                                     ThreadFunc2, NULL )) != 0) {
              fprintf(stderr, "%s\n", strerror(result));
              exit(1);
        }

        result = pthread_atfork(NULL, NULL, child);
        if (result) {
              fprintf(stderr, "%s\n", strerror(result));
              exit(1);
        }

        pidChild = fork();
        if (pidChild == -1) {
              perror("fork");
              exit(1);
        }
        if (pidChild == 0) {
              printf("Child process created!..\n");
              exit(1);
        }
        wait(NULL);

        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);

        return 0;
}
```

Thread Özelliklerinin Değiştirilmesi

Bir thread'in özellikleri denilince onun stack alanının uzunluğu, stack bölgesinin başlangıç adresi ve joinable olup olmadığı anlaşılır. Thread'lerin diğer özellikleri nispeten daha önemsizdir. Thread yaratılırken pthread_create fonksiyonunun bir parametresi thread özelliklerinin belirlenmesinde kullanılan (pthread_attr_t *) türündendir. pthread_attr_t türü genellikle bir yapıdır. Yani fonksiyon bizden bu yapının adresini istemektedir. Bu yapı POSIX tarafından standart hale getirilmemiştir. Bu nedenle bu yapıya erişip işlem yapmak için arabirim fonksiyonlar düşünülmüştür. Bu durumda programcı pthread_attr_t türünden bir değişken tanımlar. Standart fonksiyonlarla bu değişkenin belirttiği yapının içini doldurur. Bu yapının adresini de pthread_create fonksiyonuna geçirir.

pthread_attr_t türünden nesneyi önce ilk değerlemek gerekir. Bu işlem pthread_attr_init fonksiyonuyla yapılır.

```
int pthread_attr_init(pthread_attr_t *attr);
```

Thread yaratıldıktan sonra pthread_attr_destroy fonksiyonuyla boşaltılmalıdır.

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Şüphesiz thread özelliklerini barındıran pthread_attr_t türü çeşitli gösterici elemanları içermektedir. pthread_attr_destroy fonksiyonu bu göstericiler için tahsis edilmiş alanları boşaltmaktadır. pthread_attr_destroy işlemi thread yaratıldıktan hemen sonra yapılabilir.

Thread özellikleri ile ilgili fonksiyonlar şunlardır:

```
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

Sistemlerdeki default stack uzunluğu standart olarak belirlenmemiştir. Ancak pthread.h dosyası içerisinde `PTHREAD_STACK_MIN` sembolik sabiti ile default stack uzunluğu belirtilmiştir.

```
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

Programcı thread'i yaratmadan önce isterse kendi tahsis ettiği alanı thread'in stack'i olarak kullanabilir. Eğer adres belirlemesi yapılmazsa stack alanı sistem tarafından tahsis edilir.

```
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Bu fonksiyonlar thread'in detach ya da joinable olma durumunu belirler.

```
/* threadattr.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>

void check_error(const char *msg, int result)
{
      if (result) {
            fprintf(stderr, "%s: %s\n", msg, strerror(result));
            exit(EXIT_FAILURE);
      }
}

void *thread_func(void *param)
{
      int i;

      for (i = 0; i < 10; ++i) {
            printf("%d\n", i);
            sleep(1);
      }
      return NULL;
}

int main(void)
{
      int result;
      pthread_t tid;
      pthread_attr_t tattr;

      result = pthread_attr_init(&tattr);
      check_error("pthread_attr_init", result);

      result = pthread_attr_setstacksize(&tattr, 100000);
      check_error("pthread_attr_setstacksize", result);
```

```
	result = pthread_create(&tid, &tattr, thread_func, NULL);
	check_error("pthread_create", result);

	result = pthread_join(tid, NULL);
	check_error("pthread_join", result);

	result = pthread_attr_destroy(&tattr);
	check_error("pthread_attr_destroy", result);

	return 0;
}
```

Thread'lerin Dışarıdan Sonlandırılması

Bilindiği gibi bir thread'in normal olarak sonlandırılması iki biçimde yapılmaktadır:

1) Thread fonksiyonunun sona ermesi
2) pthread_exit fonksiyonunun çağırılması

Bunların dışında da thread'ler dışarıdan pthread_cancel fonksiyonu ile sonlandırılabilirler. Bu fonksiyon Win32 sistemlerindeki TerminateThread fonksiyonuna benzemektedir. Ancak Win32 sistemlerinden farklı olarak POSIX sistemlerinde thread'in sonlandırılma seçenekleri daha fazladır.

POSIX sistemlerinde thread'ler senkron ya da asenkron biçimde sonlandırılabilmektedir. Asenkron sonlandırma demek thread'in o anda zorla sonlandırılması demektir (Win32 sistemlerindeki TerminateThread bu biçimde çalışır). Senkron sonlandırmada pthread_cancel uygulansa bile thread hemen sonlandırılmaz, belirli noktalarda sonlandırılır. Bu durumda sonlandırılma isteği sisteme iletilir, ancak sistem uygun noktayı bekler. Thread yaratıldığında default olarak senkron sonlandırma durumu söz konusudur. Thread'in sonlandırma biçimi pthread_setcanceltype fonksiyonu ile belirlenir.

```
int pthread_setcanceltype(int type, int *oldtype);
```

type parametresi senkron sonlanma için `PTHREAD_CANCEL_DEFERRED`, asenkron sonlanma için `PTHREAD_CANCEL_ASYNCHRONOUS` değerini alır. Görüldüğü gibi fonksiyonda bir thread ID'si yoktur, yani fonksiyon kendi thread'i üzerinde işlemler yapmaktadır.

Senkron sonlanma hangi noktalarda gerçekleşmektedir? Çeşitli sistem fonksiyonları senkron sonlanmaya duyarlıdır. Bunların bir listesi vardır. Bu fonksiyonlardan bazıları şunlardır:

```
open, close, sleep, system, read, write, wait, waitpid
```

Ayrıca senkron sonlanma noktası `pthread_testcancel` fonksiyonu çağırılarak da belirlenebilmektedir. Örneğin, bir döngü içerisinde uygun bir noktada bu fonksiyon çağırılarak sonlandırma noktası oluşturulabilir. Bu fonksiyon şüphesiz thread için sonlandırma isteği olup olmamasına bakar ve kendi thread'ini sonlandırır.

Bazen programcı kritik bazı işlemleri yaparken thread'in sonlandırma biçimi ne olursa olsun bu sonlandırma işlemini devre dışı bırakabilir. Bunun için `pthread_setcancelstate` fonksiyonu kullanılır. Fonksiyonun prototipi aşağıdaki gibidir:

```
int pthread_setcancelstate(int style, int *oldstyle);
```

Her iki fonksiyon da ikinci parametreye NULL değeri geçilmesi durumunda eski değeri yerleştirmemektedir. Bu fonksiyonun birinci parametresi `PTHREAD_CANCEL_ENABLE` ya da `PTHREAD_CANCEL_DISABLE` olabilir.

Thrad'in sonlandırılması nihayet pthread_cancel fonksiyonu ile yapılabilir.

```
int pthread_cancel(pthread_t tid);
```

Fonksiyonun parametresi sonlandırılacak thread'in ID değeridir.

```
/* thread_cancel.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>

void check_error(const char *msg, int result)
{
      if (result) {
      fprintf(stderr, "%s: %s\n", msg, strerror(result));
            exit(EXIT_FAILURE);
      }
}

void *thread_func(void *param)
{
      int i = 0, result;
      int status;
      long d;

      result = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &status);
      check_error("pthread_setcancelstate", result);

      for (;;) {
            printf("%d\n", i++);
            sleep(1);
            for (d = 0; d < 100000000; ++d)
                  ;
            pthread_testcancel();
            if (i++ > 10) {
                  result = pthread_setcancelstate(status, NULL);
                  check_error("pthread_setcancelstate", result);
                  break;
            }
      }
      return NULL;
}

int main(void)
{
```

```
        int result;
        pthread_t tid;

        result = pthread_create(&tid, NULL, thread_func, NULL);
        check_error("pthread_create", result);

        getchar();

        result = pthread_cancel(tid);
        check_error("pthread_cancel", result);

        result = pthread_join(tid, NULL);
        check_error("pthread_join", result);

        return 0;
}
```

Thread'in sonlandırmaya karşı kapatılmış ise sonlandırma durumu ister senkron olsun, ister asenkron olsun kapatılma durumunda oluşan kapatma isteği saklanır ve açıldığında etki göstererek thread sonlandırılır.

Thread'ler ve Signal Mekanizması

POSIX sistemlerinde eskiden thread'ler yoktu. Thread mekanizması eklendiğinde zaten var olan signal mekanizmasının bu yeni sisteme uyumlu olması gerekmişti. Thread'li sistemde signal mekanizması için gerekli olan birkaç durum vardır:

- Signal fonksiyonu set edilmemişse, bir thread mi, yoksa tüm process mi sonlanacaktır.
- Signal fonksiyonu set edilmişse signal oluştuğunda hangi thread bu fonksiyonu çağıracaktır.
- Signal'ın mask yapılması thread başına mı gerçekleşecektir, yoksa process temelinde mi gerçekleşecektir.

Thread'li sistemde de bir signal oluştuğunda çağırılacak fonksiyon belirlenmemişse yalnızca tek bir thread değil, yine tüm process sonlandırılmaktadır.

Bir signal oluştuğunda birden fazla thread çalışmaktaysa yalnızca bir thread'in mi bu signal'ı işleyeceği, yoksa hepsinin mi işleyeceği POSIX standartlarında belirlenmemiştir (fakat işlenmediğinde bundan tüm process'in etkileneceği belirlenmiştir). Bu durum belirsizliğe yol açtığı için thread başına çalışan çeşitli signal fonksiyonları düşünülmüştür. Thread'siz versiyonlarda anımsanacağı gibi `sigprocmask` fonksiyonu signal'i maskelemek için kullanılıyordu. Yani process'in toplam bir tane signal mask değeri vardı. Thread'li sistemde her thread'in ayrı bir signal mask değeri vardır. Bu değer pthread_sigmask fonksiyonu ile set edilir.

```
int pthread_sigmask(int how, sigset_t *set, sigset_t *oldset);
```

Görüldüğü gibi bu fonksiyon ile belirli bir signal bazı thread'ler için devre dışı bırakılabilir. Her thread'in signal mask değeri birbirinden farklı olabilir. Bu durumda bir signal oluştuğunda signal fonksiyonunu tek bir thread'in çalıştırması isteniyorsa pthread_sigmask fonskiyonu ile bir thread dışındaki tüm thread'ler bu signal'a kapatılır, yalnızca tek thread bu signal'a açılır. Bunu yapmanın başka taşınabilir bir yöntemi yoktur.

kill sistem fonksiyonu process'e signal göndermektedir. Yani bu fonksiyon thread'siz sistem zamanlarında tasarlandığı için yukarıda açıklanan taşınabilirlik problemi oluşmuştur. POSIX1.c'de bir thread'e signal gönderme kavramı da eklenmiştir. Bir thread'e signal gönderildiğinde signal fonksiyonu o thread tarafından işletilir. Ancak signal fonksiyonu set edilmemiş ise yine bundan tüm process etkilenmektedir. POSIX sisteminde bir thread'in signal fonksiyonunu kendisinin set etmesi diye bir kavram yoktur. signal fonksiyonu yine `signal` ya da `sigaction` fonksiyonu ile process temelinde set edilir. `pthread_kill` yalnızca process temelinde set edilmiş signal fonksiyonunu hangi thread'in ele alıp işleyeceğini belirleyerek signal'ı gönderir.

```
int pthread_kill(pthread_t thread, int signo);
```

Görüldüğü gibi fonksiyonun birinci parametresi thread'in ID değeri, ikinci parametresi yollanan signal numarasıdır. Örneğin:

```
pthread_kill(tid, SIGUSR1);
```

Burada;

1) `SIG_USR1` signal ya da sigaction fonksiyonu ile set edilmemişse process sonlanır.
2) `SIG_USR1` signal'ı set edilmişse ve signal'ın gönderildiği thread bu signal'a açıksa thread signal'ı işler.
3) kill ile aynı işlem yapılsaydı signal'ı hangi thread'in işleyeceği ayrıca yukarıda anlatılan yöntem kullanılarak belirlenmeliydi.

Ayrıca pthread_kill ile başka bir process'in thread'ine signal gönderilmez. Çünkü thread'in ID değeri process'e özgü bir değerdir. Başka process'lere signal yine kill ile gönderilmek zorundadır.

Thread'lerin Senkronizasyonu

POSIX sistemlerinde thread'lerin senkronizasyonu için çeşitli yöntemler kullanılmaktadır. Bu yöntemlerden bazıları farklı process'lerin thread'leri arasında kullanılabilirken, bazıları aynı process'in thread'leri arasında kullanılabilmektedir.

Mutex Kullanımı

Mutex neredeyse her türlü işletim sisteminde olan bir senkronizasyon nesnesidir. Mutex bir thread tarafından ele geçirilir. Mutex'i ele geçiren thread mutex'i istediği zaman serbest bırakabilmektedir. Genellikle sistemlerde bir thread mutex'i ele geçirdiği zaman başka bir thread bu mutex'i ele geçirmeye çalışınca bloke olur. Mutex'in sahibi olan thread mutex nesnesini bırakana kadar thread bekler. Bazı sistemlerde (örneğin Win32 sistemlerinde) mutex nesnesi aynı thread tarafından birden fazla kez ele geçirilmektedir. Bu durumda thread'in mutex'i bırakması için yine ele geçirme sayısı kadar bırakma işlemi yapması gerekir. Şüphesiz böyle sistemlerde mutex nesnesi içerisinde bir sayaç bulundurulmaktadır.

Mutex Nesnesinin Yaratılması

Mutex nesnesinin yaratılması için pthread_mutex_t türünden bir nesne tanımlanır. Mutex nesnesini birden fazla thread kullanacağına göre bu nesnenin global ya da heap üzerinde yaratılması uygundur. Mutex nesnesi tanımlandıktan sonra ona ilk değer vermek gerekir. İlk değer verme işlemi nesneyi tanımlar tanımlamaz `PTHREAD_MUTEX_INITIALIZER` makrosu ile ya da `pthread_mutex_init` fonksiyonu ile yapılabilir. Örneğin;

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Bu işlem ilk değer verme biçiminde yapılmalıdır.

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutex_attr_t *attr);
```

Fonksiyonun birinci parametresi mutex nesnesinin adresi, ikinci parametresi mutex özelliklerinin değerlerine ilişkin yapının adresidir. Bu parametre NULL olarak geçilebilir. Bu durumda mutex default özelliklere sahip olur (PTHREAD_MUTEX_INITIALIZER da default özellikleri belirler).

Mutex Nesnesinin Ele Geçirilmesi ve Bırakılması

Mutex kullanımı oldukça basittir. Bir thread mutex nesnesini `pthread_mutex_lock` fonksiyonu ile ele geçirir. Thread mutex nesnesinin `pthread_mutex_unlock` ile bırakmaktadır. Mutex kullanılarak tipik bir kritik kod şöyle oluşturulur:

```
pthread_mutex_lock(...);
// ...
Kritik kod !
// ...
pthread_mutex_unlock(...);
```

Kritik kod (critical section) yalnızca tek bir akış tarafından işlenecek kod parçası için kullanılan bir terimdir. Mutex ilk kez pthread_mutex_lock fonksiyonunu çağıran thread tarafından ele geçirilir. Mutex bir thread tarafından ele geçirilmiş ise bu durumda başka bir thread pthread_mutex_lock fonksiyonunu aynı mutex nesnesi ile kullanırsa bloke olarak bekler. Mutex'e sahip olan thread mutex nesnesini bıraktığında beklemekte olan en yüksek öncelikli thread mutex'in kontrolünü ele alarak mutex'i ele geçirir.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Örnek Uygulama

Global bir bağlı listenin birden fazla thread tarafından asenkron biçimde kullanıladığını düşünelim. Bu durumda bir thread bağlı liste işi yaparken diğerinin beklemesi gerekir.

```
/* listapp.cpp */

#include <cstdio>
#include <cstdlib>
```

```
#include <pthread.h>
#include <unistd.h>
#include <algorithm>
#include <list>

#define SIZE      10000

using namespace std;

list<int> g_list;
pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;

void check_error(const char *msg, int result)
{
      if (result) {
            fprintf(stderr, "%s: %s\n", msg, strerror(result));
            exit(EXIT_FAILURE);
      }
}

void *Thread1(void *param)
{
      for (int i = 0; i < SIZE; ++i) {
            for (int k = 0; k < 100000; ++k)
                  ;
            pthread_mutex_lock(&g_mutex);
            g_list.push_back(i);
            pthread_mutex_unlock(&g_mutex);
      }
}

void *Thread2(void *param)
{
      for (int i = 0; i < SIZE; ++i) {
            pthread_mutex_lock(&g_mutex);
            g_list.push_back(-i);
            pthread_mutex_unlock(&g_mutex);
            for (int k = 0; k < 100000; ++k)
                  ;
      }
}

int main(void)
{
      int result;
      pthread_t tid1, tid2;

      result = pthread_create(&tid1, NULL, Thread1, NULL);
      check_error("pthread_create", result);

      result = pthread_create(&tid2, NULL, Thread2, NULL);
      check_error("pthread_create", result);

      result = pthread_join(tid1, NULL);
      check_error("pthread_join", result);

      result = pthread_join(tid2, NULL);
      check_error("pthread_join", result);

      copy(g_list.begin(), g_list.end(), ostream_iterator<int>(cout, " "));
```

```
	return 0;
}
```

Bazı durumlarda programcı mutex nesnesi eğer başka bir thread tarafından ele geçirilmediyse ele geçirmek ister. Yani mutex'i başka bir thread ele geçirdiyse bloke olmak istemez. Bunun için `pthread_mutex_trylock` fonksiyonu kullanılmaktadır.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Fonksiyon eğer mutex nesnesi boşta ise nesneyi ele geçirir ve başarı ile geri döner. Eğer mutex nesnesi başka bir thread tarafından ele geçirilmişse fonksiyon bloke olmaz EBUSY error kodu ile geri döner.

Anahtar notlar: *Win32 sistemlerinde mutex nesnesi CreateMutex API fonksiyonu ile yaratılır. Mutex nesnesini yaratan kişi tarafından ele geçirilip geçirilmeyeceği bu fonksiyonun parametresinde belirlenir. Bu sistemlerde mutex nesnesini ele geçirmek için WaitForSingleObject fonksiyonuna bırakılır. Bu fonskiyon adeta pthred_mutex_lock gibi işlem yapar. Mutex nesnesi ele geçirilmişken başka bir thread WaitForSingleObject uygularsa blokede kalır, thread mutex fonksiyonunu ReleaseMutex fonksiyonu ile bırakılır. ReleaseMutex pthread_mutex_unlock gibi işlem yapar. POSIX sistemlerinde mutex'e sahip olan thread'in bir kere daha pthread_mutex_lock uygulaması geçerli bir işlem değildir. Oysa Win32 sistemlerinde mutex nesnesine sahip olan thread birden fazla WaitForSingleObject uygulayabilir. Fakat nesneyi geri bırakmak için o sayıda ReleaseMutex yapmak gerekir.*

Mutex Nesnesinin Process'ler Arasında Kullanılması

Mutex nesnesinin process'ler arasında kullanılabilmesi için paylaşılmış bir alanda bulunması gerekir. Ancak öncelikle mutex nesnesini yaratırken bunun process'ler arasında ortak kullanılacağı belirtilmelidir. Bilindiği gibi pthread_mutex_init fonksiyonu ile mutex'e ilk değer verirken bu fonksiyon parametre olarak bir özellik almaktadır. Bu özellik bilgisi pthread_mutexattr_t türü ile temsil edilir. Bunun için önce pthread_mutexattr_init fonksiyonu çağırılır, sonra işlem sonunda pthread_mutexattr_destroy yapılır. Özelliği set eden tek bir fonksiyon vardır. Bu fonksiyon da `pthread_mutexattr_setpshared` fonksiyonudur. Bu fonksiyonun ikinci parametresinde paylaşım için `PTHREAD_PROCESS_SHARED` girilir. Default olarak eğer mutex nesnesi `PTHREAD_MUTEX_INITIALIZER` ile ilk değer verilerek tanımlanmışsa ya da pthread_mutex_init fonksiyonunda özellik parametresine NULL geçilerek oluşturulmuşsa nesne process'ler arasında paylaşılamaz.

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_getpshared(pthread_mutexattr_t *attr, int *pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

Özetle, process'ler arasında paylaşılan bir mutex nesnesi yaratmak için şunlar yapılmalıdır:

1) Her iki process'te de mutex nesnesi paylaşılan bellekte yaratılır. Bunu sağlamak için her iki programda da pthread_mutex_t türünden global bir gösterici alınır. Sonra shmget fonksiyonu ile paylaşılan bellekte mutex nesnesi için ortak alan tahsis edilir.
2) Mutex nesnesinin özellikleri için pthread_mutexattr_t türünden bir nesne tanımlanır. Bu nesne için önce pthread_mutexattr_init fonksiyonu çağırılır. Sonra

pthread_mutexattr_setpshared fonksiyonu çağırılarak mutex'in process'ler arasında paylaşılacağı bilgisi verilir.
3) pthread_mutex_init fonskiyonu ile paylaşılan alanda yaratılmış olan mutex yaratılmış olan özellikle set edilir.
4) Bu noktada artık özellik nesnesi pthread_mutexattr_destroy ile geri bırakılabilir.
5) Artık mutex işlemleri yapılır. İşlemin sonunda mutex nesnesi ve paylaşılan bellek alanı geri bırakılır.

Anahtar notlar: Linux sistemlerinde pthread_mutexattr_setpshared fonksiyonu yanı sıra pthread_mutexattr_getkind_np fonksiyonu kullanılmaktadır.

Make Dosyalarının Oluşturulması

Birden fazla modülle ve kütüphanelerle çalışıldığı durumlarda etkin bir build işlemi için yalnızca değişen dosyaların yeniden derlenmesi gerekir. Tabii bütün dosyaların ne olursa olsun hep birlikte link edilmesi mutlaka gerekmektedir (son yıllarda daha önce link edilmiş bir dosyayı daha kolay link edebilmek için artırımlı linker'lar geliştirilmiştir). Birden fazla modülle çalışıldığı durumlarda derleyicilerin tümleşik çevreli uyarlamalarında proje dosyası kavramı bu işi yürütmek için kullanılmaktadır. Bu yöntemde projeye çeşitli dosyalar eklenir, programcının yapacağı tek şey projeyi build yapmaktır. Komut satırından idare edilen derleyici sistemlerinde proje dosyası yerine make dosyası kullanılır.

Make dosyası bir text dosyası biçimindedir. Sanki kuralı olan küçük bir dile sahiptir. Programcı make dosyası içerisinde hangi dosyalar değiştiğinde hangi işlemlerin yapılacağını belirtir. Make dosyaları make isimli bir program tarafından okunarak yorumlanır.

Make programı komut satırı argümanı vermeden çalıştırılırsa otomatik olarak `makefile` ve `Makefile` isimli dosyaları araştırır. Yani böyle bir dosya varsa tek yapacağımız şey ilgili dizine geçip make programını çalıştırmaktır. Make dosyasını belirleyerek make yapmak için

```
make –f <dosya_ismi>
```

yapılır.

cc, gcc ve ld Programları

Unix sistemlerinin C derleyicileri klasik olarak cc ismine sahiptir. Pek çok kurum tarafından yazılmış ayrı cc derleyicileri vardır. Linux sistemlerinde GNU projesi kapsamında yazılmış olan gcc derleyicisi kullanılır. Ancak bu sistemlerde cc softlink olarak gcc'ye bağlı bir biçimde de bulundurulmaktadır. cc derleyicisi –c seçeneği kullanılmamışsa önce derleme işlemini yapar, sonra ld linker'ını çağırarak link işlemini yapar. cc linker'ı çağırırken startup modülü ve libc.a kütüphanesini de işleme dahil etmektedir. libc.a ya da libc.so hem standart C fonksiyonlarını hem de POSIX fonksiyonlarını içermektedir (POSIX'e sonradan katılmış olan fonksiyonlar başka kütüphanelerde bulundurulmuştur. Örneğin thread fonksiyonları libpthread.a ya da libpthread.so dosyalarında bulunur). cc derleyicisi kendi içerisinde linker'ı da çağırdığına göre cc sanki bir linker'mış gibi de kullanılabilmektedir. Örneğin x.c isimli bir dosya ayrı ayrı derlenip link edilecekse şöyle bir yöntem izlenir:

```
cc –c x.c
ld –o x –lc /usr/lib/crt0 x.o
```

Burada görüldüğü gibi startup modül ve C kütüphanesi ayrıca işleme sokulmuştur. startup modül değişik bir dizinde ve değişik bir isimde olabilir. Buradaki aynı işlem hiç linker kullanılmadan şöyle de yapılabilirdi:

```
cc –c x.c
cc –o x x.o
```

Aslında –o seçeneği derleyicinin değil, linker'ın bir seçeneğidir. Yani bazı seçenekler doğrudan linker'a aktarılırlar:

```
cc –o x x.c
```

Make Dosyalarının İçeriği

Make dosyaları kurallardan oluşur. Bir kuralın genel biçimi şöyledir:

```
dosya_ismi : dosya_ismi  à Bağımlılık (dependency)
      yapılacak işlem
```

`:`'nin solunda ve sağında aralarında boşluk bırakılmış biçimde birden fazla dosya bulunabilir. `:` satırına bağımlılık (dependency) denilmektedir. Bağımlılık şu anlama gelir: `:`'nin sağındaki dosyalardan herhangi birinin tarih ya da zamanı solundaki dosyalardan her hangi birinin tarih ya da zamanından daha yeniyse aşağıda belirtilen işlem yapılacaktır. Kurallar sondan başa doğru yazılırlar.

```
# deneme

X : a.o b.o
      cc –o x a.o b.o

a.o : a.c
      cc –c a.c

b.o : b.c
      cc –c b.c

a.o b.o : x.h
```

karakteri yorumlama için kullanılır. Kuralların sırası önemlidir. Bir kural başka bir kurala bağlı ise bağlı olan kural yukarı yazılır. Yani sondan başa doğru bir yazım biçimi uygulanmalıdır. Kurallar eylem içermek zorunda değildir. Örneğin;

```
a.o : x.h
```

Burada x.h değiştiğinde a.o değişmiş kabul edilsin anlamı vardır.

Make Dosyalarında Makro Kullanımı

Uzun make dosyalarında aynı satırların defalarca yazılması zor olabilmektedir. Bunun için makro kullanılabilir. Makro tanımlamanın genel biçimi şöyledir:

```
isim = değer
```

Örneğin;

```
CC = gcc
SOURCE = a.c b.c
OBJS = a.o b.o
```

Makrolar `$(isim)` ya da `${isim}` biçiminde kullanılabilir. Örneğin;

```
#deneme

X : ${OBJS}
      CC –o x ${OBJS}

a.o : a.c
      CC –c a.c

b.o : b.c
      CC –c b.c

${OBJS} : x.h
```

Make dosyaları içerisinde shell'in çevre değişkenleri kullanılabilir. Ayrıca bir makro komut satırından da girilebilmektedir. Örneğin;

```
make "OBJS = a.o b.o"
```

make içerisinde hiç tanımlamadan kullanılabilecek önceden tanımlanmış çeşitli makrolar da vardır. Bunların bazıları şunlardır:

```
CC = cc
LD = ld
AR = ar
SHELL = /bin/sh
```

CSD Heap Sistemi İçin make Dosyasının Oluşturulması

Heap sistemi için kullanılan C kaynak dosyaları şunlardır: heap.c, syserr.c, util.c. Bu dosyalar başlık dosyalarına şöyle bir bağımlılık içerisindedir.

```
heap.c -> kassert.h + syserr.h + util.h + kparam.h + heap.h + general.h
syserr.c -> syserr.h + general.h
util.c -> util.h

#CSD heap system makefile

CC = gcc
OBJS = heap.o syserr.o util.o
EXE = heap

$(EXE): $(OBJS)
```

```
	CC –o $(EXE) $(OBJS)

heap.o: heap.c
	CC –c heap.c

syserr.o: syserr.c
	CC –c syserr.c

util.o: util.c
	CC –c util.c

heap.o: kassert.h syserr.h util.h kparam.h heap.h general.h

syserr.o: syserr.h general.h

util.o: util.h general.h
```

Semaforlar

Mutex Unix/Linux ve Win32 sistemlerinde thread tarafından ele geçirilen senkronizasyon nesnesidir. Oysa bazı durumlarda bir kritik koda tek bir kişi değil, örneğin n değişik kişinin girmesi istenebilir. Semaforlar bu bakımdan mutex nesnelerinin sayaçlı biçimleridir. Semaforlar aşağı yukarı tüm işletim sistemlerinde benzer kullanıma sahiptir. Semafora girmek ile çıkmak biçiminde iki kavram vardır. Semaforların bir sayacı bulunur. Bu sayaç başlangıçta semaforu yaratan kişi tarafından set edilir. Bir kod semafora girdiğinde sayaç azaltılır, çıktığında artırılır. Bir kod semafora girerken işletim sistemi semafor sayacına bakar. Sayaç 0'dan büyükse girme hakkını elde eder ve sayaç bir eksiltilir. Sayaç 0 ise semafora girmek isteyen kod semafor içerisinde bloke edilir. Başka bir kod semafordan çıktığında sayaç artırılır. Bekleyen kodlardan en yüksek öncelikli olan kod semafora giriş sağlar. Böylece mutex'lerde olduğu gibi kritik koda sadece tek bir kişi değil, aynı anda n kişi girebilmektedir.

Win32 sistemlerinde semafor CreateSemaphore API fonksiyonuyla yaratılır. Yaratan kişi sayacı belirler. Kritik kod WaitForXXX ile ReleaseSemaphore fonksiyonları arasına alınır. WaitForXXX fonksiyonları semafor sayacı 0'dan büyükse azaltma işlemini yapıp kritik koda giriş işlemini yaparlar. Semafor sayacı 0 ise WaitForXXX fonksiyonları threadi çizelge dışı bırakmaktadır. ReleaseSemaphore fonksiyonu semafor sayacını artırmaktadır.

```
WaitForSingleObject(hSemphore, INFINITE);
// ...
ReleaseSemaphore(hSemaphore, 1);
```

Processlerarası Haberleşme ve Senkronizasyon Konusundaki SystemV ve POSIX Farklılıkları

Processlerarası haberleşme ve senkronizasyon konusunda Unix/Linux programcıları için iki seçenek söz konusudur:

1) Oldukça yaygın olarak bilinen SystemV fonksiyonlarını kullanmak. Bu fonksiyonlar AT&T Unix sistemleri tarafından tasarlanmıştır, ama sistemlerin hemen hepsinde geçerli olarak kullanılmaktadır.
2) POSIX standartlarında belirtilen fonksiyonlardır. Bu fonksiyonlar daha taşınabilir ve daha iyi tasarlanmış olmalarına karşın daha az kişi tarafından kullanılmaktadır.

POSIX standartlarının birincisi yani POSIX.1 ya da 1003.1 Unix sistemlerindeki sistem fonksiyonlarını tanımlar. POSIX.2 shell programlarına ilişkin standarttır. POSIX.3 test yöntemlerine ilişkin belirlemeleri içerir. 1993-94 senelerinde POSIX fonksiyonlarına yeni eklemeler yapılmıştır. Bu eklemelerin numaralandırılması konusunda anlaşmazlıklar çıkmıştır. Eklemeler üç aşamada gerçekleştirilmiştir. Bir grup bu eklentileri POSIX.1.b, POSIX.1.c, POSIX.1.d biçiminde, bir grup ise POSIX.4.a, POSIX.4.b ve POSIX.4.c biçiminde isimlendirmiştir. Sonradan yapılan bu eklemelerin bazıları şunlardır:

- thread fonksiyonları
- paylaşılan bellek alanı (shared memory) ve mesaj kuyrukları
- mutex ve semaphore'lar
- diğer bazı konular...

Process'lerarası haberleşme ve senkronizasyon konusunda yalnızca pipe kullanımı SystemV ve POSIX standartlarında ortaktır. Bu zamana kadar görülmüş olan shared memory ve mesaj kuyrukları SystemV'e özgü olanlardır, çünkü bu konuda SystemV fonksiyonları çok daha yaygın kullanılmaktadır, ancak maalesef SystemV'in semaphore fonksiyonları POSIX biçimlerine göre oldukça kullanışsızdır.

SystemV'in senkronizasyon nesneleri key_t türünden sayılarla teşhis edilmektedir. Yani bu yönteme göre farklı procesler aynı sayıyı verirlerse aynı senkronizasyon nesneleri üzerinde işlem yaparlar. İsimli pipe kullanımında istisna olarak bir dosya ismi temasının geçtiğini görüyoruz. İşte POSIX'in bütün senkronizasyon nesneleri isimli pipe'larda olduğu gibi bir dosya ismi ile teşhis edilmektedir. POSIX'in senkronizasyon nesneleri için verilecek dosya isimleri POSIX standartlarına göre root dizininde olmalıdır. Standartlara göre bu isim '/' ile başlar, fakat root'un altındaki başka bir dizinle devam ederse ne olacağı, yani işlemin geçerli olup olmayacağı işletim sistemini yazanlara bırakılmıştır. Maalesef bazı sistemlerde normal bir kullanıcının root dizinine yazma hakkı yoktur. Bu durumda standartlardaki durum hepten belirsiz hale gelmiştir. Yapılacak en iyi şey önce root üzerinde isimleri yaratmaya çalışmak, eğer bu mümkün olamıyorsa alt dizinler üzerinde işlemlere devam etmektir.

POSIX Semaphore'ları

POSIX semaphore sistemi tıpkı pipe'larda olduğu gibi isimli ve isimsiz olmak üzere iki gruptur. İsimsiz semaphore'lar aynı process'in threadleri arasında, isimli semaphore'lar farklı process'lerin thread'leri arasında kullanılabilirler.

İsimsiz POSIX Semaphore'ları

İsimsiz semaphore'lar şu fonksiyonlarla kullanılır:

```
sem_init
sem_destroy
sem_wait
sem_post
sem_trywait
```

sem_init semaphore'u yaratır; sem_destroy yok eder; sem_wait semaphore sayacına bakar, sayaç 0 ise thread'i bloke eder, sayaç 0'dan büyükse sayacı 1 azaltarak akışı devam ettirir;

sem_post ise sayacı artırmaktadır; sem_trywait bloke etmeden giriş hakkı kontrolü yapmaktadır. Bu durumda kritik kod aşağıdaki gibi oluşturulmalıdır:

```
sem_init(....);
...
sem_wait(....);
...
/* Kritik kod */
...
sem_post(....);
...
sem_destroy(....);
```

Aynı işlemler Win32 sistemlerinde şöyle yapılmaktadır:

```
CreateSemaphore(....);
...
WaitForSingleObject(....);
...
/* Kritik kod */
...
ReleaseSemaphore(....);
...
CloseHandle(....);
```

İsimsiz POSIX semaphore fonksiyonlarının prototipleri şöyledir:

```
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

pshared mutex'lerde olduğu gibi semaphore'un processler arasında paylaşılıp paylaşılmayacağını belirlemekte kullanılır. Bu değer 0 olarak geçilirse böyle bir paylaşım söz konusu olmaz. Eğer bu parametre PTHREAD_PROCES_SHARED biçiminde geçilirse processler arası paylaşım gerçekleşir. sem_init fonksiyonunun value parametresi (üçüncü parametre) semaphore sayacının başlangıç değeridir.

sem_t genellikle bir yapı biçiminde olan türdür. Sistemin tipik kullanımı şöyledir:

```
sem_t *sem;

sem_init(sem, 0, 2); // Kritik koda aynı anda iki thread girebilecektir..
...
...
sem_wait(sem);
...
...
sem_post(sem);
...
...
sem_destroy(sem);
```

Semaphore'un sayaç değeri istenildiği zaman sem_getvalue fonksiyonuyla alınabilir.

```
int sem_getvalue(sem_t *sem, int *sem_value);
```

İsimli Semaphore Kullanımı

İsimli semaphore kullanımı tamamen isimsizlerde olduğu gibidir. Ancak sem_init yerine semaphore yaratmak için sem_open fonksiyonu kullanılır.

```
sem_t *sem_open(const char *name, int, ...);
```

Fonksiyonun birinci parametresi semaphore nesnesinin ismidir. İkinci parametre açış modudur. Bu parametre O_CREAT ya da O_CREAT | O_EXCL olabilir. İkinci parametre 0 geçilebilir. Fonksiyonun üçüncü parametresi semaphore yeni yaratılmışsa erişim haklarını belirtir. Semaphore yeni yaratılıyorsa fonksiyonun dördüncü parametresi olarak sayaç değeri verilebilir. Fonksiyon başarılıysa semaphore nesnesinin ID değerine, başarısızsa –1 değerine geri döner. Semaphore'u silmek için sem_unlink fonksiyonu kullanılabilir.

```
sem_t *sem_unlink(const char *name);
```

Semaphore Nesneleri ile Mutex Nesneleri Arasındaki Benzerlikler ve Farklılıklar

Mutex nesnesini her zaman lock yapan thread unlock yapabilir (Win32 sistemlerinde bir sayaç olduğu için aynı thread mutex nesnesini birden fazla kez ele geçirebilir). Oysa semaphore'a giren bir thread sem_post fonksiyonunu uygulamak zorunda değildir. Yani bir thread sem_wait uygularken başka bir thread sem_post uygulayabilir. Hem mutex hem de semaphore kritik kod oluşturmakta kullanılabilmektedir. Ancak semaphore kritik koda birden fazla kez giriş sağlayabilir.

Semaphore'lar tipik olarak üretici-tüketici (producer-consumer) tarzı problemler için kullanılırlar. Örneğin üretici-tüketici problemi bir üreticinin senkronizasyon eşliğinde bir yere bir şey koyması ve tüketicinin de bunu alması biçiminde bir problemdir. Tipik olarak iki thread tarafından paylaşılan bir alana bir thread'in bir bilgiyi yazması diğer thread'in de bilgi yazıldıktan sonra bu bilgiyi alması uygulaması böyledir. Eğer bu işlem aynı process'in thread'leri arasında yapılacaksa paylaşılan alan global bir nesne olabilir, semaphore'lar da isimsiz açılabilir. Eğer farklı process'lerin thread'leri arasında işlem yapılacaksa bu durumda paylaşılan alan (shared memory) biçiminde oluşturulmalı, semaphore da isimli olmalıdır. Bu problemin tipik algoritması şöyledir:

```
sem_t g_sem1, g_sem2;

sem_init(&g_sem1, 0, 1);
/* yazan threadin sayacı başlangıçta 1  */
sem_init(&g_sem2, 0, 0);
/* okuyan threadin sayacı başlangıçta 0 */

/* thread1 (yazan thread) */

for (;;) {
      sem_wait(&g_sem1);
      ...
      sem_post(&g_sem2);
```

```
}

/* thread2 (okuyan thread) */

for (;;) {
      sem_wait(&g_sem2);
      ...
      ...
      sem_post(&g_sem1);
}
```

/* sem1.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>
#include <semaphore.h>

#define SHMKEY        123456

int main(void)
{
      sem_t *sem1;
      sem_t *sem2;
      int shmid, i;
      int *pi;

      if ((sem1 = sem_open("/home/student/sema1",
                                    O_CREAT, 0666, 1)) == (sem_t *) -1) {
            fprintf(stderr, "Cannot create semaphore!..\n");
            exit(1);
      }

      if ((sem2 = sem_open("/home/student/sema2",
                                    O_CREAT, 0666, 1)) == (sem_t *) -1) {
            fprintf(stderr, "Cannot create semaphore!..\n");
            exit(1);
      }
      if ((shmid = shmget(SHMKEY, sizeof(int), IPC_CREAT|0666)) == -1) {
            perror("shmget");
            exit(1);
      }
      pi = (int *) shmat(shmid, NULL, 0);
      if (pi == (void *) -1) {
            perror("shmat");
            exit(1);
      }

      for (i = 0; i < 100; ++i) {
            sem_wait(sem1);
            *pi = i;
            sem_post(sem2);
            if (i == 10)
                  break;
      }
      shmdt(pi);
```

```
      return 0;
}
```

/* sem2.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>
#include <semaphore.h>

#define SHMKEY        123456

int main(void)
{
      sem_t *sem1;
      sem_t *sem2;
      int shmid, i;
      int *pi;

      if ((sem1 = sem_open("/home/student/sema1",
                           O_CREAT|O_RDWR, 0666, 1)) == (sem_t *) -1) {
            fprintf(stderr, "Cannot create semaphore!..\n");
            exit(1);
      }

      if ((sem2 = sem_open("/home/student/sema2",
                           O_CREAT|O_RDWR, 0666, 1)) == (sem_t *) -1) {
            fprintf(stderr, "Cannot create semaphore!..\n");
            exit(1);
      }
      if ((shmid = shmget(SHMKEY, sizeof(int), IPC_CREAT|0666)) == -1) {
            perror("shmget");
            exit(1);
      }
      pi = (int *) shmat(shmid, NULL, 0);
      if (pi == (void *) -1) {
            perror("shmat");
            exit(1);
      }

      for (i = 0; i < 100; ++i) {
            sem_wait(sem2);
            if (*pi == 10)
                  break;
            printf("%d\n", *pi);
            sem_post(sem1);
      }
      sem_close(sem1);
      sem_close(sem2);

      return 0;
}
```

/* sema.c */

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

```
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>
#include <semaphore.h>

#define SIZE      10

int g_x;
sem_t g_sem1, g_sem2;

void *ThreadFunc(void *parg)
{
      for (;;) {
            sem_wait(&g_sem2);
            if (g_x == 10)
                  break;
            printf("%d\n", g_x);
            sem_post(&g_sem1);
      }
      return NULL;
}

int main(void)
{
      pthread_t tid;
      int i;
      void *pReturn;
      int result;

      if (sem_init(&g_sem1, 0, 1) == -1) {
            fprintf(stderr, "Cannot create semaphore!..\n");
            exit(1);
      }

      if (sem_init(&g_sem2, 0, 0) == -1) {
            fprintf(stderr, "Cannot create semaphore!..\n");
            exit(1);
      }

      if ((result = pthread_create(&tid, NULL, ThreadFunc, NULL)) != 0) {
            fprintf(stderr, "%s\n", strerror(result));
            exit(1);
      }

      for (i = 0; i < 100; ++i) {
            sem_wait(&g_sem1);
            g_x = i;
            sem_post(&g_sem2);
            if (i == 10)
                  break;
      }

      if ((result = pthread_join(tid, NULL)) != 0) {
            fprintf(stderr, "%s\n", strerror(result));
            exit(1);
      }
      return 0;
}
```

poo

Blokesiz I/O İşlemleri

Normal olarak dosya açılırken O_NONBLOCK özelliği kullanılmamışsa dosya blokeli modda açılır. Blokeli I/O işlemlerinde read ve write fonksiyonları belirtilen miktarda byte'ın tamamı yazılana kadar ya da okunana kadar process çizelge dışına çıkarılarak bloke edilir. Eğer dosya disk üzerinde açılmışsa modern sistemlerde bu işlemler için geniş cache alanları kullanıldığı için genellikle bir bloke durumu oluşmaz. Ancak donanım aygıtlarından okumalar sırasında, pipe'lara okuma yazma sırasında, mesaj kuyruklarına okuma yazma sırasında bloke durumunun oluşmasına sıklıkla rastlanır. Blokeli çalışma bazı durumlarda problemli olabilmektedir.

1) Yavaş aygıtlardan okuma yazma yapıldığı zaman aynı zamanda arka planda başka işlemler de yapılacaksa bloke durumu istenmez.
2) Birden fazla yavaş aygıttan okuma yapılmak istensin. Bu durumda blokeli çalışma problemli olur, çünkü bir betimleyicide process bloke edilebilir, ancak diğerlerine bilgi gelmiş olabilir. Halbuki istenen şey hangi betimleyiciye bilgi gelirse onu işleme sokmaktır.

Blokesiz moda geçebilmek için `open` fonksiyonunda `O_NONBLOCK` kullanılır. SystemV IPC mekanizmasında `msgrcv` ve `msgsnd` fonksiyonlarında `IPC_NOWAIT` parametresi kullanılırsa mesaj kuyruğu blokesiz moda geçirilebilir. Benzer biçimde soketler de blokesiz çalışabilmektedir.

Blokesiz modda read fonksiyonu hiç bekleme yapmadan o anda okuyabildiği kadar bilgiyi okur. Örneğin biz 100 byte okumak isteyelim; read fonksiyonu hazırda 10 byte varsa bu 10 byte'ı okur ve geri döner. Eğer read fonksiyonu o anda hiçbir okuyacak bilgiye sahip değilse başarısızlık anlamında –1 değerine geri döner ve errno EAGAIN olarak set edilir. read fonksiyonu 0 değerine geri döndüyse bu durum EOF anlamına gelir.

Blokesiz olarak SIZE uzunlukta bir bilgi bir döngü içerisinde okunarak bir diziye yerleştirilecek olsun. Bu işlem şöyle yapılabilir:

```
char buf[SIZE];
int nleft, nindex, nread;

nleft = SIZE;
nindex = 0;
while (nleft >= 0) {
      nread = read(fd, buf + nindex, nleft);
      if (nread == -1) {
            if (errno == EAGAIN)
                  continue;
            else {
                  perror("read");
                  exit(1);
            }
      }
      nleft -= nread;
      nindex += nread;
}
```

Birden fazla betimleyiciden okuma yapılması gerektiğinde yukarıdaki algoritma nasıl düzenlenmelidir?

```
typedef struct _MSG {
      int fd;
      int nindex;
      int nleft;
      char buf[SIZE];
} MSG;

MSG msg[10];
Initiaize(....);  /* tüm dizideki elemanlar initialize edilir... */

for (;;) {
      for (i = 0; i < 10; ++i) {
            nread = read(msg[i].fd, msg[i].buf + msg[i].nindex,
                                                      msg[i].nleft);
            if (nread == -1) {
                  if (errno == EAGAIN)
                        continue;
                  else {
                        perror("read");
                        exit(1);
                  }
            }
            msg[i].nleft -= nread;
            msg[i].nindex += nread;
            if (msg[i].nleft == 0) {
                  Proc(&msg[i]);
                  Initialize(....); // ilgili eleman initialize edilir...
            }
      }
}
```

Yukarıdaki algoritmalar blokesiz okumada kullanılan klasik algoritmalardır. Ancak blokesiz okumada küçük bir problem daha vardır. Yukarıdaki gibi bir döngüde eğer aygıttan bilgi çok yavaş geliyorsa gereksiz yinelemeler oluşacaktır. Bu da gereksiz bir CPU zamanı harcanmasına neden olur. Bu durumda en iyi seçenek process'in herhangi bir betimleyiciye bilgi gelene kadar bloke edilmesini sağlamaktır. İşte `select` fonksiyonu böyle bir işlem yapmaktadır. Select fonksiyonuna bir grup betimleyici verilir. select fonksiyonu bunlardan herhangi birinde okuma ya da yazma oluşana kadar process'i bloke eder. select fonksiyonu yalnızca bu olayların gerçekleştiğini belirtmektedir. Ancak kaç byte'lık bir okuma ya da yazma olduğunu vermemektedir.

select Fonksiyonu

select fonksiyonu bir POSIX fonksiyonu olmamakla birlikte POSIX'in son eklemelerine yerleştirileceği düşünülmektedir. select fonksiyonunun benzeri BSD sistemlerinde `poll` fonksiyonu biçiminde kullanılmaktadır. Bu iki fonksiyon da modern Unix sistemlerinin hemen hepsi tarafından desteklenmektedir. Ancak select fonksiyonu çok daha yaygın kullanıma sahiptir.

```
int select(int maxfd, fd_set *readfds, fd_set *writefds,
      fd_set *exceptfds, struct timeval *tv);
```

fd_set türü bitsel bir yapıya sahip olan bir türü temsil eder. Bu tür genellikle bir yapı biçimindedir. Fonksiyonun fd_set türünden gösterici parametrelerine NULL geçilebilir. Bu durum bu parametrelerle ilgilenilmediği anlamına gelir. fd_set bir betimleyici kümesi belirtir. fd_set sistemden sisteme değişebilecek bir türü belirttiğine göre bu kümeye betimleyici eklemek ya da bu kümeden betimleyici çıkartmak için özel makrolar kullanılmalıdır. fd_set genellikle long türden bir dizi elemanına sahip bir yapı biçimindedir. İlgili betimleyici kümeye dahilse ilgili bit 1, değilse 0 biçimindedir. fd_set için kullanılan makrolar şunlardır:

```
FD_ZERO(fd_set *fdset);
FD_SET(int fd, fd_set *fdset);
FD_CLR(int fd, fd_set *fdset);
FD_ISSET(int fd, fd_set *fdset);
```

FD_ZERO tüm kümeyi sıfırlar. FD_SET belirli bir betimleyiciyi kümeye ekler. FD_CLR belirli bir betimleyiciyi kümeden çıkartır. FD_ISSET ise belirli bir betimleyicinin kümeye dahil olup olmadığına bakar. select fonksiyonu okuma, yazma ve istisna durumları kontrol eder. Genellikle programcılar tek bir kümeyi belirleyip diğerlerini NULL olarak geçerler. Fonksiyon verilen kümeler içerisindeki herhangi bir betimleyici içerisinde okuma ya da yazma olayı gerçekleştiğinde sonlanır. select fonksiyonunun geri dönüş değeri üç biçimde olabilir.

1) Geri dönüş değeri –1 ise bu durum error anlamına gelir. Örneğin bir signal oluştuğunda fonksiyon –1 değeriyle geri döner.
2) Geri dönüş değeri 0 ise herhangi bir betimleyicide istenilen olay gerçekleşmemiştir. Fonksiyon timeout nedeniyle çıkmıştır.
3) Fonksiyonun geri dönüş değeri 0'dan büyükse bu değer tüm kümelerdeki gerçekleşen toplam olay sayısını belirtir.

select fonksiyonunun birinci parametresi işletim sisteminin işini kolaylaştırmak için düşünülmüştür. Bu parametre tüm kümelerdeki arama için yapılacak maksimum betimleyici sayısını belirtir. Örneğin birinci parametreyi 10 olarak girsek işletim sistemi fd_set kümelerinin yalnızca ilk 10 slotu için araştırma yapar (yani 0-9 arası). Normal olarak bu sayının toplam kümelere dahil edilmiş olan en büyük betimleyici numarasının bir fazlası biçiminde verilmesi uygundur. Fonksiyonun son parametresi timeval türünden bir yapı değişkeninin adresini alır.

```
struct timeval {
      long tv_sec;
      long tv_usec;
};
```

Programcı son parametreyi NULL geçerse zaman aşımı vermemiş olur. Yapının tv_sec ve tv_usec elemanlarının her ikisi de 0 olarak girilirse select fonksiyonu hiç bekleme yapmadan olayın gerçekleştiği betimleyicileri rapor eder. select fonksiyonu başarılı bir biçimde geri döndüğünde fd_set kümeleri yalnızca olayın gerçekleştiği betimleyicileri içerecek hale getirilmiş olur. Yani bu durumda fonksiyon çıkışında FD_ISSET uygulanarak ilgilenilen betimleyicide olayın gerçekleşip gerçekleşmediği sorgulanabilir. Aşağıda select fonksiyonu kullanılarak bir döngü içersinde etkin bekleme sağlayan kalıp verilmiştir.

```
int fd1, fd2, maxfd;
fd_set rdset;
```

```
for (;;) {
      FD_ZERO(&rdset);
      FD_SET(fd1, &rdset);
      FD_SET(fd2, &rdset);
      maxfd = MAX(fd1, fd2);
      if (select(maxfd + 1, &rdset, NULL, NULL, NULL) == -1) {
            perror(“select”);
            exit(1);
      }
      if (FD_ISSET(fd1, &rdset)) {
            ...
      }
      if (FD_ISSET(fd2, &rdset)) {
            ...
      }
}
```

Bu döngü sırasında bazı betimleyiciler zamanla kapanabilir. Kapanan betimleyicilerin kümeden çıkartılması kolay değildir. Zaten kümeden çıkartılmalarına da gerek yoktur, nasıl olsa kapanan betimleyicilere ilişkin olaylar hiç gerçekleşmeyecektir. FD_ZERO taşınabilirlik açısından döngü içerisinde tutulmalıdır. Çünkü tüm betimleyicilerin sıfırlanması işleminde fd_set içerisindeki başka elemanlar çeşitli değerlerle set ediliyor olabilir. select fonksiyonundan çıkıldığında yalnızca olay gerçekleşen betimleyiciler fonksiyon tarafından kümede bırakılır diğerleri küme dışına alınır.

/* select.c */

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

#define MAX(a, b) ((a) > (b) ? (a) : (b))

#define PIPE_BUF  10

typedef struct _PIPEREC {
      int nleft;
      int index;
      char buf[PIPE_BUF + 1];
} PIPEREC;

void process(int pipeno, char *str)
{
      str[PIPE_BUF] = '\0';
      printf("pipe %d: %s\n",  pipeno, str);
}

int main(void)
{
      int pipe1, pipe2, maxfd, n;
      fd_set rdset;
      PIPEREC prec1 = {PIPE_BUF, 0 }, prec2 = {PIPE_BUF, 0};

      if ((pipe1 = open("pipe1", O_RDONLY|O_NONBLOCK)) == -1) {
            perror("pipe1");
            exit(1);
```

```
        }
        if ((pipe2 = open("pipe2", O_RDONLY|O_NONBLOCK)) == -1) {
                perror("pipe1");
                exit(1);
        }
        maxfd = MAX(pipe1, pipe2);

        for (;;) {
                FD_ZERO(&rdset);
                FD_SET(pipe1, &rdset);
                FD_SET(pipe2, &rdset);

                if (select(maxfd + 1, &rdset, NULL, NULL, NULL) == -1) {
                        perror("select");
                        exit(1);
                }
                if (FD_ISSET(pipe1, &rdset)) {
                        n = read(pipe1, prec1.buf + prec1.index, prec1.nleft);
                        if (n == -1) {
                                perror("read");
                                exit(1);
                        }
                        if (n == 0) {
                                prec1.nleft = PIPE_BUF;
                                prec1.index = 0;
                        }
                        prec1.nleft -= n;
                        prec1.index += n;
                        if (prec1.nleft == 0) {
                                process(1, prec1.buf);
                                prec1.index = 0;
                                prec1.nleft = PIPE_BUF;
                        }
                }

                if (FD_ISSET(pipe2, &rdset)) {
                        n = read(pipe2, prec2.buf + prec2.index, prec2.nleft);
                        if (n == -1) {
                                perror("read");
                                exit(1);
                        }
                        if (n == 0) {
                                prec2.nleft = PIPE_BUF;
                                prec2.index = 0;
                        }
                        prec2.nleft -= n;
                        prec2.index += n;
                        if (prec2.nleft == 0) {
                                process(2, prec2.buf);
                                prec2.index = 0;
                                prec1.nleft = PIPE_BUF;
                        }
                }
        }
        close(pipe1);
        close(pipe2);
}
```

Unix/Linux Sistemlerinde Aygıt Dosyaları ve Aygıt Sürücüler

Aygıt sürücüler Win32 ve Unix/Linux sistemlerinde çekirdek modunda çalışan özel programlardır. Aygıt sürücüler koruma mekanizmasından etkilenmeyecek serbestlikte çalışırlar. Örneğin haberleşme portlarına istedikleri gibi erişebilirler, bütün öncelikli makine komutlarını kullanabilirler. Genellikle aygıt sürücüler bir donanım biriminin yazılımcılar tarafından kullanılmasını sağlamak amacıyla kullanılırlar. Aygıt sürücüler pek çok işletim sisteminde (Örneğin Win32 ve Unix/Linux sistemlerinde) user mode programlar tarafından bir dosya gibi kullanılırlar. Yani aygıt sürücüler aslında adeta çekirdeğin bir parçası gibi çalışırlar ve normal programlara hizmet verirler. Bu sistemlerde user mode programlar aygıt sürücüleri bir dosya gibi açarlar. Dosyadan okuma yazma yapan fonksiyonları kullanarak veri transferini gerçekleştirirler. Tabii aygıt sürücüler yalnızca veri transferi için kullanılmazlar. Örneğin user mode programlar aygıt sürücüleri kullanarak belirli işlemleri aygıt sürücülere yaptırmak isterler. Genellikle bu işlemler için özel fonksiyonlar kullanılır.

Unix/Linux sistemlerin `ioctl` fonksiyonu, Win32 sistemlerinde DeviceIOControl fonksiyonu aygıt sürücülerle iletişim kurarak onların çeşitli işlemleri yapmasını sağlarlar. Örneğin CD-ROM aygıtını düşünelim. Bu aygıt bir aygıt sürücü tarafından kontrol edilir. Biz Unix/Linux sistemlerinde CD-ROM üzerinde işlem yapacaksak önce o aygıt sürücüyü dosya gibi açarak bu işlemden bir betimleyici elde etmeliyiz. Artık read fonksiyonu ile CD-ROM'daki bilgileri okuyabiliriz. Ancak CD-ROM yalnızca okunan bir aygıt değildir. Örneğin biz programımızdan CD-ROM'u eject etmek isteyebiliriz. Bu tür işlemler için ilgili betimleyici ile ioctl fonksiyonundan faydalanırız. Unix/Linux sistemlerindeki ioctl ya da Win32 sistemlerindeki DeviceIOControl fonksiyonu sürücü programındaki istenilen fonksiyonları çalıştırmak için kullanılır. Bu modelde sürücü programını oluşturan fonksiyonların birer numarası vardır. Programcı bu fonksiyonlarda numarayı belirterek sürücü fonksiyonunu çağırabilir. Şüphesiz sürücü fonksiyonlarının giriş ve çıkış parametreleri de olacaktır. ioctl fonksiyonunun prototipi şöyledir.

```
int ioctl(int d, int request, ...);
```

Fonksiyonun birinci parametresi aygıt sürücüye ilişkin betimleyici, ikinci parametresi işlem kodudur. Bu parametre aygıt sürücünün hangi fonksiyonunun çağrılacağını belirtir. ioctl fonksiyonun geri dönüş değeri başarı durumunda 0, başarısızlık durumunda –1 değeridir. Standart aygıt sürücülerine ilişkin işlem kodları `<sys/ioctl.h>` içerisinde sembolik sabitler biçiminde tanımlanmıştır. ioctl fonksiyonunun diğer parametreleri aygıt sürücüye bağlı olarak değişebilmektedir. Örneğin CD-ROM sürücüyü eject etmek isteyelim;

```
int fd;

fd = open("/dev/hdc", O_RDONLY);
...
ioctl(fd, CDROMEJECT);
...
close(fd);
```

Şüphesiz aygıt sürücüler sistem yöneticisinin nezaretinde sisteme install edilmelidir. Bir aygıt sürücüsünü kullanmak doğrudan ioctl ya da DeviceIOControl fonksiyonlarıyla yapılmayabilir. Örneğin sürücüyü yazan programcı çeşitli işlemler için user mode kütüphane de oluşturmuş olabilir. Şüphesiz bu user mode kütüphane, işlemleri yapmak için ioctl ya da DeviceIOControl gibi ilişki kuran fonksiyonları kullanmaktadır.

Aygıt sürücülerin yazım biçimi işletim sisteminden işletim sistemine değişmektedir. Aygıt sürücü programlar birden çok uygulama programına hizmet verebilecek biçimde yazılmalıdır. Aygıt sürücü programlar bu nedenle kuvvetli bir senkronizasyon içerirler. Aygıt sürücü programları yazarken bu programın çekirdeğin bir parçasıymış gibi yükleneceği düşünülmelidir. Yani bu programlar yalnızca çekirdek fonksiyonlarını kullanmalıdır. Örneğin, Linux çekirdeği içerisinde ekrana bir şeyler yazmak için kprintf isimli printf benzeri bir fonksiyon vardır. Ya da çekirdek alanından dinamik tahsisat yapmak için malloc fonksiyonu gibi bir kmalloc fonksiyonu vardır.

Aygıt sürücüler /dev dizininin altında dosyalar biçiminde bulunurlar. Aygıt sürücüler tamamen bir dosya gibi açılmaktadır. Her aygıtın bir numarası vardır. Aygıt numarası büyük ve küçük olmak üzere ikiye ayrılır. Büyük numara kategori belirtirken küçük numara tür belirtmektedir. Örneğin floppy sürücüsünün büyük aygıt numarası 2'dir. A sürücüsünün küçük aygıt numarası 0, B sürücüsünün ise 1'dir. Örneğin IDE sürücülerinin büyük aygıt numarası 3'tür. Birinci hard diskin tamamı /dev/hda ile temsil edilir. Bu aygıtın büyük numarası 3, küçük numarası 0'dır. Birinci hard diskteki partition'lar /dev/hda1, /dev/hda2 gibi giderler. Bunların küçük numaraları 1, 2 biçiminde gitmektedir. İkinci IDE hard diskin büyük numarası yine 3'tür ama küçük numarası 64'tür.

Terminaller ve seri portların büyük aygıt numaraları 4'tür. Birinci sanal terminalin küçük aygıt numarası 1, ikincisinin 2'dir ve böyle gider. Bunlara ilişkin aygıt dosyaları /dev/tty1, /dev/tty2 biçiminde gider. Bu dosyalar kullanılarak doğrudan klavyeden okuma ve ekrana yazma yapılabilir.

Şüphesiz aygıt dosyalarının açılarak kullanılabilmesi için root önceliğine sahip olunması gerekir. Bu dosyaların erişim hakları daha sonra değiştirilebilir.

COM portların büyük numarası 4'tür, küçük numaraları 64, 65 biçiminde gider. Sırasıyla /dev/st0, /dev/st1 gibi isimlerle kullanılırlar.

Paralel portlara /dev/lp0 veya /dev/par0 biçiminde erişilebilir. Paralel portların büyük aygıt numaraları 6'dır. LPT1 için küçük aygıt numarası 0, LPT2 için 1'dir.

Aygıtların numarası stat fonksiyonuyla alınabilir.

Linux'taki Özel Aygıt Dosyaları

/dev/null dosyasına yazılan her şey tamamen atılmaktadır. Yani bu gerçek bir dosya değildir. /dev/null dosyasından okuma yapıldığında her zaman EOF okunur.

/dev/zero dosyası içerisinde sonsuz tane 0 bulunan bir dosya gibi işlem görmektedir. Bu dosyaya yazma yapılamamaktadır.

/dev/full dosyası yazma yapmakta kullanılır. Bu dosyayı ne zaman yazma yapılmaya çalışılırsa "disk full" okunur. Programların tamamen disk dolduğundaki davranışlarını test etmek amacıyla kullanılır.

proc Dosya Sistemi

Unix sistemlerinin çekirdek özelliklerine ilişkin taşınabilir sistem fonksiyonları çok azdır. Örneğin process listesi alınacak olsa bunu yapmanın taşınabilir bir yolu yoktur. ps programı da bunu tamamen aşağı seviyeli çekirdek özellikleriyle yapmaktadır. Unix sistemlerindeki proc dosya sistemi /proc dizinine mount edilmiştir. Bu dosya sistemi gerçek bir dosya sistemi değildir. Bu dosya sistemi içerisindeki çeşitli dosyalar sistemin durumu hakkında bilgiler içerirler. Yani o dosyaları açıp onların içerisinden okuma yapmakla çekirdeğe ilişkin bilgiler edinilebilir. Proc dosya sistemi tüm Unix sistemlerinde olan bir sistem değildir. Ancak Unix sistemleri gittikçe daha fazla bu dosya sistemini desteklemektedir. Proc dosya sisteminin önemli elemanları şunlardır.

/proc/version: Linux sisteminin versiyon numarasını ve hangi gcc derleyicisiyle derlendiğini verir.

/proc/cpuinfo: Bu dosya çalışılan işlemci hakkında bilgiler verir.

/proc/filesystems: O anda register ettirilmiş olan dosya sistemlerini gösterir.

/proc/ioports: Kullanılan IO portlarının hangi işlemcilere tahsis edildiğini gösterir.

O anda çalışan tüm process'ler için process numarasına ilişkin dizinler yaratılmıştır. /proc/self dizini o anda çalışmakta olan process'e link edilmiştir. Process'leri belirten dizinler içerisinde standart dosyalar vardır. cmdline processin komut satırı argümanlarını belirtir.

```
/* dev.c */

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

#define SECT_SIZE       512

typedef unsigned char BYTE;

int main(int argc, char *argv[])
{
      int fd, i;
      BYTE buf[SECT_SIZE];

      if (argc != 3) {
            fprintf(stderr, "Wrong number of arguments!..\n");
            exit(1);
      }

      if ((fd = open(argv[1], O_RDWR)) == -1) {
            perror("open");
            exit(1);

      }

      if (read(fd, buf, SECT_SIZE) == -1) {
            perror("read");
            exit(1);
```

```
	}

	i = 0;
	do {
		putchar(buf[i]);
	} while (buf[i++] != '\n');

	close(fd);

	return 0;
}
```

```
/* sector.c */

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

#define SECT_SIZE	512

typedef unsigned char BYTE;

int main(int argc, char *argv[])
{
	int fd, i;
	BYTE buf[SECT_SIZE];

	if (argc != 3) {
		fprintf(stderr, "Wrong number of arguments!..\n");
		exit(1);
	}

	if ((fd = open(argv[1], O_RDONLY)) == -1) {
		perror("open");
		exit(1);
	}

	lseek(fd, atoi(argv[2]) * SECT_SIZE, SEEK_SET);

	if (read(fd, buf, SECT_SIZE) == -1) {
		perror("read");
		exit(1);
	}

	for (i = 0; i < 64; ++i)
		printf("%x%c", buf[i], (i % 16 == 15) ? '\n' : ' ');

	close(fd);

	return 0;
}
```

Karakter ve Blok Aygıtları

Unix sistemlerinde aygıtlar blok ve karakter aygıtları olmak üzere ikiye ayrılır. Blok aygıtları rastgele erişime izin veren aygıtlardır. Yani lseek fonksiyonu ile rastgele konumlandırma

yapılabilir. Halbuki karakter aygıtları byte byte okuma yapılabilen aygıtlardır. Tipik olarak disk sistemleri blok aygıtları, seri port, paralel port gibi aygıtlar da karakter aygıtlarıdır.

Curses Kütüphanesi

Unix sistemlerine çok çeşitli terminaller bağlanabilmektedir. Doğal olarak bu terminallerin görüntü kapasiteleri birbirlerinden farklıdır. Terminallerde bir ara birim oluşturmak amacıyla Escape kodlaması düşünülmüştür. Yani bir terminale önce 27 numaralı Escape karakteri gönderip, sonra bir sol köşeli parantez"[" ve bir takım karakterler verildiğinde bunları doğrudan görüntülemez. Bunlar yerine bir takım işlemler yapar. Bu escape karakterlerinin kullanımı ve anlamı terminaller arasında farklılık göstermektedir. Bazı terminaller ANSII uyumludur. ANSII'nin ayrı bir terminal kapasitesi vardır. Unix sistemlerinde genellikle sisteme bağlı bulunan terminaller hakkındaki genel escape sistemi ve diğer özellikler `/etc/termcap` dosyası içerisinde yer almaktadır. Curses kütüphanesi aslında arka planda bu termcap dosyasına bakarak uygun escape kodlamalarını oluşturmaktadır.

Curses İşlemlerinin Başlatılması ve Bitirilmesi

Curses kullanırken önce initscr isimli bir başlangıç fonksiyonu çağrılır. Prototipi;

```
WINDOW *initscr(void);
```

Bu fonksiyon bir takım ilk işlemleri yapar ve ekranın bütününe ilişkin pencereyi betimleyen bir handle değerine geri döner. Fonksiyon başarısızlık durumunda NULL değerine geri döner. İşlemler bitirildiğinde programcı endwin fonksiyonunu çağırmalıdır. Fonksiyonun prototipi;

```
int endwin(void);
```

curses kütüphanesinin kullanılabilmesi için curses.h dosyasının include edilmesi gerekir. Bütün curses fonksiyonları özel bir kütüphane içerisindedir. Bu kütüphane normal curses için libcurses.a ve eski curses için libncurses.a dir.

```
/*cur.c*/

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>

int main(void)
{
      WINDOW *win;

      if ((win = initscr()) == NULL) {
            fprintf(stderr, "Can not initialize curses..\n");
            exit(1);
      }
      return 0;
}
```

Curses Fonksiyonlarını İsimlendirme Biçimi

Önek almayan curses fonksiyonları o andaki aktif pencere üzerinde işlem yaparlar. O andaki aktif pencere `stdscr` değişkeni tarafından belirlenmektedir. "w" öneki alan fonksiyonlar birinci parametre olarak pencere handle değerini alırlar ve o pencereye dayalı işlem yaparlar. "mv" öneki alan fonksiyonlar koordinat parametresi alıp önce curser'ı o koordinata çeker, sonra işlemi yaparlar. mv ve w önekleri birlikte mvw olarak kullanılabilmektedir. Genel olarak curses fonksiyonlarının çoğunun geri dönüş değerleri int türündendir. Bu fonksiyonlar başarısızlık durumunda ERR değerine, başarı durumunda ise OK değerine geri dönerler.

Curser Konumlandıran Fonksiyonlar

Bu fonksiyonlar move ve wmove fonksiyonlarıdır. Fonksiyonların prototipleri;

```
int move(int x, int y);
int wmove(WINDOW *win, int x, int y);
```

curses fonksiyonlarında ekran koordinatları önce satır sonra sütün parametreleri ile belirlenir.

```
/*curses.c*/

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>

int main(void)
{
      WINDOW *win;

      if ((win = initscr()) == NULL) {
            fprintf(stderr, "Can not initialize curses..\n");
            exit(1);
      }

      move(10, 10);
      getch();
      wmove(win, 0, 0);
      getch();
      endwin();
      return 0;
}
```

Klavyeden Karakter Alan Fonksiyonlar

Klavyeden tuş alma ile ilgili fonksiyonlar şunlardır; bu fonksiyonlar <curses.h> içerisindedir:

```
int getch(void);
int wgetch(WINDOW *win);
int mvgetch(int x, int y);
int mvwgetch(WINDOW *win, int x, int y);
int ungetch(int ch);
int has_key(int ch);
```

Bu fonksiyonlar default olarak enter tuşuna gereksinim duymazlar. Eğer `nocbreak` fonksiyonu çağrılır ise yeniden `cbreak` fonksiyonu çağrılana kadar enter tuşuna gereksinim duyulur. Bu fonksiyonlar default olarak basılan tuşu görüntülemektedir. Eğer `echo` fonksiyonu çağrılır ise bir daha `noecho` fonksiyonu çağrılana kadar bu durum ortadan kaldırılır.

```
/*cursesecho.c*/

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>

int main(void)
{
      WINDOW *win;
      if ((win = initscr()) == NULL) {
            fprintf(stderr, "Can not initialize curses..\n");
            exit(1);
      }
      cbreak();
      noecho();
      move(10, 10);
      getch();
      echo();
      wmove(win, 0, 0);
      getch();
      endwin();
      return 0;
}
```

Bu fonksiyonları geri dönüş değerleri özel tuşlar için bazı sembolik sabitler ile ifade edilebilmektedir. Örneğin; KEY_F0 ya da KEY_HOME, KEY_ENTER gibi.basılan tuş normal bir tuş ise; düşük anlamlı byte'inda ASCII kodu bulunmaktadır.

```
/*cursesKEY.c*/

#include <stdio.h>
#include <stdlib.h>
#include <curses.h>


int main(void)
{
      WINDOW *win;

      if ((win = initscr()) == NULL) {
            fprintf(stderr, "Can not initialize curses..\n");
            exit(1);
      }
      move(10, 10);
      ch = getch();
      if (ch == KEY_F(1))
            printw("F1 key press..\n");
      else if (ch == KEY_F(2))
            printw("F2 key press..\n");
      else if (ch == KEY_HOME)
            printw("Home key press..\n");
      else
            putchar(ch);
```

```
	refresh();  // Önemli bir fonksiyon

	endwin();
	return 0;
}
```

Görüntünün Tazelenmesi ve refresh Fonksiyonu

Curses modelinde ekrana yazma yapan fonksiyonlar doğrudan görüntüyü ekranda oluşturmazlar. Görüntü önce bellekte bir tamponda oluşturulur, refresh fonksiyonu çağrılınca ekrana aktarılır. wrefresh fonksiyonu ise sadece bir pencereye özgü bu işi yapmaktadır. Klavyeden giriş alan fonksiyonlar da kendi içerisinde refresh yapmaktadır.

Formatlı Yazdırma İşlemleri

printw printf fonksiyonunun curses versiyonudur.
`printw` cursor'ın bulunduğu yere yazar.
`wprintw` belli bir pencere için bu işlemi yapar.
`mvprintw` cursor'ı taşıyarak yazma işlemini yapar.
`mvwprintw` fonksiyonu özel bir pencere için cursor'ı taşıyarak yazma işlemi yapar.

Özellikli Yazdırma İşlemleri

Normal olarak yazdırma işleminde renk ya da özellik kullanılmaz. Ancak programcı isterse yazdırma işlemini özellikli ya da renkli yapabilir. Genel olarak `attron` fonksiyonuyla bir özellik set edilir ve o özellik `attroff` ile reset edilene kadar geçerli olur. attron ve attroff fonksiyonlarının `wattron` ve `wattroff` biçimleri de vardır. Örneğin

```
attron(A_BOLD);
printw("bold");
attroff(A_BOLD);
printw("not bold");
refresh();
```

Renksiz özellikler şunlardır:

```
A_NORMAL
A_UNDERLINE
A_REVERSE
A_BLINK
A_BOLD
```

attron ve attroff belirli bir özelliği ekleyip çıkartmak için kullanılır. Özelliği tümden değiştirmek için `attrset` yapmak gerekir.

Renkli Yazım

Bilindiği gibi bir terminal renkli olmak zorunda değildir ve eski terminallerin çoğu renksizdir. Bu nedenle programcı önce has_colors fonksiyonu ile renk testi yapmalıdır. Örneğin;

```
if (has_colors() == FALSE) {
      endwin();
      exit(1);
}
```

Monitörün renkli olduğu tespit edildikten sonra `start_color` fonksiyonuyla temel başlangıç işlemleri yapılmalıdır. Bundan sonra `init_pair` fonksiyonuyla şekil zemin renkleri set edilir. Sonra da attron gibi fonksiyonlarla `COLOR_PAIR` makrosu kullanılarak renkli yazım moduna geçilir.

```
int init_pair(short pair, short f, short b);
```

Fonksiyonun birinci parametresi rengin sayısal değeri, ikinci ve üçüncü parametreleri sırasıyla şekil ve zemin renkleridir. Örneğin;

```
init_pair(1, COLOR_RED, COLOR_BLACK);
attron(COLOR_PAIR(1));
```

Formatsız Yazdırma Yapan Fonksiyonlar

Formatsız yazdırma yapan fonksiyonlar add ile isimlendirilmişlerse ekleme yaparlar, ins ile isimlendirilmişlerse insert ederek yazarlar. Örneğin addch tek bir karakteri ezerek yazar. Burada karakterin yanı sıra özellikler ile or'lama yapılabilir.

```
int addch(chtype ch);
```

Örneğin;

```
addch(ch | A_BOLD);
```

Şu versiyonları da vardır:

```
waddch
mvaddch
mvwaddch
```

Tek karakter yazan fonksiyonların insert versiyonları şunlardır:

```
insch
winsch
mvinsch
mvwinsch
```

Çerçeve Çizmek

Programcı box fonksiyonu ile çerçeve çizebilir. Ancak bu fonksiyonu kullanmak yerine ayrı çerçeve fonksiyonları da yazılabilir.

```
int box(WINDOW *win, chtype verch, chtype horch);
```

box fonksiyonu yalnızca bir pencerenin çerçevesini çizer. Normal çerçeveler için `border` fonksiyonu kullanılabilir.

Yeni bir pencere `newwin` fonksiyonu ile, bir pencere içerisinde başka bir pencere `subwin` fonksiyonuyla açılır. scanf fonksiyonuna benzer `scanw` fonksiyonu vardır. Ayrıca curses bulunan sistemlerde menu, form ve panel kütüphaneleri de bulunmaktadır. Form kütüphanesi ekrandan string alan fonksiyonları barındırır. menu kütüphanesi menü açmaya yarar. Bunların dışında ncurses fare kullanımını da desteklemektedir. getch fonksiyonu aynı zamanda fareye basılıp basılmadığını da tespit edebilir. Eğer getch fonksiyonunun geri dönüş değeri `KEY_MOUSE` ise tuşa basılmıştır, programcı da `getmouse` fonksiyonu ile farenin pozisyonunu alabilir.

Linux'ta Sembolik Makine Dili ile Programlama

Linux'un sembolik makine dili programlama dili modeli tamamen DOS'ta olduğu gibidir. Sistem fonksiyonlarının ana girişi 80h kesmesidir. 80h kesmesine yerleştirilen tuzak kapısı yoluyla process'in akışı kernel moda geçer ve sistem fonksiyonları çalıştırılır. 80h kesmesi çağrılırken eax yazmacına fonksiyon numarası yerleştirilir. Sonra sistem fonksiyonlarının parametreleri sırasıyla ebx, ecx, edx, esi, edi yazmaçlarına yerleştirilir. Örneğin exit fonksiyonu çağrılıyor olsun. Bu fonksiyonun numarası 1'dir. Bu sayı eax yazmacına yerleştirilir. exit fonksiyonunun tek bir parametresi olduğuna göre bu parametre de ebx yazmacına yerleştirilecektir. Derleme ve link işlemi şöyle yapılabilir:

```
nasm –f elf x.asm ß
gcc –o x x.o ß
```

(nasm ve ndisasm Linux sistemlerinde `binutil` paketi içerisindedir.)

```
; sample.asm
; Sample hello world program

[BITS 32]

[SECTION .text]

GLOBAL _start

_start:
      mov   eax,  4           ; write syscall number
      mov   ebx,  1           ; stdout
      mov   ecx,  message     ; buffer
      mov   edx,  16          ; number of characters to write
      int   80h               ; system call


      mov   eax,  1           ; exit syscall number
      mov   ebx,  0           ; exit code
      int   80h               ; system call

[SECTION .data]

message     db    "hello world...", 10

; compile: nasm -f elf sample.asm
```

```
; link: ld -o sample -e _start sample.o
```

Linux Çekirdeğinin Derlenmesi

Linux açık bir sistemdir. Sistem programcıları çekirdek üzerinde çeşitli değişiklikler yaparak işletim sistemini yeniden derleyebilirler. Örneğin tipik olarak sisteme yeni bir sistem fonksiyonu eklenebilir ve bu durumda çekirdek yeniden derlenmek istenebilir. Çekirdeği derlemenin adımları şöyledir:

1) Çekirdek kaynak kodlarının bulunduğundan emin olunur.

2) /usr/src/linux dizinine geçilir.

3) Çekirdek parametrelerine ilişkin belirlemeler yapılır. Bunun için

```
make config
```

ya da

```
make menuconfig
```

yapılır (root olmak gerekir).

4) Bundan sonra dependency durumu için

```
make dep
```

yapılabilir ya da yapılmadan geçilebilir.

5) Zorunlu olmamakla birlikte daha önceki derlemeden kalan dosyalar

```
make clean
```

ile silinebilir.

6) Nihayet asıl kernel derlemesi başlatılır. En fazla zaman alacak işlem bu işlemdir. Bunun için `/usr/src/linux/makefile` dosyası kullanılır. Bu makefile dosyası içerisinde çekirdeği oluşturan bütün dosyaların derlenme ve link edilme bilgileri vardır. Eski bir dosya üzerinde değişiklik yapılmışsa bu dosyada değişiklik yapılmaya gerek yoktur. Çekirdek derlendiğinde zaten o değişiklik devreye girer. Ancak yeni bir dosyayı da çekirdek derlemesi içerisine alacaksak bunu bu dosyada belirlemeliyiz. Kernel derlemesi için

```
make zImage
```

yapılır. Burada değişen dosyalar yeniden derlenecektir ve sonra hep beraber link edilerek yeni çekirdek dosyası oluşturulacaktır.

7) Kernel image dosyasının oluşturulması

```
make zImage
```

ile bitirilir. image dosya `/usr/src/linux/arch/i386/boot/zImage` klasörüne kopyalanır. Artık bu kernel image dosyasının boot işlemine sokulması aşamasına gelinmiştir.

8) Oluşturulan kernel dosyasının boot işleminde devreye girmesi için lilo bootloader programından faydalanılabilir. lilo programı `lilo.conf` isimli bir dosyaya bakarak ne yapacağını tespit eder. lilo.conf içerisinde sistemde hangi işletim sistemlerinin olduğu ve Linux'un hangi kernel image dosyasıyla boot edileceği gibi bilgiler vardır. Bu değişiklikler yapıldıktan sonra lilo programı çalıştırılarak değişiklikler gerçekleştirilir. lilo.conf dosyasında kernel image ismi ve yeri belirtilmektedir. Çekirdeği derleyen kişi yeni yarattığı kernel image dosyasını burada belirtilen yere aynı isimle kopyalayabilir ya da lilo.conf içerisinde değişiklik yaparak yeni yaratılan image dosyasının kullanılmasını sağlar.

Bitiş tarihi: 06-05-2003