

Sistem Programlama ve İleri C Uygulamaları - I

Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

Son Güncelleme Tarihi: 03/02/2022

Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabilir.

1. Giriş

Kursumuzun başında bazı temel kavramlar üzerinde durulacaktır.

1.1. Sistem Programlama Nedir?

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

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

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

1.2. Bilgisayar Donanımlarının Tarihsel Gelişimi

Elektronik düzeyde bugün kullandığımız bilgisayarlara benzer ilk aygıtlar 1940'lı yıllarda geliştirilmeye başlanmıştır. Ondan önce hesaplama işlemlerini yapmak için pek çok mekanik aygıt üzerinde çalışılmıştır. Bunların bazıları kısmen başarılı olmuş ve belli bir süre kullanılmıştır. Mekanik bilgisayarlardaki en önemli girişim Charles Babbage tarafından yapılan "Analytical Engine" ve "Difference Engine" aygıtlarıdır. "Analytical Engine" tam olarak bitirilememiştir. Fakat bunlar pek çok çalışmaya ilham kaynağı olmuştur. Hatta bir dönem Babbage'in asistanlığını yapan Ada Lovelace bu

"Analytical Engine" üzerindeki çalışmalarından dolayı dünyanın ilk programcısı kabul edilmektedir. Şöyle ki: Rivayete göre Babbage Ada'dan "Analytical Engine" için Bernolli sayılarının bulunmasını sağlayan bir yönerge yazmasını istemiştir. Ada'nın yazdığı bu yönergeler dünyanın ilk programı kabul edilmektedir. (Gerçi bu yönergelerin bizzat Babbage'in kendisi tarafından yazılmış olduğu neredeyse ispatlanmış olsa bile böyle atıf vardır.) Daha sonra 1800'lü yılların ortalarından itibaren elektronikte hızlı bir ilerleme yaşanmıştır. Bool cebri ortaya atılmış, çeşitli devre elemanları kullanılmaya başlanmış ve mantık devreleri üzerinde çalışmalar başlatılmıştır. 1900'lü yılların başlarında artık yavaş yavaş elektromekanik bilgisayar fikri belirmeye başlamıştır. 1930'lu yıllarda Alan Turing konuya matematiksel açıdan yaklaşmış ve bugünkü bilgisayar benzeri bir makinenin hangi matematik problemleri çözebileceği üzerine kafa yormuştur. Turing bir şerit üzerinde ilerleyen bir kafadan oluşan ve ismine "Turing Makinesi" denilen soyut makine tanımlamıştır ve bu makinenin neler yapabileceği üzerinde kafa yormuştur. ACM Turing'in anısına bilgisayarın Nobel ödülü gibi kabul edilen Turing ödelleri vermektedir.

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

Modern bilgisayar tarihi üç döneme ayrılarak incelenebilir:

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

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

Transistör 1947 yılında John Bardeen, William Shockley ve Walter Brattain tarafından Bell Lab'ta icad edildi. Fakat 1950'li yılların ortalarına doğru kullanıma girdi. İlk transistörlü radyo ve ilk transistörlü bilgisayar (TRADIC) 1954 yılında yapıldı. Transistörler 1950'li yıllarda yavaş yavaş bilgisayar devrelerine de girmeye başladı. Bu sayede bilgisayar devreleri küçüldü ve kuvvetlendi. O zamanların en önemli firmaları IBM, Honeywell, DEC gibi firmalardı.

Entegre devreye benzer ilk çalışma aslında ilk olarak 1949 yılında Alman mühendis Werner Jacobi tarafından yapıldı. Ancak entegre devre fikri 1952 yılında İngiliz Geoffrey Dummer tarafından ortaya atıldı. Fakat gerçek anlamda ilk gerçekleştirimi 1958 yılında Texas Instruments şirketi çalışanı Jack Kilby tarafından yapıldı. Kilby'den habersiz olarak yaklaşık altı ay sonra benzer entegre devre gerçekleştirimi Fairchild Semiconductor firmasında Robert Noyce tarafından da yapıldı. Kilby ile Noyce patent konusunda mahkemelik olmuşlarsa da sonra anlaşma sağlanmış ve her iki kişi adına patentleme yapılmıştır. Robert Noyce aslında transistörü bulan ekipteki William Shockley'nin yanında çalışıyordu. Bu ekipte Gordon Moore da vardı. Shockley'nin yönetiminden memnun olmayan bu ekip Fairchild Semiconductor şirketine geçmiştir. Noyce şirketin genel müdürü, Moore da arge müdürü olmuştur. Daha sonra 1968 yılında Robert Noyce, Gordon Moore Fairchild Semiconductor firmasından ayrılarak Intel'i kurdu. İkili Intel'i kurduktan sonra şirkete Fairchild Semiconductor'dan Andrew S. Grove da yanlarına aldı. Dünyanın entegre devre olarak üretilen ilk mikroişlemcisi Intel'in 8080'i kabul edilmektedir. Intel daha önce 4004, 8008 gibi entegre devreler yaptıysa da bunlar tam bir mikroişlemci olarak kabul edilmemektedir. Entegre devreler kullanılarak mikroişlemciler yapılmaya başlanınca artık bilgisayar dünyası yeni bir döneme girmiş oldu.

Intel 8080'i tasarladığında bundan bir kişisel bilgisayar yapılabileceği onların aklına gelmemiştir. Kişisel bilgisayar fikri Ed Roberts isimli bir girişimci tarafından ortaya atıldı. Ed Roberts 8080'i kullanarak Altair isimli ilk kişisel bilgisayarı yaptı ve "Popular Electronics" isimli dergiye kapak oldu. Altair makine dilinde kodlanıyordu. Roberts buna Basic derleyicisi yazacak kişi aradı ve Popular Electronics dergisine ilan verdi. İlan o zaman Harvard'ta öğrenci olan Bill Gates ve Paul

Allen başvurdular. Böylece Altair daha sonra Basic ile piyasaya sürüldü. Gates ve Allen okuldan ayrıldılar ve 1975 yılında Microsoft firmasını kurdular. (O zamanlar bu yeni kişisel bilgisayarlara mikrobilgisayarlar denilmekteydi). Amerika'da bu süreç içerisinde bilgisayar kulüpleri kuruldu ve pek çok kişi kendi kişisel bilgisayarlarını yapmaya çalıştı. Steve Jobs ve Steve Wozniak Apple'ı 1976 yılında böyle bir süreçte kurmuştur.

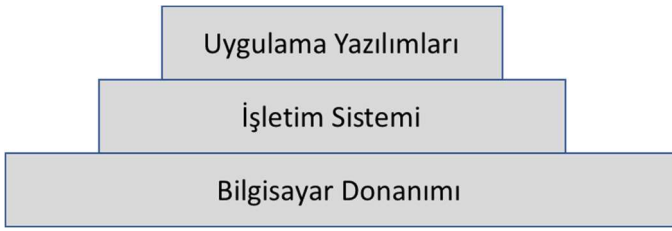
IBM kişisel bilgisayar konusunu hafife aldı. Fakat yine de bir ekip kurarak bugün kullandığımız PC'lerin donanımını tasarlamıştır. Ancak IBM küçük iş olduğu gerekçesiyle bunlara işletim sistemini kendisi yazmadı, taşeron bir firmaya yazdırmak istedi. Bu süreç içerisinde Microsoft IBM ile anlaşarak DOS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

1.3. İşletim Sistemleri

Bu bölümde işletim sistemleri giriş düzeyinde ele alınmaktadır.

1.3.1. İşletim Sistemi Nedir?

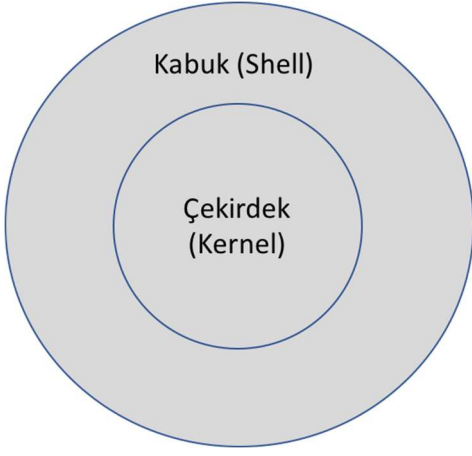
İşletim sistemleri bilgisayar donanımının kaynaklarını yöneten, bilgisayar donanımı ile kullanıcı arasında arayüz oluşturan temel bir sistem programıdır. Pek çok teorisyen işletim sistemini bir kaynak yöneticisi (resource manager) olarak tanımlamıştır.



İşletim sistemlerinin yönettiği kaynakların en önemlileri şunlardır:

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

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



İşletim sistemleri çeşitli biçimlerde sınıflandırılabilir.

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

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

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

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

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

Donanım Özelliğine Göre: Masaüstü, mobil, embedded. Neredeyse her yaygın masaüstü işletim sisteminin bir mobil versiyonu da oluşturulmuştur. Windows'un mobil versiyonuna genel olarak Windows CE denilmektedir. Windows CE'nin

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

Bugün için (2018 Aralık) en yaygın kullanılan mobile işletim sistemi Android'tir (%72'den fazla). Bunu IOS (%20 civarı), onu da Windows Mobil izlemektedir (%0.5 civarı). Mobil işletim sistemlerinin doğal programlama ortamları sistemden sisteme değişebilmektedir. Android'in doğal programla ortamı Java, IOS'un Objective-C ve Swift, Windows Mobile'in ise C#'tır. Tabii bu ortamlarda bu dillerin dışında başka dillerle de uygulamalar geliştirilebilmektedir.

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

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

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

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

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

1.3.2. İşletim Sistemlerinin Doğuşu ve İlk İşletim Sistemleri

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

1950'li yıllarda IBM gibi pek çok bilgisayar üreticisi firma yalnızca donanım satıyordu. İşletim sistemi gibi programları yazmak kullanıcıların yapması gereken bir işti. Böylece donanımı satın alan her kurum işletim sistemine benzeyen programları da kendisi yazıyordu. Bu anlamda standart bir işletim sistemi yoktu. Bugünkü anlamda ilk işletim sisteminin General Motors'un 1955 yılında IBM'in 701 sistemi için yazdığı GMOS ve 1956 yılında aynı donanım için yazdığı NAA IO olduğu söylenebilir.

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

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

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

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

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

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

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

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

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

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

1.3.3. UNIX Türevi İşletim Sistemlerinin Tarihsel Gelişimi

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

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

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

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

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

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

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

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

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

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

FreeBSD, NETBSD ve OpenBSD: 4.3BSD sistemleri temel alınarak geliştirilmiştir. FreeBSD NetBSDve 1993 yılında, OpenBSD ise 1996 yılında piyasaya çıkmıştır. Sürüdürülmeye devam etmektedir. Önemli bir UNIX varyantı durumundadır. Bu üç sistem de birbirlerine çok benzemektedir. FreeBSD genel amaçlı client ve server işletim sistemi olma niyetindedir. NetBSD daha taşınabilir ve geniş bir porta sahiptir. Daha çok bilimsel çalışmalarda tercih edilmektedir. OpenBSD güvenliğin önemli olduğu alanlarda tercih edilmektedir.

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

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

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

1.3.4. Linux Sistemlerinin Tarihsel Gelişimi

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

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

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

1.3.5. Dağıtım Kavramı ve Linux Dağıtımları

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

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

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

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

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

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

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

1.3.6. Mac OS X Sistemlerinin Tarihsel Gelişimi

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

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

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

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

Mac OS X sistemlerinin versiyonları şunlardır:

- Mac OS X 10.0 (Cheetah, 2001)
- Mac OS X 10.1 (Puma, 2001)
- Mac OS X 10.2 (Jaguar, 2002)
- Mac OS X 10.3 (Panther, 2003)
- Mac OS X 10.4 (Tiger, 2005)
- Mac OS X 10.5 (Leopard, 2007)
- Mac OS X 10.6 (Snow Leopard, 2009)
- Mac OS X 10.7 (Lion, 2011)
- Mac OS X 10.8 (Mountain Lion, 2012)
- Mac OS X 10.9 (Maverics, 2013)
- Mac OS X 10.10 (Yosemite, 2014)
- Mac OS X 10.11 (El Capitan, 2015)
- Mac OS X 10.12 (Sierra, 2017)
- Mac OS X 10.13 (High Sierra, 2017)
- Mac OS X 10.14 (Mojave, 2018)
- Mac OS X. 10.15 (Catalina, 2019)

Mac OS X büyük ölçüde POSIX uyumlu bir sistemdir.

1.3.7. Masaüstü ve Mobil İşletim Sistemleri

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

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

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

1.3.8. Orijinal Kod Temeline Sahip İşletim Sistemleri ve Dağıtımlar

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

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

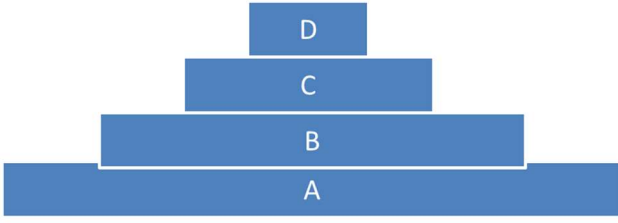
Mac OS X'in çekirdeği olan Darwin Carnegie Mellon Üniversitesi tarafından yazılan Mach çekirdeği ile BSD çekirdeğinin kodlarını hala kullanmaktadır. Bu anlamda Mac OS X orijinal bir kod temeline sahip değildir. Benzer biçimde Android ve IOS da orijinal kod temeline sahip işletim sistemleri değildir.

Dağıtımlar (distributions) bir işletim sistemi çekirdeğinin ve kabuğunun çeşitli araçlarla donatılmış biçimde bir araya getirilmesiyle oluşturulmuş olan yazılım paketleridir. Örneğin Linux çekirdeğini kullanan pek çok dağıtım oluşturulmuştur.

Peki "bir işletim sistemi yazdık" diyebilmek için ne gerekir? Yanıt: Yapılan katkının çekirdek düzeyinde ve belli bir yoğunlukta olması gerekir. Bu tanıma göre örneğin Ubuntu gibi, Suse gibi, Mint gibi, Pardus gibi dağıtımlar bir işletim sistemi olarak isimlendirilmemelidir.

1.4. Programlamadaki Katmanlı Yapılar

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



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

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

1.5. API (Application Programming Interface) Kavramı

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

1.6. Kütüphane (Library) ve Framework ve Toolkit Kavramları

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

- 1) Karmaşıklığın kullanıcıya daha basit gösterilmesi ve yük oluşturan bazı hammaliye işlemlerin kullanıcının üzerinden alınması.
- 2) Kod akışının ele geçirilmesi ve duruma göre programcıya belli zamanlarda verilmesi (inversion of control).

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

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

Toolkit çok yaygın kullanılan bir terim değildir. Farklı konularda kütüphanelerin bir araya getirilmesi ile oluşturulan kütüphane paketlerine "toolkit" denilmektedir. Örneğin Qt için hem "framework" hem de "toolkit" terimi kullanılabilir.

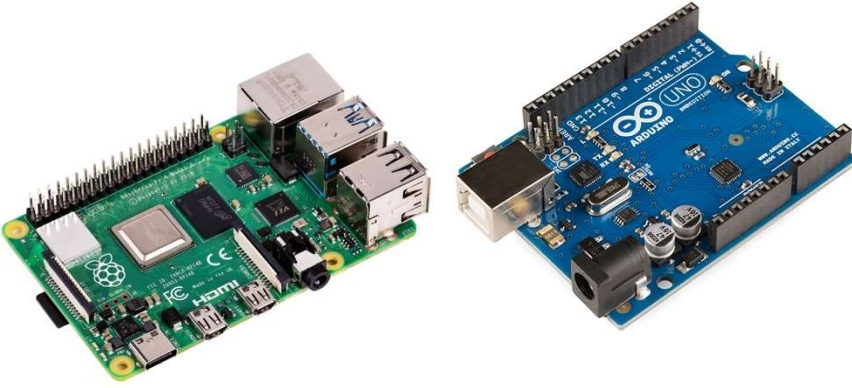
1.7. CPU, Mikroişlemci, Mikrodenetleyici, SOC ve SBC (Single Board Computer Kavramaları)

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

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

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

SBC (Single Board Computer) küçük bir kit (baskılı devre) üzerine monte edilmiş bilgisayar devreleri için kullanılan bir terimdir. Genellikle bu kitlerde SOC'lar, başka çevre birimleri ve IO işlemleri için soketler bulunur. Örneğin Raspberry Pi ve Arduino ismi verilen bilgisayarlar SBC olarak üretilmektedir. Aşağıda Raspberry Pi 4 ve Arduino Uno R3 modellerinin resimlerini görüyorsunuz:

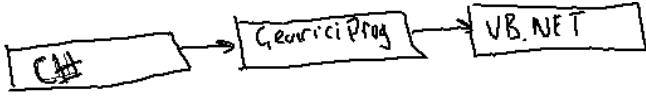


1.8. Gömülü Sistemler (Embedded Systems)

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

1.9. Çevirici Programlar (Translators), Derleyiciler (Compilers) ve Yorumlayıcılar (Interpreters)

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



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

Eğer bir çevirici programda hedef dil aşağı seviyeli bir dil ise (saf makine dili, arakod ve sembolik makine dilleri alçak seviyeli dillerdir) böyle çevirici programlara derleyici (compiler) denilmektedir. Her derleyici bir çevirici programdır fakat her çevirici program bir derleyici değildir. Bir çevirici programa derleyici diyebilmek için hedef dile bakmak gerekir. Örneğin arakodu gerçek makine koduna dönüştüren CLR bir derleme işlemi yapmaktadır. Sembolik makine dilini saf makina diline dönüştüren program da bir derleyicidir.

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

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

1.10. Decompiler'lar ve Disassembler'lar

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

1.11. IDE (Integrated Development Environment) Kavramı

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

Microsoft'un ünlü IDE'sinin ismi "Visual Studio"dur. Apple'ın "X-Code" isimli IDE'si vardır. Bunların dışında başka şirketlerin malı olan ya da "open source" olan pek çok IDE mevcuttur. Örneğin "Eclipse" ve "Netbeans" yaygın kullanılan cross-platform "open source" IDE'lerdir. Linux altında Mono'da "Mono Develop" isimli bir IDE tercih edilmektedir. Bu IDE'nin Windows versiyonu da vardır.

UNIX/Linux sistemlerinde C ve C++ IDE'si olarak QtCreator, KDevelop, Eclipse (CDT), NetBeans, ve MonoDevelop kullanılabilir. Ancak kursumuzdaki uygulamalarda bir IDE kullanmayacağız. Kodları bir Kate, Visual Studio Code gibi bir editörde yazıp komut satırından derleyeceğiz.

1.12. Doğal Kodlu Çalışma, Arakodlu Çalışma ve JIT Derlemesi

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

Bazı sistemlerde derleyiciler doğrudan doğal kod üretmek yerine hiçbir CPU'nun makine dili olmayan (dolayısıyla hiçbir CPU tarafından işletilemeyen) yapay bir kod üretmektedir. Bu yapay kodlara genel olarak "ara kodlar (intermediate codes)" denilmektedir. Bu arakodlar doğrudan CPU tarafından çalıştırılmazlar. Arakodlu çalışma Java ve .NET dünyasında ve daha başka ortamlarda kullanılmaktadır. Java dünyasında Java derleyicilerinin ürettikleri ara koda "Java Bytecode", .NET (CLI) dünyasında ise "CIL (Common Intermediate Language)" denilmektedir. Pekiyi bu arakodlar ne işe yaramaktadır? İşte bu arakodlar çalıştırılmak istendiğinde ilgili ortamın alt sistemleri devreye girerek önce bu arakodları o anda çalışılan CPU'nun doğal makine diline dönüştürüp sonra çalıştırmaktadır. Bu sürece (yani arakodun doğal makine koduna dönüştürülmesi sürecine)" tam zamanında derleme (just in time compilation)" ya da kısaca "JIT derlemesi" denilmektedir. Java ortamında bu JIT derlemesi yapıp programı çalıştıran alt sisteme "Java Sanal Makinesi (Java Virtual Machine)", .NET ortamında ise CLR (Common Language Runtime)" denilmektedir.

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

1.13. C'de Tanımsız Davranış (Undefined Behavior), Derleyiciye Bağlı Davranış (Implementation Dependent Behavior) ve Belirsiz Davranış (Unspecified Behavior) Kavramları

C'de bazı kodlar için nasıl bir sonuç elde edileceği tam olarak tanımlanmamıştır. Bunlara tanımsız davranışa yol açan kodlar denilmektedir. Bu tür kodlardan kaçınmak gerekir. Tanımsız davranışa yol açan kodlar derleme aşamasında herhangi bir sorun oluşturmazlar. Bu kodlar dilin sentaks yapısına tamamen uygundur. Fakat bu tür kodları içeren programlar çalışırken artık her şey olabilir. Program çökebilir, yanlış çalışabilir ya da hiçbir şey olmayabilir. Örneğin nereyi gösterdiği belli olmayan bir göstericinin gösterdiği yere * ya da [] operatörleriyle erişmek tanımsız davranışa yol açar. Ya da örneğin bir ifadede bir nesne ++ ya da -- operatörüyle kullanılmışsa o ifadede artık o nesne gözükmemelidir. Aksi halde tanımsız davranış oluşur.

Bazı durumlarda standartlar kesin belirlemeyi dokümanete etmek koşuluyla derleyicileri yazarlara bırakmıştır. Bu tür kodlara derleyiciye bağlı kodlar denir. Örneğin C'de işaretli bir tamsayının sağa ötelenmesinde işaret bitinin korunup korunmayacağı, büyük tamsayı türünden küçük işaretli tamsayı türüne yapılan dönüştürmelerde bilgi kaybının nasıl oluşacağı böyledir.

Bazı durumlarda ise birkaç seçenek vardır. Bu seçeneklerden herhangi birini derleyiciyi yazarlar seçmiş olabilir. Fakat bunu dokümanete etmek zorunda değildirler. Böyle kodlara da "belirsiz davranışa yol açan kodlar" denilmektedir. Örneğin argümanlardan parametre değişkenlerine aktarım sırasının soldan-sağa mı sağdan-sola mı olacağı derleyiciden derleyiciye değişebilir. Derleyicileri yazarlar bunu dokümanete etmek zorunda değildirler.

1.14. Derleyicilerin Hata Mesajları ve Standartlara Uyum

C standartlarında geçersiz bir programın derleyici tarafından başarılı biçimde derlenip derlenmeyeceği konusunda bir belirlemede bulunulmamıştır. Yani geçersiz bir programı derleyici isterse birtakım hataları görmezden gelerek

derleyebilir ya da hiç derlemeyebilir. Ancak standartlarda geçersiz durumlar için derleyicilerin en az bir tane durumu anlatan bir mesaj vermesi öngörülmüştür. Bu durumda örneğin derleyici geçersiz program için bir mesaj vererek onu derleyebilir. Tabii bu durumda söz konusu bu mesaj bir uyarı mesajı olacaktır. Örneğin:

```
char buf[10];
int *pi;

pi = buf;      /* geçersiz! */
```

Burada char türden bir adres bilgisi int türden bir göstericiye atanmıştır. Bu geçersizdir. Ancak derleyici bir uyarı mesajı vererek derleme işlemini başarıyla bitirebilir. Tabii aynı kodu başka bir derleyici "error" mesajı vererek derlemeyebilir. Bu durumda bir programın belli bir C derleyicisinde derleniyor olması onun standartlar bağlamında geçerli olduğu anlamına gelmemektedir. Tabii geçerli olan programların derleyici tarafından başarılı bir biçimde derlenmesi gerektiği standartlarda belirtilmiştir. Şüphesiz derleyici geçerli bir kod için de birtakım uyarı mesajları verebilmektedir.

1.15. UNIX/Linux Ortamlarının Windows ve Mac OS X Sistemlerinde Oluşturulması

Windows'ta Linux ortamının oluşturulması için iki yöntem kullanılabilir.

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

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

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

1.16. UNIX/Linux Sistemlerinde ve Windows Sistemlerinde C Programlarının Derlenerek Çalıştırılması

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

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

ya da :

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

Örneğin:

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

ya da örneğin:

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


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

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

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

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

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

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

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

```
./sample
```

"." karakterinin "bulunulan dizini temsil ettiğini" anımsayınız.

Örneğin:

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

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

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

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

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

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

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

UNIX/Linux sistemlerinde çeşitli C/C++ IDE'leri de kullanılabilir. Örneğin "Qt Creator" IDE'si "cross platform" bir IDE olarak bu sistemlerde kullanılabilir. Qt Creator IDE'sinde C projesi oluşturmak için önce "File/New File or Project" menü elemanı seçilir. Açılan diyalog penceresinde de "None Qt Project/Plain C Application" seçilerek yalın bir C projesi oluşturulabilir. Benzer biçimde Eclipse ve Netbeans IDE'leri de C/C++ için "cross platform" IDE'ler olarak tercih edilebilirler. Diğer bir seçenek ise Mono ortamı için geliştirilen "Mono Develop" IDE'sidir. Bu IDE de aslında C# için geliştirilmiş olsa da C ve C++ dilleri için de kullanılabilir.

Windows sistemlerinde Microsoft'un komut satırı derleyicisi "cl.exe" isimli programdır. Bu programla derleme ve bağlama işlemi aşağıdaki gibi yapılır:

```
cl Sample.c
```

cl default durumda yine bağlama işlemi için bağlayıcı programı kendisi çalıştırmaktadır. cl programı kaynak dosya ile aynı isimli çalıştırılabilir dosya oluşturur. Tabii biz cl ile yine derleme ve bağlama işlemlerini ayrı ayrı yapabiliriz. cl programının /C seçeneği "yalnızca derle, bağlayıcıyı çalıştırma" anlamına gelmektedir. Örneğin:

```
d:\Dropbox\Kurslar\SysProg-1\Src\Sample>cl /c sample.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.00.23506 for x86
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
sample.c
```

```
d:\Dropbox\Kurslar\SysProg-1\Src\Sample>dir sample.obj
Volume in drive D has no label.
Volume Serial Number is 58BE-3327
```

```
Directory of d:\Dropbox\Kurslar\SysProg-1\Src\Sample
```

```
09.02.2019 10:17          2,885 sample.obj
             1 File(s)          2,885 bytes
             0 Dir(s) 1,672,332,947,456 bytes free
```

```
d:\Dropbox\Kurslar\SysProg-1\Src\Sample>link sample.obj
Microsoft (R) Incremental Linker Version 14.00.23506.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
d:\Dropbox\Kurslar\SysProg-1\Src\Sample>
```

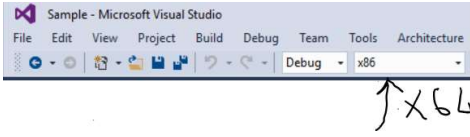
cl derleyicisinde Komut satırı seçenekleri '/' karakteriyle verilmektedir. Yine tıpkı gcc'de olduğu gibi cl derleyicisi de hiç derleme yapmadan doğrudan bağlayıcıyı çalıştırabilmektedir. Örneğin:

```
d:\Dropbox\Kurslar\SysProg-1\Src\Sample>link sample.obj
Microsoft (R) Incremental Linker Version 14.00.23506.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
d:\Dropbox\Kurslar\SysProg-1\Src\Sample>
```

Microsoft'un bağlayıcı programı link.exe isimli programdır. Aslında Visual Studio derleme ve bağlama işlemlerini bu programları çalıştırarak gerçekleştirmektedir.

Windows ortamında Visual Studio IDE'sinde default derleme 32 bit biçimde yapılmaktadır. Microsoft maalesef 32 bit ve 64 bit derleyicileri aynı isimli fakat ayrı programlar olarak bulundurmaktadır. Her iki derleyicinin de ismi "cl.exe" biçimindedir. Dolayısıyla biz "cl.exe"yi çalıştırdığımızda PATH çevre değişkeninde hangi "cl.exe"nin dizini daha önce geliyorsa o devreye girer. Tabii biz tam yol ifadesi vererek istediğimiz "cl.exe"nin çalıştırılmasını da sağlayabiliriz. Örneğin "Visual Studio 2015"te 32 bit derleme yapan "cl.exe" "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin" dizini içerisinde, 64 bit derleme yapan "cl.exe" ise "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64" dizini içerisindedir. Ancak Visual Studio IDE'si ile yapıyorsanız 64 bit derleme basit bir biçimde konfigürasyon seçeneklerinden X64 seçilerek de yapılabilir:



1.17. Bazı Az Kullanılan Standart C Fonksiyonları

Bu bölümde kursumuzda kullanılabilecek bazı yaygın C fonksiyonları ele alınacaktır.

1.17.1. strtok Fonksiyonu

strtok fonksiyonu bir yazıyı bazı karakterlere göre ayrıştırmakta (parse etmekte) kullanılan standart bir C fonksiyonudur. Örneğin aşağıdaki gibi bir yazı olsun:

```
"ankara,adana,izmir,kars"
```

Burada yazı ',' karakterlerinden ayrıştırılarak "ankara", "adana", "izmir", "kars" yazıları elde edilmek istenebilir. Ya da örneğin:

```
"10/12/2007"
```

Burada yazı '/' karakterlerinden ayrıştırılarak tarihin gün, ay, yıl bileşenleri elde edilmek istenebilir.

strtok fonksiyonunun prototipi şöyledir:

```
#include <string.h>

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

Fonksiyon birinci parametresiyle belirtilen yazı içerisinde ikinci parametresiyle belirtilen karakterleri arar. Bunlardan birini bulursa oraya '\0' yerleştirip, o kısmın adresiyle geri döner. Eğer birinci parametre NULL geçilirse fonksiyon kaldığı yerden devam eder. Eğer bulunacak hiçbir atom kalmamışsa fonksiyon NULL adresle geri dönmektedir. Tipik kullanım şöyledir:

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

int main(void)
{
    char text[] = "ankara,izmir,adana,eskisehir";
    char *str;

    str = strtok(text, ",");
    while (str != NULL) {
        puts(str);
        str = strtok(NULL, ",");
    }

    return 0;
}
```

Tabii while yerine for döngüsünü de tercih edebilirsiniz:

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

int main(void)
{
    char text[] = "ankara,izmir,adana,eskisehir";
    char *str;
```

```

    for (str = strtok(text, ","); str != NULL; str = strtok(NULL, ","))
        puts(str);

    return 0;
}

```

strtok fonksiyonu ile karmaşık ayrıştırma işlemleri yapılamaz. Örneğin biz bu fonksiyonla bir C programını atomlarına ayıramayız. strtok ile ancak basit ayrıştırma işlemlerini yapılabiliriz.

strtok fonksiyonunun birinci parametresinin const olmayan bir gösterici olduğuna dikkat ediniz. Fonksiyon ayrıştırılacak yazıyı bozmaktadır. Bu nedenle ayrıştırılacak yazıyı iki tırnak içerisinde bir string olarak girmeye çalışmayınız (C'de string ifadelerinin güncellenmesinin tanımsız davranışa (undefined behavior) yol açtığını anımsayınız). strtok fonksiyonu aşağıdaki gibi yazılabilir:

```

#include <stdio.h>
#include <string.h>
#include <ctype.h>

char *mystrtok(char *str, const char *delim)
{
    static char *pos;
    char *beg;

    if (str != NULL)
        pos = str;

    while (*pos != '\0' && strchr(delim, *pos) != NULL)
        ++pos;
    if (*pos == '\0')
        return NULL;
    beg = pos;
    while (*pos != '\0' && strchr(delim, *pos) == NULL)
        ++pos;
    if (*pos != '\0')
        *pos++ = '\0';

    return beg;
}

int main(void)
{
    char text[] = "ankara,      izmir";
    char *str;

    for (str = mystrtok(text, " ,"); str != NULL; str = mystrtok(NULL, " ,"))
        puts(str);

    return 0;
}

```

strtok fonksiyonunun asıl yazı üzerinde değişiklik yapmayan bir biçimi şöyle yazılabilir:

```

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

char *mystrtok(const char *str, const char *delim, char *dest)
{
    static const char *pos;
    const char *beg;

    if (str != NULL)

```

```

    pos = str;

    while (*pos != '\0' && strchr(delim, *pos) != NULL)
        ++pos;
    if (*pos == '\0')
        return NULL;
    beg = pos;
    while (*pos != '\0' && strchr(delim, *pos) == NULL)
        ++pos;
    strncpy(dest, beg, pos - beg);
    dest[pos - beg] = '\0';

    return dest;
}

int main(void)
{
    char *text = "ankara,    izmir, istanbul";
    char s[1000];
    char *str;

    for (str = mystrtok(text, " ,", s); str != NULL; str = mystrtok(NULL, " ,", s))
        puts(str);

    return 0;
}

```

strok ile bir dosyayı satır satır okuyarak onların üzerinde ayrıştırma yapabiliriz. Örneğin:

```

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

#define MAX_LINE    1024

int main(void)
{
    FILE *f;
    char line[MAX_LINE];
    char *str;

    if ((f = fopen("test.txt", "r")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    while (fgets(line, MAX_LINE, f) != NULL) {
        for (str = strtok(line, "\t\n"); str != NULL; str = strtok(NULL, "\t\n"))
            puts(str);
        printf("-----\n");
    }

    fclose(f);

    return 0;
}

```

1.17.2. remove Fonksiyonu

remove bir dosyayı silmek için kullanılan standart bir C fonksiyonudur. Prototipi şöyledir:

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

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

Fonksiyon parametre olarak silinecek dosyanın yol ifadesi (path name) alır. Başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner.

Örneğin:

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

int main(void)
{
    if (remove("test.txt") == -1) {
        fprintf(stderr, "cannot delete file!..\n");
        exit(EXIT_FAILURE);
    }
    printf("Ok\n");

    return 0;
}
```

1.17.3. rename Fonksiyonu

rename dosyanın ismini değiştirmek için kullanılan standart bir C fonksiyonudur. Prototipi şöyledir:

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Fonksiyonun birinci parametresi dosyanın eski yol ifadesini, ikinci parametresi ise yeni yol ifadesini belirtir. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda -1 değerine geri döner. Örneğin:

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

int main(void)
{
    if (rename("test.txt", "x.txt") != 0) {
        fprintf(stderr, "cannot rename file!..\n");
        exit(EXIT_FAILURE);
    }
    printf("Ok\n");

    return 0;
}
```

rename fonksiyonu Windows ve UNIX/Linux sistemlerinde dosyayı taşıma işlemini de yapabilmektedir.

1.17.4. system Fonksiyonu

system fonksiyonu kabuk programını interaktif olmayan modda çalıştırarak parametresiyle belirtilen kabuk komutunun kabuk tarafından çalıştırılmasını sağlar. Fonksiyonun prototipi şöyledir:

```
#include <stdlib.h>

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

Fonksiyon kabuk komutunu yazı olarak alır. Kabuk programını (komut yorumlayıcı programı) çalıştırarak komutu kabuk programına işletir. Fonksiyonun geri dönüş değeri sistemden sisteme değişebilmektedir (implementation dependent). Pek çok sistemde system fonksiyonu fonksiyon başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri

dönmektedir. Fonksiyon özel bir durum olarak NULL adresle çağrılabilir. Bu durumda fonksiyon sıfır dışı bir değere geri dönerse ilgili sistemde kabuk programı vardır, sıfıra geri dönerse yoktur.

Örneğin:

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

int main(void)
{
    if (system("ren x.txt a.txt & copy a.txt b.txt") != 0) {
        fprintf(stderr, "system failed!..\n");
        exit(EXIT_FAILURE);
    }

    printf("Ok\n");

    return 0;
}
```

Komut satırlarında bir komut kendi içerisinde komut parçalarına ayrılabilir. Bu işlem Windows'ta '&' karakteri ile, UNIX/Linux sistemlerinde de ';' karakteri ile yapılmaktadır. Örneğin biz bu sayede UNIX/Linux sistemlerinde birden fazla kabuk komutunu aralarına ';' karakteri getirerek tek bir komut gibi işletebiliriz.

UNIX/Linux ve Windows sistemlerinde kabuk (shell) programları temelde iki moda çalıştırılmaktadır: "İnteraktif mod" ve "interaktif olmayan (yani tek komutluk)" mod. İnteraktif modda bir prompt çıkar. Kullanıcı komutu girer, o komut çalışır, sonra yeniden prompt'a düşülür. Ta ki exit ya da logout komutu uygulanana kadar. Halbuki interaktif olmayan moda tek bir komut çalıştırılıp kabuktan çıkılmaktadır. İşte system fonksiyonu kabuk programını böyle çalıştırmaktadır.

Windows sistemlerinde kabuk program "cmd.exe" ismiyle UNIX/Linux sistemlerinde ise "bash" ismiyle bulunmaktadır. Aslında UNIX/Linux sistemlerinde tek bir kabuk programı da yoktur. Ancak bugünlerde en çok kullanılan dolayısıyla "default" durumda olan "bash (Bourne Again Shell)" kabuğudur. Tek komutluk çalıştırma için Windows sistemlerinde "cmd.exe" programı "/C" seçeneği ile, UNIX/Linux sistemlerinde ise "bash" programı "-c" seçeneği ile çalıştırılmaktadır.

Anahtar Notlar: Linux aslında bir çekirdek projesidir. Bir Linux dağıtımıyla bilgisayarımıza yüklediğimiz tüm yazılımlar farklı proje grupları tarafından oluşturulmuş durumdadır. Bunlar açık kaynak kodlu olduğu için gerekirse kaynak kodları indirilerek incelenebilir. Örneğin bash, gnome, kde, web tarayıcıları, metin editörleri vs. hep farklı kişi uya da gruplar tarafından gerçekleştirilmiş yazılımlardır.

1.17.5. Geçici Dosya Kavramı ve Geçici Dosya Oluşturan Fonksiyonlar

Bazen çeşitli nedenlerden dolayı bir dosya yaratıp onun içerisine birtakım bilgileri yazıp o dosya üzerinde bazı işlemleri yapmak isteyebiliriz. Genellikle hedeflediğimiz işlem bittiğinde dosyaya gereksinimiz de kalmaz ve biz onu sileriz. Bu amaçla kullanılan dosyalara geçici dosyalar (temporary files) denilmektedir. Örneğin tipik olarak C derleyicilerindeki önışlemci modülü ilgili kaynak dosyayı okur, onu #'li ifadelerden arındırır ve bir geçici dosyaya yazar. Derleme modülü de aslında önışlemcinin oluşturduğu bu dosyayı alarak derler. Derleme işlemi bittiğinde de önışlemcinin yaratmış olduğu bu geçici dosya silinmektedir. Örneğin bir dosyanın herhangi bir yerine bir grup bilgiyi eklemek (insert etmek) isteyelim. Bu işlem tipik olarak bir geçici dosya yaratılarak yapılmaktadır.

Geçici dosyaların oluşturulmasında dikkat edilmesi gereken en önemli noktalardan biri isim çakışmasıdır. Yani geçici dosyaya vereceğimiz isim zaten var olan bir dosya ismiyle çakışmamalıdır. İşte isim çakışmasına yol açmadan geçici dosya açmakta kullanılan iki standart C fonksiyonu vardır: tmpfile ve tmpnam.

1.17.5.1. tmpfile Fonksiyonu

Fonksiyonun prototipi şöyledir:

```
#include <stdio.h>

FILE *tmpfile(void);
```

Fonksiyon "w+b" modunda yeni bir dosyayı isim çakışması olmadan (yani çakışmayan bir isimle) yaratır, onu açar ve bize o dosyanın dosya bilgi göstericisi (stream) ile geri döner. Biz de bu dosyayı kullanırız. İşimiz bittiğinde de fclose fonksiyonuyla dosyayı kapatırız. tmpfile fonksiyonunun açarak bize verdiği dosya fclose işlemiyle kapatıldığında otomatik olarak silinmektedir. Tabii tmpfile fonksiyonu başarısız da olabilir. Bu durumda NULL adrese geri döner.

Örneğin:

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

int main(void)
{
    FILE *f;
    int i, val;

    if ((f = tmpfile()) == NULL) {
        fprintf(stderr, "cannot create temporary file!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 100; ++i)
        if (fwrite(&i, sizeof(int), 1, f) != 1) {
            fprintf(stderr, "cannot write file!..\n");
            exit(EXIT_FAILURE);
        }

    fseek(f, 0, SEEK_SET);

    while (fread(&val, sizeof(int), 1, f) == 1)
        printf("%d ", val);

    if (ferror(f)) {
        fprintf(stderr, "cannot read file!...\n");
        exit(EXIT_FAILURE);
    }
    printf("\n");

    fclose(f);

    return 0;
}
```

Örneğimizde geçici dosya tmpfile fonksiyonuyla açılmış sonra içine bir şeyler yazılmış sonra da dosya göstericisi dosyanın başına çekilerek okuma yapılmıştır. tmpfile fonksiyonuyla açmış olduğumuz dosyayı kapatmazsak dosya program bittiğinde otomatik olarak kapatılmaktadır. Yani bu durumda da dosya silinmiş olacaktır.

1.17.5.2. tmpnam Fonksiyonu

Bu fonksiyon geçici dosyayı kendisi açmaz. Bize çakışmayan bir geçici dosya ismi verir. Prototipi şöyledir:

```
#include <stdio.h>

char *tmpnam(char *s);
```

Fonksiyon parametre olarak bizden geçici dosya isminin yerleştirileceği char türden dizinin adresini alır. Fakat bu parametre NULL adres olarak da girilebilir. Bu durumda fonksiyon kendi içerisindeki static bir dizine dosya ismini yerleştirir ve o static alanın adresiyle geri döner. Fonksiyon başarısızlık durumunda ise NULL adrese geri dönmektedir. (Tabii böyle bir başarısızlığın oluşma olasılığı yok denecek kadar zayıftır.) tmpnam fonksiyonunun dosyayı açmadığına yalnızca geçici dosyanın ismini bize verdiği dikkat ediniz. Dosyayı açmak programcının sorumluluğundadır.

Aşağıdaki örnekte bir dosyadaki #'li satırlar silinmektedir:

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

#define MAX_LINE      8192

int issharp(const char *line);

int main(int argc, char *argv[])
{
    FILE *f, *ftemp;
    char line[MAX_LINE];
    char *path;

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

    if ((path = tmpnam(NULL)) == NULL) {
        fprintf(stderr, "cannot get temporary file!..\n");
        exit(EXIT_FAILURE);
    }

    if ((f = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    if ((ftemp = fopen(path, "w")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    while (fgets(line, MAX_LINE, f) != NULL) {
        if (!isharp(line))
            if (fputs(line, ftemp) == EOF) {
                fprintf(stderr, "cannot write file temporary file!..\n");
                goto FAILURE;
            }
    }

    if (ferror(f)) {
        fprintf(stderr, "cannot read file: %s\n", argv[1]);
        goto FAILURE;
    }

    fclose(f);
    fclose(ftemp);

    if (remove(argv[1]) != 0) {
        fprintf(stderr, "cannot remove file!..\n");
        goto FAILURE;
    }

    if (rename(path, argv[1]) != 0) {
        fprintf(stderr, "cannot rename file!..\n");
        exit(EXIT_FAILURE);
    }

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

    exit(EXIT_SUCCESS);
}
```



```

FAILURE:
    fclose(ftemp);
    if (remove(path) != 0)
        fprintf(stderr, "cannot remove file!..\n");

    exit(EXIT_FAILURE);

    return 0;
}

int issharp(const char *line)
{
    while (isspace(*line))
        ++line;

    return *line == '#';
}

```

1.18. UNIX/Linux ve Windows Sistemlerinde Programların Komut Satırı Argümanları

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

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

- GNU stilinde komut satırı argümanları üç biçimde bulunabilmektedir:

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

```
cat sample.c
```

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

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

```
ls -l
```

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

```
ls -l -i
```

ile:

```
ls -li
```

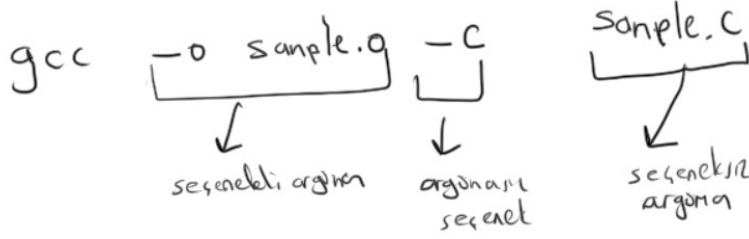
aynı anlamdadır. Eğer seçenek birden fazla karakterden oluşuyorsa bu durumda '-' yerine '--' kullanılmaktadır. Örneğin:

```
ls --version
```

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

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

Örneğin:



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

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

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

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

1.18.1. getopt Fonksiyonunun Kullanımı

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

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

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

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

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

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

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

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

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

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

```
...  
}
```

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

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

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

```
#include <stdio.h>  
#include <unistd.h>  
  
int main(int argc, char **argv)  
{  
    int ch;  
    char *carg;  
    int aflag = 0, bflag = 0, cflag = 0;  
  
    while ((ch = getopt(argc, argv, "abc:")) != -1) {  
        switch (ch) {  
            case 'a':  
                aflag = 1;  
                break;  
            case 'b':  
                bflag = 1;  
                break;  
            case 'c':  
                carg = optarg;  
                cflag = 1;  
                break;  
        }  
    }  
  
    if (aflag)  
        printf("-a switch is given\n");  
  
    if (bflag)  
        printf("-b switch is given\n");  
  
    if (cflag)  
        printf("-c switch is given and its arument is: %s\n", carg);  
  
    return 0;  
}
```

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

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

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

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

```
// abc:
./sample -a ali -b veli selami -c ayse fatma NULL
./sample -a -b -c ayse ali veli selami fatma NULL
                ↑
                optind
```

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

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

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

Tipik bir getopt kullanımı şöyle olabilir:

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

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

    opterr = 0;
    while ((result = getopt(argc, argv, "abc:d:")) != -1) {
        switch (result) {
            case 'a':
                a_flag = 1;
                break;
            case 'b':
                b_flag = 1;
                break;
            case 'c':
                c_flag = 1;
                c_arg = optarg;
                break;
            case 'd':
                d_flag = 1;
                d_arg = optarg;
                break;
        }
    }
}
```

```

    case '?':
        if (optopt == 'c' || optopt == 'd')
            fprintf(stderr, "-%c option given without an argument\n", optopt);
        else
            fprintf(stderr, "invalid switch: %c\n", optopt);
        exit(EXIT_FAILURE);
    }
}

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

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

return 0;
}

```

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

Programın kullanımı şöyledir:

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

- Programın dört seçeneği şöyledir:

```

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

```

Program iki argümanın dört işlem değerini alarak stdout dosyasına yazdırmalıdır.

1.18.2. getopt_long Fonksiyonun Kullanımı

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

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

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

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

```
#include <getopt.h>

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

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

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

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

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

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

Yapının flag elemanı int türden bir nesnenin adresini almaktadır. Bu eleman NULL adres olarak geçilebilir. Eğer bu eleman NULL adres olarak geçilmemişse getopt_long fonksiyonu girişte ilgili seçenek belirtildiğinde bu adresin gösterdiği nesneye yapının val elemanındaki değeri atar ve 0 ile geri döner. Yapının bu flag elemanına NULL adres atanırsa getopt_long doğrudan yapının val elemanı ile belirtilen değere geri dönmektedir. Özellikle kısa seçenekle eşdeğer olan uzun seçenekler söz konusu olduğunda programcı yapının bu flag elemanına NULL değerini, val elemanına da kısa seçeneğin karakter değerini yerleştirip durumu daha kolay ele alabilmektedir.

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

getopt_long fonksiyonunun geri dönüş değerinin beş biçimde olabileceğine dikkat ediniz:

- 1) Eğer getopt_long komut satırı argüman incelemesinin sonuna geldiyse -1 değerine geri döner.
- 2) Eğer getopt_long uzun bir seçeneği bulduysa ve o seçeneğe ilişkin option yapı nesnesinin flag elemanı NULL ise bu durumda o yapı nesnesinin val elemanındaki değerle geri döner.
- 3) Eğer getopt_long uzun bir seçeneği bulduysa ve o seçeneğe ilişkin yapı nesnesinin flag elemanı NULL değilse fonksiyon yapı nesnesinin val elemanındaki değeri flag elemanı ile belirtilen int nesneye yerleştirir ve sıfır ile geri döner.

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

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

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

`getopt_long` için değişik kullanımları içeren aşağıdaki örneği verebiliriz:

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

int main(int argc, char *argv[])
{
    int result;
    int a_flag, b_flag, c_flag, help_flag, count_flag, line_flag;
    char *b_arg, *count_arg, *line_arg;
    int err_flag;
    int i;

    struct option options[] = {
        {"help", no_argument, &help_flag, 1},
        {"count", required_argument, NULL, 2},
        {"line", optional_argument, NULL, 3},
        {0, 0, 0, 0 }
    };

    opterr = 0;

    a_flag = b_flag = c_flag = help_flag = count_flag = line_flag = 0;
    err_flag = 0;

    while ((result = getopt_long(argc, argv, "ab:c", options, NULL)) != -1) {
        switch (result) {
            case 'a':
                a_flag = 1;
                break;
            case 'b':
                b_flag = 1;
                b_arg = optarg;
                break;
            case 'c':
                c_flag = 1;
                break;
            case 2: /* --count */
                count_flag = 1;
                count_arg = optarg;
                break;
            case 3: /* --line */
                line_flag = 1;
                line_arg = optarg;
                break;
            case '?':
                if (optopt == 'b')
```



```

        fprintf(stderr, "-b option must have an argument!..\n");
    else if (optopt == 2)
        fprintf(stderr, "argument must be specified with --count option\n");
    else if (optopt != 0)
        fprintf(stderr, "invalid option: -%c\n", optopt);
    else
        fprintf(stderr, "invalid long option!..\n");

    err_flag = 1;

    break;
}
}

if (err_flag)
    exit(EXIT_FAILURE);

if (a_flag)
    printf("-a option used...\n");

if (b_flag)
    printf("-b option used with argument \"%s\"...\n", b_arg);

if (c_flag)
    printf("-c option used...\n");

if (help_flag)
    printf("--help option used...\n");

if (count_flag)
    printf("--count option specified with \"%s\"...\n", count_arg);

if (line_flag) {
    if (line_arg != NULL)
        printf("--line option used with optional argument \"%s\"\n", line_arg);
    else
        printf("--line option used without optional argument...\n");
}

if (optind != argc) {
    printf("Seçeneksiz (yani normal) argümanlar:\n");
    for (i = optind; i < argc; ++i)
        printf("%s\n", argv[i]);
}

return 0;
}

```

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

```

struct option options[] = {
    {"help", no_argument, &help_flag, 1},
    {"count", required_argument, NULL, 2},
    {"line", optional_argument, NULL, 3},
    {0, 0, 0, 0 },
};

```

Bu yapıda birinci uzun seçeneğin isminin "help" olduğu görülmektedir. Bu uzun seçenek argüman almaz. Birinci seçeneğin flag değişkenine help_flag nesnesinin adresinin atandığını görüyorsunuz. Dolayısıyla bu flag değişkeni getopt_long fonksiyonunun kendisi tarafından set edilecektir. Yapının ikinci elemanında uzun seçeneğin "count" isminde olduğuna ve bunun argüman almadığına dikkat ediniz. Kullanıcı bu uzun seçeneği girdiğinde getopt_long 2 değeri ile geri dönecektir. Nihayet yapının üçüncü elemanının "line" isminde uzun seçenek aldığı görülmektedir. Bu uzun seçenek

isteğe bağlı olarak (optional) arüman almaktadır. Bu uzun seçenek seçildiğinde optarg eğer uzun seçenek argüman ile kullanılmışsa argümanın adresini, argüman ile kullanılmamışsa NULL adres içerecektir.

Yukarıda da belirttiğimiz gibi bazen programcı kısa bir seçenekle aynı anlamda olan bir uzun seçenek de kullanabilmektedir. Bu durumda struct option yapısının ilgili elemanına val değeri olarak (yapının son elemanı) kısa seçenek harfinin sayısal değeri yerleştirilebilir. Böylece tek bir flag değişkeni tutularak bu değişkenin set edilmesi aynı yerde yapılabilir. Yukarıdaki örnekteki programda -help seçeneği ile -h seçeneğinin aynı anlama geldiğini varsayalım. Bu durumda program şu hale gelecektir:

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

int main(int argc, char *argv[])
{
    int result;
    int a_flag, b_flag, c_flag, h_flag, count_flag, line_flag;
    char *b_arg, *count_arg, *line_arg;
    int err_flag;
    int i;

    struct option options[] = {
        {"help", no_argument, NULL, 'h'},
        {"count", required_argument, NULL, 2},
        {"line", optional_argument, NULL, 3},
        {0, 0, 0, 0 },
    };

    opterr = 0;

    a_flag = b_flag = c_flag = h_flag = count_flag = line_flag = 0;
    err_flag = 0;

    while ((result = getopt_long(argc, argv, "ab:ch", options, NULL)) != -1) {
        switch (result) {
            case 'a':
                a_flag = 1;
                break;
            case 'b':
                b_flag = 1;
                b_arg = optarg;
                break;
            case 'c':
                c_flag = 1;
                break;
            case 'h':
                h_flag = 1;
                break;
            case 2: /* --count */
                count_flag = 1;
                count_arg = optarg;
                break;
            case 3: /* --line */
                line_flag = 1;
                line_arg = optarg;
                break;
            case '?':
                if (optopt == 'b')
                    fprintf(stderr, "-b option must have an argument!..\n");
                else if (optopt == 2)
                    fprintf(stderr, "argument must be specified with --count option\n");
                else if (optopt != 0)

```

```

        fprintf(stderr, "invalid option: -%c\n", optopt);
    else
        fprintf(stderr, "invalid long option!..\n");

    err_flag = 1;

    break;
}
}

if (err_flag)
    exit(EXIT_FAILURE);

if (a_flag)
    printf("-a option used...\n");

if (b_flag)
    printf("-b option used with argument \"%s\"...\n", b_arg);

if (c_flag)
    printf("-c option used...\n");

if (h_flag)
    printf("-h or --help option used...\n");

if (count_flag)
    printf("--count option specified with \"%s\"...\n", count_arg);

if (line_flag) {
    if (line_arg != NULL)
        printf("--line option used with optional argument \"%s\"\n", line_arg);
    else
        printf("--line option used without optional argument...\n");
}

if (optind != argc) {
    printf("Seçeneksiz (yani normal) argümanlar:\n");
    for (i = optind; i < argc; ++i)
        printf("%s\n", argv[i]);
}

return 0;
}

```

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

- 1) Fonksiyonda belirttiğiniz her argümanlı ya da argümansız seçenek için bir flag değişkeni, her argümanlı seçenek için de o argümanları gösteren bir gösterici tutabilirsiniz.
- 2) Hem uzun hem de kısa biçimi olan seçeneklerde yapının flag elemanı NULL değerinde tutulup fonksiyonun kısa seçenek ile geri dönmesi sağlanabilir.
- 3) Yalnızca uzun biçimi olan seçeneklerde yapının flag elemanına doğrudan o seçenek için tanımladığınız flag değişkeninin adresini atayabilirsiniz.
- 4) Fonksiyondan çıktıktan sonra switch içerisinde flag değişkenlerinin durumuna bakabilirsiniz.

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

```
mycat [seçenekler] [dosyalar]
```

Seçenekler:

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

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

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

/* Symbolic Constants */

#define DEF_LINE          10
#define HEX_OCTAL_LINE_LEN 16

/* Function Prototypes */

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

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

    opterr = 0;
    while ((result = getopt_long(argc, argv, "xoth", options, NULL)) != -1) {
        switch (result) {
            case 'x':
                x_flag = 1;
                break;
            case 'o':
                o_flag = 1;
                break;
            case 't':
                t_flag = 1;
                break;
            case 'h':
                header_flag = 1;
                break;
            case 1:
                top_flag = 1;
                top_arg = optarg;
                break;
        }
    }
}
```

```

        case '?':
        if (optopt != 0)
            fprintf(stderr, "invalid switch: -%c\n", optopt);
        else
            fprintf(stderr, "invalid switch: %s\n", argv[optind - 1]); /* argv[optind - 1]
dokümanete edilmemiş */
            err_flag = 1;
    }
}

if (err_flag)
    exit(EXIT_FAILURE);

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

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

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

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

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

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

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

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

    fclose(f);
}

return 0;
}

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

    if (nline == -1)
        while ((ch = fgetc(f)) != EOF)

```

```

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

return !ferror(f);
}

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

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

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

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

    return !ferror(f);
}

```

1.15. İşletim Sisteminin Sistem Fonksiyonları, POSIX Fonksiyonları, Windows API Fonksiyonları ve Standart C Fonksiyonları

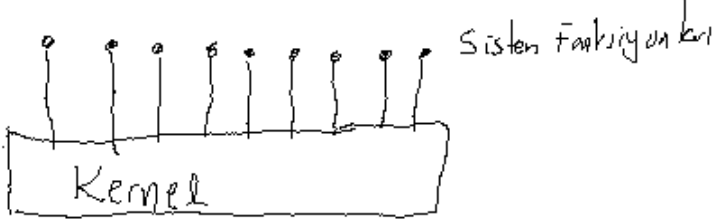
İşletim sistemleri genellikle prosedürel teknik kullanılarak C Programlama Diliyle yazılmaktadır. İşletim sistemleri kabaca çekirdek ve kabuk kısımlarından oluşur.



Çekirdek (kernel) işletim sisteminin ana işlevlerini gerçekleştiren motor kısmıdır. Kabuk (shell) ise işletim sisteminin kullanıcıyla arayüz oluşturan kısmıdır. Örneğin Windows'ta masaüstü, UNIX/Linux sistemlerinde bash gibi komut satırı

programları bu sistemlerin kabuk kısımlarını oluşturmaktadır. Tabii UNIX/Linux sistemlerinde de Windows'taki gibi grafik kabuk sistemleri de (pencere yöneticisi sistemler) bulunmaktadır.

İşletim sistemlerinin çekirdeklerinde binlerce fonksiyon bulunur. Bunların küçük bir kısmı dışarıdan da (kullanıcı modundan) önemli bazı işleri yapmak için çağrılabilir. Bunlara sistem fonksiyonları (system calls) denilmektedir. Her işletim sisteminin sistem fonksiyonlarının isimleri, parametrik yapıları farklı olabilmektedir. Yani işletim sistemlerinin sistem fonksiyonları taşınabilir değildir. Ancak ne olursa olsun biz C Programcısı olarak bu sistem fonksiyonlarını doğrudan çağırabiliriz.

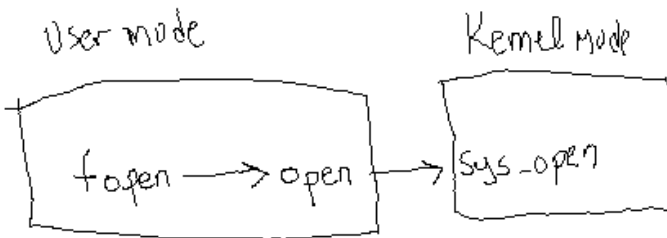


Programlarda sık karşılaşılan bazı işlemler aslında aşağı seviyede tamamen işletim sisteminin kontrolü altındadır. O faaliyetleri gerçekleştirmek isteyen herkes aslında eninde sonunda işletim sisteminin ilgili sistem fonksiyonlarını çağırmak zorundadır. Örneğin bir dosyayı silmek için bir sistem fonksiyonu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

POSIX standartları UNIX türevi sistemlerdeki kabuk komutlarını ve C'den çağrılacak ortak fonksiyonları belirlemektedir. Yani POSIX fonksiyonları UNIX türevi sistemlerdeki ortak fonksiyonlardır. Dolayısıyla POSIX fonksiyonları Linux gibi, BSD gibi, Solaris gibi hatta MAC OS X gibi sistemlerde aynı isimle ve aynı parametrik yapıyla kullanılabilir. Bazı POSIX fonksiyonları doğrudan o sistemdeki bir sistem fonksiyonu çağırır. Bazı POSIX fonksiyonları ise hiçbir sistem fonksiyonunu çağırmaz. Bazıları da birden fazla sistem fonksiyonunu çağırabilmektedir. Örneğin dosya açmak için open isimli bir POSIX fonksiyonu kullanılmaktadır. Bu fonksiyon Linux sistemlerinde sys_open isimli sistem fonksiyonunu çağırılmaktadır. Öter yandan getopt fonksiyonu herhangi bir sistem fonksiyonunu çağırmaz.

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

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



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

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

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

1.19. Proses Kavramı

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

Bir proses yaratıldığında (yani bir program çalıştırıldığında) işletim sistemi onu izlemek için çekirdek alanında bir veri yapısı oluşturur. Bu veri yapısına kavramsal olarak "Proses Kontrol Bloğu (Process Control Block)" denilmektedir. Örneğin Linux sistemlerinde process kontrol bloğu task_struct isimli yapıyla temsil edilmektedir.

Pekiye Proses Kontrol Bloğunda hangi bilgiler saklanmaktadır? İşte tipik bazı bilgiler şunlardır:

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

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

Aslında Proses Kontrol Bloğunu içerisinde göstericilerin de bulunduğu bir ağaç gibi (daha doğru bir deyişle graf gibi) düşünebiliriz. Yani proses kontrol bloğunun içerisindeki bazı elemanlar başka birtakım yapıları gösteriyor olabilir. O yapılar da başka yapıları gösteriyor olabilir. Biz kursumuzda bir bilginin proses kontrol bloğunda olduğunu söylediğimizde o bilginin doğrudan ya da dolaylı olarak proses kontrol bloğu yoluyla erişilebilecek bir yerde olduğunu kastetmiş olacağız.

Bir proses işletim sisteminin sistem fonksiyonuyla yaratılır ve işletim sisteminin bir sistem fonksiyonuyla sonlandırılır. Prosesler arasında altlık-üstlük (parent-child) ilişkisi de vardır. Prosesi yaratan procese üst-proses (parent process), yeni oluşturulan procese de alt-proses (child process) denilmektedir. Pek çok sistemde bir proses yaratıldığında üst prosesin proses kontrol bloğundaki bazı bilgiler alt prosesin proses kontrol bloğuna aktarılmaktadır. Örneğin bizim derleyerek çalışabilir hale getirdiğimiz "sample.exe" isimli programı çalıştırmış olalım. Bu "sample.exe" prosesi de "notepad.exe" programını çalıştırıp yeni bir prosesin yaratılmasına yol açıyor olabilir. İşte bu durumda "sample.exe" prosesi üst proses, "notepad.exe" prosesi ise alt proses olacaktır. Tabii alt prosesin de alt prosesleri söz konusu olabilir.

1.20. Macar Notasyonu ve Windows API Fonksiyonlarında Kullanılan typedef Türleri

Windows'un API fonksiyonlarının büyük çoğunluğunun prototipleri <windows.h> başlık dosyası içerisinde yer almaktadır. API fonksiyonlarında taşınabilirliği artırmak için çeşitli typedef isimleri kullanılmıştır. Bunların typedef bildirimleri de <windows.h> içerisinde bulunmaktadır.

Windows API fonksiyonlarının harflendirilmesinde (capitalization) ve isimlendirilmesinde Macar Notasyonu (Hungarian Notation) denilen bir notasyon kullanılmıştır. (Bu notasyona Macar Notasyonu denmesinin nedeni bu notasyonu geliştiren Charles Simonyi'nin Macar olmasından kaynaklanmaktadır. Charles Simonyi Microsoft'un erken dönem çalışanlarından ve uzunca bir süre Office paketinin proje yöneticiliğini yapmıştır.)

Macar notasyonunun anahtar özellikleri şöyledir:

- Fonksiyonlar Pascal stili ile isimlendirilmiştir. (Yani her sözcüğün ilk harfi büyüktür. Örneğin CreateFile, FindFirstFile gibi.) Genellikle önce eylem sonra onun nesnesi gelir. (Örneğin CreateWindow, CreateProcess, OpenProcess, CloseHandle gibi.)

- Değişken isimleri onların türlerini belirten küçük harfli örneklerle başlatılmıştır. Tipik kullanılan örnekler şunlardır:

p ya da lp	Gösterici (lp "long pointer" dan gelme. Eskiye uyum için hala kullanılıyor.)
l	long
w	WORD
dw	DWORD
h	HANDLE (genel olarak void *)
sz	char * (yazıyı gösteren gösterici, "sz" "zero terminating string" sözcüklerinden geliyor.)
b	BOOL
f	float
d	double

int türü için genel olarak örnek kullanılmaz.

- Yapı isimleri yine yapıyı temsil eden örneklerle başlatılır. Örneğin:

```
RECT rectWindow;  
POINT ptRef;
```

- Yerel ve global değişkenlerin (fonksiyonların değil) harflendirmesinde "deve notasyonu (camel casting)" kullanılmaktadır. Eğer değişken tür belirten bir örnek içeriyorsa bu örnek küçük harfle yazılır. Sonraki sözcüklerin ilk harfleri büyük yazılır. Eğer değişken tür belirten bir örnek içermiyorsa değişkenin ilk sözcüğü küçük harflerle sonraki sözcüklerin yalnızca ilk harfleri büyük harflerle harflendirilir. Örneğin:

```
dwNumberOfSectors  
rectWindow  
szName  
studentId
```

Global deęişkenlerin farkedilmesi için "g_" gibi bir ekle öneklendirilmesi çok kullanılan bir yöntemdir. Örneęin:

```
g_dwNumberOfSectors  
g_dwCount
```

- typedef tür isimleri büyük harflerle isimlendirilmektedir.

Şimdi Windows API programlamasında kullanılan typedef isimlerinin üzerinde duralım. Aşaęıda en çok kullanılan typedef isimlerini görüyorsunuz:

BYTE	Bir byte'lık işaretli tamsayı türü (unsigned char)
WORD	İki byte'lık işaretli tamsayı türü (tipik olarak unsigned short int)
DWORD	Dört byte'lık işaretli tamsayı türü (duruma göre unsigned long int ya da unsigned int olabilir)
QWORD	Sekiz byte'lık işaretli tamsayı türü (tipik olarak unsigned long long int)
HANDLE	Handle türü (void *)
PXXX, LPXXX	XXX türünden adres türü (örneğin LPVOID, PVOID, LPDWORD, LPCHAR)
PCXXX, LPCXXX	XXX türünden gösterdiği yer const olan adres (Örneğin LPCVOID tür ismi const void * anlamına gelmektedir.)
PSTR, LPSTR	Yazıyı gösteren adres (char *)
LPTSTR, LPCTSTR	Yazıyı gösteren UNICODE destekli adres (duruma göre char * ya da wchar_t *)
BOOL	int türünü belirtir. Fakat anlam olarak başarı ve başarısızlık düşünülmelidir. Geri dönüş değeri BOOL olan API fonksiyonları başarı durumunda sıfır dışı değere, başarısızlık durumunda sıfır değerine geri dönerler.
INT_PTR	O sistemdeki bir göstericinin uzunluğu kadar uzunlukta olan işaretli tamsayı türü
INT, LONG, ...	Pek çok standart tür ayrıca büyük harflerle de typedef edilmiştir
VOID, PVOID, LPVOID, PCVOID, LPCVOID	VOID void türünü belirtir. PVOID ve LPVOID void * biçiminde PCVOID ve LPCVOID ise const void * biçiminde typedef edilmiştir

Bia ara Microsoft API fonksiyonlarının prototiplerinde gösterici olan parametre deęişkenlerinin önünde __in, __out ve __in_out sözcüklerini kullanıyordu. Bunlar okunabilirliği artırmak için düşünülmüştü. Bu makrolar boş bir biçimde aşağıdaki gibi define edilmişlerdi:

```
#define __in  
#define __out  
#define __in_out
```

Yani bu makrolar önışlemci tarafından silinmekteydi. __in makrosu fonksiyonun parametre deęişkenindeki bilgiyi kullanacağı fakat ona bir değeri yerleştirmeyeceęi anlamına geliyordu. __out tam tersine fonksiyonun parametre deęişkenindeki değeri deęiştireceęi anlamına geliyordu. __in_out ise fonksiyonun hem parametre deęişkenindeki değeri kullanacağı hem de ona yeni bir değeri yerleştireceęi anlamına geliyordu. Fakat Microsoft birkaç sene önce bu yazımı terk etti. Zaten bu makroların gereklilięi de tartışmalıdır. Çünkü gösterici olmayan parametre deęişkenleri __in olmak zorundadır. Gösterici parametre deęişkenlerinde __in ya da __out durumu göstericinin const olup olmamasıyla zaten anlaşılmaktadır. O halde bunun tek faydası __in_out durumu için olabilir. Yukarıda da belirttiğimiz gibi Microsoft yeni başlık dosyalarında artık bu makroları kullanmamaktadır.

1.21. Fonksiyonlar İçin Hata Kontrolleri

Hata kontrolü bakımından fonksiyonları iki gruba ayırabiliriz:

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

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

1.22. Windows API Fonksiyonlarının Başarısızlık Nedenlerinin Elde Edilmesi

Windows sistemlerinde bir API fonksiyonu başarısız olduğunda onun hangi nedenden dolayı başarısız olduğu bize doğrudan verilmez. Her başarısızlık için DWORD türden bir değer tanımlanmıştır. Son çağrılan API fonksiyonunun başarısızlık nedeni GetLastError isimli API fonksiyonu çağrılarak elde edilmektedir:

```
DWORD GetLastError(void);
```

Hangi hata kodlarının ne anlama geldiği MSDN yardım dokümanlarında belirtilmiştir. Aşağıda ilgili MSDN dokümanından bir parça görüyorsunuz:

Constant/value	Description
ERROR_SUCCESS 0	The operation completed successfully.
ERROR_INVALID_FUNCTION 1	Incorrect function.
ERROR_FILE_NOT_FOUND 2	The system cannot find the file specified.
ERROR_PATH_NOT_FOUND 3	The system cannot find the path specified.
ERROR_TOO_MANY_OPEN_FILES 4	The system cannot open the file.
ERROR_ACCESS_DENIED 5	Access is denied.

Windows'ta ayrıca her hata kodu <windows.h> içerisindeki ERROR_XXXX biçiminde bir sembolik sabitle de define edilmiştir. 0 (sıfır) değeri özel bir anlama gelir ve şöyle define edilmiştir:

```
#define ERROR_SUCCESS 0
```

GetLastError için 0 diğeri kimi zaman "hata olmama durumunu" belirtmektedir. Ancak başarılı bir fonksiyonun "last error" değerini 0'a çekmesi zorunlu değildir. Bu nedenle GetLastError fonksiyonunun ancak fonksiyon başarısızsa çağrılması tavsiye edilmektedir. Fonksiyonun başarılı olduğu durumda bu fonksiyonu çağırdığımızda hata kodu olarak 0 değerini almak zorunda değiliz. Ayrıca bazı API fonksiyonları başarılı olduğunda bile "last error" değerini set edebilmektedir. Ancak bu durum çok seyrekdir. Ayrıca programlarda hatanın sayısal kodu kullanıcı için pek bir anlam ifade etmemektedir. İşte bir hata koduna karşılık onun hata yazısını veren FormatMessage isimli bir API fonksiyonu da vardır. Bu fonksiyonun kullanımı biraz detaylıdır. Burada bu detay ele alınmayacaktır. Biz kursumuzda bir API fonksiyonu başarısız olduğunda hatayı GetLastError ile alan ve bunu FormatMessage API fonksiyonuna vererek hata yazısını elde eden bunu da stderr dosyasına yazdırıp programı sonlandıran ExitSys isimli bir fonksiyonu kullanacağız:

```
void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}
```

```
}
```

ExitSys fonksiyonunun bir API fonksiyonu olmadığına bizim tarafımızdan yazılmış yardımcı bir fonksiyon olduğuna dikkat ediniz. Örneğin bir API fonksiyonu başarısız olduğunda hata mesajı ExitSys ile şöyle yazdırılabilir:

```
HANDLE hFileFind;
WIN32_FIND_DATA finfo;
...
if ((hFileFind = FindFirstFile("test.c", &finfo)) == INVALID_HANDLE_VALUE)
    ExitSys("FindFirstFile");
```

ExitSys fonksiyonu hata mesajını şu kalıpla stderr dosyasına yazdırmaktadır:

```
xxxxx: hata yazısı
```

Burada ‘:’ karakterinin solundaki xxxxx yazısı ExitSys fonksiyonuna argüman olarak geçirdiğimiz yazıdır. ExitSys fonksiyonunun programı sonlandırdığına dikkat ediniz.

1.23. UNIX/Linux Sistemlerindeki POSIX Fonksiyonlarının Başarısızlık Nedenlerinin Elde Edilmesi

POSIX fonksiyonlarının çok büyük çoğunluğunun geri dönüş değeri int türündendir. Bu int türden geri dönüş değeri bize fonksiyonun başarılı mı yoksa başarısız mı olduğu bilgisini verir. POSIX fonksiyonları başarı durumunda sıfır (dikkat ediniz), başarısızlık durumunda ise -1 değerine geri dönerler. Böylece tipik olarak bir POSIX fonksiyonunun başarısızlığı şöyle tespit edilmektedir:

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

Fakat bazı programcılar kontrolü aşağıdaki gibi de yapabilmektedir:

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

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

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

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

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

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

Ya da örneğin şöyle bir kod da yazabilir:

```

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

```

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

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

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

```

#include <string.h>

char *strerror(int errnum);

```

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

```

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

int main(void)
{
    int fd;

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

    return 0;
}

```

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

```

#include <stdio.h>

void perror(const char *s);

```

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

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

Örneğin:

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

int main(void)
{
    int fd;

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

    return 0;
}
```

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

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

    exit(EXIT_FAILURE);
}
```

exit_sys fonksiyonunun kullanımı da şöyle olabilir:

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;

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

    printf("Ok\n");

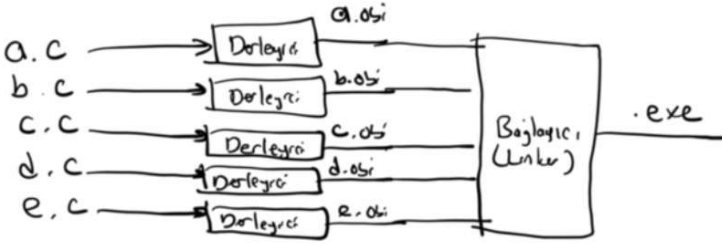
    return 0;
}

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

```
    exit(EXIT_FAILURE);  
}
```

1.23. C Programlarında Dosya Organizasyonu

Orta ve büyük çaplı C projeleri tek bir kaynak dosya ile oluşturulmamalı, birden fazla kaynak dosya ile düzenlenmelidir. Böylece bu kaynak dosyalar ayrı ayrı derlenip birlikte bağlama (link) işlemine sokulurlar. Her defasında yalnızca üzerinde değişiklik yapılmış olan dosyaların derlenmesi ve derlenmiş olan bu dosyaların hep beraber bağlanması işlemini otomatize etmek için çeşitli "build" araçlarından faydalanılmaktadır. Bu konu ileride ele alınacaktır.



Eğer biz IDE ile çalışıyorsak değişen dosyaların yeniden derlenip hep birlikte bağlanması işlemini IDE kendisi arka planda yapmaktadır. Yani biz yalnızca bu IDE'lerde projeye yeni bir .c dosyası eklediğimizde her şey arka planda otomatik yapılmaktadır.

Bir projeyi oluşturan ve bağımsız olarak derlenen her bir C dosyasına modül (module) denilmektedir. Tipik olarak bir modül iki dosya biçiminde organize edilebilir: Çeşitli bildirimlerin ve önışlemci komutlarının bulunduğu bir başlık dosyası ve fonksiyon kodlarının bulunduğu bir kaynak dosya. Bu çeşit çalışma özellikle kütüphanelerin oluşturulması sürecinde de sıklıkla uygulanmaktadır.

Başlık dosyalarının içerisinde önışlemci komutları, fonksiyon prototipleri, extern bildirimleri, yapı bildirimleri gibi nesne yaratmayan bildirim işlemleri bulundurulur. Başlık dosyasının doğrudan ya da dolaylı biçimde birden fazla kez include edilmesini engellemek için mutlaka include koruması uygulanmalıdır. Başlık dosyasının ismi "x.h" olmak üzere, include koruması tipik olarak şöyle uygulanabilir:

```
#ifndef X_H_  
#define X_H_  
....  
#endif
```

Burada X_H_ dosya isminden hareketle uydurulmuş olan bir makro isimdir. Başlık dosyası ilk kez önışlemci tarafından ele alındığında bu makro define edilmediği için koşul içerisindeki kodlar derleme işlemine sokulacaktır. Ancak dosya önışlemci tarafından daha sonra ele alınırsa artık bu makro tanımlanmış olduğu için dosyanın içeriği derleme işlemine sokulmayacaktır.

include korumaları için Microsoft ve gcc derleyicileri dahil olmak üzere pek çok derleyicide #pragma once isimli özel bir pragma direktifi de bulunmaktadır. Bu direktif şöyle kullanılmaktadır:

```
#pragma once  
....
```

Başlık dosyalarının başka başlık dosyalarına gereksinim duymaması gerekmektedir. Dolayısıyla bir başlık dosyasının içerisinde kullanılan programcı tanımlı türler vs. eğer başka bir başlık dosyası içerisinde bildirilmişse bunların bildirildiği başlık dosyasının bunları kullanan başlık dosyasının içerisinde include edilmesi uygun olur. Örneğin:

```
#ifndef X_H_  
#define X_H_  
  
#include <stddef.h>
```

```
size_t GetCount(void);
```

```
/* ... */
```

```
#endif
```

Burada size_t türü için <stddef.h> dosyası "x.h" başlık dosyasının içerisinde include edilmiştir. Böylece söz konusu "x.h" dosyası başka bir dosyaya bağımlı olmayacaktır.

static fonksiyonların prototipleri başlık dosyalarında bulundurulmamalıdır. Bunlar modüle özgü oldukları için kaynak dosya içerisinde bulundurulmalıdırlar. Başlık dosyalarında asla nesne yaratan tanımlamaların bulunmaması gerektiğini biliyorsunuz. (Çünkü bu durumda başlık dosyası farklı modüllerden include edildiğinde aynı nesnenin birden fazla kez tanımlanması söz konusu olmaktadır.)

Başlık dosyaları hem ana kaynak dosyadan hem de diğer gereken kaynak dosyalardan include edilebilirler. Zaten bizim ilgili modülü iki dosya biçiminde organize etmemizin temel nedeni budur. Örneğin:

```
/* a.h */
```

```
#ifndef A_H_
```

```
#define A_H_
```

```
#include <stddef.h>
```

```
/* Symbolic Constants */
```

```
#define ASCENDING 0
```

```
#define DESCENDING 1
```

```
/* Function Prototypes */
```

```
void sort(int *pi, size_t size);
```

```
#endif
```

```
/* a.c */
```

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

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

```
#include "a.h"
```

```
void sort(int *pi, size_t size, int order)
```

```
{
```

```
    size_t i, k;
```

```
    int temp;
```

```
    for (i = 0; i < size - 1; ++i)
```

```
        for (k = 0; k < size - 1 - i; ++k) {
```

```
            if (order == ASCENDING) {
```

```
                if (pi[k] > pi[k + 1]) {
```

```
                    temp = pi[k];
```

```
                    pi[k] = pi[k + 1];
```

```
                    pi[k + 1] = temp;
```

```
                }
```

```
            }
```

```
            else
```

```
                if (pi[k] < pi[k + 1]) {
```

```
                    temp = pi[k];
```

```
                    pi[k] = pi[k + 1];
```

```
                    pi[k + 1] = temp;
```

```
                }
```

```
        }
```

```
}
```



```

/* b.h */

#ifndef B_H_
#define B_H_

#include <stddef.h>

/* Function Prototypes */

int getmax(const int *pi, size_t size);

#endif

/* b.c */

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

int getmax(const int *pi, size_t size)
{
    int max = pi[0];
    size_t i;

    for (i = 1; i < size; ++i)
        if (pi[i] > max)
            max = pi[i];

    return max;
}

/* app.c */

#include <stdio.h>
#include "a.h"
#include "b.h"

int main(void)
{
    int a[10] = { 34, 56, 23, 78, 12, 90, 69, 22, 54, 18 };
    int i, max;

    max = getmax(a, 10);
    printf("%d\n", max);

    sort(a, 10, ASCENDING);
    for (i = 0; i < 10; ++i)
        printf("%d ", a[i]);
    printf("\n");

    return 0;
}

```

1.24. Handle Kavramı ve Handle Sistemleri

Handle bir veri yapısına erişmekte kullanılan tekil bir anahtar değeridir. Handle bir tamsayı biçiminde olabilir. Bu durumda muhtemelen global bir dizide bir indeks belirtiyor durumdadır. Handle bir adres biçiminde olabilir. Bu durumda doğrudan bir veri yapısını gösteriyor olabilir. Bazen handle void bir adres olarak karşımıza çıkar. (Örneğin Windows API fonksiyonlarında sıkça karşılaştığımız HANDLE tür ismi aslında void * olarak typedef edilmiştir.) Bu durumda aslında o adresin gösterdiği yerde bir veri yapısı vardır. Fakat sistemi tasarlayan kişi bunu açıklamak istememiştir. Bu nedenle programcıya handle değerini sanki void bir adres gibi vermiştir. Bazen handle bozulmuş (şifrelenmiş de diyebiliriz) bir

biçimde de bize verilebilir. Sistem onu düzelterek ilgili veri yapısına erişir. Böylece handle değerine sahip olan kişi oraya doğrudan erişemez. Handle ile erişilen veri yapısına handle alanı denilmektedir:



Bir handle sisteminde üç grup fonksiyon bulunur.

1) Handle sistemlerini yaratan ya da açan fonksiyonlar: (Bunlar genellikle Windows sistemlerinde CreateXXX ya da OpenXXX biçiminde isimlendirilmektedir.) Bu fonksiyonlar veri yapısını (handle alanını) tahsis ederler. Onun elemanlarına çeşitli ilkdeğerleri verirler ve handle değeri ile geri dönerler.

2) Handle sistemini kullanan fonksiyonlar: Bunlar handle değerini bizden alıp hedef veri yapısına erişerek oradaki bilgileri kullanan fonksiyonlardır. Bizim için faydalı birtakım işlemleri yaparlar.

3) Handle sistemini kapatan fonksiyonlar: Bunlar handle alanını boşaltıp birtakım son işlemleri yaparlar. (Windows sistemlerinde genellikle bu fonksiyonlar CloseXXX ya da DestroyXXX biçiminde isimlendirilmektedir.)

Bir handle sisteminde programcının handle alanını bilmesi gerekmez. (Hatta sistemi tasarlayanlar yukarıda da belirtildiği gibi handle alanını gizlemeye bile çalışabilirler.) Programcı yalnızca handle sistemini yaratan ya da açan fonksiyondan elde edilen handle değerini handle sistemini kullanan fonksiyonlara parametre olarak geçirir. Böylece hedeflediği işlemleri bu fonksiyonlara yaptırmış olur.

Handle sistemine çeşitli örnekler verilebilir. Örneğin standart C'nin dosya fonksiyonları böyle bir sistemi kullanmaktadır. Örneğin C'nin standart kütüphanesindeki fopen bu bağlamda handle sistemini yaratan bir fonksiyondur. Bize handle değerini FILE * biçiminde verir. fgets, fputs, fread gibi fonksiyonlar handle sistemini kullanan fonksiyonlardır. fclose da handle sistemini kapatan fonksiyondur.

Örneğin bir matris soyut veri yapısını (abstract data structure) bir handle sistemi biçiminde aşağıdaki gibi oluşturabiliriz:

```
/* matrix.h */

#ifndef MATRIX_H_
#define MATRIX_H_

#include <stddef.h>

typedef int DATATYPE;

/* Type Declarations */

typedef struct tagMATRIX {
    DATATYPE *pMatrix;
    size_t rowSize;
    size_t colSize;
} MATRIX, *HMATRIX;

/* Function Prototypes */

HMATRIX CreateMatrix(size_t rowSize, size_t colSize);
void SetMatrix(HMATRIX hMatrix, const DATATYPE *vals);
DATATYPE GetElem(HMATRIX hMatrix, size_t row, size_t col);
void PutElem(HMATRIX hMatrix, size_t row, size_t col, DATATYPE val);
void DispMatrix(HMATRIX hMatrix);
void CloseMatrix(HMATRIX hMatrix);
```

```

#endif

/* matrix.c */

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

/* Function Definitions */

HMATRIX CreateMatrix(size_t rowSize, size_t colSize)
{
    HMATRIX hMatrix;

    if ((hMatrix = (HMATRIX)malloc(sizeof(MATRIX))) == NULL)
        return NULL;

    if ((hMatrix->pMatrix = (DATATYPE *)malloc(rowSize * colSize * sizeof(DATATYPE))) == NULL) {
        free(hMatrix);
        return NULL;
    }
    hMatrix->rowSize = rowSize;
    hMatrix->colSize = colSize;

    return hMatrix;
}

void SetMatrix(HMATRIX hMatrix, const DATATYPE *vals)
{
    size_t i;

    for (i = 0; i < hMatrix->rowSize * hMatrix->colSize; ++i)
        hMatrix->pMatrix[i] = vals[i];
}

DATATYPE GetElem(HMATRIX hMatrix, size_t row, size_t col)
{
    return hMatrix->pMatrix[hMatrix->colSize * row + col];
}

void PutElem(HMATRIX hMatrix, size_t row, size_t col, DATATYPE val)
{
    hMatrix->pMatrix[hMatrix->colSize * row + col] = val;
}

void DispMatrix(HMATRIX hMatrix)
{
    size_t i;

    for (i = 0; i < hMatrix->rowSize * hMatrix->colSize; ++i)
        printf("%-5d%c", hMatrix->pMatrix[i], i % hMatrix->colSize == hMatrix->colSize - 1 ? '\n' : ' ');
}

void CloseMatrix(HMATRIX hMatrix)
{
    free(hMatrix->pMatrix);
    free(hMatrix);
}

/* app.c */

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

```

```

int main(void)
{
    HMATRIX hMatrix;
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    if ((hMatrix = CreateMatrix(3, 3)) == NULL) {
        fprintf(stderr, "Cannot create matrix!..\n");
        exit(EXIT_FAILURE);
    }

    SetMatrix(hMatrix, a);
    DispMatrix(hMatrix);

    printf("\n\n");

    PutElem(hMatrix, 1, 2, 100);
    DispMatrix(hMatrix);

    printf("%d\n", GetElem(hMatrix, 1, 1));

    CloseMatrix(hMatrix);

    return 0;
}

```

Handle alanı istenirse onu kullananlardan gizlenebilir. Bunun için fonksiyonlar derlenerek bir kütüphaneye yerleştirilir. Başlık dosyasına ise prototipler yazılır. Ancak handle alanı bu başlık dosyasında bildirilmez. Handle türü void * olarak alınır. Matris veri yapısı bu biçimde aşağıdaki gibi düzenlenebilir:

```

/* matrix.h */

#ifndef MATRIX_H_
#define MATRIX_H_

#include <stddef.h>

typedef int DATATYPE;

/* Type Declarations */

typedef void *HMATRIX;

/* Function Prototypes */

HMATRIX CreateMatrix(size_t rowSize, size_t colSize);
void SetMatrix(HMATRIX hMatrix, const DATATYPE *vals);
DATATYPE GetElem(HMATRIX hMatrix, size_t row, size_t col);
void PutElem(HMATRIX hMatrix, size_t row, size_t col, DATATYPE val);
void DispMatrix(HMATRIX hMatrix);
void CloseMatrix(HMATRIX hMatrix);

#endif

/* Matrix.c */

/* matrix.c */

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

/* Type Declarations
*/
typedef struct tagMATRIX {
    DATATYPE *pMatrix;
    size_t rowSize;

```

```

    size_t colSize;
} MATRIX;

/* Function Definitions */

HMATRIX CreateMatrix(size_t rowSize, size_t colSize)
{
    MATRIX *hMatrix;

    if ((hMatrix = (HMATRIX)malloc(sizeof(MATRIX))) == NULL)
        return NULL;

    if ((hMatrix->pMatrix = (DATATYPE *)malloc(rowSize * colSize * sizeof(DATATYPE))) == NULL) {
        free(hMatrix);
        return NULL;
    }
    hMatrix->rowSize = rowSize;
    hMatrix->colSize = colSize;

    return hMatrix;
}

void SetMatrix(HMATRIX hMatrix, const DATATYPE *vals)
{
    MATRIX *matrix = (MATRIX *)hMatrix;
    size_t i;

    for (i = 0; i < matrix->rowSize * matrix->colSize; ++i)
        matrix->pMatrix[i] = vals[i];
}

DATATYPE GetElem(HMATRIX hMatrix, size_t row, size_t col)
{
    MATRIX *matrix = (MATRIX *)hMatrix;

    return matrix->pMatrix[matrix->colSize * row + col];
}

void PutElem(HMATRIX hMatrix, size_t row, size_t col, DATATYPE val)
{
    MATRIX *matrix = (MATRIX *)hMatrix;

    matrix->pMatrix[matrix->colSize * row + col] = val;
}

void DispMatrix(HMATRIX hMatrix)
{
    MATRIX *matrix = (MATRIX *)hMatrix;
    size_t i;

    for (i = 0; i < matrix->rowSize * matrix->colSize; ++i)
        printf("%-5d%c", matrix->pMatrix[i], i % matrix->colSize == matrix->colSize - 1 ? '\n' : ' ');
}

void CloseMatrix(HMATRIX hMatrix)
{
    MATRIX *matrix = (MATRIX *)hMatrix;

    free(matrix->pMatrix);
    free(matrix);
}

/* matrix.c */

#include <stdio.h>

```

```

#include <stdlib.h>
#include "matrix.h"

/* Type Declarations
*/
typedef struct tagMATRIX {
    DATATYPE *pMatrix;
    size_t rowSize;
    size_t colSize;
} MATRIX;

/* Function Definitions */

HMATRIX CreateMatrix(size_t rowSize, size_t colSize)
{
    MATRIX *hMatrix;

    if ((hMatrix = (HMATRIX)malloc(sizeof(MATRIX))) == NULL)
        return NULL;

    if ((hMatrix->pMatrix = (DATATYPE *)malloc(rowSize * colSize * sizeof(DATATYPE))) == NULL) {
        free(hMatrix);
        return NULL;
    }
    hMatrix->rowSize = rowSize;
    hMatrix->colSize = colSize;

    return hMatrix;
}

void SetMatrix(HMATRIX hMatrix, const DATATYPE *vals)
{
    MATRIX *matrix = (MATRIX *)hMatrix;
    size_t i;

    for (i = 0; i < matrix->rowSize * matrix->colSize; ++i)
        matrix->pMatrix[i] = vals[i];
}

DATATYPE GetElem(HMATRIX hMatrix, size_t row, size_t col)
{
    MATRIX *matrix = (MATRIX *)hMatrix;

    return matrix->pMatrix[matrix->colSize * row + col];
}

void PutElem(HMATRIX hMatrix, size_t row, size_t col, DATATYPE val)
{
    MATRIX *matrix = (MATRIX *)hMatrix;

    matrix->pMatrix[matrix->colSize * row + col] = val;
}

void DispMatrix(HMATRIX hMatrix)
{
    MATRIX *matrix = (MATRIX *)hMatrix;
    size_t i;

    for (i = 0; i < matrix->rowSize * matrix->colSize; ++i)
        printf("%-5d%c", matrix->pMatrix[i], i % matrix->colSize == matrix->colSize - 1 ? '\n' : ' ');
}

void CloseMatrix(HMATRIX hMatrix)
{
    MATRIX *matrix = (MATRIX *)hMatrix;

```

```

    free(matrix->pMatrix);
    free(matrix);
}

/* app.c */

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

int main(void)
{
    HMATRIX hMatrix;
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    if ((hMatrix = CreateMatrix(3, 3)) == NULL) {
        fprintf(stderr, "Cannot create matrix!..\n");
        exit(EXIT_FAILURE);
    }

    SetMatrix(hMatrix, a);
    DispMatrix(hMatrix);

    printf("\n\n");

    PutElem(hMatrix, 1, 2, 100);
    DispMatrix(hMatrix);

    printf("%d\n", GetElem(hMatrix, 1, 1));

    CloseMatrix(hMatrix);

    return 0;
}

```

Handle sistemleri nesne yönelimli programlama tekniğinde sınıflarla oldukça kolay biçimde temsil edilebilirler. Başka bir deyişle biz nesne yönelimli teknikte bir handle sistemini bir sınıf biçiminde oluşturabiliriz. Şöyle ki: Handle alanı sınıfın private veri elemanları olarak bildirilir. Sınıfın üye fonksiyonları (metotları) bunları ortak kullanmaktadır. Sınıfın başlangıç fonksiyonu (constructor) handle alanında tahsisatlar yaparak elemanlara ilkdeğerlerini verir. Sınıfın bitiş fonksiyonu (destructor) ise handle alanı yok edilirken gereken son işlemleri yapar.

1.25. C'de typedef Bildirimlerine Yeniden Bakış

C'de typedef anahtar sözcüğü "storage class specifier" olarak gramere yerleştirilmiştir. Bu nedenle her türlü bildirimde typedef belirleyicisi getirilebilir. Bir bildirimde typedef belirleyicisini eklersek artık o bildirimdeki değişken ismi, o değişkenin türüne ilişkin tür ismi haline gelir. Örneğin:

```
int I;
```

I burada int türden bir değişken ismidir. Fakat:

```
typedef int I;
```

Burada I artık int türünü temsil eden bir tür ismi haline gelmiştir. typedef bildirimleri de yerel ya da global düzeyde yapılabilir. Örneğin:

```
int A, B, C;
```

Burada A, B, C int türden değişkenlerdir. Fakat:

```
typedef int A, B, C;
```

Burada hem A, hem B, hem C int türünü temsil etmektedir. Örneğin:

```
int *PI;
```

Burada PI "int *" türündendir. Yani int türden bir adres bilgisini tutan bir nesnesid. Fakat:

```
typedef int *PI;
```

Burada PI "int *" türünü temsil etmektedir. Yani:

```
int *pi;
```

ile

```
PI pi;
```

aynı anlamdadır.

Mademki C'de belirleyiciler bildirimde herhangi bir sırada yazılabiliyor o halde typedef anahtar sözcüğünün başa getirilmesi de zorunlu değildir. Örneğin:

```
int typedef I;
```

Örneğin:

```
int ARY[10];
```

Burada ARY "int[10]" (10 elemanlı int türden dizi türünden) türündendir. Fakat:

```
typedef int ARY[10];
```

Burada ARY "int[10]" türü anlamına gelmektedir. Yani:

```
int a[10];
```

ile,

```
ARY a;
```

aynı anlamdadır. Örneğin:

```
struct tagCMD {  
    ...  
} CMD;
```

Burada CMD "struct tagCMD" türündendir. Fakat:

```
typedef struct tagCMD {  
    ...  
} CMD;
```

Burada CMD "struct tagCMD" türünü temsil etmektedir.

Fonksiyon prototipine typedef uygulanabilir. Örneğin:

```
typedef void F(double);
```


Burada F geri dönüş değeri void, parametresi double olan bir fonksiyon prototipini temsil etmektedir.

```
F foo, bar;
```

Bu bildirim eşdeğeri:

```
void foo(double);  
void bar(double);
```

typedef bildirimlerinin fonksiyon göstericileriyle kullanılması "Fonksiyon Göstericileri"nin anlatıldığı bölümde ele alınacaktır.

1.26. Fonksiyonların Çağırma Biçimleri (Calling Conventions)

Fonksiyonların çağırma biçimleri C standartlarında yer alan bir konu değildir. Derleyicilerde birer eklenti (extension) biçiminde bulunmaktadır. Hem Microsoft derleyicilerinde hem de gcc ve clang derleyicilerinde fonksiyonların çağırma biçimleri ek birtakım anahtar sözcüklerle temsil edilmektedir.

Fonksiyon çağırma biçimi fonksiyon parametlerinin fonksiyona nasıl aktarılacağı ve geri dönüş değerlerinin nasıl elde edileceği hakkındaki belirlemelerden oluşmaktadır. Her çağırma biçimi bir isimle temsil edilmektedir. Windows sistemlerinde çağırma biçimleri fonksiyon isminin hemen soluna (yani geri dönüş değeri ile fonksiyon isminin arasına) getirilen anahtar sözcüklerle belirlenir. gcc ve clang derleyicilerinde ise çağırma biçimleri geri dönüş değerlerinin solunda `__attribute__((...))` sentaksıyla belirtilmektedir. Genel olarak Intel işlemcileri için Windows sistemleriyle UNIX/Linux sistemlerindeki çağırma biçimleri benzerdir. Ancak bu çağırma biçimleri 32 bit ve 64 bit mimarilerde farklılıklar göstermektedir. Biz burada yalnızca Intel'in işlemcileri için kullanılan çağırma biçimlerinin temel özelliklerini belirteceğiz. Çağırma biçimi (calling convention) konusunun ayrıntıları ancak iyi bir sembolik makine dili bilgisi ile tam olarak anlaşılabilir. Derneğimizde "80X86 ve ARM Sembolik Makine Dili" kurslarında bu konu ayrıntılarıyla ele alınmaktadır.

x86 cdecl Çağırma Biçimi: Bu Microsoft, gcc ve clang C derleyicileri için default çağırma biçimidir. Yani fonksiyon prototipinde ya da tanımlamasında çağırma biçimi hiç belirtilmemişse sanki "cdecl" belirtilmiş gibi işlem görür. Bu çağırma biçimi Microsoft derleyicilerinde `__cdecl` anahtar sözcüğü ile, gcc ve clang derleyicilerinde `__attribute__((cdecl))` anahtar sözcükleriyle belirtilmektedir. cdecl çağırma biçiminde fonksiyon parametreleri sağdan sola stack'e atılır ve geri dönüş değerleri yazmaçlardan alınır. Stack'in dengelenmesi çağırılan fonksiyonun (caller) sorumluluğundadır.

x86 stdcall Çağırma Biçimi: Windows'un bütün API fonksiyonları ve özel birtakım fonksiyonlar hep bu çağırma biçimiyle derlenmişlerdir. Bu çağırma biçimi Windows derleyicilerinde `__stdcall`, gcc ve clang derleyicilerinde ise `__attribute__((stdcall))` anahtar sözcükleriyle belirtilmektedir. stdcall çağırma biçiminde yine parametreler sağdan sola stack'e atılır. Geri dönüş değerleri yine yazmaçlardan elde edilir. Ancak stack'in dengelenmesi çağırılan fonksiyon (callee) tarafından yapılmaktadır. Genel olarak bu çağırma biçimi Intel mimarisinde cdecl çağırma biçiminden daha etkindir. Microsoft derleyicilerinde ayrıca WINAPI isimli makro `__stdcall` olarak bildirilmiştir.

```
#define WINAPI __stdcall
```

Böylece siz bir API fonksiyonun prototipinde WINAPI gördüğünüzde aslında orada `__stdcall` olduğunu düşünmelisiniz.

x86 fastcall Çağırma Biçimi: Bu çağırma biçiminde parametreler yazmaç yoluyla fonksiyona aktarılmaktadır. Geri dönüş değerleri yine yazmaçlardan alınır. Microsoft derleyicilerinde `__fastcall`, gcc ve clang derleyicilerinde `__attribute__((fastcall))` anahtar sözcükleriyle temsil edilmişlerdir.

X64 Windows ve Linux Çağırma Biçimleri: 64 bit Windows ve Linux sistemlerinde yukarıdaki gibi çağırma biçimleri yoktur. Parametreler tıpkı fastcall işleminde olduğu gibi yazmaçlar yoluyla aktarılmaktadır. Eğer yazmaçlar yetmezse geri kaalan parametreler stack yoluyla aktarılır. Ancak aktarımın hangi yazmaçlar kullanılarak yapılacağı Microsoft ve Linux sistemlerinde değişiklik göstermektedir.

2. Aşağı Seviyeli Fonksiyonlarla Dosya İşlemleri

İşletim sistemlerinin dosyalarla ilgili işlemler yapan alt sistemine "dosya sistemi (file system)" denilmektedir. Dosyaların yaratılması, silinmesi, dosyalardan bilgi okunması, dosyalara bilgilerin yazılması gibi işlemler hep dosya sisteminin kontrolündedir. İşletim sistemlerinin bu işlemleri yapan sistem fonksiyonları vardır. Biz hangi dille ya da hangi kütüphaneye çalışıyor olursak olalım eninde sonunda dosya işlemleri işletim sisteminin sistem fonksiyonlarıyla gerçekleştirilmektedir. Yukarıda da belirttiğimiz gibi dosya işlemlerini yapan Windows sistemlerindeki API fonksiyonları ve UNIX/Linux sistemlerindeki POSIX fonksiyonları aslında arka planda işletim sisteminin sistem fonksiyonlarını çağırarak dosya işlemlerini yapmaktadır.

Peki neden standart C fonksiyonları varken işletim sisteminin aşağı seviyeli sistem fonksiyonlarıyla dosya işlemi yapmak isteriz? Standart C fonksiyonları her sistemde olabilecek düşük bir işlevsellik ve dolayısıyla parametrik yapı sunmaktadır. Oysa sistemlerin kendine özgü özellikleri vardır. Örneğin Windows'ta açmış olduğumuz dosyayı başka birisinin de açmasını istemeyebiliriz. Bunu fopen ile sağlayamayız. Çünkü fopen fonksiyonunda böyle bir açış modu yoktur. Ya da örneğin aygıt sürücülerle çalışırken mecburen fopen fonksiyonunda olmayan bazı açış modlarını kullanmak zorunda kalabiliriz. Bunların dışında ayrıca bir sistem programcısı olarak işletim sisteminin aşağı seviyeli dosya fonksiyonları bize işletim sisteminin dosya sisteminin tasarımı hakkında da fikirler verebilmektedir.

Genel olarak pek çok işletim sisteminde dosya işlemleri için 5 temel sistem fonksiyonu bulundurulmaktadır:

- 1) Dosyayı açan bir sistem fonksiyonu
- 2) Dosyadan okuma yapmakta kullanılan bir sistem fonksiyonu
- 3) Dosyaya yazma yapmakta kullanılan bir sistem fonksiyonu
- 4) Dosya göstericisini konumlandıran bir sistem fonksiyonu
- 5) Dosyayı kapatan bir sistem fonksiyonu.

Biz kursumuzda işletim sisteminin sistem fonksiyonlarını doğrudan çağırılmayacağız. Bunun yerine Windows sistemlerinde Windows API fonksiyonlarını, UNIX/Linux sistemlerinde de POSIX fonksiyonlarını kullanacağız.

2.1. Windows Sistemlerinde Aşağı Seviyeli Dosya İşlemleri

Windows sistemlerinde temel dosya işlemlerini yapmak için 5 API fonksiyonu bulunmaktadır:

```
CreateFile  
ReadFile (ReadFileEx)  
WriteFile (WriteFileEx)  
CloseHandle  
SetFilePointer (SetFilePointerEx)
```

ReadFile, WriteFile ve SetFilePointer fonksiyonlarının daha sonra Ex sonek'li (IO Completion port özelliği olan) daha yetenekli versiyonları da oluşturulmuştur. Fakat biz kursumuzda bu fonksiyonların Ex'li versiyonlarından hiç bahsetmeyeceğiz.

CreateFile fonksiyonu var olan bir dosyayı açmak ya da olmayan bir dosyayı yaratarak açmak için kullanılmaktadır. Prototipi şöyledir:

```
HANDLE WINAPI CreateFile(  
    LPCTSTR LpFileName,  
    DWORD dwDesiredAccess,  
    DWORD dwShareMode,  
    LPSECURITY_ATTRIBUTES LpSecurityAttributes,  
    DWORD dwCreationDisposition,  
    DWORD dwFlagsAndAttributes,  
    HANDLE hTemplateFile  
);
```

Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesini, ikinci parametresi ise açış modunu belirtir. Bu parametre en azından GENERIC_READ, GENERIC_WRITE bayraklarından yalnızca birini ya da her ikisini içermelidir. Diğer bayraklar için MSDN dokümanlarına başvurulabilir. Üçüncü parametre dosyanın paylaşım modunu belirtir. Bu parametre FILE_SHARE_READ, FILE_SHARE_WRITE ve FILE_SHARE_DELETE bayraklarıyla oluşturulmaktadır. FILE_SHARE_READ bayrağı "ben açtıktan sonra başkası bu dosyayı açsın ama okuma modunda açsın" anlamına gelir. Benzer biçimde FILE_SHARE_WRITE da "ben açtıktan sonra başkası bu dosyayı yazma modunda açabilir" anlamına gelmektedir. Açış modunda FILE_SHARE_DELETE kullanılırsa dosya açıkken başkası dosyayı silebilir. (Tabii bu durumda dosyanın meta data bilgileri silinmektedir. Dosyanın gerçekten silinmesi bunu açan son prosesin dosyayı kapatması sırasında yapılmaktadır.) Programcı bu bayrakların birden fazlasını kullanabilir. Eğer bu parametre sıfır girilirse başkası dosyayı açamaz. (Örneğin fopen fonksiyonu dosyayı her zaman FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_SHARE_DELETE) modunda açmaktadır. Fonksiyonun dördüncü parametresi kernel nesnesinin güvenlik özelliklerini (security attributes) belirtir. Dosyaların güvenlik özelliklerini biz bu kursta ele almayacağız. Bu konu "Windows Sistem Programlama" kursumuzda ele alınmaktadır. Bu parametreyi NULL geçebilirsiniz. Beşinci parametre yaratım modunu belirtir. Bu parametrede OPEN_EXISTING bayrağı "dosya varsa aç, yoksa başarısız ol" anlamına gelir. Bu bayrak fopen fonksiyonundaki "r" moduna benzetilebilir. CREATE_ALWAYS bayrağı "dosya varsa sıfırla ve aç, dosya yoksa yarat ve aç" anlamına gelir. Bu da fopen fonksiyonundaki "w" modu gibidir. CREATE_NEW "dosya yoksa yarat ve aç, dosya varsa başarısız ol" anlamına, TRUNCATE_EXISTING ise "dosya yoksa başarısız ol, varsa sıfırla ve aç" anlamına gelmektedir. Fonksiyonun altıncı parametresi yaratılacak dosyanın dosya özelliklerini belirtir. Bu parametre FILE_ATTRIBUTE_NORMAL geçilebilir. Bu durumda tipik bazı dosya özellikleri (ayrıntılar için MSDN'e bakınız) verilmiş olur. Eğer istenirse bu parametre FILE_ATTRIBUTE_XXX biçiminde isimlendirilmiş sembolik sabitlerin Bit Or (|) işlemine sokulmasıyla da oluşturulabilir. Tabii bu parametre dosya ilk kez yaratılacaksa anlamlıdır. Eğer olan dosya açılacaksa bu parametre sıfır olarak da geçilebilir. Son parametre zaten açık bir dosyanın handle'li elimizde varsa onun özelliklerine sahip yeni bir dosyayı açmak için kullanılır. Bu parametre de NULL geçilebilir. Fonksiyon başarı durumunda dosyanın handle değerine başarısızlık durumunda INVALID_HANDLE_VALUE özel değerine geri döner. Örneğin:

```
int main(void)
{
    HANDLE hFile;
    DWORD dwRead;
    char buf[10 + 1];

    if ((hFile = CreateFile("Sample.c", GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
        0, NULL)) == INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    /* ... */

    return 0;
}
```

Burada "Sample.c" isimli dosya okuma amaçlı açılmak istenmiştir. Biz açtıktan sonra başkaları da bu dosyayı ancak okuma amaçlı açabilir.

Dosyadan okuma yapmak için ReadFile API fonksiyonu kullanılmaktadır:

```
BOOL WINAPI ReadFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped
);
```

Fonksiyonun birinci parametresi okunacak dosyanın handle değerini, ikinci parametresi okunan bilgilerinin yerleştirileceği adresi, üçüncü parametresi okunacak byte miktarını belirtir. Dördüncü parametre DWORD bir nesnenin adresini almaktadır. Fonksiyon adresini aldığı bu nesneye başarılı olarak okunabilen byte sayısını yerleştirir. Bu parametrenin DWORD türünden bir gösterici olduğuna dikkat ediniz. Son parametre "overlapped IO" işlemleri için

gereken bir yapı türündendir. "Overlapped IO" konusu "Windows Sistem Programlama" kursunda ele alınmaktadır. Bu parametre NULL geçilebilir. ReadFile API fonksiyonu başarı durumunda sıfır dışı bir değere başarısızlık durumunda 0 değerine geri döner. Fonksiyon talep edilen miktardan daha az sayıda byte'ı okuyabilir. Bu durum bir başarısızlık gerekçesi oluşturmaz. Örneğin:

```
int main(void)
{
    HANDLE hFile;
    DWORD dwRead;
    char buf[10 + 1];

    if ((hFile = CreateFile("Sample.c", GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
        0, NULL)) == INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    if (!ReadFile(hFile, buf, 10, &dwRead, NULL))
        ExitSys("ReadFile");

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

    /* ... */

    return 0;
}
```

Örneğin biz ReadFile fonksiyonunu bir döngü içerisinde çağırırsak dosyanın sonuna kadar bilgileri kısım kısım okuyabiliriz:

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

#define BUF_SIZE      4096

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    HANDLE hFile;
    char buf[BUF_SIZE + 1];
    DWORD dwRead;

    if ((hFile = CreateFile("Test.txt", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL)) ==
        INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    for (;;) {
        if (!ReadFile(hFile, buf, BUF_SIZE, &dwRead, NULL))
            ExitSys("ReadFile");

        if (dwRead == 0)
            break;

        buf[dwRead] = '\0';
        printf("%s", buf);
    }

    CloseHandle(hFile);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
```

```

DWORD dwLastErr = GetLastError();
LPTSTR lpszErr;

if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastErr,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
    fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
    LocalFree(lpszErr);
}

exit(EXIT_FAILURE);
}

```

Burada döngü yapısı şöyle de oluşturulabilirdi:

```

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

#define BUF_SIZE    4096

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    HANDLE hFile;
    char buf[BUF_SIZE + 1];
    DWORD dwRead;
    BOOL bResult;

    if ((hFile = CreateFile("Test.txt", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL)) ==
        INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    while ((bResult = ReadFile(hFile, buf, BUF_SIZE, &dwRead, NULL)) && dwRead > 0) {
        buf[dwRead] = '\0';
        printf("%s", buf);
    }

    if (!bResult)
        ExitSys("ReadFile");

    CloseHandle(hFile);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastErr = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastErr,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Dosyaya yazma yapmak için WriteFile API fonksiyonu kullanılmaktadır. Bu fonksiyonun parametrik yapısı ReadFile fonksiyonundaki gibidir:

```

BOOL WINAPI WriteFile(
    HANDLE hFile,

```

```

LPCVOID LpBuffer,
DWORD nNumberOfBytesToWrite,
LPDWORD LpNumberOfBytesWritten,
LPOVERLAPPED LpOverlapped
);

```

Fonksiyonun birinci parametresi dosyanın handle değerini alır. İkinci parametre dosyaya yazılacak bilgilerin bellekteki başlangıç adresini belirtir. Üçüncü parametre yazılacak toplam byte miktarını, dördüncü parametre ise başarılı olarak yazılan byte miktarını belirtmektedir. Son parametre yine "Overlapped IO" için kullanılan yapının adresini alır. Bu parametre NULL olarak geçilebilir. Aşağıda Windows sistemlerinde aşağı seviyeli API fonksiyonları kullanılarak bir dosya kopyalama örneği verilmiştir. Bu örnekte ReadFile fonksiyonu ile kaynak dosyadan belli uzunlukta okumalar yapıp hedef dosyaya WriteFile fonksiyonu ile bunlar yazdırılmaktadır.

```

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

#define BUF_SIZE      4096

void ExitSys(LPCSTR lpszMsg);

int main(int argc, char *argv[])
{
    HANDLE hFileSource, hFileDest;
    char buf[BUF_SIZE];
    DWORD dwRead, dwWritten;
    BOOL bResult;

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

    if ((hFileSource = CreateFile(argv[1], GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL)) ==
INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    if ((hFileDest = CreateFile(argv[2], GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, 0, NULL)) ==
INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    while ((bResult = ReadFile(hFileSource, buf, BUF_SIZE, &dwRead, NULL)) && dwRead > 0) {
        if (!WriteFile(hFileDest, buf, dwRead, &dwWritten, NULL))
            ExitSys("WriteFile");
        if (dwWritten != dwRead) {
            fprintf(stderr, "Cannot file write, maybe disk full!..\n");
            exit(EXIT_FAILURE);
        }
    }
    if (!bResult)
        ExitSys("WriteFile");

    CloseHandle(hFileSource);
    CloseHandle(hFileDest);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastErr = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastErr,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {

```

```

    fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
    LocalFree(lpszErr);
}

exit(EXIT_FAILURE);
}

```

Aslında dosya kopyalama işlemi için Windows sistemlerinde CopyFile isimli özel bir API fonksiyonu da bulundurulmuştur. CopyFile fonksiyonu dosya kopyalamayı disk üzerinde doğrudan yaptığı için daha hızlı çalışma eğilimindedir.

Anımsanacağı gibi her açılan dosya için işletim sistemi bir imleç görevi yapan bir "dosya göstericisi (file pointer)" tutmaktadır. Dosya göstericisi dosyanın o anda hangi byte'ı üzerinde işlem yapılacağını belirten bir offset numarası içerir. Dosyadan okuma ve yazma yapan fonksiyonlar okunan ya da yazılan miktar kadar otomatik olarak dosya göstericisini ilerletirler. Dosya ilk kez açıldığında dosya göstericisi 0'ıncı offset'tedir. Tabii biz istersek dosya göstericisini belli bir offset'e konumlandırarak doğrudan oradan okuma ve yazma yapabiliriz. Bunun için Windows sistemlerinde SetFilePointer API fonksiyonu kullanılmaktadır:

```

DWORD WINAPI SetFilePointer(
    HANDLE hFile,
    LONG lDistanceToMove,
    PLONG lpDistanceToMoveHigh,
    DWORD dwMoveMethod
);

```

Fonksiyonun birinci parametresi dosyanın handle değerini, ikinci ve üçüncü parametreleri de sırasıyla konumlandırma offset'inin düşük ve yüksek anlamlı dörder byte'lık değerlerini almaktadır. Konumlandırma offset'inin fonksiyona iki ayrı LONG parametreyle aktarıldığına dikkat ediniz. Offset'in yüksek anlamlı LONG değeri bir gösterici yoluyla aktarılmaktadır. Fonksiyon bu üçünü parametreye aynı zamanda (eğer NULL geçilmemişse) konumlandırılan offset'in yüksek anlamlı LONG değerini yerleştirmektedir. Tabii bu üçüncü parametre istenirse NULL geçilebilir. Son parametre konumlandırma orijini belirtmektedir. Bu parametre FILE_BEGIN, FILE_CURRENT ya da FILE_END biçiminde üç seçenekten biri olarak girilir. FILE_BEGIN konumlandırmanın dosyanın başından itibaren yapılacağını belirtir. Bu durumda konumlandırma offset'i sıfır ya da sıfırdan büyük bir değer olmak zorundadır. FILE_CURRENT o anda dosya göstericisinin gösterdiği offset'ten itibaren görece konumlandırma yapılacağı anlamına gelir. Bu durumda konumlandırma offset'i pozitif, negatif ya da sıfır olabilir. FILE_END ise konumlandırmanın EOF konumundan itibaren yapılacağı anlamına gelir. Bu durumda konumlandırma offset'i 0 ya da negatif olabilir. Fonksiyonun geri dönüş değeri dosya göstericisinin yeni offset'inin düşük anlamlı LONG değeridir. (Yukarıda da belirtildiği gibi yüksek anlamlı LONG değer üçüncü parametreyle belirtilen adresin gösterdiği yere yerleştirilmektedir.) Fonksiyon başarısızlık durumunda INVALID_SET_FILE_POINTER değerine geri dönmektedir. Fakat INVALID_SET_FILE_POINTER değeri eğer üçüncü parametre NULL girilmemişse geçerli bir offset de belirttiğinden fonksiyonu çağıran kişi GetLastError ile fonksiyonun başarısını da ayrıca kontrol etmelidir. Şöyle ki: Eğer fonksiyonun üçüncü parametresi NULL geçilmişse geri dönüş değerinin INVALID_SET_FILE_POINTER olması başarısızlık için yeterli olmaktadır. Ancak fonksiyonun üçüncü parametresi NULL geçilmemişse geri dönüş değerinin INVALID_SET_FILE_POINTER olması yeterli değildir. Başarısızlık için ayrıca GetLastError fonksiyonunun NO_ERROR dışında bir değere geri dönüyor olması gerekir. Örneğin:

```

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

#define BUF_SIZE 4096

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    HANDLE hFile;
    char buf[BUF_SIZE + 1];
    DWORD dwRead;
    BOOL bResult;

```



```

    if ((hFile = CreateFile("Sample.c", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL)) ==
INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    if (SetFilePointer(hFile, 100, NULL, FILE_BEGIN) == INVALID_SET_FILE_POINTER)
        ExitSys("SetFilePointer");

    if (!ReadFile(hFile, buf, BUF_SIZE, &dwRead, NULL))
        ExitSys("ReadFile");

    buf[dwRead] = '\\0';
    printf("%s", buf);

    CloseHandle(hFile);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastErr = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastErr,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Bu örnekte dosya CreateFile fonksiyonuyla açılmış ve hemen SetFilePointer fonksiyonuyla dosya göstericisi 100'üncü offset'e konumlandırılmıştır. Sonra oradan itibaren en fazla BUF_SIZE kadar bilgi okunup ekrana (stdout dosyasına) yazdırılmıştır.

NTFS kullanan Windows sistemlerinde ve Linux sistemlerinde dosya göstericisi dosyanın uzunluğundan öteye başarılı biçimde konumlandırılabilir. Bu tür durumlarda dosyaya yazma yapıldığında dosya deliği (file hole) oluşur. Dosya deliklerini ne anlam ifade ettiği ileride ele alınmaktadır.

Dosya nihayet CloseHandle API fonksiyonuyla kapatılır. CloseHandle yalnızca dosyaları kapatmak için değil ismine "kernel nesnesi" denilen tüm handle alanlarını kapatmak için ortak kullanılan bir fonksiyondur. Eğer dosyayı CloseHandle fonksiyonuyla kapatmazsak proses sonlandığında işletim sistemi dosyayı kendisi kapatmaktadır. Yani dosyanın kapatılmaması genellikle bir soruna yol açmaz. Tabii kullanılmayan dosyaların ilk fırsatta kapatılması iyi bir tekniktir. Pek çok işletim sisteminde bir prosesin açık durumda tutabileceği maksimum dosya sayısı vardır. CloseHandle fonksiyonunun prototipi şöyledir:

```

BOOL WINAPI CloseHandle(
    HANDLE hObject
);

```

Fonksiyon başarısızlık durumunda 0 değerine, başarı durumunda sıfır dışı bir değere geri dönmektedir.

2.2. UNIX/Linux Sistemlerinde Aşağı Seviyeli Dosya İşlemleri

UNIX/Linux sistemlerinde aşağı seviyeli dosya işlemleri için 5 POSIX fonksiyonu kullanılmaktadır. Bu fonksiyonlar pek çok UNIX türevi sistemde doğrudan işletim sisteminin sistem fonksiyonlarını çağırılmaktadır. UNIX/Linux sistemlerindeki temel POSIX dosya fonksiyonları şunlardır:

```

open
read

```

```
write
close
lseek
```

open fonksiyonun prototipi şöyledir:

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

```
int open(const char *pathname, int flags, ...);
```

open fonksiyonu iki ya da üç argümanla çağrılabilir. Fonksiyonun prototipi ve açış bayrakları <fcntl.h> dosyası içerisinde, erişim haklarına ilişkin sembolik sabitler de <sys/stat.h> dosyası içerisinde bildirilmiştir.

Fonksiyonun birinci parametresi açılacak dosyanın yol ifadesini, ikinci parametresi ise açış modunu belirtmektedir. Açış modu en azından şunlardan yalnızca birini içermek zorundadır:

```
O_RDONLY
O_WRONLY
O_RDWR
```

Buna ilaveten aşağıdaki bayraklar da bit düzeyinde OR'lanarak açış modunda kullanılabilir:

O_CREAT: Bu modda dosya yoksa yaratılır ve açılır, varsa olan dosya açılır.

O_TRUNC: Bu modda dosya açılırken eğer dosya zaten varsa aynı zamanda sıfırlanmaktadır. Örneğin O_CREAT|O_TRUNC modu "dosya yoksa yarat ve aç, varsa sıfırla ve aç" anlamına gelir. (fopen fonksiyonundaki "w" modu da POSIX sistemlerinde bu bayraklarla gerçekleştirilmektedir.) Bu bayrak O_RDONLY bayrağı ile birlikte kullanılamaz.

O_APPEND: Bu modda dosyadan okuma yapılabilir. Ancak her yazma işlemi önce dosya göstericisinin dosyanın sonuna çekilmesiyle yapılmaktadır. Yani her write işlemi sona eklemeye yol açar.

O_EXCL: Bu modda yaratım sırasında dosya varsa open fonksiyonu başarısız olur. (Yani bu mod olmayan bir dosyayı yaratmanın garanti altına alınması için kullanılmaktadır.) O_EXCL tek başına kullanılamaz ancak O_CREAT ile birlikte kullanılabilir.

Bu mod bayraklarının dışında open fonksiyonunda kullanılan başka mod bayrakları da vardır. Diğer mod bayrakları için ilgili dokümanlara başvurabilirsiniz.

open fonksiyonunun üçüncü parametresi dosyanın erişim haklarını belirtir. Fakat bu üçüncü parametre dosyanın yaratılma olasılığı varsa kullanılmalıdır. Başka bir deyişle ancak ikinci parametrede O_CREAT bayrağı belirtilmişse üçüncü parametre girilmelidir.

Dosyaların erişim hakları ls -l komutunda aşağıdaki gibi görüntülenmektedir:

```
-rwxrwxrwx
```

Görüldüğü gibi erişim hakları için toplam 9 karakter kullanılmaktadır. En soldaki karakter dosyanın türünü belirtir. Bu karakter şunlar biri olabilir:

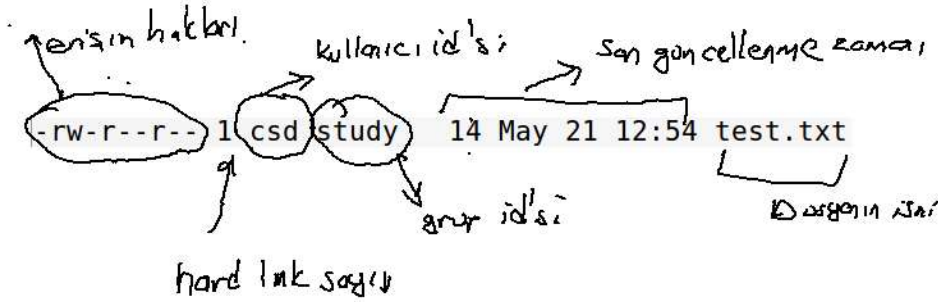
'-'	Normal dosya (regular file)
'd'	Dizin (directory)
'c'	Karakter aygıt sürücüsü (character device driver)
'b'	Blok aygıt sürücüsü (block device driver)
'p'	Boru (pipe)

's' Soket (socket)

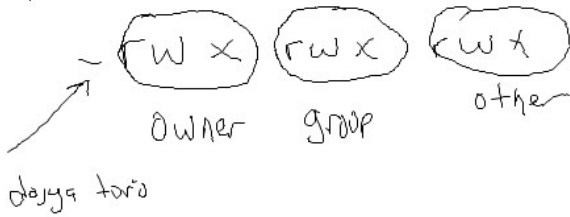
Diğer erişim hakları 3'lük 3 grup oluşturmaktadır:

UNIX/Linux sistemlerinde her prosesin bir "kullanıcı id'si (user id)" ve "grup id'si (group id)" değeri vardır. Bu değerler sayısaldır ve prosesin kontrol bloğunda saklanmaktadır. Kullanıcı id'si ve grup id'si üst procesten alt prosese fork işlemi sırasında aktarılır ve exec işlemi sırasında değişmez. Ayrıca sistem "kullanıcı id"lerini "kullanıcı isimleriyle (user name)", "grup id"lerini de "grup isimleriyle (group name) ilişkilendirmiştir. (Bu ilişkilendirme Linux sistemlerinde "/etc/passwd" ve "/etc/group" dosyalarında tutulmaktadır.) Böylece konuşurken ya da görüntüleme sırasında kullanıcı id'lerinin sayısal değerleri yerine kullanıcı isimleri, grup id'lerin sayısal değerleri yerine de grup isimleri kullanılmaktadır. Tabii işletim sisteminin çekirdeği gerçekten sayısal olan kullanıcı id'leri ve grup id'leri ile çalışır. Aslında UNIX/Linux sistemlerinde ayrıca proseslerin bir de "etkin kullanıcı id'leri (effective user ids)" ve "etkin grup id'leri (effective group ids)" de vardır. Aslında yetki denetimlerine bu etkin kullanıcı id'leri ve etkin grup id'leri sokulmaktadır. Çoğu zaman normal kullanıcı id'si ile etkin kullanıcı id'si, normal grup id'si ile etkin grup id'si aynıdır. Bunlar yalnızca bazı durumlarda değişmektedir. Siz şimdilik normal id'ler ile etkin id'ler arasında fark olmadığını varsayabilirsiniz.

UNIX/Linux sistemlerinde her dosyanın ayrıca bir "kullanıcı id'si (user id)" ve bir "grup id'si (group id)" de vardır. (Fakat dosyaların etkin kullanıcı id'leri ve etkin grup id'sleri yoktur). Is -l komutunda dosyaların kullanıcı ve grup id'leri de görüntülenmektedir. Örneğin:



Bir dosyanın erişim haklarının ilk üç karakterine "sahiplik (owner)", sonraki üç karakterine "grupluk (group)" ve sonraki üç karakterine "diğer (other)" hakları denilmektedir:



Is -l komutunda ilgili erişim hakkı varsa r, w, x sembollerinden biri yoksa '-' sembolü bulundurulmaktadır. Bu durumda erişim haklarındaki üçlü grupların ilk karakterleri ya 'r' ya da '-', ikinci karakterleri ya 'w' ya da '-' ve üçüncü karakterleri de 'x' ya da '-' olabilir. 'r' ilgili dosyadan okuma yapabilme hakkını, 'w' ilgili dosyaya yazma yapabilme hakkını ve 'x' de ilgili dosyayı çalıştırabilme hakkını belirtmektedir. '-' ise ilgili hakkın verilmediğini anlatmaktadır.

Örneğin:

-rw-r-----

Burada dosyanın sahiplik erişim hakları "rw-" biçimindedir. Bu dosyanın sahibi olan proses (yani etkin kullanıcı id'leri dosyanın kullanıcı id'si ile aynı olan prosesler) dosyadan okuma yapabilir, dosyaya yazma yapabilir ancak dosyayı çalıştıramaz. Dosyanın grupluk hakları ise "r--" biçimindedir. Dosyanın sahibi olmayan ancak dosyayla aynı grupta olan prosesler (yani etkin grup id'leri dosyanın grup id'si ile aynı olan prosesler) bu dosyadan yalnızca okuma yapabilirler. Dosyanın diğer haklarının "---" biçiminde olduğunu görüyorsunuz. Herhangi diğer prosesler (yani etkin kullanıcı ve grup id'leri dosyanın kullanıcı ya da grup id'si ile aynı olmayan prosesler) bu dosya üzerinde hiçbir işlem yapamamaktadır.

Çalıştırılabilir olmayan bir dosyanın sahiplik, grupluk ve diğer erişim haklarının '-' biçiminde olması normaldir. Ancak dosya çalıştırılabilir olduğu halde çalıştırma hakkı ilgili üçlük gruba verilmemiş de olabilir.

Erişim hakları open fonksiyonu tarafından kontrol edilmektedir. Yani dosya işin başında açılırken open fonksiyonu (aslında open fonksiyonunun çağırıldığı sistem fonksiyonu) talep edilen erişim hakkıyla dosyanın izin verilen erişim haklarını karşılaştırır. Eğer talep edilen erişim hakları izin verilen erişim haklarından daha genişse open fonksiyonu başarısız olur. Yani örneğin biz open fonksiyonuyla yalnızca okuma hakkı verilmiş bir dosyayı O_RDWR modunda açmaya çalışalım. Bu durumda dosya başarılı açılıp yazma sırasında hata oluşmaz, dosya açma işlemi işin başında başarısız olur.

open fonksiyonun denetim algoritması şöyledir (maddeleri else-if olarak değerlendiriniz):

1) Önce open fonksiyonu kendisini çağıran prosesin etkin kullanıcı id'si 0 mı diye bakar. Sıfır numaralı kullanıcı id'sine sahip procese "super user", "privileged user" ya da "root user" denilmektedir. Eğer dosyaya erişmek isteyen prosesin user id'si 0 ise erişim kabul edilir. 0 numaralı user id'ye sahip proses dosya erişimlerinde hiçbir engelle karşılaşmaz.

2) open fonksiyonu dosyaya erişmek isteyen prosesin etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynı mı diye bakar. (Yani erişimi yapmak isteyen proses dosyanın sahibi midir?). Eğer böyleyse sahiplik (owner) erişim haklarını dikkate alır.

3) open fonksiyonu dosyaya erişmek isteyen prosesin etkin grup id'si dosyanın grup id'si ile aynı mı diye bakar. (Yani erişim yapmak isteyen proses dosyanın grubuyla aynı gruptan mıdır?). Eğer böyleyse grupluk (group) erişim haklarını dikkate alır.

4) Bu durumda erişimi yapmak isteyen proses ne dosyanın sahibi ne de dosya ile aynı gruptan olan bir proses değildir. O artık herhangi bir prostedir. Bu durumda open fonksiyonu dosyanın diğer erişim haklarını (other) dikkate alır.

Örneğin "- rw- r-- r--" erişim haklarına sahip "x.txt" dosyasını open ile açmaya çalışalım:

```
if ((fd = open("x.txt", O_RDWR)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

Eğer bu çağrıyı dosyanın sahibi olan proses yaparsa open başarılı olur. Aynı gruptan ya da herhangi biri yaparsa başarısız olacaktır.

Peki bir dosyanın kullanıcı id'sinin grup id'sinin ve erişim haklarının ne olacağına kim nasıl karar vermektedir? İşte UNIX/Linux sistemlerinde tüm dosyalar şu ya da bu biçimde open fonksiyonuyla yaratılırlar. Dosyaların kullanıcı ve grup id'lerinin ve erişim haklarının nasıl olacağı open işlemi sırasında belirlenmektedir. Şöyle ki:

1) Yeni yaratılan dosyanın kullanıcı id'si her zaman onu yaratan prosesin etkin kullanıcı id'si olarak alınır. (Yani dosyayı hangi kullanıcı yaratmışsa dosyanın kullanıcı id'si de o olacaktır.)

2) Yeni yaratılan dosyanın grup id'si POSIX standartlarında iki seçenekten biri olarak belirlenmektedir: Dosyayı yaratan prosesin group id'si olarak ya da dosyanın içinde bulunduğu dizinin group id'si olarak. Bu farklı belirleme tarzı POSIX standartları oluşturulduğu sırada izlenen iki farklı stratejiden kaynaklanmaktadır. POSIX standartları oluşturulurken bu konuda farklı yöntem izlemiş olan sistemlerin standart uyumunun korunması istenmiştir. Örneğin Linux'ta default olarak yeni yaratılan dosyanın grup id'si onu yaratan prosesin etkin grup id'si olarak atanmaktadır. Ancak bu durum "mount parametreleri" ile değiştirilebilmektedir.

3) Dosyanın erişim hakları dosya yaratılırken open fonksiyonun üçüncü parametresiyle belirlenir. Bu üçüncü parametre aşağıdaki sembolik sabitlerin bit düzeyinde OR'lanmasıyla oluşturulmaktadır:

```
S_IRUSR
S_IWUSR
S_IXUSR
```

S_IRGRP
S_IWGRP
S_IXGRP

S_IROTH
S_IWOTH
S_IXOTH

Bu sembolik sabitlerin hepsinin başının S_I ile başladığına dikkat ediniz. Bunu R, W ve X harfleri, bunu daUSR, GRP ya da OTH karakterleri izlemektedir.

S_I R USR
W GRP
X OTH

Ayrıca bu sembolik sabitlerin dışında aşağıdaki üç sembolik sabit de bulunmaktadır:

S_IRWXU
S_IRWXG
S_IRWXO

Bu sembolik sabitler aslında aşağıdakilerle eşdeğerdir:

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

Örneğin:

```
if ((fd = open("x.txt", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR|S_IRGRP)) < 0) {  
    perror("open");  
    exit(EXIT_FAILURE);  
}
```

Burada dosya "rw-r----" haklarıyla yaratılmak istenmiştir. Yukarıdaki S_IXXX sembolik sabitleri <sys/stat.h> içerisinde bildirilmiştir. (Böylece eğer open fonksiyonunda ikinci parametre olarak O_CREAT girilmişse bu dosyanın da include edilmesi gerekir.)

Open fonksiyonu başarı durumunda dosya betimleyicisi (file descriptor) denilen handle değerine başarısızlık durumunda ise -1 değerine geri döner. errno değişkeni uygun biçimde set edilmektedir. open fonksiyonundan elde edilen dosya betimleyicisi read, write, lseek ve close gibi fonksiyonlarda dosyayı "betimlemek" için kullanılacaktır.

Burada bir kez daha bir uyarıda bulunmak istiyoruz. Eğer open fonksiyonun ikinci parametresinde O_CREAT girilmemişse dosyanın yaratılma olasılığı yoktur. Bu durumda biz open fonksiyonun üçüncü parametresini girsek bile fonksiyon onu zaten kullanmayacaktır. Yani bizim bu durumda bu üçüncü parametreyi girmemizin bir anlamı yoktur. Benzer biçimde bizim open fonksiyonunun ikinci parametresinde O_CREAT bayrağını girmiş olmamız da üçüncü parametredeki erişim haklarının kullanılacağı anlamına gelmemektedir. Bizim üçüncü parametre için girdiğimiz erişim hakları dosya yalnızca ilk kez yaratılacaksa dikkate alınır. Olan dosyanın açılması ya da sıfırlanması (truncation) işleminde bu üçüncü parametre dikkate alınmamaktadır.

UNIX/Linux sistemlerinde ayrıca bir de creat isimli bir POSIX fonksiyonu da vardır. Bu fonksiyon aslında taban bir fonksiyon değildir. Kendi içerisinde open fonksiyonunu çağırılmaktadır:

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

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

Yukarıdaki creat fonksiyonu ikinci parametresi O_WRONLY|O_CREAT|O_TRUNC biçiminde olan open fonksiyonuyla eşdeğerdir. Başka bir deyişle fonksiyon şöyle yazılmıştır:

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

Burada biraz daha kafanızı karıştıracakız. Aslında open fonksiyonunun üçüncü parametresinde girilen erişim hakları nihai durumu belirtmemektedir. Bu erişim hakları prosesin "umask" denilen bir maskeleye değerinin bitset tersi ile bitset AND işlemine sokularak gerçek erişim hakları belirlenmektedir. Başka bir deyişle umask değerinde 1 olan bitler open fonksiyonunda belirtilmiş olsa bile silinmektedir. Örneğin prosesin umask değeri S_IWGRP|S_IROTH|S_IWOTH biçiminde olsun. Bu durumda open fonksiyonunun üçüncü parametresinde bu bayraklar kullanılmış olsa bile silinecektir. Yani bu umask değeri, belirtilmiş olsa bile grup "write" hakkını diğerleri için "read" ve "write" hakkını kaldır" anlamına gelmektedir. Prosesin umask değeri umask isimli POSIX fonksiyonuyla değiştirilmektedir:

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

mode_t umask(mode_t mask);
```

Fonksiyon prosesin yeni umask değerini parametre olarak alır, değişikliği yapar, eski umask değerine geri döner. Örneğin:

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;

    umask(S_IWOTH);

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

    printf("Ok\n");

    return 0;
}

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

    exit(EXIT_FAILURE);
}
```

Prosesin umask değeri üst prosten alt prosese fork işlemi sırasında aktarılmaktadır. Örneğin biz kabuk prosesinin (bash) umask değerini değiştirirsek bu değer alt proseslere aktarılacaktır. Böylece artık kabuktan çalıştırılan programların da umask değerleri kabuğun umask değeri ile aynı olacaktır. Kabuğun umask değeri kabuk üzerinde umask komutu uygulanarak elde edilebilir. Örneğin:

```
csd@csd-virtual-machine ~ $ umask
0022
csd@csd-virtual-machine ~ $
```

Kabukta umask değerinin 4 octal digit biçiminde belirtildiğine dikkat ediniz. Düşük anlamlı üç oktal digit sırasıyla user, group ve other için rwx haklarının olup olmadığını belirtmektedir.

0022
↓ ↓ ↓
000 010 010
---w--w-
UJR GRP OTH

Yine biz istersek kabuğun umask değerini de aynı komutla değiştirebiliriz. Örneğin:

```
csd@csd-virtual-machine ~ $ umask 0066
csd@csd-virtual-machine ~ $ umask
0066
csd@csd-virtual-machine ~ $
```

Peki kabuk prosesinin umask değeri nasıl belirlenmektedir? Aslında kabuk umask değerini onu çalıştıran üst prosesten almaktadır. Tabii kullanıcı isterse kabuğun çalıştırdığı çeşitli script dosyaları içerisine umask komutunu yerleştirerek kabuğun umask değerini otomatik biçimde değiştirebilir.

UNIX/Linux sistemlerinde dizinler aslında bir çeşit dosya gibi ele alınmaktadır. Öyle ki, dizin (directory) aslında bir dosyadır. Ancak dizin dosyasının içerisinde o dizindeki dosyaların isimleri tutulur. (Zaten ileride göreceğimiz opendir fonksiyonu dizin dosyasını açar, readdir ise oradan okuma yapar.) Bir dizin içerisinde bir dosyanın yaratılması aslında o dizin dosyasına yazma yapma anlamındadır. O halde bir dizin içerisinde bizim bir dosya yaratabilmemiz için o dizine "w" hakkına sahip olmamız gerekir. Benzer biçimde bir dizinden bir dosya silmek de aslında o dizin dosyasına yazma yapmak anlamına gelmektedir. O halde bir dizinden bir dosyayı silebilmemiz için o dizine "w" hakkımızın olması gerekir. Dizinin içeriğinin elde edilmesi ise o dizinden okuma yapma anlamına gelmektedir. Bizim bir dizinin listesini alabilmemiz için o dizine "r" hakkımızın olması gerekir. Gerçekten de dizini opendir fonksiyonu ile açabilmemiz için bizim dizine "r" hakkımızın olması gerekmektedir. Dizinlerde 'x' hakkı "çalıştırma" anlamına gelmez. Başka bir anlama gelmektedir. Biz dizine "x" hakkımız varsa (ki genellikle olur) yol ifadesi olarak onun içinden geçebiliriz. Örneğin "/home/csd/a/b/test.txt" biçiminde bir yol ifadesi vererek open fonksiyonuyla "test.txt" dosyasını açmak isteyelim. Bizim bu dosyaya ulaşabilmemiz için sırasıyla home, csd, a ve b dizinlerinin içinden geçebilmemiz gerekir. Bu nedenele bu dizinlere "x" hakkımızın olması gerekmektedir. Eğer yol ifadesi görel olarak "test.txt" biçiminde verilseydi bile bizim yine prosesin çalışma dizinine "x" hakkımızın olması gerekirdi. Görüldüğü "x" hakkı dizinler için çok önemlidir. Biz bir dizini komut satırında mkdir komutuyla yarattığımızda dizine "x" hakkı verilmektedir. Tersten gidersek biz kendi dizinimize ilişkin bir ağacı dış dünyadan gizlemek istediğimizde tek yapacağımız şey ağacın o kök kısmındaki dizinden "x" hakkını kaldırmamız olacaktır. Gerçekten de bir dizinin "x" hakkı kaldırıldığında artık başkaları o dizine ve o dizinin alt dizinlerine erişememektedir. Bir yol ifadesindeki hedef dosyaya erişebilmek için bizim o dosyaya giden dizinlerin hepsine "x" olması gerekir. Ancak bu dizinlere "r" hakkına sahip olması gerekmez. Örneğin bizim open fonksiyonuyla "/home/csd/a/b/test.txt" yol ifadesiyle belirtilen "test.txt" dosyasını O_RDONLY modunda açabilmemiz için yol ifadesinde belirtilen tüm dizinler için "x" hakkına sahip olmamız gerekir. Ancak "r" hakkına sahip olmamız gerekmez. Tabii "test.txt" dosyası için "r" hakkına sahip olmamız gerekmektedir. Bir dizin içerisindeki bir dosyayı remove ya da unlink fonksiyonuyla silebilmemiz için o dosyaya "w" hakkına sahip olmamız gerekmez. O dosyanın içinde bulundığı dizine "w" hakkına sahip olmamız gerekir.

Bir dosyanın erişim hakları dosyanın sahibi tarafından (yani etkin kullanıcı id'si dosyanın kullanıcı id'si ile aynı olan proses tarafından) chmod isimli POSIX fonksiyonuyla istenildiği zaman değiştirilebilir. Aynı zamanda komut satırında bu fonksiyonu kullanarak yazılmış olan chmod isimli bir kabuk komutu da vardır. Bu konu ileride ele alınmaktadır.

UNIX/Linux sistemlerinde dosyadan okuma yapmak için read isimli POSIX fonksiyonu, dosyaya yazma yapmak için write isimli POSIX fonksiyonu kullanılmaktadır.


```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Fonksiyonların birinci parametreleri okuma yazma işlemlerinin hangi dosyadan yapılacağını belirten dosya betimleyicisini alır. İkinci parametreler bellekteki transfer adresini son parametreler ise yazılacak ya da okunacak byte sayısını belirtmektedir. read fonksiyonu okuyabildiği byte sayısına geri döner. Dosyada dosya göstericisinin gösterdiği yerden dosya sonuna kadar olandan daha fazla byte okunmak istenebilir. Bu durumda read fonksiyonu başarısız olmaz. Okuyabildiği kadar byte'ı okur, okuyabildiği byte sayısına geri döner. Eğer dosya göstericisi EOF durumundaysa read dosyadan hiç okuma yapamaz. Bu durumda 0 değerine geri döner. write fonksiyonu da benzer biçimde yazabildiği byte sayısı ile geri dönmektedir. Normalde write fonksiyonu talep edilen kadar bilgiyi yazma konusunda bir sorun çıkartmaz. Ancak disk doluysa ve dosya sisteminin sınırları ile ilgili bir durum söz konusuysa write fonksiyonu talep edilenden daha az byte'ı dosyaya yazabilmektedir. Fonksiyonların geri dönüş değerlerindeki ssize_t türü POSIX standartlarına göre işaretli bir tamsayı türü olarak typedef edilmek zorundadır. Pek çok sistemde bu tür signed long int olarak typedef edilmektedir. Her iki fonksiyon da başarısızlık durumunda -1 değerine geri dönmektedir. Şimdi dosyadan okuma yapan aşağıdaki gibi bir örnek program yazalım:

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

#define BUF_SIZE 10

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    char buf[BUF_SIZE + 1];
    ssize_t n;

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

    while ((n = read(fd, buf, BUF_SIZE)) > 0) {
        buf[n] = '\0';
        printf("%s", buf);
    }

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

    close(fd);

    return 0;
}

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

UNIX/Linux sistemlerinde dosya göstericisini konumlandırmak için lseek fonksiyonu kullanılmaktadır.

```
#include <unistd.h>

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


Iseek fonksiyonu standart C'nin fseek fonksiyonu gibidir. Fonksiyonun birinci parametresi dosya betimleyicisini, ikinci parametresi konumlandırma offsetini, üçüncü parametresi ise konumlandırma orijini belirtir. Fonksiyon başarı durumunda konumlandırma offset'ine başarısızlık durumunda -1 değerine geri döner. Prototipteki off_t türü POSIX standartlarına göre işaretli bir tamsayı türü olarak typedef edilmek zorundadır. Pek çok sistemde off_t signed long int biçiminde typedef edilmiştir. Iseek fonksiyonu ile dosya göstericisi dosyanın uzunluğunun ötesine konumlandırılabilir. Bu durumda ilk yazma yapıldığında dosya deliği (file hole) oluşmaktadır.

Nihayet dosya close fonksiyonuyla kapatılır. close fonksiyonunun prototipi şöyledir:

```
#include <unistd.h>

int close(int fd);
```

Fonksiyon parametre olarak açılmış dosyanın betimleyicisini alır. Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönmektedir. Diğer işletim sistemlerinde olduğu gibi UNIX/Linux sistemlerinde de eğer programcı dosyayı kapatmamışsa proses sonlandığında işletim sistemi tüm açık dosyaları kapatmaktadır. Ancak kullanılmayan dosyaların işlem bitince kapatılması iyi bir tekniktir. Çünkü açık dosyalar belli bir sistem kaynağının kullanılmasına yol açmaktadır. Aynı zamanda UNIX/Linux sistemlerinde her prosesin en fazla açık durumda tutabileceği dosya sayısında bir sınırlama vardır. (Bu sınır Linux sistemlerinde 1024'tür.)

UNIX/Linux sistemlerinde genellikle dosya kopyalamak için ayrı sistem fonksiyonu bulundurulmaz. Kopyalama işlemi bir dosyadan blok blok bilgilerin okunarak diğerine yazılmasıyla gerçekleştirilmektedir. Aşağıda dosya kopyalamaya bir örnek veriyoruz. Ancak bu örnekte kaynak dosya ile hedef dosyanın erişim haklarının aynı olmasına dikkat edilmemiştir. Oysa komut satırındaki cp komutu hedef dosyayı kaynak dosya erişim haklarının aynısına sahip olacak biçimde yaratmaktadır.

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

#define BUF_SIZE 8192

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fds, fdd;
    char buf[BUF_SIZE];
    int oflag = 0;
    int result;
    ssize_t n;

    opterr = 0;
    while ((result = getopt(argc, argv, "o")) != -1) {
        switch (result) {
            case 'o':
                oflag = 1;
                break;
            case '?':
                fprintf(stderr, "invalid switch: %c\n", optopt);
                break;
        }
    }

    if (argc - optind != 2) {
        fprintf(stderr, "wrong number of arguments!..\n");
        fprintf(stderr, "usage: mycp [-o] <source file path> <dest file path>\n");
        exit(EXIT_FAILURE);
    }

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

```

if (access(argv[optind + 1], F_OK) == 0 && !oflag) {
    fprintf(stderr, "file already exists, cannot overwrite, please use -o switch to overwrite!..\n");
    exit(EXIT_FAILURE);
}

if ((fdd = open(argv[optind + 1], oflag ? O_WRONLY | O_CREAT : O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
    exit_sys("open");

while ((n = read(fds, buf, BUF_SIZE)) > 0)
    if (write(fdd, buf, n) != n)
        exit_sys("write");

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

printf("1 file copied...\n");

close(fds);
close(fdd);

return 0;
}

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

```

2.3. Yardımcı Dosya Fonksiyonları

Windows ve UNIX/Linux sistemlerinde temel dosya fonksiyonlarının yanı sıra bir de yardımcı çeşitli fonksiyonlar bulunabilmektedir. Bu bölümde her iki sistemde de önemli yardımcı dosya fonksiyonlarını göreceğiz.

2.3.1. Windows Sistemlerinde Yardımcı Dosya Fonksiyonları

Windows sistemlerinde temel dosya fonksiyonlarının yanı sıra dosya üzerinde bütünsel işlemler yapan yardımcı API fonksiyonları bulunmaktadır.

2.3.1.1. DeleteFile Fonksiyonu

Windows'ta dosya silmek için DeleteFile isimli API fonksiyonu kullanılmaktadır. Aslında remove isimli standart C fonksiyonu Windows sistemlerinde DeleteFile API fonksiyonu çağırılmaktadır. Fonksiyonun prototipi şöyledir:

```

BOOL DeleteFile(
    LPCTSTR lpFileName
);

```

Fonksiyonun geri dönüş değeri işlemin başarısını belirtir. Örneğin:

```

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

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    if (!DeleteFile("test.txt"))
        ExitSys("DeleteFile");

    printf("Ok\n");
}

```

```

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

2.3.1.2. CopyFile ve MoveFile Fonksiyonları

Windows sistemlerinde dosya kopyalamak ve dosyayı taşımak için hızlı çalışan iki API fonksiyonu bulundurulmuştur. Fonksiyonların prototipleri şöyledir:

```

BOOL CopyFile(
    LPCTSTR lpExistingFileName,
    LPCTSTR lpNewFileName,
    BOOL bFailIfExists
);

```

```

BOOL MoveFile(
    LPCTSTR lpExistingFileName,
    LPCTSTR lpNewFileName
);

```

CopyFile fonksiyonun son parametresi hedef dosya varsa ne yapılacağını belirtmektedir. MoveFile fonksiyonunda hedef dosya zaten varsa fonksiyon her zaman başarısız olmaktadır. Her iki fonksiyon da işlemin başarılı olup olmadığı bilgisine geri dönmektedir. MoveFile genellikle dosyanın ismini değiştirmek için tercih edilmektedir.

2.3.1.3. GetFileSize ve GetFileAttributes Fonksiyonları

Bu API fonksiyonları sırasıyla dosyanın uzunluğunu ve özelliklerini elde etmek için kullanılmaktadır. Prototipleri şöyledir:

```

DWORD GetFileSize(
    HANDLE hFile,
    LPDWORD lpFileSizeHigh
);

```

```

DWORD GetFileAttributes(
    LPCTSTR lpFileName
);

```

GetFileSize fonksiyonu açılmış bir dosyanın uzunluğunu elde etmek için kullanılmaktadır. Bu fonksiyonun birinci parametresi açılmış dosyanın handle değerini, ikinci parametresi ise dosya uzunluğunun yüksek anlamlı 4 byte'lık kısmının yerleştirileceği DWORD nesnenin adresini almaktadır. İkinci parametre NULL geçilebilir. GetFileSize fonksiyonu başarısızlık durumunda INVALID_FILE_SIZE özel değerine geri döner. Ancak bu INVALID_FILE_SIZE sembolik sabiti 0xFFFFFFFF değerine sahip olduğu için ve bu değer de geçerli bir dosya uzunluğu olabildiği için programcının ayrıca fonksiyonun başarısını GetLastError fonksiyonuyla belirlemesi gerekmektedir. Yani eğer fonksiyon INVALID_FILE_SIZE değerine geri dönmüşse programcının GetLastError fonksiyonunu çağırması gerekir. GetLastError eğer NO_ERROR değerini verirse fonksiyon başarısız olmamıştır. Dosya gerçekten 0xFFFFFFFF uzunluğundadır. Örneğin:

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

```

#include <stdlib.h>
#include <windows.h>

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    HANDLE hFile;
    DWORD dwSize;

    if ((hFile = CreateFile("Test.txt", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL)) ==
INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    if ((dwSize = GetFileSize(hFile, NULL)) == INVALID_FILE_SIZE)
        if (GetLastError() != NO_ERROR)
            ExitSys("GetFileSize");

    printf("%lu\n", (unsigned long)dwSize);

    CloseHandle(hFile);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastErr = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastErr,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

GetFileAttributes ise dosyanın yol ifadesini alarak dosya özelliklerini geri dönüş değeri ile verir. Fonksiyonun geri dönüş değeri bitsel olarak kodlanmıştır. Bu fonksiyon da başarısızlık durumunda INVALID_FILE_ATTRIBUTES özel değerine geri dönmektedir.

2.3.1.4. CreateDirectory ve RemoveDirectory Fonksiyonları

Bu fonksiyonlar sırasıyla yeni bir dizin yaratmak ya da yaratılmış bir dizini silmek için kullanılmaktadır. Prototipleri şöyledir:

```

BOOL CreateDirectory(
    LPCTSTR lpPathName,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);

BOOL RemoveDirectory(
    LPCTSTR lpPathName
);

```

CreateDirectory fonksiyonun birinci parametresi yaratılacak dizinin yol ifadesini ikinci parametresi de güvenlik bilgilerini içerir. İkinci parametre NULL geçilebilir. RemoveDirectory fonksiyonu içi boş olan bir dizini silmektedir. Her iki fonksiyon da işlemin başarısını anlatan BOOL değerler geri dönmektedir.

2.3.2. UNIX/Linux Sistemlerinde Yardımcı Dosya Fonksiyonları

Bu bölümde UNIX/Linux sistemlerindeki temel dosya fonksiyonlarının dışındaki dosya sistemi ile ilgili önemli işlemler yapan POSIX fonksiyonları tanıtılacaktır.

2.3.2.1. remove ve unlink Fonksiyonları

remove ve unlink fonksiyonları dosyayı silmek için kullanılmaktadır. Bu iki fonksiyon tamamen eşdeğerdir. remove bir standart C fonksiyonudur. Yani her sistemde bulunmaktadır. unlink ise POSIX fonksiyonudur. Yalnızca UNIX/Linux türevi sistemlerde bulunur. remove fonksiyonunun prototipi <stdio.h> dosyasındadır:

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

```
int remove(const char *pathname);
```

unlink fonksiyonunun prototipi ise <unistd.h> dosyasındadır.

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

```
int unlink(const char *pathname);
```

Fonksiyonlar başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönerler. Örneğin:

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

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

```
int main(void)
{
    if (unlink("test.txt") == -1)
        exit_sys("unlink");

    printf("ok\n");

    return 0;
}
```

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

2.3.2.2 chmod ve fchmod Fonksiyonları

chmod isimli POSIX fonksiyonu zaten var olan bir dosyanın erişim haklarını değiştirmek için kullanılır. Fonksiyonun prototipi şöyledir:

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

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

Fonksiyonun birinci parametresi erişim hakları değiştirilecek dosyanın yol ifadesini ikinci parametresi ise onun erişim haklarını alır. Fonksiyon başarı durumda 0 değerine, başarısızlık durumunda -1 değerine geri döner. Dosyanın (dizinler de birer dosya gibidir) erişim hakları dosyanın sahibi olan prosesler tarafından ya da etkin kullanıcı id'si 0 olan prosesler (root/super user) tarafından değiştirilebilir. Prosesin umask değerinin chmod fonksiyonu üzerinde bir etkisi yoktur.

fchmod fonksiyonu tamamen chmod fonksiyonu gibidir. Ancak açılmış dosyanın dosya betimleyicisinde hareketle dosyanın erişim haklarını değiştirir. Yani elimizde zaten open fonksiyonuyla açmış olduğumuz bir dosya varsa biz fchmod

fonksiyonuyla bu dosyanın erişim haklarını hemen değiştirebiliriz. Bu fonksiyonun chmod fonksiyonundan daha hızlı çalışacağı varsayılabilir. fchmod fonksiyonun prototipi şöyledir:

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

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

Fonksiyonun birinci parametresi açılmış dosyanın dosya betimleyicisini, ikinci parametresi ise erişim haklarını belirtmektedir. Bu fonksiyon da başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri döner.

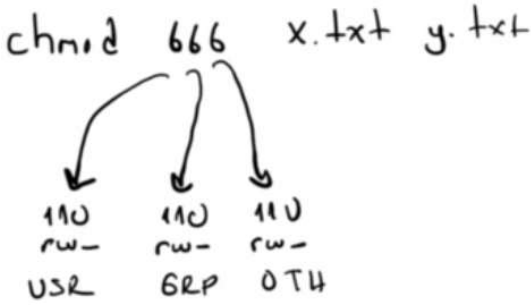
Komut satırında da chmod isimli POSIX kabuk komutu dosyanın erişim haklarını değiştirmek için kullanılmaktadır. chmod komutunun en basit kullanımı şöyledir:

```
chmod <octal sayı olarak erişim hakları> <dosya listesi>
```

Örneğin:

```
csd@csd-virtual-machine ~/Study/SysProg-2019 $ chmod 666 x.txt y.txt
csd@csd-virtual-machine ~/Study/SysProg-2019 $ ls -l x.txt y.txt
-rw-rw-rw- 1 csd study 23 Şub  3 11:33 x.txt
-rw-rw-rw- 1 csd study 23 Şub  3 11:38 y.txt
csd@csd-virtual-machine ~/Study/SysProg-2019 $
csd@csd-virtual-machine ~/Study/SysProg-2019 $
csd@csd-virtual-machine ~/Study/SysProg-2019 $
csd@csd-virtual-machine ~/Study/SysProg-2019 $
```

Buradaki octal digitler sırasıyla owner, group, other haklarının ikilik sistemdeki karşılıklarıdır.



chmod komutunun başka kullanım biçimleri de vardır. Örneğin +r, -r, +w, -w, +x, -x ifalarının başına ugo harflerinden biri getirilirse sırasıyla user, group, other için ilgili özellik set ya da reset edilir. a harfi hem user (owner), hem group hem de other'ı kapsama (all) anlamındadır. Örneğin:

```
chmod a+w x.txt
```

Burada user, group ve other için w hakkı verilmiştir. Örneğin:

```
chmod o-w x.txt
```

Burada other'dan w hakkı çıkartılmıştır.

Aşağıdaki örnekte -chmod komutunda olduğu gibi- octal digit'lerle girilen erişim hakları chmod fonksiyonuyla dosyalara uygulanmıştır:

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

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

```

{
mode_t modes[] = {S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH};
unsigned smode;
mode_t dmode;
int i;

if (argc < 3) {
    fprintf(stderr, "wrong number of arguments!\nUsage: mychmod <mode> <file list>\n");
    exit(EXIT_FAILURE);
}

sscanf(argv[1], "%o", &smode);

dmode = 0;
for (i = 8; i >= 0; --i)
    if ((smode >> i) & 1)
        dmode |= modes[8 - i];

for (i = 2; i < argc; ++i)
    if (chmod(argv[i], dmode) == -1)
        fprintf(stderr, "cannot change mode: %s\n", argv[i]);

return 0;
}

```

2.4.3. mkdir ve rmdir Fonksiyonları

Bir dizin yaratmak için mkdir isimli POSIX fonksiyonu, dizini silmek için ise rmdir isimli POSIX fonksiyonu kullanılmaktadır.

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

```
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

mkdir fonksiyonu yaratılacak dizinin yol ifadesini ve erişim haklarını parametre olarak alır. rmdir ise silinecek dizinin yol ifadesini almaktadır. Fonksiyonlar başarı durumunda 0 değerine başarısızlık durumunda -1 değerine geri dönerler. rmdir ile bir dizinin silinmesi için dizinin boş olması gerekir. mkdir yine proses umask değerinden etkilenmektedir. Örneğin:

```
/* mymkdir */
```

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

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

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

```
{
mode_t modes[] = { S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH
};
unsigned int omode;
mode_t mode;
int m_flag, err_flag;
char *m_arg;
int result, i;

struct option options[] = {
    {"mode", required_argument, NULL, 'm'},
    {0, 0, 0, 0 },
};

```

```

opterr = 0;

m_flag = 0;
err_flag = 0;

while ((result = getopt_long(argc, argv, "m:", options, NULL)) != -1) {
    switch (result) {
        case 'm':
            m_flag = 1;
            m_arg = optarg;
            break;
        case '?':
            if (optopt == 'm')
                fprintf(stderr, "-m option must have an argument!..\n");
            else if (optopt != 0)
                fprintf(stderr, "invalid option: -%c\n", optopt);
            else
                fprintf(stderr, "invalid long option!..\n");

            err_flag = 1;
        }
    }

if (err_flag)
    exit(EXIT_FAILURE);

if (m_flag) {
    if (strlen(m_arg) > 3) {
        fprintf(stderr, "mode must have maximum 3 octal digits!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; m_arg[i] != '\0'; ++i)
        if (m_arg[i] < '0' || m_arg[i] > '7') {
            fprintf(stderr, "invalid octal digit!..\n");
            exit(EXIT_FAILURE);
        }

    sscanf(m_arg, "%o", &omode);

    mode = 0;
    for (i = 0; i < 9; ++i)
        mode |= (omode >> (8 - i) & 1) ? modes[i] : 0;
}
else
    mode = S_IRWXU | S_IRWXG | S_IRWXO;

umask(0);
for (i = optind; i < argc; ++i)
    if (mkdir(argv[i], mode) == -1)
        fprintf(stderr, "%s: %s\n", argv[i], strerror(errno));

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Bu örnekte mymkdir programını -m ya da --mode parametresiyle yaratılacak dizinlerin erişim haklarını alabilmektedir. Eğer bu seçenek belirtilmediyse default erişim hakları rwxrwxrwx biçimindedir.

UNIX/Linux sistemlerinde izin yaratmak için ayrıca mkdir, izin silmek için de rmdir POSIX komutları vardır. Tabii yine mkdir komutu kabuğun umask değerinden etkilenmemektedir.

mkdir ve rmdir POSIX fonksiyonlarının izin dosyaları üzerinde değişiklik yapmaktadır. Bu nedenle yaratılacak ve silinecek dizinlerin içinde bulunduğu dizinlere ilgili prosesin 'w' hakkına sahip olması gerekir.

2.4.4. stat, fstat ve lstat POSIX Fonksiyonları

Bir dosyanın yol ifadesi biliniyorsa ona ilişkin önemli bilgiler (metadata bilgileri) stat fonksiyonu ile elde edilebilir. stat fonksiyonunun fstat ve lstat isimli kardeşleri de vardır. stat fonksiyonu UNIX türevi sistemler için çok önemli bir fonksiyondur:

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

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

stat Fonksiyonunun birinci parametresi bilgisi elde edilecek dosyanın yol ifadesini, ikinci parametresi ise dosya bilgilerinin yerleştirileceği struct stat türünden yapı nesnesinin adresini almaktadır. Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. Eğer dosya zaten açılmışsa fstat fonksiyonu dosya betimleyicisini alarak dosya bilgilerini daha hızlı bulabilmektedir. lstat ise stat fonksiyonunun sembolik bağlantıları izlemeyen biçimidir. Yani bir sembolik bağlantı dosyasının kendisine stat uygulanırsa o bağlantı dosyasının referans ettiği dosyanın bilgileri elde edilmektedir. Sembolik bağlantı dosyaları l-Node tabanlı dosya sistemlerinin anlatıldığı bölümde ele alınmaktadır. Ancak lstat fonksiyonu uygulanırsa bağlantı dosyasının kendisine ilişkin dosya bilgileri elde edilir.

stat yapısı <sys/stat.h> içerisinde aşağıdaki gibi bildirilmiştir:

```
struct stat {
    dev_t    st_dev;    /* ID of device containing file */
    ino_t    st_ino;    /* inode number */
    mode_t   st_mode;   /* protection */
    nlink_t  st_nlink;  /* number of hard links */
    uid_t    st_uid;    /* user ID of owner */
    gid_t    st_gid;    /* group ID of owner */
    dev_t    st_rdev;   /* device ID (if special file) */
    off_t    st_size;   /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks; /* number of 512B blocks allocated */
    time_t   st_atime;  /* time of last access */
    time_t   st_mtime;  /* time of last modification */
    time_t   st_ctime;  /* time of last status change */
};
```

Yapının elemanları önemli bilgiler vermektedir. Ancak bu elemanların hepsi kursun bu noktasında ele alınmayacaktır. Yapının st_mode elemanı dosyanın erişim bilgilerini, st_size elemanı dosyanın uzunluğunu, st_atime, st_mtime ve st_ctime elemanları ise dosyanın erişim zamanlarını bize verir. off_t işaretli bir tamsayı türündendir. time_t ise (C'den de anımsayınız) 01/01/1970 tarihinden geçen saniye sayısını belirten aritmetik bir türdür.

Örneğin:

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

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

```

int i;

if (argc == 1) {
    fprintf(stderr, "too few parameters!..\n");
    exit(EXIT_FAILURE);
}

for (i = 1; i < argc; ++i) {
    if (stat(argv[i], &finfo) == -1) {
        fprintf(stderr, "%s: %s\n", strerror(errno), argv[i]);
        continue;
    }
    printf("%s\n", argv[i]);
    printf("-----\n");
    printf("User id: %lu\n", (unsigned long)finfo.st_uid);
    printf("Group id: %lu\n", (unsigned long)finfo.st_gid);
    printf("Size: %lu\n", (unsigned long)finfo.st_size);
    printf("-----\n");
}

return 0;
}

```

Dosyanın erişim bilgileri stat yapısı içerisindeki st_mode elemanına yerleştirilmektedir. Bu elemanın mode_t türünden olduğuna dikkat ediniz. mode_t işaretli bir tamsayı türünü belirtmektedir. Bunun hangi bitlerinin hangi anlama geldiği sistemden sisteme değişebileceği için erişim haklarının tek tek S_IXXX sembolik sabitleriyle bit AND işlemine sokularak elde edilmesi uygun olur. Dosyanın türü de yine yapının st_mode elemanında kodlanmıştır. <sys/stat.h> başlık dosyası içerisindeki S_ISXXX makroları bizden bu elemanın değerini alıp sıfır ya da sıfır dışı bir değere geri dönmektedir. Örneğin S_ISDIR makrosu dosyanın bir dizin dosyası olup olmadığını test eder. Bu makroların listesi şöyledir:

Makro	Anlamı
S_ISREG	Dosya normal bir dosya mı?
S_ISDIR(m)	Dosya bir dizin mi belirtmektedir?
S_ISCHR(m)	Dosya karakter aygıt sürücüsü mü belirtiyor?
S_ISBLK(m)	Dosya blok aygıt sürücüsü mü belirtiyor?
S_ISFIFO(m)	Dosya isimli bir boru mu belirtiyor?
S_ISLNK(m)	Dosya bir sembolik bağlantı mı belirtiyor?
S_ISSOCK(m)	Dosya bir socket dosyası mı?

Aynı bilgiler istenirse S_IFXXX biçimindeki sembolik sabitlerle maskeleme yapılarak da elde edilebilir:

```

S_IFREG
S_IFDIR
S_IFCHR
S_IFBLK
S_IFIFO
S_IFLNK
S_IFSOCK

```

Yani örneğin:

```

if (S_ISDIR(m)) {
    ...
}

```

kontrolü ile,

```

if (m & S_IFDIR) {
    ...
}

```

kontrolü aynı işleve sahiptir.

Şimdi yol ifadesi ile verilen bir dosyanın erişim haklarını ls -l formatına uygun bir yazı biçimine dönüştüren bir fonksiyon yazmak isteyelim. Bu fonksiyonu yazarken henüz bahsetmediğimiz birkaç POSIX fonksiyonunu daha kullanacağız. Yazacağımız fonksiyonun prototipi şöyle olsun:

```
const char *get_ls_info(mode_t mode);
```

Fonksiyon dosyanın mode_t türünden erişim bilgilerini parametre olarak alarak onun "ls -l" formatındaki bilgilerinin bulunduğu char türden static yerel bir dizinin adresiyle geri dönmektedir. Örnek bir gerçekleştirim şöyle olabilir:

```
const char *get_ls_info(mode_t mode)
{
    static char str[11];
    static mode_t modes[] = { S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH,
S_IXOTH };
    int i;

    if (S_ISREG(mode))
        str[0] = '-';
    else if (S_ISDIR(mode))
        str[0] = 'd';
    else if (S_ISCHR(mode))
        str[0] = 'c';
    else if (S_ISBLK(mode))
        str[0] = 'b';
    else if (S_ISFIFO(mode))
        str[0] = 'p';
    else if (S_ISLNK(mode))
        str[0] = 'l';
    else if (S_ISSOCK(mode))
        str[0] = 's';

    for (i = 0; i < 9; ++i)
        str[i + 1] = (mode & modes[i]) ? "rwx"[i % 3] : '-';
    str[10] = '\0';

    return str;
}
```

struct stat yapısı içerisinde dosyanın kullanıcı id'si ve grup id'si sayısal biçimde kodlanmıştır. Gerçekten de işletim sistemi çekirdeği kullanıcı id'lerini ve grup id'lerini hep sayısal düzeyde işlemlere sokar. Ancak bu biçim bizim için okunabilir değildir. Biz kullanıcı ve grup id'lerini daha okunabilir biçimde yazısal olarak elde etmek isteriz.

UNIX/Linux sistemlerinde kullanıcılar için okunabilirliği artırmak amacıyla kullanıcı id'leri ve grup id'leri birer isimle eşleştirilmiştir. Bu isimlere kullanıcı isimleri ve grup isimleri denilmektedir. İşletim sisteminin çekirdeği aslında bu isimlerle değil bunların sayısal değerleriyle işlemleri yapmaktadır. Pek çok UNIX türevi sistemde kullanıcı id'leriyle kullanıcı isimleri /etc/passwd isimli bir text dosyada bulundurulmaktadır. Benzer biçimde grup id'leri ile grup isimleri de /etc/group isimli dosyada bulundurulmuştur.

/etc/passwd ve /etc/group dosyalarının kullanıcı id'leri (user id) ve grup id'leri (group) id root biçimindedir. Bu dosya sıradan kişilere yalnızca okuma hakkı vermektedir. Yani sıradan kişiler bu dosyanın içerisinde değişiklik yapamazlar.

```
csd@csd-virtual-machine ~/Study/SysProg-2019 $ ls -l /etc/passwd /etc/group
-rw-r--r-- 1 root root 1054 May 31 2017 /etc/group
-rw-r--r-- 1 root root 2434 Kas 12 16:08 /etc/passwd
```

/etc/passwd dosyası text bir dosyadır ve satırlardan oluşmuştur. Bu dosyanın içeriğinin bir bölümünü aşağıda görüyorsunuz:

```
csd@csd-virtual-machine ~/Study/SysProg-2019 $ tail /etc/passwd
pulse:x:116:122:PulseAudio daemon,,,:/var/run/pulse:/bin/false
mdm:x:117:124:MDM Display Manager:/var/lib/mdm:/bin/false
nm-openvpn:x:118:126:NetworkManager OpenVPN,,,:/var/lib/openvpn/chroot:/bin/false
rtkit:x:119:127:RealtimeKit,,,:/proc:/bin/false
saned:x:120:128:./var/lib/saned:/bin/false
usbmux:x:121:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
csd:x:1000:1001:CSD,,,:/home/csd:/bin/bash
telnetd:x:122:131:./nonexistent:/bin/false
ftp:x:123:132:ftp daemon,,,:/srv/ftp:/bin/false
kaan:x:1001:1001:CSD,,,:/home/kaan:/bin/bash
```

Bu dosyadaki her satır 7 içerğe sahiptir. İçerikler ':' karakteri ile birbirlerinden ayrılmıştır. İlk içerikte kullanıcının ismi, ikinci içerikte parolası bulunmaktadır. Eğer parola yerinde x varsa bu durum parolanın şifreli bir biçimde /etc/shadow dosyasında saklandığını belirtir. Üçüncü içerikte kullanıcı id'si ve dördüncü içerikte grup id'si bulunmaktadır. Diğer içeriklerden başka konuların içerisinde bahsedilecektir.

```
csd@csd-virtual-machine ~/Study/SysProg-2019 $ tail /etc/passwd
pulse:x:116:122:PulseAudio daemon,,,:/var/run/pulse:/bin/false
mdm:x:117:124:MDM Display Manager:/var/lib/mdm:/bin/false
nm-openvpn:x:118:126:NetworkManager OpenVPN,,,:/var/lib/openvpn/chroot:/bin/false
rtkit:x:119:127:RealtimeKit,,,:/proc:/bin/false
saned:x:120:128:./var/lib/saned:/bin/false
usbmux:x:121:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
csd:x:1000:1001:CSD,,,:/home/csd:/bin/bash
telnetd:x:122:131:./nonexistent:/bin/false
ftp:x:123:132:ftp daemon,,,:/srv/ftp:/bin/false
kaan:x:1001:1001:CSD,,,:/home/kaan:/bin/bash
```

↓
Kullanıcı ismi
↓
Kullanıcı id'si
↓
Grup id'si

/etc/group isimli dosyada da gruplara ilişkin bilgiler vardır. Bu dosya da satırlardan oluşmaktadır. Örneğin:

```
csd@csd-virtual-machine ~/Study/SysProg-2019 $ tail /etc/group
nm-openvpn:x:126:
rtkit:x:127:
saned:x:128:
vboxsf:x:129:
csd:x:1000:
sambashare:x:130:csd
telnetd:x:131:
ftp:x:132:
study:x:1001:
wireshark:x:133:
csd@csd-virtual-machine ~/Study/SysProg-2019 $
```

/etc/group dosyası da ':' karakterleriyle birbirlerinden ayrılmış sütunlardan oluşmaktadır. O halde /etc/passwd ve /etc/group dosyaları parse edilip kullanıcı id'lerine ve grup id'lerine karşı gelen kullanıcı isimleri ve grup isimleri bulunabilir. Aslında programcının kullanıcı id'lerini ve grup id'lerini kullanıcı isimlerine ve grup isimlerine dönüştürmek için bu dosyaları parse etmesi gerekmez. Zaten bu işlemi yapan birkaç POSIX fonksiyonu bulundurulmuştur. Bu fonksiyonlar bu dosyalara bakarak bize istediğimiz bilgileri vermektedir.

getpwnam isimli fonksiyon bizden kullanıcının ismini alır bize o kullanıcının bilgilerini verir. Benzer biçimde getpwuid fonksiyonu da bizden kullanıcının id'sini alıp kullanıcı bilgilerini vermektedir. Kullanıcının bilgileri /etc/passwd dosyasının satırlarındaki bilgilerdir ve bu bilgiler struct passwd isimli yapıyla temsil edilmiştir:

```
struct passwd {
    char    *pw_name;          /* username */
    char    *pw_passwd;       /* user password */
    uid_t   pw_uid;           /* user ID */
    gid_t   pw_gid;           /* group ID */
    char    *pw_gecos;        /* user information */
    char    *pw_dir;          /* home directory */
    char    *pw_shell;        /* shell program */
};
```

Fonksiyonların prototipleri de şöyledir:

```
#include <pwd.h>

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

getpwnam fonksiyonu kullanıcının ismini alarak, getpwuid ise kullanıcının kullanıcı id'sini alarak onun bilgilerini vermektedir. Fonksiyonlar başarısızlık durumunda NULL adrese geri dönerler.

Benzer biçimde aynı işlemleri grup için de yapan getgrgid ve getgrnam isimli fonksiyonlar da vardır. Grup bilgileri de struct group isimli bir yapıyla temsil edilmiştir:

```
struct group {
    char *gr_name;          /* group name */
    char *gr_passwd;       /* group password */
    gid_t gr_gid;         /* group ID */
    char **gr_mem;        /* NULL-terminated array of pointers to names of group members */
};
```

Fonksiyonların da prototipleri şöyledir:

```
#include <grp.h>

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

getgrnam fonksiyonu grubun ismini alarak, getgrgid ise grubun grup id'sini alarak grup bilgilerini vermektedir. Bu iki fonksiyon da yine başarısızlık durumunda NULL adrese geri dönmektedir.

Şimdi yukarıdaki programı artık kullanıcı grup id'lerini de yazdırarak biçimde yeniden düzenleyelim:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>

const char *getls_mode(mode_t mode);

int main(int argc, char *argv[])
{
    struct stat finfo;
    int i;
    struct passwd *pwd;
    struct group *grp;

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

    for (i = 1; i < argc; ++i) {
        if (stat(argv[i], &finfo) == -1) {
            fprintf(stderr, "stat: %s\n", strerror(errno));
            continue;
        }
    }
}
```

```

    pwd = getpwuid(finfo.st_uid);
    grp = getgrgid(finfo.st_gid);

    printf("-----\n");
    printf("%s\n\n", argv[i]);
    printf("Mode: %s\n", getls_mode(finfo.st_mode));
    printf("%-20s%lld\n", "Size", (long long)finfo.st_size);
    printf("%-20s%llu\n", "I-Node", (unsigned long long)finfo.st_ino);
    printf("%-20s%s(%lld)\n", "User id", pwd ? pwd->pw_name : "?????", (long long)finfo.st_uid);
    printf("%-20s%s(%lld)\n", "Group id", grp ? grp->gr_name : "?????", (long long)finfo.st_gid);
    printf("%-20s", "Last Update", ctime(&finfo.st_mtime));
    printf("%-20s", "Last Read", ctime(&finfo.st_atime));
    printf("%-20s", "Last I-Node Update", ctime(&finfo.st_ctime));
}

return 0;
}

const char *getls_mode(mode_t mode)
{
    static char tmode[11];
    mode_t modef[] = { S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH
};
    char *modec = "rwx";
    int i;

    if (S_ISREG(mode))
        tmode[0] = '-';
    else if (S_ISDIR(mode))
        tmode[0] = 'd';
    else if (S_ISCHR(mode))
        tmode[0] = 'c';
    else if (S_ISBLK(mode))
        tmode[0] = 'b';
    else if (S_ISFIFO(mode))
        tmode[0] = 'p';
    else if (S_ISLNK(mode))
        tmode[0] = 'l';
    else if (S_ISSOCK(mode))
        tmode[0] = 's';

    for (i = 0; i < 9; ++i)
        tmode[i + 1] = mode & modef[i] ? modec[i % 3] : '-';
    tmode[i + 1] = '\\0'; /* may be omitted */

    return tmode;
}

```

getpwnam ve getpwuid fonksiyonlarını kullanabilmemiz için kullanıcının kullanıcı ismini ya da kullanıcı id'sini biliyor olmamız gerekir. Pekiyi tüm kullanıcıların bilgilerini nasıl elde edebiliriz? İşte bunun üç POSIX fonksiyonu daha bulundurulmuştur:

```

#include <pwd.h>

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

```

getpwent fonksiyonu her çağrıldığında bir kullanıcıya ilişkin kullanıcı bilgileri elde edilir. Listenin sonuna gelindiğinde fonksiyon NULL adrese geri dönmektedir. İşlemin sonunda endpwent fonksiyonu çağrılmalıdır. Eğer aynı işlem bir kez daha yapılacaksa tüm işlemleri yenden başlatmak için setpwent fonksiyonu çağrılmalıdır. İşin başında setpwent fonksiyonun çağrılmasına gerek yoktur.

Kullanıcı bilgilerinin tek tek elde edilmesi için kullanılan yukarıdaki arayüzün aynısı grup bilgilerinin elde edilmesi için de bulundurulmuştur. Grup bilgilerinin elde edilmesinde kullanılan POSIX fonksiyonları da şunlardır:

```
#include <grp.h>

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

Benzer biçimde getgrent fonksiyonu her çağrıldığında bize bir grubun bilgisi verilir. Listenin sonuna gelindiğinde fonksiyon NULL adrese geri döner. İşlemin sonunda endgrent fonksiyonu çağrılmalıdır. İşlem yeniden başlatılmak isteniyorsa setgrent fonksiyonu çağrılır. İşin başında bu fonksiyonun çağrılmasına gerek yoktur.

Aşağıda sistemdeki tüm kullanıcıların ver gruplarının isimlerini ekrana yazdıran örnek bir program görüyorsunuz:

```
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>
#include <grp.h>

int main(void)
{
    struct passwd *pwd;
    struct group *grp;

    while ((pwd = getpwent()) != NULL)
        printf("%s\n", pwd->pw_name);

    endpwent();

    printf("-----\n");

    while ((grp = getgrent()) != NULL)
        printf("%s\n", grp->gr_name);

    endgrent();

    return 0;
}
```

Şimdi de ls -l komutundaki tüm satırı yazdıran yeni bir get_ls fonksiyonunu yazalım. Bu fonksiyon bizden dosyanın yol ifadesini alsın ve bize tamamen ls -l formatında kodlanmış bir yazı versin:

```
const char *get_ls(const char *path);
```

Fonksiyon şöyle yazılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>

void exit_sys(const char *msg);
const char *get_ls(const char *path);

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



```

struct stat finfo;
int i;

if (argc == 1) {
    fprintf(stderr, "too few parameters!..\n");
    exit(EXIT_FAILURE);
}

for (i = 1; i < argc; ++i) {
    if (stat(argv[i], &finfo) == -1) {
        fprintf(stderr, "%s: %s\n", strerror(errno), argv[i]);
        continue;
    }
    printf("%s\n", get_ls(argv[i]));
}

return 0;
}

const char *get_ls(const char *path)
{
    struct stat finfo;
    static char buf[4096];
    static mode_t modes[] = { S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH,
S_IXOTH };
    struct passwd *pass;
    struct group *gr;
    int index = 0;
    int i;

    if (stat(path, &finfo) == -1)
        return NULL;

    if (S_ISREG(finfo.st_mode))
        buf[index] = '-';
    else if (S_ISDIR(finfo.st_mode))
        buf[index] = 'd';
    else if (S_ISCHR(finfo.st_mode))
        buf[index] = 'c';
    else if (S_ISBLK(finfo.st_mode))
        buf[index] = 'b';
    else if (S_ISFIFO(finfo.st_mode))
        buf[index] = 'p';
    else if (S_ISLNK(finfo.st_mode))
        buf[index] = 'l';
    else if (S_ISSOCK(finfo.st_mode))
        buf[index] = 's';
    ++index;

    for (i = 0; i < 9; ++i)
        buf[index++] = (finfo.st_mode & modes[i]) ? "rwx"[i % 3] : '-';
    buf[index] = '\0';

    index += sprintf(buf + index, " %llu", (unsigned long long)finfo.st_nlink);

    if ((pass = getpwuid(finfo.st_uid)) == NULL)
        return NULL;
    index += sprintf(buf + index, " %s", pass->pw_name);

    if ((gr = getgrgid(finfo.st_gid)) == NULL)
        return NULL;
    index += sprintf(buf + index, " %s", gr->gr_name);

    index += sprintf(buf + index, " %lld", (long long)finfo.st_size);
    index += strftime(buf + index, 100, " %b %e %H:%M", localtime(&finfo.st_mtime));
}

```



```

    sprintf(buf + index, "%s", path);

    return buf;
}

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

    exit(EXIT_FAILURE);
}

```

Burada yazdığımız fonksiyonda önce stat fonksiyonunu çağırarak dosya bilgilerini elde ettik. stat fonksiyonunun bize dosyanın kullanıcı id'sini (user id) ve grup id'sini (group id) sayı olarak verdiğini anımsayınız. Biz de getpwuid ve getgrgid fonksiyonlarıyla bu sayıları kullanıcı ismine (user name) ve grup ismine (group name) dönüştürdük. Anımsanacağı gibi stat yapısı içerisinde dosyaya ilişkin üç tarih bilgisi vardır: Bunlar st_atime, st_mtime ve st_ctime elemanlarıyla temsil edilmiştir. Bu tarih bilgilerinin time_t türünden olduğuna dikkat ediniz. Yapının bu alanları 01/01/1970'ten geçen saniye saniye sayısını tutmaktadır. İşte biz de bu bilgiyi alıp protipleri <time.h> içerisinde olan localtime ve strptime standart C fonksiyonlarını kullanarak dosyanın son değiştirilme zamanını (st_mtime) ls -l komutunda verildiği biçime dönüştürdük.

3. Dizin İçerisindeki Dosyaların Elde Edilmesi

Bir dizin içerisindeki dosyaların elde edilmesi isteğiyle uygulamalarda çok karşılaşılmaktadır. Bu işlem aslında en düşük düzeyde işletim sistemlerinin sistem fonksiyonları tarafından yapılmaktadır. Ancak programcılar Windows sistemlerinde dizin listesini almak için API fonksiyonlarından, UNIX/Linux sistemlerinde de POSIX fonksiyonlardan faydalanırlar. .NET ve Java gibi ortamlarda ise bu işlemler neredeyse tek bir fonksiyona (metoda) indirgenmiş durumdadır. Tabii aslında bu fonksiyonlar da eninde sonunda Windows sistemlerinde ilgili API fonksiyonlarını, UNIX/Linux sistemlerinde de ilgili POSIX fonksiyonlarını çağırılmaktadır.

3.1. Windows Sistemlerinde Dizin İçerisindeki Dosyaların Elde Edilmesi

Windows sistemlerinde dizin içerisindeki dosyaların listesi sırasıyla şu aşamalardan geçilerek elde edilmektedir:

1) Öncelikle FindFirstFile isimli API fonksiyonuyla belirtilen koşulu sağlayan ilk dosyanın bilgileri elde edilir. FindFirstFile fonksiyonunun prototipi şöyledir:

```

HANDLE WINAPI FindFirstFile(
    LPCTSTR lpFileName,
    LPWIN32_FIND_DATA lpFindFileData
);

```

Fonksiyona birinci parametresi bir yol ifadesi (pathname) biçiminde girilir. Bu yol ifadesinde '*' ve '?' joker karakterleri kullanılabilir. '*' joker karakteri "herhangi bir ya da birden fazla karakter" anlamına gelirken '?' joker karakteri ise "herhangi tek bir karakter" anlamına gelmektedir. Bu parametre örneğin "C:\\Windows*.*" ya da "C:\\Windows*.exe" biçiminde verilebilir. İkinci parametre koşulu sağlayan ilk dosyanın bilgilerinin yerleştirileceği WIN32_FIND_DATA isimli bir yapı nesnesinin adresini almaktadır. Fonksiyon başarı durumunda HANDLE türüyle (Windows'ta başı H ile başlayan türlerin void * olarak typedef edildiğini anımsayınız) temsil edilen bir handle değerine, başarısızlık durumunda ise INVALID_HANDLE_VALUE değerine geri dönmektedir. (INVALID_HANDLE_VALUE değeri <windows.h> içerisinde 0xFFFFFFFF olarak define edilmiştir.) FindFirstFile fonksiyonunun geri döndürdüğü handle değeri CreateFile fonksiyonunun geri döndürdüğü handle değeri ile aynı değildir. Yani biz FindFirstFile fonksiyonundan elde ettiğimiz handle değeri ile ReadFile ve WriteFile fonksiyonlarını çağıramayız. WIN32_FIND_DATA yapısı şöyle bildirilmiştir:

```

typedef struct _WIN32_FIND_DATA {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD nFileSizeHigh;

```

```

DWORD nFileSizeLow;
DWORD dwReserved0;
DWORD dwReserved1;
TCHAR cFileName[MAX_PATH];
TCHAR cAlternateFileName[14];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA, *LPWIN32_FIND_DATA;

```

Yapının dwFileAttributes elemanı dosyanın özellik bilgisini belirtir. Bu eleman bit bit anlamlıdır. Her bit bir özelliğin olup olmadığını belirtmektedir. Bitleri maskeleyerek için FILE_ATTRIBUTE_XXX biçiminde makrolar bulundurulmuştur. Örneğin biz bu elemanı FILE_ATTRIBUTE_ARCHIVE makrosuyla Bit And (&) işlemine sokarsak ve bundan sıfır dışı bir değer elde edersek o özelliğin var olduğunu anlamalıyız. Windows -dosya sistemine de bağlı olarak- her dosya için ilk yaratılma zamanını, son erişim zamanını ve son yazma zamanını tutmaktadır. WIN32_FIND_DATA yapısında bu tarih ve zamanlar FILETIME olarak belirtilmiştir. FILETIME bir yapı türündendir fakat programcı için kullanışlı bir tür değildir. FILETIME türü bize 01/01/1601'den geçen 100 nanosaniyelerin sayısını vermektedir. Buradaki tarih ve zaman UTC biçimindedir. İstenirse FILETIME yapısı ile belirtilen tarih ve zaman FileTimeToLocalFileTime fonksiyonuyla yerel tarih ve zamana dönüştürülebilir. FILETIME değerini normal tarih ve zaman bilgisine dönüştürmek için ise FileTimeToSystemTime fonksiyonu kullanılmaktadır. Örneğin:

```

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

void ExitSys(LPCSTR lpszMsg);

typedef unsigned my_t;

int main(void)
{
    WIN32_FIND_DATA finfo;
    HANDLE hFindFile;
    SYSTEMTIME st;

    if ((hFindFile = FindFirstFile("c:\\windows\\*.exe", &finfo)) == INVALID_HANDLE_VALUE)
        ExitSys("FindFirstFile");

    printf("%s\n", finfo.cFileName);
    printf("%d\n", finfo.nFileSizeLow);
    FileTimeToLocalFileTime(&finfo.ftLastWriteTime, &finfo.ftLastWriteTime);
    FileTimeToSystemTime(&finfo.ftLastWriteTime, &st);
    printf("Last Write Time: %02d/%02d/%04d %02d:%02d:%02d\n",
        st.wDay, st.wMonth, st.wYear, st.wHour, st.wMinute, st.wSecond);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Yapının nFileSizeLow ve nFileFileSizeHight elemanlarında dosyanın uzunluğu tutulmaktadır. Windows dosya sistemlerinde (fakat hepsinde değil) bir dosyanın olası maksimum uzunluğu 64 bitle temsil edilmektedir. (Dolayısıyla bu değer en fazla 2⁶⁴ byte olabilir.) Fakat bu 64 bitlik değer 32 bit sistemlerde iki ayrı 32 bitlik parçayla temsil edilmiştir. (32 bit sistemlerde 64 bit uzunluğunda doğal bir türün olmadığına dikkat ediniz.) Yapının cFileName elemanı dosyanın ismini

(yol ifadesi olmadan) belirtmektedir. `cAlternateFileName` ise dosyanın DOS uyumlu 11 karakterli (8 + 3) dosya ismini temsil etmektedir. Ancak bu eleman her dosya sisteminde bulunmamaktadır.

2) Belirtilen koşulu sağlayan diğer dosyaların bilgileri bir döngü içerisinde `FindNextFile` fonksiyonu çağrılarak elde edilmektedir. `FindNextFile` fonksiyonunun prototipi şöyledir:

```
BOOL WINAPI FindNextFile(  
    HANDLE hFindFile,  
    LPWIN32_FIND_DATA lpFindFileData  
);
```

Fonksiyonun birinci parametresi `FindFirstFile` fonksiyonundan elde edilen handle değerini, ikinci parametresi ise bulunan dosyanın bilgilerinin yerleştirileceği `WIN32_FIND_DATA` türünden nesnenin adresini alır. Fonksiyon yeni bir dosyayı bulamamışsa sıfır değerine bulmuşsa sıfır dışı bir değere geri döner.

3) En sonunda `FindFirstFile` fonksiyonuyla elde edilen handle `FindClose` fonksiyonuyla kapatılmalıdır. `FindClose` fonksiyonunun prototipi şöyledir:

```
BOOL WINAPI FindClose(  
    HANDLE hFindFile  
);
```

Fonksiyon `FindFirstFile` fonksiyonundan elde edilen handle değerini parametre olarak alır, başarı durumunda sıfır değerine başarısızlık durumunda sıfır dışı bir değere geri döner.

Dizin listesini elde eden bir örnek şöyle verilebilir:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <Windows.h>  
  
void ExitSys(LPCSTR lpszMsg);  
  
typedef unsigned my_t;  
  
int main(void)  
{  
    WIN32_FIND_DATA finfo;  
    HANDLE hFindFile;  
    SYSTEMTIME st;  
  
    if ((hFindFile = FindFirstFile("c:\\windows\\*.exe", &finfo)) == INVALID_HANDLE_VALUE)  
        ExitSys("FindFirstFile");  
  
    do {  
        printf("%s\n", finfo.cFileName);  
        printf("%d\n", finfo.nFileSizeLow);  
        FileTimeToLocalFileTime(&finfo.ftLastWriteTime, &finfo.ftLastWriteTime);  
        FileTimeToSystemTime(&finfo.ftLastWriteTime, &st);  
        printf("Last Write Time: %02d/%02d/%04d %02d:%02d:%02d\n",  
            st.wDay, st.wMonth, st.wYear, st.wHour, st.wMinute, st.wSecond);  
        printf("-----\n");  
    } while (FindNextFile(hFindFile, &finfo));  
  
    FindClose(hFindFile);  
  
    return 0;  
}  
  
void ExitSys(LPCSTR lpszMsg)  
{  
    DWORD dwLastError = GetLastError();
```

```

LPTSTR lpszErr;

if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
    fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
    LocalFree(lpszErr);
}

exit(EXIT_FAILURE);
}

```

FindFirstFile ve FindNextFile fonksiyonları hem normal dosyaları hem de dizin dosyalarını bulmaktadır. Başka bir deyişle dizinler de birer dosya olarak kabul edilmektedir. Bu nedenle bulunan bir dosyanın normal bir dosya mı yoksa bir dizin dosyası mı olduğu dosyanın özelliklerine bakılarak anlaşılabilir. Eğer dosya özelliklerinde FILE_ATTRIBUTE_DIRECTORY biti set edilmişse bu dosya aslında bir dizin belirtmektedir. Şimdi yukarıdaki programı dizin isimlerinin yanına "<DIR>" yazacak biçimde yeniden düzenleyelim:

```

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

void ExitSys(LPCSTR lpszMsg);

typedef unsigned my_t;

int main(void)
{
    WIN32_FIND_DATA finfo;
    HANDLE hFindFile;
    SYSTEMTIME st;

    if ((hFindFile = FindFirstFile("c:\\windows\\*.\"", &finfo)) == INVALID_HANDLE_VALUE)
        ExitSys("FindFirstFile");

    do {
        printf("%s %s\n", finfo.cFileName, finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY ? "<DIR>"
: "");
        if (!(finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY))
            printf("%d\n", finfo.nFileSizeLow);
        FileTimeToLocalFileTime(&finfo.ftLastWriteTime, &finfo.ftLastWriteTime);
        FileTimeToSystemTime(&finfo.ftLastWriteTime, &st);
        printf("Last Write Time: %02d/%02d/%04d %02d:%02d:%02d\n", st.wDay, st.wMonth, st.wYear,
st.wHour, st.wMinute, st.wSecond);
        printf("-----\n");
    } while (FindNextFile(hFindFile, &finfo));

    FindClose(hFindFile);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Windows'ta dizinlerin uzunlukları 0 olarak elde edilmektedir. Tabii buradaki 0 o dizinin içinin boş olduğu anlamına gelmez. Biz de örneğimizde eğer bulunan dosya bir dizin dosyasıysa onun uzunluğunu yazdırmadık.

Hem Windows hem de UNIX/Linux sistemlerinde bir alt dizin yaratıldığında onun içerisinde otomatik olarak ismi "." olan ve ".." olan iki dizin yaratılmaktadır. (Eskiden DOS ve Windows bu dizinleri görüntülüyordu. Ancak Windows'un arayüzü varsayılan durumda artık bu dizinleri görüntülememektedir. Fakat komut satırında "dir" komutuyla bu dizinler görüntülenebilmektedir.) "." dizini bulunulan dizini, ".." dizini ise üst dizini (parent directory) belirtmektedir.

3.2. UNIX/Linux Sistemlerinde Dizin İçerisindeki Dosyaların Elde Edilmesi

UNIX/Linux sistemlerinde bir dizin içerisindeki dosyalar sırasıyla şu aşamalardan geçilerek elde edilir:

1) Öncelikle dizin opendir isimli POSIX fonksiyonuyla açılır:

```
#include <dirent.h>
```

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

Fonksiyon parametre olarak içeriği elde edilecek dizinin yol ifadesini alır, DIR türünden bir yapı nesnesinin adresine geri döner. Programcının bu DIR yapısının içeriğini bilmesi gerekmemektedir. (Tıpkı fopen fonksiyonunun geri döndürdüğü FILE yapısının içeriğinin bilinmesi gerekmediği gibi). Bu adres bir handle olarak kullanılmaktadır. opendir fonksiyonu başarısızlık durumunda NULL adrese geri dönmektedir. opendir fonksiyonuyla dizinin açılması için prosesin dizine okuma hakkının bulunuyor olması gerekir.

2) Dizindeki dosyalar bir döngü içerisinde readdir fonksiyonu çağrılarak tek tek elde edilir:

```
#include <dirent.h>
```

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

Fonksiyon opendir fonksiyonundan elde edilen handle değerini parametre olarak alır ve struct dirent türünden static ömürlü bir yapı nesnesinin adresiyle geri döner. Fonksiyon elde ettiği dosyanın ismini ve i-node numarasını bu nesnesinin içerisine yerleştirmektedir. dirent yapısı şöyle bildirilmiştir:

```
struct dirent {
    ino_t d_ino;
    char d_name[256];
};
```

Çeşitli UNIX türevi sistemler bu yapıya eklenti biçiminde bazı eleman ekleyebilmektedir. Ancak POSIX standartlarında yapının en azından yukarıdaki elemanları içermesi öngörülmektedir.

readdir dizin listesinin sonuna gelmişse ya da okuma hatası (IO error) durumunda NULL adrese geri dönmektedir. Fonksiyonun dizin listesinin sonuna geldiğinden dolayı mı yoksa okuma hatasından dolayı mı NULL değeri geri döndüğünü anlamak için şu yöntem izlenmelidir: Fonksiyon çağrılmadan önce errno değişkenine 0 değeri atanır. Fonksiyon NULL değeri ile geri dönmüşse errno değişkeninin değerine bakılır. Eğer errno 0'dan farklı bir değerdeseyse okuma hatasının söz konusu olduğu anlaşılır. Anımsanacağı gibi errno değeri hiçbir fonksiyon tarafından 0 değerine çekilmemektedir. Ancak bir POSIX fonksiyonu başarılı olsa bile errno değerini -eğer dokümantasyonda açıkça belirtilmemişse- set edebilmektedir.

Görüldüğü gibi readdir bize dosyanın isminden ve i-node numarasından başka bir bilgi vermemektedir. i-node numarası dosyayı betimleyen sistem genelinde tek olan (unique) bir numaradır. i-node numarasının ne anlam ifade ettiği başka bir bölümde ele alınacaktır.

readdir fonksiyonu bize hem normal dosyaları hem de dizin dosyalarını vermektedir. readdir ile elde ettiğimiz dizin girişinin sıradan bir dosya mı yoksa bir dizin mi olduğunu stat fonksiyonu uygulayarak anlayabiliriz.

3) İşlem bitince opendir fonksiyonuyla açılmış olan dizin closedir fonksiyonuyla kapatılır. Tabii dosyalarda olduğu gibi biz dizini kapatmamışsak proses sonlandığında dizin sorunsuz olarak işletim sistemi tarafından zaten kapatılmaktadır.

```
#include <dirent.h>
```

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

Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. Her şey düzgün yapılmışsa başarının kontrol edilmesine gerek yoktur. Ancak debug amaçlı kontrol yapılabilir.

Örneğin:

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

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

```
int main(int argc, char *argv[])
{
    DIR *dir;
    struct dirent *dent;

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

    if ((dir = opendir(argv[1])) == NULL)
        exit_sys("open");

    while (errno = 0, (dent = readdir(dir)) != NULL)
        printf("%s\n", dent->d_name);

    if (errno != 0)
        exit_sys("readdir");

    closedir(dir);

    return 0;
}
```

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

    exit(EXIT_FAILURE);
}
```

UNIX/Linux sistemlerindeki işlemlerle Windows sistemlerindeki işlemler arasındaki iki önemli farklılığa dikkat ediniz:

1) Windows'ta dizin içerisinde belli kalıptaki dosyalar joker karakterleriyle elde edilebilmektedir. Oysa UNIX/Linux sistemlerinde böyle bir kullanım yoktur.

2) Windows sistemlerinde elde edilen dosyaların bütün bilgileri (uzunlukları, özellikleri vs.) bize WIN32_FIND_DATA yapısıyla verilmektedir. Halbuki UNIX/Linux sistemlerinde struct dirent yapısında yalnızca bize dosyaların isimleri ve inode numaraları verilmektedir.

Dizin listesi elde edilirken her bulunan dosya ayrıca stat fonksiyonuna sokularak onun bilgileri de elde edilebilir. Aslında POSIX ls komutu da dosya bilgilerini bu biçimde bularak ekrana yazdırmaktadır. Örneğin:

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

#define MAX_PATH    4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    DIR *dir;
    struct dirent *dent;
    struct stat finfo;
    char path[MAX_PATH];

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

    if ((dir = opendir(argv[1])) == NULL)
        exit_sys("open");

    while (errno = 0, (dent = readdir(dir)) != NULL) {
        sprintf(path, "%s/%s", argv[1], dent->d_name);
        if (stat(path, &finfo) == -1)
            exit_sys("stat");

        printf("-----\n");
        printf("%-25s\n", dent->d_name, S_ISDIR(finfo.st_mode) ? "<DIR>" : "");
        printf("%lld\n", (long long)finfo.st_size);
    }

    if (errno != 0)
        exit_sys("readdir");

    closedir(dir);

    return 0;
}

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

    exit(EXIT_FAILURE);
}
```

Şimdi de bir dizindeki dosyaların hepsini "ls -l" formatında yazdırmaya çalışalım. Bu programı daha önce yazmış olduğumuz get_ls fonksiyonu çağırarak kolay biçimde yazabiliriz:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
```

```

#include <dirent.h>

#define MAX_PATH    4096

void exit_sys(const char *msg);
const char *get_ls(const char *path);

int main(int argc, char *argv[])
{
    DIR *dir;
    struct dirent *dent;
    struct stat finfo;
    char path[MAX_PATH];

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

    if ((dir = opendir(argv[1])) == NULL)
        exit_sys("open");

    while (errno = 0, (dent = readdir(dir)) != NULL) {
        sprintf(path, "%s/%s", argv[1], dent->d_name);
        if (stat(path, &finfo) == -1)
            exit_sys("stat");

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

    if (errno != 0)
        exit_sys("readdir");

    closedir(dir);

    return 0;
}

const char *get_ls(const char *path)
{
    struct stat finfo;
    static char buf[4096];
    static mode_t modes[] = { S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH,
S_IXOTH };
    struct passwd *pass;
    struct group *gr;
    char *str;
    int index = 0;
    int i;

    if (stat(path, &finfo) == -1)
        return NULL;

    if (S_ISREG(finfo.st_mode))
        buf[index] = '-';
    else if (S_ISDIR(finfo.st_mode))
        buf[index] = 'd';
    else if (S_ISCHR(finfo.st_mode))
        buf[index] = 'c';
    else if (S_ISBLK(finfo.st_mode))
        buf[index] = 'b';
    else if (S_ISFIFO(finfo.st_mode))
        buf[index] = 'p';
    else if (S_ISLNK(finfo.st_mode))
        buf[index] = 'l';

```



```

else if (S_ISSOCK(finfo.st_mode))
    buf[index] = 's';
++index;

for (i = 0; i < 9; ++i)
    buf[index++] = (finfo.st_mode & modes[i]) ? "rwx"[i % 3] : '-';
buf[index] = '\\0';

index += sprintf(buf + index, " %llu", (unsigned long long)finfo.st_nlink);

if ((pass = getpwuid(finfo.st_uid)) == NULL)
    return NULL;
index += sprintf(buf + index, " %s", pass->pw_name);

if ((gr = getgrgid(finfo.st_gid)) == NULL)
    return NULL;
index += sprintf(buf + index, " %s", gr->gr_name);

index += sprintf(buf + index, " %lld", (long long)finfo.st_size);
index += strftime(buf + index, 100, " %b %e %H:%M", localtime(&finfo.st_mtime));

str = strrchr(path, '/');
sprintf(buf + index, " %s", str ? str + 1 : path);

return buf;
}

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

    exit(EXIT_FAILURE);
}

```

Örnek çıktı şöyle olacaktır:

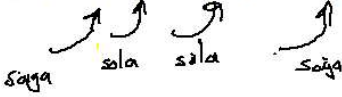
```

csd@csd-virtual-machine ~/Study/SysProg-2019 $ ./mys /home/csd/Study/SysProg-2019
-rwxr-xr-x 1 csd study 13744 Mar 10 09:58 ls-l
-rw-r--r-- 1 csd study 986 Mar  2 12:19 mample.c
-rw-r--r-- 1 csd study 3538 Mar 10 09:54 ls-l-align.c
drwxr-xr-x 4 csd study 4096 Feb  9 10:03 QtCreator-Projects
-rw-r--r-- 1 csd study 90 Feb  9 09:31 Untitled-1
-rwxr-xr-x 1 csd study 8608 Feb  9 09:33 a.out
drwxr-xr-x 10 csd csd 4096 Feb  3 11:33 ..
-rwxr-xr-x 1 csd study 13952 Mar 10 09:41 ls-l-align
-rw-r--r-- 1 csd study 183 Mar  9 12:15 test.c
-rw-r--r-- 1 csd study 2155 Mar 10 09:49 x.txt
-rw-r--r-- 1 csd study 1512 Feb  9 09:46 sample.o
-rw-r--r-- 1 csd study 2128 Mar  9 12:37 sample.c
-rw-r--r-- 1 csd study 2130 Mar 10 09:58 ls-l.c
-rwxr-xr-x 1 csd study 13744 Mar 10 09:59 mys
drwxr-xr-x 4 csd study 4096 Mar 10 09:58 .
-rw-rw-r-- 1 csd study 0 Mar  2 09:47 test.txt
drwxr-xr-x 2 csd study 4096 Mar  3 11:27 xxx
-rw-r--r-- 1 csd study 869 Mar 10 09:47 x
-rwxr-xr-x 1 csd study 13704 Mar 10 09:28 sample
-rw-rw-rw- 1 csd study 23 Feb  3 11:38 y.txt
-rwxr-xr-x 1 csd study 8608 Mar  9 12:23 test
csd@csd-virtual-machine ~/Study/SysProg-2019 $

```

Ancak buradaki çıktıda hizalamanın biraz bozuk olduğunu görüyorsunuz. Gerçekten de aslında standart "ls -l" komutu 4 alanı kendi aralarında hizalamaktadır. Örneğin:

```
csd@csd-virtual-machine ~/Study/SysProg-2017 $ ls -l
toplam 60
drwxr-xr-x 2 csd csd 4096 May 14 09:55 build-QtCreator-Sample-Desktop-Debug
-rw-r--r-- 1 csd csd 0 May 6 12:42 c
-rw-r--r-- 1 csd study 1466 May 27 12:50 mest.txt
-rwxr-xr-x 1 csd study 13416 May 27 12:49 mycp
drwxr-xr-x 2 csd csd 4096 May 17 19:19 QtCreator-Sample
-rwxr-xr-x 1 csd study 13608 May 28 11:17 sample
-rw-r--r-- 1 csd study 2201 May 28 11:25 sample.c
-rw-r--r-- 1 csd csd 4232 May 7 09:35 sample.o
-rw-rw-rw- 1 csd study 1466 May 27 12:49 test.txt
```



Anahtar Notlar: GCC derleyicisi ve ld bağlayıcısı default durumda matematik fonksiyonların bulunduğu libm.a ya da libm.so kütüphanelerine bakmamaktadır. Bu nedenle <math.h> içerisinde prototipi bulunan kodları derlerken komut satırında "-lm" seçeneğinin belirtilmesi gerekmektedir. Örneğin:

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

Bir dizinin tamamını hizalayarak yazdıran programı şöyle yazabiliriz:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <dirent.h>

#define MAX_PATH 4096

void exit_sys(const char *msg);
const char *get_ls(const char *path, int hlink_digit, int uname_digit, int gname_digit, int size_digit);

int main(int argc, char *argv[])
{
    DIR *dir;
    struct dirent *dent;
    struct stat finfo;
    char path[MAX_PATH];
    struct passwd *pass;
    struct group *gr;
    int len;
    int hlink_digit, uname_digit, gname_digit, size_digit;

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

    if ((dir = opendir(argv[1])) == NULL)
        exit_sys("open");

    hlink_digit = uname_digit = gname_digit = size_digit = 0;

    while (errno = 0, (dent = readdir(dir)) != NULL) {
        sprintf(path, "%s/%s", argv[1], dent->d_name);
        if (stat(path, &finfo) == -1)
            exit_sys("stat");
    }
}
```

```

len = (int)log10(finfo.st_nlink) + 1;
if (len > hlink_digit)
    hlink_digit = len;

if ((pass = getpwuid(finfo.st_uid)) == NULL)
    exit_sys("getppuid");

len = (int)strlen(pass->pw_name);
if (len > uname_digit)
    uname_digit = len;

if ((gr = getgrgid(finfo.st_gid)) == NULL)
    exit_sys("getgrgid");

len = (int)strlen(gr->gr_name);
if (len > gname_digit)
    gname_digit = len;

len = (int)log10(finfo.st_size) + 1;
if (len > size_digit)
    size_digit = len;
}

if (errno != 0)
    exit_sys("readdir");

rewinddir(dir);
while (errno = 0, (dent = readdir(dir)) != NULL) {
    sprintf(path, "%s/%s", argv[1], dent->d_name);
    if (stat(path, &finfo) == -1)
        exit_sys("stat");

    printf("%s\n", get_ls(path, hlink_digit, uname_digit, gname_digit, size_digit));
}

if (errno != 0)
    exit_sys("readdir");

closedir(dir);

return 0;
}

const char *get_ls(const char *path, int hlink_digit, int uname_digit, int gname_digit, int size_digit)
{
    struct stat finfo;
    static char buf[4096];
    static mode_t modes[] = { S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH,
S_IXOTH };
    struct passwd *pass;
    struct group *gr;
    char *str;
    int index = 0;
    int i;

    if (stat(path, &finfo) == -1)
        return NULL;

    if (S_ISREG(finfo.st_mode))
        buf[index] = '-';
    else if (S_ISDIR(finfo.st_mode))
        buf[index] = 'd';
    else if (S_ISCHR(finfo.st_mode))
        buf[index] = 'c';
    else if (S_ISBLK(finfo.st_mode))

```

```

    buf[index] = 'b';
else if (S_ISFIFO(finfo.st_mode))
    buf[index] = 'p';
else if (S_ISLNK(finfo.st_mode))
    buf[index] = 'l';
else if (S_ISSOCK(finfo.st_mode))
    buf[index] = 's';
++index;

for (i = 0; i < 9; ++i)
    buf[index++] = (finfo.st_mode & modes[i]) ? "rwx"[i % 3] : '-';
buf[index] = '\\0';

index += sprintf(buf + index, "%*llu", hlink_digit, (unsigned long long)finfo.st_nlink);

if ((pass = getpwuid(finfo.st_uid)) == NULL)
    return NULL;
index += sprintf(buf + index, "%-*s", uname_digit, pass->pw_name);

if ((gr = getgrgid(finfo.st_gid)) == NULL)
    return NULL;
index += sprintf(buf + index, "%-*s", gname_digit, gr->gr_name);

index += sprintf(buf + index, "%*lld", size_digit, (long long)finfo.st_size);
index += strftime(buf + index, 100, "%b %e %H:%M", localtime(&finfo.st_mtime));

str = strrchr(path, '/');
sprintf(buf + index, "%s", str ? str + 1 : path);

return buf;
}

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

    exit(EXIT_FAILURE);
}

```

Bu örnekte izin listesinin elde edilmesi ve stat çağruları birden fazla kez yapılmıştır. Bunu engellemek için önce dosyalar bulunarak dinamik bir diziye yerleştirilebilir. Sonra işlemlere bu diziden devam edilebilir. (ls komutu bu biçimde bir yöntem izlemektedir.) Ayrıca standart ls komutu dizini varsayılan durumda dosyaları isme göre sıralı göstermektedir. Oysa yukarıdaki kodda biz izin listesini sıralamadık.

4. Fonksiyon Göstericileri (Pointers to Functions / Function Pointers)

Fonksiyonlar da aslında ardışıl makine komutlarından oluşmaktadır. Bir fonksiyonun çağırılması için makine dillerinde genellikle CALL isimli makine komutları kullanılır. Tipik olarak fonksiyon çağırma kullanılan CALL makine komutu çağrılacak fonksiyonun başlangıç adresini operand olarak almaktadır. CALL makine komutu sonraki komutun adresini stack'e atarak o adrese daller. Fonksiyondaki RET makine komutu da stack'ten geri dönüş adresini alarak akışın çağırılan yerden devam etmesini sağlar. Özetle bir fonksiyonun çağrılabilmesi için aslında onun yalnızca başlangıç adresinin bilinmesi gerek ve yeter koşuldur. Örneğin:

```

int add(int a, int b)
{
    return a+b;
}

```

Gibi bir fonksiyonun makine komutları 32 bit Intel mimarisinde aşağıdaki gibidir:

```

push ebp
mov  ebp, esp
mov  eax, [ebp+8]
add  eax, [ebp+12]
pop  ebp
ret

```

Görüldüğü gibi bu makine komutları ardışıl bir biçimde bellekte bulunmaktadır. (Yukarıdaki sembolik makine komutları aslında ikilik sistemde byte'lardan oluşan değerlere karşılık gelmektedir. Yani fonksiyonların makine komutları da aslında bellekte ikilik sistemde sayılar biçiminde tutulmaktadır.)

İşte fonksiyonların başlangıç adreslerini tutabilen göstericilere fonksiyon göstericileri denilmektedir.

Fonksiyon gösterici bildirimlerinin genel biçimi şöyledir:

```
[geri dönüş değerinin türü]<(*<gösterici ismi>)[parametre bildirimi];
```

Örneğin:

```

void (*pf1)(int);
int (*pf2)(long, long);
void (*pf3)(void);

```

Biz bir fonksiyon göstericisine her türden fonksiyonun adresini yerleştiremeyiz. Ancak geri dönüş değerinin türü ve parametre türleri belirli biçimde olan fonksiyonların adreslerini yerleştirebiliriz. Yukarıdaki örnekte pf1 göstericisine geri dönüş değeri void, parametresi int olan herhangi bir fonksiyonun adresi yerleştirilebilir. pf2'ye geri dönüş değeri int, parametreleri long, long olan fonksiyonların başlangıç adresleri yerleştirilebilir. pf3'e ise ancak geri dönüş değeri ve parametresi void olan fonksiyonların başlangıç adresleri yerleştirilebilir.

Fonksiyon gösterici bildirimindeki parantezlere dikkat ediniz. Bu parantezler önemlidir. Eğer bu parantezler olmasa bu bildirimler fonksiyon prototip bildirimleri haline gelirdi değil mi?. Aşağıdaki farka dikkat ediniz:

```

void (*pf1)(int);      /* fonksiyon göstericisi tanımlanmış */
void *pf2(int);       /* fonksiyon prototip bildirimi yapılmış */

```

Fonksiyon gösterici bildiriminde parametre parantezi içerisine parametre değişken isimleri de yazılabilir. Fakat programcılar genel olarak bunu pek tercih etmemektedir. Örneğin:

```
void (*pf2)(long a, long b);    /* geçerli fakat pek tercih edilmiyor */
```

Tabii buradaki parametre isimleri tıpkı prototipte olduğu gibi istenildiği gibi verilebilmektedir. Ancak yukarıda da belirttiğimiz gibi programcılar genellikle fonksiyon gösterici bildiriminde parametre isimlerini kullanmazlar.

C'de bir fonksiyonun yalnızca ismi (yani parantezler olmadan yalnızca ismi) onun başlangıç adresi anlamına gelmektedir. Fonksiyon çağırırken kullandığımız parantezler operatör görevindedir. Örneğin:

```

#include <stdio.h>

void foo(void)
{
    printf("foo\n");
}
...
void (*pf)(void);

pf = foo;

```

Burada foo fonksiyonunun adresi pf göstericisine atanmıştır.

foo gibi bir fonksiyonun yalnızca foo biçiminde kullanılmasıyla foo(...) biçiminde kullanılması arasındaki farka dikkat ediniz. foo ifadesi fonksiyonun başlangıç adresi anlamına gelirken foo(...) ifadesi foo adresinden başlayan fonksiyonun çağrılmasından sonra elde edilen geri dönüş değeri anlamına gelmektedir.

pf bir fonksiyon göstericisi olmak üzere, bu göstericinin gösterdiği yerdeki fonksiyonu çağırmak için iki eşdeğer ifade kullanılabilir: pf(...) ya da (*pf)(...). Örneğin:

```
#include <stdio.h>

void foo(void)
{
    printf("foo\n");
}

int main(void)
{
    void (*pf)(void);
    pf = foo;
    pf();      /* foo çağrılır */
    (*pf)();  /* foo çağrılır */

    return 0;
}
```

Aslında normal fonksiyonlar da (*foo)(...) ifadesiyle çağrılabilirler. Ancak bazı programcılar okunabilirliği artırmak için gösterici yoluyla fonksiyonu çağırırken (*pf)(...) sentaksını, normal fonksiyon çağırmalarında da pf(..) sentaksını tercih etmektedirler. Biz kursumuzda fonksiyon göstericileri ile fonksiyonları çağırırken de pf(...) biçimindeki doğal sentaksı tercih edeceğiz.

Benzer biçimde her ne kadar fonksiyonun ismi zaten fonksiyonun başlangıç adresini belirtiyorsa da fonksiyon ismine & operatörü uygulamak da geçerli kabul edilmektedir. Örneğin:

```
pf = foo;
```

ile,

```
pf = &foo;
```

tamamen aynı anlamdadır.

Fonksiyon göstericileri yoluyla bir fonksiyonu çağırdığımızda yine parametre sayısı kadar argüman bulundurmak zorundayız. Çağırım işleminden yine çağırdığımız fonksiyonun geri dönüş değeri elde edilmektedir. Örneğin:

```
#include <stdio.h>

int square(int a)
{
    return a * a;
}

int main(void)
{
    int (*pf)(int);
    int result;

    pf = square;
    result = pf(10);
    printf("%d\n", result);
}
```

```
    return 0;
}
```

Fonksiyon gösterici bildiriminde parametre parantezinin içinin boş bırakılmasıyla oraya void yazılması arasında farklılık vardır. Eğer parametre parantezinin içi boş bırakılırsa bu durum o göstericinin parametrik yapısı herhangi bir biçimde olan fakat geri dönüş değeri belli türde olan fonksiyonların adreslerini tutabileceği anlamına gelir. Örneğin:

```
void (*pf1)(void);
void (*pf2)();
```

Burada pf1'e geri dönüş değeri void, parametresi olmayan bir fonksiyonun adresini yerleştirebiliriz. Halbuki pf2'ye geri dönüş değeri void olan fakat parametrik yapısı herhangi bir biçimde olan fonksiyonların adreslerini yerleştirebiliriz.

Fonksiyon göstericilerine ilkdeğer de verebiliriz. Örneğin:

```
void(*pf)(void) = foo;
```

Bir fonksiyonun parametresi bir fonksiyon göstericisi olabilir. Örneğin:

```
void foo(void (*pf)(void))
{
    ...
}
```

Burada foo fonksiyonu geri dönüş değeri void, parametresi olmayan bir fonksiyonun adresini almaktadır. Örneğin:

```
#include <stdio.h>

void foo(void (*pf)(void))
{
    pf();
}

void bar(void)
{
    printf("bar\n");
}

void tar(void)
{
    printf("tar\n");
}

int main(void)
{
    foo(bar);
    foo(tar);

    return 0;
}
```

Bir "fonksiyon gösterici dizisi (array of pointer to function)" söz konusu olabilir. Örneğin:

```
int (*a[3])(void);
```

Burada a geri dönüş değeri int, parametresi void olan fonksiyonların adreslerini tutan 3 elemanlı bir dizidir.

Tabii bir fonksiyon gösterici dizisine de de ilkdeğer verilebilir. Örneğin:

```
void (*pf)(void) = foo;
```

Örneğin:

```
int (*a[3])(void) = {foo, bar, tar};
```

Fonksiyon göstericilerine NULL adres ataanabilir. Örneğin:

```
void (*pf)(void) = NULL;    /* geçerli */
```

Örneğin:

```
#include <stdio.h>

void foo(void)
{
    printf("foo\n");
}

void bar(void)
{
    printf("bar\n");
}

void tar(void)
{
    printf("tar\n");
}

void car(void)
{
    printf("car\n");
}

int main(void)
{
    void(*pfs[])(void) = { foo, bar, tar, car, NULL };
    int i;

    for (i = 0; pfs[i] != NULL; ++i)
        pfs[i]();

    return 0;
}
```

Bilindiği gibi C'de türlerin sembolik gösterimleri vardır. Bu gösterimler tür dönüştürmelerinde kullanılmaktadır. Pekiyi bir fonksiyon adresinin tür ifadesi nasıl temsil edilmektedir? Fonksiyon adreslerine ilişkin tür ifadeleri diğer tür ifadelerinde olduğu gibi bildirimdeki değişken ismi atılarak elde edilir. Örneğin:

Bildirim	Tür İfadesi
int a;	int
int *pi;	int *
int a[10];	int [10]
char *names[10];	char *[10]
int (*pf)(int a, int b);	int (*)(int, int)
void foo(int a);	void (*)(int)
(int)(*pfs[10])(int);	int (*[10])(int)

Pekiyi bir fonksiyonun geri dönüş değeri bir fonksiyon adresi olabilir mi? Evet! Bu durumda fonksiyon isminin soluna '*' atomunu koyup parantezlemek gerekir. Sonra bu parantezlerin soluna geri dönüş değerine ilişkin fonksiyonun geri dönüş değerinin türünü, sağında da geri dönüş değerine ilişkin fonksiyonun parametre bildirimi yerleştirilir. Örneğin:


```
long (*foo(int a))(double)
{
    ...
}
```

Burada foo'nun kendi parametresi int türündendir. foo öyle bir fonksiyon adresine geri dönmektedir ki, onun geri dönüş değeri long, parametresi double türündendir. Bu bildirim aşama aşama nasıl oluşturulduğunu gösterelim:

foo(int a)	Fonksiyonun ismi foo'dur ve parametresi int türündendir.
(*foo(int a))	Parantezlemeden dolayı fonksiyonun geri dönüş değeri bir fonksiyon adresidir.
long (*foo(int a))	Fonksiyonun geri dönüş değerine ilişkin fonksiyonun geri dönüş değeri long türündendir.
long (*foo(int a))(double)	Fonksiyonun geri dönüş değerine ilişkin fonksiyonun parametresi double türündendir.

Örneğin:

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int(*foo(void))(int, int)
{
    return add;
}

int main(void)
{
    int(*pf)(int, int);
    int result;

    pf = foo();
    result = pf(10, 20);
    printf("%d\n", result);

    result = foo()(10, 20);
    printf("%d\n", result);

    return 0;
}
```

Bu örnekte fonksiyonunun geri dönüş değeri, geri dönüş değeri int, parametreleri int, int olan bir fonksiyon adresidir. Foo fonksiyonun geri dönüş değerinin aynı türden bir fonksiyon göstericisine atandığına dikkat ediniz. Aşağıdaki çağrı da size ilginç gelebilir:

```
result = foo()(10, 20);
```

Fonksiyon çağırma operatörünün de soldan-sağa öncelikli grupta bulunduğunu anımsayınız. Bu durumda önce foo fonksiyonu çağrılacak sonra da bu çağrıdan elde edilen adresteki fonksiyon çağrılacaktır.

Bildirimler daha karmaşık olabilir. Örneğin, "öyle bir fonksiyon yazalım ki fonksiyonumuzun kendi parametresi int türden olsun fakat geri dönüş değeri bir fonksiyon adresi olsun. Ama öyle bir fonksiyonun adresi olsun ki, onun parametresi long, onun geri dönüş değeri de geri dönüş değeri double parametresi double olan bir fonksiyon adresi olsun". Bu fonksiyonun bildirimini adım adım aşağıdaki gibi yazabiliriz:

```
foo(int a)
```

```
(*foo(int a))
(*foo(int a))(long)
>(*foo(int a))(long)
>(*foo(int a))(long)(double)
double (*foo(int a))(long)(double)
```

Bu fonksiyonu kısaca şöyle ifade edebiliriz: "foo parametresi int, geri dönüş değeri, parametresi long, geri dönüş değeri parametresi double, geri dönüş değeri double türden olan bir fonksiyon adresidir".

Örneğin:

```
#include <stdio.h>

double foo(double a)
{
    printf("foo: %d\n", a);

    return 0;
}

double (*bar(long a))(double)
{
    return foo;
}

double>(*tar(int a))(long)(double)
{
    return bar;
}

int main(void)
{
    double>(*pf1)(long)(double);
    double(*pf2)(double);

    pf1 = tar(0);
    pf2 = pf1(0);
    pf2(0);

    return 0;
}
```

main'deki fonksiyon çağrısı şöyle de yapılabilirdi:

```
int main(void)
{
    tar(0)(0)(0);

    return 0;
}
```

Bildirimler daha karışık da olabilir. Fakat uygulamada böylesi karmaşık bildirimlerle karşılaşılmamaktadır.

Bir fonksiyon gösterici dizisi de söz konusu olabilir. Örneğin:

```
int (*a[10])(void);
```

Burada a geri dönüş değeri int, parametresi void olan fonksiyon adreslerinden oluşan 10 elemanlı bir dizidir. Bildirimin şöyle elde edildiğine dikkat ediniz:

```
a[10]
(*a[10])
```

```
int (*a[10])(void);
```

Örneğin:

```
#include <stdio.h>

void foo(void)
{
    printf("foo\n");
}

void bar(void)
{
    printf("bar\n");
}

void tar(void)
{
    printf("tar\n");
}

int main(void)
{
    void(*a[]) (void) = { foo, bar, tar, NULL };
    int i;

    for (i = 0; a[i] != NULL; ++i)
        a[i]();

    return 0;
}
```

C'de fonksiyon parametresi olan fonksiyon göstericileri fonksiyon prototip sentaksıyla da belirtilebilmektedir. Örneğin:

```
void foo(int(*pf)(int))
{
    /* ... */
}
```

Burada foo fonksiyonu geri dönüş değeri int parametresi int olan bir fonksiyon göstericisine sahiptir. Aynı bildirim şöyle de yapılabilirdi:

```
void foo(int pf(int))
{
    /* ... */
}
```

Bu eşdeğerlik yalnızca fonksiyon parametreleri için geçerlidir.

C'de NULL adres bir fonksiyon göstericisine atanabilir. Bu durumda fonksiyon göstericisinin içerisinde NULL adres bulunduğu söylenir. Örneğin:

```
void (*pf)(void) = NULL;
...
if (pf == NULL) {
    ...
}
```

C'de (ve tabi C++'ta da) void göstericiler veri göstericisi (pointer to data) olarak düşünülmüştür. Biz void bir göstericiye herhangi bir türden nesnenin adresini atayabiliriz fakat bir fonksiyonun adresini atayamayız. Örneğin:

```

void foo(void)
{
    ...
}
...
void *pv;
pv = foo;      /* geçersiz! */

```

C ve C++ standartlarına göre tür dönüştürmesi yapılsa bile bu durum geçerli olmaz. (C standartları void bir göstericiye bir fonksiyonun adresinin doğrudan ya da tür dönüştürme operatörüyle atanmasını yasaklamış olsa da pek çok derleyici bu duruma ses çıkartmamaktadır. Ancak pek çok C++ derleyicileri bu durumu standartlarda belirtildiği gibi geçersiz kabul etmektedir.) Yukarıdaki işlemin taşınabilir bir biçimde yapılması ancak şöyle mümkündür:

```

void *pv;
void (*pf)(void) = foo;
pv = *(void **)&pf

```

Bu işlemin tersi de şöyle yapılabilir:

```

void *pv;
void (*pf)(void);
*(void **)&pf = pv;

```

Bir fonksiyon göstericisine onunla uyumlu (compatible) olmayan bir fonksiyonun adresi doğrudan atanamaz. Başka bir deyişle parametrik yapısı ve geri dönüş değeri aynı olmayan fonksiyon adresleri arasında otomatik (standard) tür dönüştürmesi yoktur. Fakat istenirse bir fonksiyon adresi tür dönüştürme operatörü ile başka bir fonksiyon adres türüne dönüştürülebilir.. Örneğin:

```

int foo(int a)
{
    ...
}
...
void (*pf)(void);
pf = foo;          /* geçersiz! Tür uyumsuzluğu var */
pf = (void (*)(void)) foo; /* geçerli */

```

Burada fonksiyon adreslerinin tür ifadelerinde parantez içerisinde yalnızca * atomunun bulunduğuna dikkat ediniz. Örneğin C'de int (*)(long) türü, geri dönüş değeri int parametresi long olan bir fonksiyon adres türünü temsil eder. Tabii yukarıdaki dönüştürmeyi typedef bildirimini ile sadeleştirebiliriz. Örneğin:

```

int foo(int a)
{
    ...
}
...
typedef void (*PF)(void);

PF pf;
pf = (PF) foo; /* geçerli */

```

C'de void fonksiyon göstericisi olmadığına göre bir fonksiyonun adresini geçici süre bir göstericide saklayacaksa bunun için herhangi bir türden fonksiyon göstericisini kullanabiliriz. Örneğin:

```

int foo(int a)
{
    ...
}

```

```
...
typedef void (*PF)(void);

PF pfv;
pfv = (PF) foo;
...
int (*pfi)(int);
pfi = (int (*)(int))pfv;
```

Görüldüğü gibi typedef bildirimini ile karmaşık fonksiyon gösterici türlerini daha sade ifade edebilmekteyiz. Örneğin:

```
void (*PF)(void);
```

PF burada geri dönüş değeri void parametresi void olan bir fonksiyon göstericisidir. Bu da C'de sembolik olarak " void (*)(void)" biçiminde temsil edilir. Şimdi bu bildirim önüne typedef belirleyicisini getirelim:

```
typedef void (*PF)(void);
```

Burada PF artık void (*)(void) türünü temsil eder. Yani:

```
void (*pf)(void);
```

ile,

```
PF pf;
```

aynı anlamdadır. Örneğin:

```
double (*foo(int a))(long)
{
    ...
}
```

Bu bildirimini typedef bildirimini ile daha sade yazabiliriz:

```
typedef double (*PF)(long);
```

```
PF foo(int a)
{
    ...
}
```

Örneğin:

```
typedef double (*PF)(long);
typedef PF (*PFF)(int);
```

```
PFF foo(int a)
{
    ...
}
```

Burada foo'yu typedef'isiz olarak bildirmek isteyelim:

```
double ((*foo(int a))(int))(long)
{
    ...
}
```

Karışık değil mi?

4.1. Fonksiyon Göstericilerine Neden Gereksinim Duyulmaktadır?

Fonksiyon göstericileri çok çeşitli gerekçelerle kullanılabilir. Ancak en yaygın kullanım gerekçesi "kodun genelleştirilmesini ya da akışın devredilmesini" sağlama amaçlıdır. Örneğin bir fonksiyon bizim için birşeyleri buluyor olabilir. Fakat onu bulunca ne yapacağını bize bırakmak isteyebilir. İşte bunu sağlamak için fonksiyon bizden bir fonksiyonun adresini alır, onu bulunca bizden adresini aldığı fonksiyonu çağırır. O fonksiyonun içeriğini de biz yazacağımız için fonksiyona istediğimiz şeyi yaptırmış oluruz. Bir olay gerçekleştiğinde bizden alınan bir fonksiyonun çağırılması durumunda bu tür fonksiyonlara İngilizce "callback function" denilmektedir. Örneğin:

```
#include <stdio.h>

void for_each(int *pi, int size, void (*pf)(int *))
{
    int i;

    for (i = 0; i < size; ++i)
        pf(&pi[i]);
}

void square(int *pi)
{
    *pi = *pi * *pi;
}

void disp(int *pi)
{
    printf("%d\n", *pi);
}

int main(void)
{
    int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    for_each(a, 10, square);
    for_each(a, 10, disp);

    return 0;
}
```

Buradaki for_each fonksiyonu diziyi dolaşmakta ve dizinin her elemanı için o elemanın adresini argüman yaparak belirlenen fonksiyonu çağırılmaktadır. square fonksiyonunun dizinin her elemanını onun karesiyle yer değiştirmek için, disp fonksiyonunun da dizi elemanlarını yazdırmak için kullanıldığına dikkat ediniz.

Örneğin bir fonksiyon belli periyotlarla bizden aldığı bir fonksiyonu çağırıyor olabilir. Biz de periyodik bir işlem yapmak için böyle bir fonksiyondan faydalanabiliriz. Örneğin:

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

void do_periodically(int period, void(*pf)(void))
{
    for (;;) {
        pf();
        Sleep(period);
    }
}

void foo(void)
{
    putchar('.');
}
```

```

int main(void)
{
    do_periodically(1000, foo);

    return 0;
}

```

Burada `do_periodically` fonksiyonu sonsuz bir döngü içerisinde belli bir periyotta bizim verdiğimiz fonksiyonu çağırılmaktadır. Örneğimizde akışı bekletmek için Windows sistemlerindeki `Sleep` isimli API fonksiyonunu kullandık. `Sleep` fonksiyonu bekleme süresini bizden milisaniye olarak almaktadır. UNIX/Linux sistemlerinde de `sleep` isiminde benzer bir fonksiyon vardır. Ancak bu sistemlerdeki `sleep` fonksiyonun argümanı zaman aralığını saniye cinsinden alır. Aynı örneği UNIX/Linux sistemleri için şöyle yazabilirdik:

```

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

void do_periodically(int period, void(*pf)(void))
{
    for (;;) {
        pf();
        sleep(period);
    }
}

void foo(void)
{
    putchar('.');
}

int main(void)
{
    do_periodically(1, foo);

    return 0;
}

```

Burada `do_peridically` isimli fonksiyonun akışı kendi içerisinde tuttuğuna dikkat ediniz. Yani biz akış bu fonksiyonun içerisindeyken başka bir şey yapamayız. Akışı tutmadan başka bir akış içerisinde periyodik işlemler thread'lerle yapılabilmektedir. Thread'ler konusu ileride ele alınmaktadır.

`do_periodically` fonksiyonuyla ekrana saati de bastırabiliriz:

```

#include <stdio.h>
#include <time.h>
#include <windows.h>

void do_periodically(int period, void(*pf)(void))
{
    for (;;) {
        pf();
        Sleep(period);
    }
}

void put_clock(void)
{
    struct tm *pt;
    time_t t;

    time(&t);
    pt = localtime(&t);
    printf("%02d:%02d:%02d\r", pt->tm_hour, pt->tm_min, pt->tm_sec);
}

```

```

int main(void)
{
    do_periodically(1000, put_clock);

    return 0;
}

```

Şimdi de belli bir zaman geldiğinde bizim verdiğimiz bir fonksiyonu çağıran do_alarm isimli bir fonksiyon yazalım:

```

#include <stdio.h>
#include <time.h>
#include <windows.h>

void do_alarm(int hour, int minute, int second, void(*pf)(void))
{
    time_t t;
    struct tm *ptime;

    for (;;) {
        t = time(NULL);
        ptime = localtime(&t);

        if (ptime->tm_hour == hour && ptime->tm_min == minute && ptime->tm_sec == second) {
            pf();
            break;
        }
        Sleep(100);
    }
}

void alarm_proc(void)
{
    printf("ALARM...\n");
}

int main(void)
{
    do_alarm(10, 19, 20, alarm_proc);

    return 0;
}

```

Fonksiyon göstericileri "callback" mekanizmasının oluşturulması dışında başka amaçlarla da sıklıkla kullanılmaktadır. Örneğin bir "komut yorumlayıcı (command interpreter)" yazarken belli bir komut yakalandığında belirlenen bir fonksiyonun çağırılması sağlanabilir. Böyle bir tasarım komut yorumlayıcısının daha pratik bir biçimde gerçekleştirilmesini sağlayabilmektedir.

```

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

/* Symbolic Constants */

#define MAX_CMD_LINE    1024
#define MAX_PARAMS      32

/* Type Declarations */

typedef struct tagCMD {
    const char *cmdText;
    void(*proc)(void);
} CMD;

/* Function Prototypes */

```



```

void parse_cmdline(void);
void proc_dir(void);
void proc_del(void);

/* Global Object Definitions */

char g_cmdLine[MAX_CMD_LINE];
CMD g_cmds[] = {
    {"dir", proc_dir},
    {"del", proc_del },
    {NULL, NULL}
};

char *g_params[MAX_PARAMS];
int g_nparams;

/* Function Definitions */

int main(void)
{
    int i;

    for (;;) {
        printf("CSD>");
        gets(g_cmdLine);
        parse_cmdline();

        if (g_nparams == 0)
            continue;
        for (i = 0; g_cmds[i].cmdText != NULL; ++i)
            if (!strcmp(g_params[0], g_cmds[i].cmdText)) {
                g_cmds[i].proc();
                break;
            }
        if (g_cmds[i].cmdText == NULL)
        {
            printf("command not found!..\n");
        }
    }

    return 0;
}

void parse_cmdline(void)
{
    char *str;

    g_nparams = 0;
    for (str = strtok(g_cmdLine, " \t"); str != NULL; str = strtok(NULL, " \t"))
        g_params[g_nparams++] = str;
}

void proc_dir(void)
{
    printf("dir command...\n");
}

void proc_del(void)
{
    if (g_nparams != 2) {
        printf("the syntax of the command is incorrect.\n");
        return;
    }

    printf("file deleted...\n");
}

```

4.2. Türden Bağımsız Her Diziyi Sıraya Dizebilen Bir Fonksiyon Nasıl Yazılabilir?

Normal olarak bir sort fonksiyonu yalnızca belli bir türdeki dizileri sıraya dizebilir. Örneğin:

```
void sort(int *pi, size_t size);
```

Buradaki sort fonksiyonu yalnızca int türden bir diziyi sıraya dizebilir. Eğer biz long türden bir diziyi sıraya dizmek istiyorsak başka bir fonksiyon yazmalıyız:

```
void sort_long(long *pi, size_t size);
```

Böyle her tür için içi aynı olan fakat türleri farklı olan fonksiyonları defalarca yazmak gerekebilir. Bu zahmeti ortadan kaldırmak için C++'ta template, Java ve C#'ta generic fonksiyonlar bulunmaktadır. Fakat template ya da generic fonksiyonlar özü değiştirmemektedir. Yalnızca zahmeti ortadan kaldırmaktadır. Pekiyi her türden diziyi sıraya dizebilen tek bir sort fonksiyonu yazılamaz mı? Böyle bir fonksiyon için ilk akla gelen yöntem fonksiyonun parametresini void bir gösterici yapmak ve fonksiyona sıraya dizilecek türü belirten ayrı bir parametre eklemektir. Örneğin:

```
void sort(void *ptr, size_t size, int type);
```

Buradaki type parametresi her tür için belli bir değer kabul edebilir. Örneğin:

```
enum SORT_TYPES {  
    INT, UINT, LONG, ULONG, SHORT, USHORT, SCHAR, UCHAR, FLOAT, DOUBLE, LDOUBLE  
};
```

Bu tasarımda yine bir sorun vardır: Kendi oluşturduğumuz bir yapıyı böyle bir fonksiyonla sıraya dizemeyiz.

İşte fonksiyon göstericilerini kullanarak türden bağımsız sıraya dizme işlemini yapan genel bir sort fonksiyonu yazılabilir. Bu fonksiyonda dizinin başlangıç adresi bir void gösterici ile alınır. Dizinin bir elemanının uzunluğu ve dizi uzunluğu fonksiyona ayrı parametrelerle geçirilir. Algoritma ne olursa olsun her sort fonksiyonunda dizinin iki elemanının karşılaştırılıp yer değiştirmesi gibi bir işlem yapılmaktadır. Gerçekten de dizinin iki elemanın yer değiştirilmesi aslında dizi elemanlarının uzunluğunu bilerek yapılabilmektedir. Ancak karşılaştırma işleminin fonksiyonu çağırın programcı tarafından yapılması gerekir. Böylece böyle bir sort fonksiyonunda bir fonksiyon gösterici parametresi bulundurulur. Bu parametre için fonksiyonu çağırın programcı kendi yazdığı bir karşılaştırma fonksiyonunu argüman olarak geçer. Fonksiyon da her karşılaştırmada bu fonksiyonu çağırır. Böylece fonksiyon her türden diziyi sıraya dizebilir. İşlemi bu biçimde yapan sort fonksiyonunun prototipi aşağıdaki gibi olacaktır:

```
void sort(void *base, size_t count, size_t width, int (*cmp)(const void *, const void *))
```

Burada fonksiyonun birinci parametresi dizinin başlangıç adresini, ikinci parametresi eleman sayısını, üçüncü parametresi ise bir elemanın byte cinsinden uzunluğunu belirtmektedir. Son parametre karşılaştırma fonksiyonunun adresini alır. Karşılaştırma fonksiyonu sıralama sırasında sort fonksiyonu tarafından dizinin iki elemanının adresi argüman yapılarak çağrılacaktır. Karşılaştırma fonksiyonunun geri dönüş değerinin int türden olduğuna dikkat ediniz. Bu fonksiyon eğer birinci parametresiyle belirtilen dizi elemanı ikinci parametresiyle belirtilen dizi elemanından büyükse pozitif herhangi bir değere, küçükse negatif herhangi bir değere, eşitse sıfır değerine geri dönecek biçimde yazılmalıdır. Şimdi bu fonksiyonu biz kabarcık sıralaması yöntemini kullanarak yazalım ve çağıralım:

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

```
void bsort(void *base, size_t count, size_t width, int(*cmp)(const void *, const void *))  
{  
    size_t i, k, m;  
    char *pcbase = (char *)base;  
    char *pc1, *pc2;  
    char temp;  
  
    for (i = 0; i < count - 1; ++i)
```

```

    for (k = 0; k < count - 1 - i; ++k) {
        pc1 = pbase + k * width;
        pc2 = pbase + (k + 1) * width;
        if (cmp(pc1, pc2) > 0) {
            for (m = 0; m < width; ++m) {
                temp = pc1[m];
                pc1[m] = pc2[m];
                pc2[m] = temp;
            }
        }
    }
}

int comparer1(const void *pv1, const void *pv2)
{
    const int *pi1 = (const int *)pv1;
    const int *pi2 = (const int *)pv2;

    if (*pi1 > *pi2)
        return 1;

    if (*pi1 < *pi2)
        return -1;

    return 0;
}

typedef struct tagPERSON {
    char name[32];
    int no;
} PERSON;

int comparer2(const void *pv1, const void *pv2)
{
    const PERSON *p1 = (const PERSON *)pv1;
    const PERSON *p2 = (const PERSON *)pv2;

    return strcmp(p1->name, p2->name);
}

int comparer3(const void *pv1, const void *pv2)
{
    const PERSON *p1 = (const PERSON *)pv1;
    const PERSON *p2 = (const PERSON *)pv2;

    return p1->no - p2->no;
}

int main(void)
{
    {
        int a[10] = { 34, 23, 45, 11, 78, 43, 34, 87, 33, 21 };
        int i;

        bsort(a, 10, sizeof(int), comparer1);

        for (i = 0; i < 10; ++i)
            printf("%d ", a[i]);
        printf("\n");
        printf("-----\n");
    }

    {
        int i;
        PERSON persons[] = {
            { "Ali Serce", 123 }, { "Kaan Aslan", 456 }, { "Necati Ergin", 321 },

```

```

        { "John Lennon", 54 }, { "Abidin Yarata", 115 }
    };

    sort(persons, 5, sizeof(PERSON), comparer2);
    for (i = 0; i < 5; ++i)
        printf("%s, %d\n", persons[i].name, persons[i].no);
    printf("-----\n");
    bsort(persons, 5, sizeof(PERSON), comparer3);
    for (i = 0; i < 5; ++i)
        printf("%s, %d\n", persons[i].name, persons[i].no);

    return 0;
}
}

```

Aslında C'nin standart kütüphanesinde yukarıdaki gibi sıraya dizme işlemi yapan qsort isimli bir fonksiyon zaten vardır. qsort fonksiyonu da yukarıda yazmış olduğumuz sort fonksiyonuyla aynı parametrik yapılara sahiptir ve genel kullanımı da aynı biçimdedir:

```
void qsort(void *base, size_t count, size_t width, int (*cmp)(const void *, const void *));
```

Her ne kadar standartlarda açık biçimde belirtilmemiş olsa da qsort fonksiyonu tipik olarak $n \log(n)$ karmaşıklığa sahip "quick sort" denilen algoritmayı kullanmaktadır. Örneğin:

```

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

int comparer1(const void *pv1, const void *pv2)
{
    const int *pi1 = (const int *)pv1;
    const int *pi2 = (const int *)pv2;

    if (*pi1 > *pi2)
        return 1;
    if (*pi1 < *pi2)
        return -1;

    return 0;
}

typedef struct tagPERSON {
    char name[32];
    int no;
} PERSON;

int comparer2(const void *pv1, const void *pv2)
{
    const PERSON *p1 = (const PERSON *)pv1;
    const PERSON *p2 = (const PERSON *)pv2;

    return strcmp(p1->name, p2->name);
}

int comparer3(const void *pv1, const void *pv2)
{
    const PERSON *p1 = (const PERSON *)pv1;
    const PERSON *p2 = (const PERSON *)pv2;

    return p1->no - p2->no;
}

int main(void)
{
    {
        int a[10] = { 34, 23, 45, 11, 78, 43, 34, 87, 33, 21 };
    }
}

```

```

int i;

qsort(a, 10, sizeof(int), comparer1);

for (i = 0; i < 10; ++i)
    printf("%d ", a[i]);
printf("\n");
printf("-----\n");
}

{
int i;
PERSON persons[] = {
    { "Ali Serce", 123 }, { "Kaan Aslan", 456 }, { "Necati Ergin", 321 },
    { "John Lennon", 54 }, { "Abidin Yarata", 115 }
};

qsort(persons, 5, sizeof(PERSON), comparer2);
for (i = 0; i < 5; ++i)
    printf("%s, %d\n", persons[i].name, persons[i].no);
printf("-----\n");
qsort(persons, 5, sizeof(PERSON), comparer3);
for (i = 0; i < 5; ++i)
    printf("%s, %d\n", persons[i].name, persons[i].no);

return 0;
}
}

```

Şimdi de belli bir dizindeki dosyaları uzunluklarına göre sıraya dizelim:

```

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

#define BLOCK_SIZE      10

void ExitSys(LPCSTR lpszMsg);
void Disp(const WIN32_FIND_DATA *files, size_t size);
int CompareBySize(const void *pv1, const void *pv2);
int CompareByName(const void *pv1, const void *pv2);

int main(void)
{
    HANDLE hFileFind;
    WIN32_FIND_DATA finfo;
    WIN32_FIND_DATA *files;
    int count;

    if ((hFileFind = FindFirstFile("c:\\windows\\*.\"", &finfo)) == INVALID_HANDLE_VALUE)
        ExitSys("FindFirstFile");

    count = 0;
    files = NULL;
    do {
        if (!(finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)) {
            if (count % BLOCK_SIZE == 0) {
                if ((files = (WIN32_FIND_DATA *)realloc(files, (count + BLOCK_SIZE) *
sizeof(WIN32_FIND_DATA))) == NULL) {
                    fprintf(stderr, "cannot allocate memory!..\n");
                    exit(EXIT_FAILURE);
                }
            }
            files[count] = finfo;
            ++count;
        }
    } while (FindNextFile(hFileFind, &finfo));
}

```

```

qsort(files, count, sizeof(WIN32_FIND_DATA), CompareBySize);
Disp(files, count);
printf("-----\n");

qsort(files, count, sizeof(WIN32_FIND_DATA), CompareByName);
Disp(files, count);

free(files);
CloseHandle(hFileFind);

return 0;
}

int CompareBySize(const void *pv1, const void *pv2)
{
    const WIN32_FIND_DATA *f1 = (const WIN32_FIND_DATA *)pv1;
    const WIN32_FIND_DATA *f2 = (const WIN32_FIND_DATA *)pv2;

    if (f1->nFileSizeLow > f2->nFileSizeLow)
        return 1;

    if (f1->nFileSizeLow < f2->nFileSizeLow)
        return -1;

    return 0;
}

int CompareByName(const void *pv1, const void *pv2)
{
    const WIN32_FIND_DATA *f1 = (const WIN32_FIND_DATA *)pv1;
    const WIN32_FIND_DATA *f2 = (const WIN32_FIND_DATA *)pv2;

    return strcmp(f1->cFileName, f2->cFileName);
}

void Disp(const WIN32_FIND_DATA *files, size_t size)
{
    size_t i;

    for (i = 0; i < size; ++i)
        printf("%-40s%lu\n", files[i].cFileName, (unsigned long int) files[i].nFileSizeLow);
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Pekiye diziye küçükten büyüğe değil de büyükten küçüğe sıraya dizmesek istesek ne yapmamız gerekir? Aslında tek yapacağımız şey karşılaştırma fonksiyonunu ters yazmaktır. Yani karşılaştırma fonksiyonu birinci parametre ikinci parametreden büyükse negatif herhangi bir değere, ikinci parametre birinci parametreden büyükse pozitif herhangi bir değere ve iki parametre birbirine eşitse sıfır değerine geri dönecek biçimde yazılmalıdır.

Şimdi de içerisinde isimlerin bulunduğu char türden bir gösterici dizisini büyükten küçüğe büyük harf küçük harf duyarlılığı olmadan (case insensitive) sıraya dizmeye çalışalım.

```

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

```

```

#include <string.h>
#include <Windows.h>

int compare(const void *p1, const void *p2);

int main(void)
{
    char *names[5] = { "Veli", "ali", "Secaettin", "Süleyman", "Aslı" };
    int i;

    qsort(names, 5, sizeof(char *), compare);
    for (i = 0; i < 5; ++i)
        printf("%s\n", names[i]);

    return 0;
}

int compare(const void *p1, const void *p2)
{
    const char **ppc1 = (const char **)p1;
    const char **ppc2 = (const char **)p2;

    return -stricmp(*ppc1, *ppc2);

    return 0;
}

```

Bu gösterici dizisinin her elemanı char * türünden olduğuna göre ve karşılaştırma fonksiyonu elemanların adresleriyle çağrılacağına göre karşılaştırma fonksiyonun parametrelerinin char ** türüne dönüştürülmesi gerekir. Büyük harf küçük harf duyarlılığı olmadan karşılaştırma yapan stricmp fonksiyonu standart bir C fonksiyonu değildir. Ancak hem Microsoft derleyicilerinde hem gcc ve clang derleyicilerinde ve diğer pek çok derleyicide bulunan bir eklenti (extention) fonksiyondur.

Aşağıda UNIX/Linux sistemlerinde belli bir dizinin içeriğini ls -l formatında isme göre sıraya sıraya dizilmiş biçimde ekrana yazdıran örnek program görüyorsunuz:

```

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <dirent.h>

#define MAX_PATH        4096
#define BLOCK_SIZE      32

struct linfo {
    struct stat finfo;
    char *name;
};

void exit_sys(const char *msg);
const char *get_ls(const struct linfo *linfo, int hlink_digit, int uname_digit, int gname_digit, int size_digit);
int cmp_name(const void *linfo1, const void *linfo2);

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

```

```

DIR *dir;
struct dirent *dent;
struct linfo *linfo;
int count;
char path[MAX_PATH];
struct passwd *pass;
struct group *gr;
int len;
int hlink_digit, uname_digit, gname_digit, size_digit;
int i;

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

if ((dir = opendir(argv[1])) == NULL)
    exit_sys("open");

linfo = NULL;
for (count = 0, errno = 0; (dent = readdir(dir)) != NULL; ++count) {
    sprintf(path, "%s/%s", argv[1], dent->d_name);
    if (count % BLOCK_SIZE == 0)
        if ((linfo = realloc(linfo, (count + BLOCK_SIZE) * sizeof(struct linfo))) == NULL) {
            fprintf(stderr, "cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
    if ((linfo[count].name = (char *)malloc(strlen(dent->d_name) + 1)) == NULL) {
        fprintf(stderr, "cannot allocate memory!...\n");
        exit(EXIT_FAILURE);
    }
    strcpy(linfo[count].name, dent->d_name);
    if (stat(path, &linfo[count].finfo) == -1)
        exit_sys("stat");
}

if (errno != 0)
    exit_sys("readdir");

closedir(dir);

hlink_digit = uname_digit = gname_digit = size_digit = 0;

for (i = 0; i < count; ++i) {
    len = (int)log10(linfo[i].finfo.st_nlink) + 1;
    if (len > hlink_digit)
        hlink_digit = len;

    if ((pass = getpwuid(linfo[i].finfo.st_uid)) == NULL)
        exit_sys("getppuid");

    len = (int)strlen(pass->pw_name);
    if (len > uname_digit)
        uname_digit = len;

    if ((gr = getgrgid(linfo[i].finfo.st_gid)) == NULL)
        exit_sys("getgrgid");

    len = (int)strlen(gr->gr_name);
    if (len > gname_digit)
        gname_digit = len;

    len = (int)log10(linfo[i].finfo.st_size) + 1;
    if (len > size_digit)
        size_digit = len;
}

```



```

qsort(lsinfo, count, sizeof(struct lsinfo), cmp_name);

for (i = 0; i < count; ++i)
    printf("%s\n", get_ls(&lsinfo[i], hlink_digit, uname_digit, gname_digit, size_digit));

for (i = 0; i < count; ++i)
    free(lsinfo[i].name);

free(lsinfo);

return 0;
}

const char *get_ls(const struct lsinfo *lsinfo, int hlink_digit, int uname_digit, int gname_digit, int
size_digit)
{
    static char buf[4096];
    static mode_t modes[] = { S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH,
S_IXOTH };
    struct passwd *pass;
    struct group *gr;
    int index = 0;
    int i;

    if (S_ISREG(lsinfo->finfo.st_mode))
        buf[index] = '-';
    else if (S_ISDIR(lsinfo->finfo.st_mode))
        buf[index] = 'd';
    else if (S_ISCHR(lsinfo->finfo.st_mode))
        buf[index] = 'c';
    else if (S_ISBLK(lsinfo->finfo.st_mode))
        buf[index] = 'b';
    else if (S_ISFIFO(lsinfo->finfo.st_mode))
        buf[index] = 'p';
    else if (S_ISLNK(lsinfo->finfo.st_mode))
        buf[index] = 'l';
    else if (S_ISSOCK(lsinfo->finfo.st_mode))
        buf[index] = 's';
    ++index;

    for (i = 0; i < 9; ++i)
        buf[index++] = (lsinfo->finfo.st_mode & modes[i]) ? "rwx"[i % 3] : '-';
    buf[index] = '\0';

    index += sprintf(buf + index, "%*llu", hlink_digit, (unsigned long long)lsinfo->finfo.st_nlink);

    if ((pass = getpwuid(lsinfo->finfo.st_uid)) == NULL)
        return NULL;
    index += sprintf(buf + index, "%-*s", uname_digit, pass->pw_name);

    if ((gr = getgrgid(lsinfo->finfo.st_gid)) == NULL)
        return NULL;
    index += sprintf(buf + index, "%-*s", gname_digit, gr->gr_name);

    index += sprintf(buf + index, "%*lld", size_digit, (long long)lsinfo->finfo.st_size);
    index += strftime(buf + index, 100, "%b %e %H:%M", localtime(&lsinfo->finfo.st_mtime));

    sprintf(buf + index, "%s", lsinfo->name);

    return buf;
}

int istrncmp(const char *s1, const char *s2)
{
    while (tolower(*s1) == tolower(*s2)) {

```

```

        if (*s1 == '\0')
            return 0;
        ++s1;
        ++s2;
    }

    return tolower(*s1) - tolower(*s2);
}

int cmp_name(const void *pv1, const void *pv2)
{
    const struct linfo *linfo1 = (const struct linfo *)pv1;
    const struct linfo *linfo2 = (const struct linfo *)pv2;

    return strcmp(linfo1->name, linfo2->name);
}

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

    exit(EXIT_FAILURE);
}

```

4.3. Göstericiyi Gösteren Göstericiler (Pointers to Pointers)

Bir gösterici başka bir göstericinin adresini tutabilir. Böyle göstericilere "göstericiyi gösteren göstericiler (pointers to pointers)" denilmektedir. Gösterici gösteren göstericiler deklarasyonda iki tane * ile belirtilirler. Örneğin:

```
int **ppi;
```

Burada ppi bir göstericiyi gösteren göstericidir. Türü int ** biçiminde temsil edilir (ppi'yi kapatıp sola bakınız). Biz bu göstericiye * operatörü uygularsak int * türünden bir nesne elde ederiz (*ppi'yi kapatıp sola bakınız). Nihayet biz bu göstericiye iki tane * operatörü uygularsak (**ppi'yi kapatıp sola bakınız) int bir nesne elde ederiz.



Örneğin:

```

#include <stdio.h>

int main(void)
{
    int a;
    int *pi;
    int **ppi;

    a = 10;
    pi = &a;
    ppi = &pi;

    **ppi = 20;
    printf("%d\n", a);

    return 0;
}

```

Bir dizinin ismi bir ifadede kullanıldığında bu isim o dizinin ilk elemanın adresini (yani dizinin adresini) belirtiyordu. Yani T türünden bir dizi işleme sokulduğunda otomatik olarak bu dizi T * türüne dönüştürülmektedir. Bu durumda bir gösterici dizisinin ismi işleme sokulduğunda gösterici adresine dönüştürülmektedir. Baljka bir deyişle biz bir gösterici dizisinin adresini aynı türden göstericiyi gösteren göstericiye atayabiliriz. Bir göstericiyi gösteren gösteri 1 artırıldığında ya da 1 eksiltildiğinde onun içerisindeki adres göstericinin uzunluğu kadar artırılıp eksiltilecektir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    char *names[] = { "ali", "veli", "selami" };
    char **ppc;

    ppc = names;
    puts(*ppc);
    ++ppc;
    puts(*ppc);
    ++ppc;
    puts(*ppc);

    return 0;
}
```

Aslında göstericiyi gösteren göstericiyi gösteren göstericiler ve daha yüksek düzeyli göstericiler de bulunabilir. Yani örneğin aşağıdaki bildirim geçerlidir:

```
int *****pppppi;
```

Ancak pratikte üç yıldızlı göstericilere bile neredeyse hiç gereksinim duyulmamaktadır.

Fonksiyon parametresi olarak göstericilerin sanki birer diziymiş gibi gösterilmeleri geçerlidir. Örneğin aşağıdaki bildirimlerin hiçbir farkı yoktur:

```
void foo(int *pi);
void foo(int pi[]);
void foo(int pi[2]);
```

Tabii bu durum yalnızca fonksiyon parametresi söz konusu olduğunda eşdeğerdir. Köşeli parantezler içerisinde yazılan uzunluğun da hiçbir önemi yoktur. Ancak bu uzunluk yazılacaksa sabit ifadesi olması zorunludur. O halde aşağıdakibildirim eşdeğeri nedir?

```
void foo(int *a[]);
```

Bunun eşdeğeri şöyledir:

```
void foo(int **a);
```

Örneğin main fonksiyonunun argv parametresi geleneksel olarak aşağıdaki gibi bildirilmektedir:

```
int main(int argc, char *argv[])
{
    /* ... */
}
```

Biz bildirimini şöyle de yapabiliriz:

```
int main(int argc, char **argv)
{
    /* ... */
}
```

C'de void bir göstericiye her türden adres atanabilir. Bir göstericinin adresi de atanabilir. Yani örneğin biz int ** türünden bir adresi void * türüne atayabiliriz. Göstericinin adresini tutabilecek genel bir gösterici türü void * göstericileridir, void ** göstericileri değildir. Örneğin:

```
int *a[10];
void *pv;
void **ppv;

pv = a;    /* geçerli */
ppv = a;   /* geçersiz */
```

Bu durumda biz türü void ** olan bir göstericiye ancak void * türünden bir göstericinin adresini atayabiliriz. Örneğin:

```
void *pv;
void **ppv;
void *pv2;

ppv = &pv;    /* geçerli */
pv2 = &pv;    /* geçerli */
```

Ancak fonksiyon göstericileri konusunda da belirtildiği gibi bir fonksiyon adresi void türünden bir göstericiye tür dönüştürmesi yapılsa bile atanamaz. Yani veri göstericilerinden (pointers to data) fonksiyon göstericilerine (pointers to function) otomatik ya da açıkça tür dönüştürmesi yasaklanmıştır.

Bir fonksiyon bir göstericinin içerisine bir adres atayacaksa o göstericinin adresini almalıdır. Bu durumda fonksiyonun parametresi göstericiyi gösteren gösterici olur. Örneğin:

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

void allocstr(char **str, size_t n)
{
    if ((*str = (char *)malloc(n)) == NULL) {
        fprintf(stderr, "fatal error: Not enough memory!..\n");
        exit(EXIT_FAILURE);
    }
}

int main(void)
{
    char *name;

    allocstr(&name, 32);
    gets(name);
    puts(name);

    return 0;
}
```

Göstericiyi gösteren göstericilerde const'luk ve volatile'lık durumu biraz karışık olabilmektedir. Burada nerenin const ya da volatile olacağı bildirimde const ya da volatile anahtar sözcüklerinin nereye getirildiğine bağlıdır. Örneğin:

```
const int **ppi;
int * const * ppi;
int ** const ppi = ilkdeğer;

volatile int **ppi;
int * volatile *ppi;
int ** volatile ppi;
```

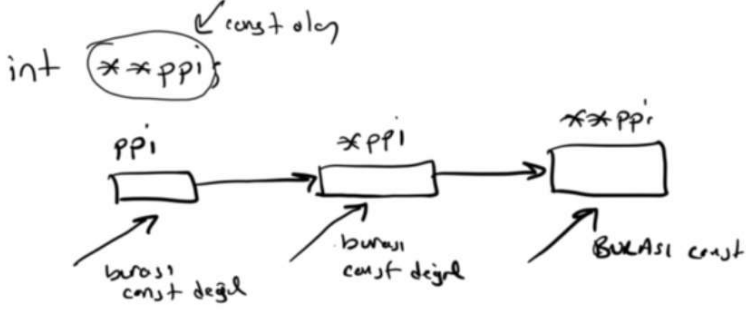
Tabii bu anahtar sözcükler birden fazla yere getirilebilirler. Örneğin:

```
const int * const * const ppi = ilkdeğer;
```

Aşağıdaki bildirimde dikkat ediniz:

```
const int **ppi;
```

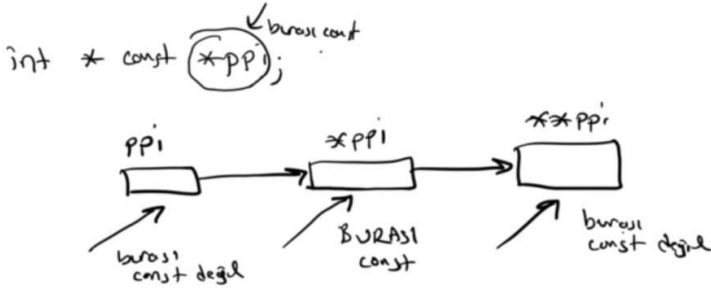
Burada const olan göstericiyi gösteren göstericinin gösterdiği göstericinin gösterdiği nesnedir. Gösterici gösteren göstericinin ya da göstericiyi gösteren göstericinin gösterdiği gösterici const değildir. Bunu şekilsel olarak şöyle ifade edebiliriz:



Şimdi aşağıdaki bildirimde inceleyiniz:

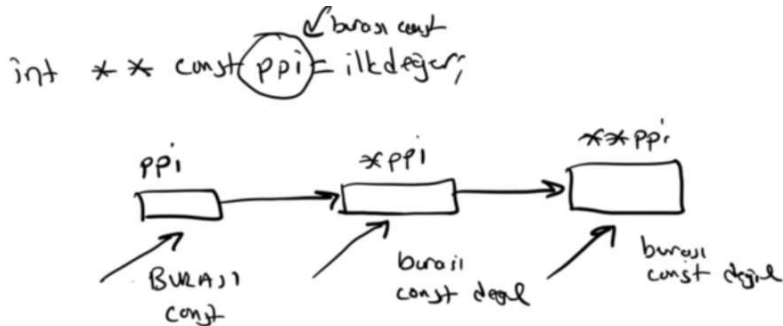
```
int * const *ppi;
```

Bu durumu da şekilsel olarak şöyle gösterebiliriz:



Şimdi de aşağıdaki bildirimde inceleyiniz:

```
int ** const ppi = ilkdeğer;
```



C++'ta göstericiyi gösteren göstericilerin const'luk volatile'lik durumunda anlaşılması zor bir tür uyumsuzluğu vardır. const (ya da volatile) olmayan bir gösterici gösteren adres const (ya da volatile) belirleyicisi en başta olan bir göstericiyi gösteren göstericiye atanamaz. Bu atamada tür uyumsuzluğu vardır. Başka bir deyişle T bir tür belirtmek üzere biz T * türünü const T * türüne atayabiliriz. Ancak T ** türünü const T ** türüne atayamayız. Örneğin:

```

int *pi;
const int **ppi;

ppi = &pi;      /* geçersiz! const T ** = T ** atanamaz */

```

Peki neden T ** türünden const T ** türüne otomatik dönüştürme yok? Halbuki T * türünden const T * türüne otomatik dönüştürme var.

İşte bu durumun neden geçersiz olduğu basit bir örnekle açıklanabilir1:

1) const int x = 10;

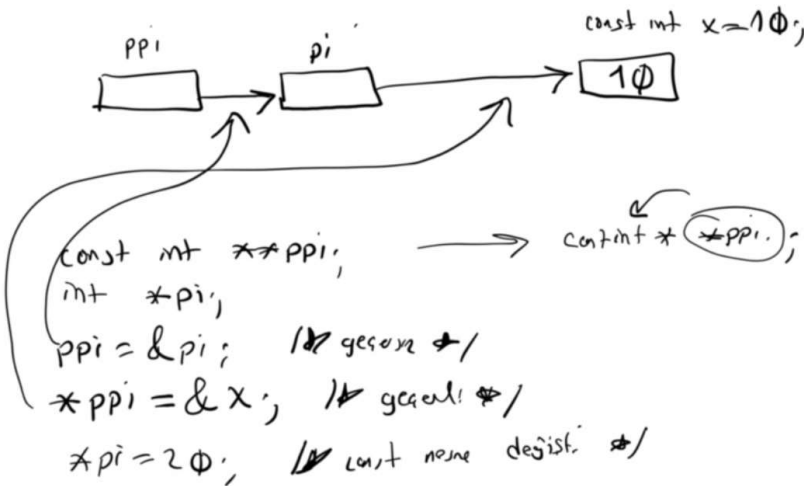
Burada x const bir nesnedir. Bunun adresi gösterdiği yer const olmayan bir göstericiye atanamaz. fakat gösterdiği yer const olan yani const int * türünden bir göstericiye atanabilir.

2) const int **ppi;
int *pi;

ppi = π /* geçersiz fakat geçersiz olmadığını varsayalım */

3) *ppi = &x; /* geçerli */

Burada aslında biz x'in adresini çaktırmadan pi'ye atamış olduk. Bu durumda artık pi vasıtasıyla biz x'i haksız yere değiştirebiliriz.



Peki biz T ** türünü const T**'a atayamıyorsak nasıl bire atayabiliriz. İşte T ** türü ancak T * const * türüne atanabilmektedir. Yani:

```

int *pi;
const int **ppi;
int * const *ppi2;

ppi = &pi;      /* geçersiz */
ppi2 = &pi;     /* geçerli */

```

Aynı sorun volatile niteleyicisi için de söz konusudur. Bunu genelleştirirsek T n tane * türünden const T n tane * türüne otomatik dönüştürme yoktur. Ancak T n tane * türünden T * const n - 1 tane * türüne otomatik dönüştürme vardır.

Aslında C'de T ** türünden const T ** türüne atama işlemi geçerli kabul edilmektedir. Ancak C derleyicilerinin hemen hepsi bu böyle bir atama işleminde uyarı mesajı vermektedir. Programcının C'de çalışılıyor olsa da bu biçimdeki atamalardan kaçınması gerekir.

4.4. Çok Boyutlu Diziler ve Dizi Göstericileri

C'de bir dizinin ismi aslında dizinin tamamını temsil eden bir nesne belirtmektedir. Örneğin:

```
int a[8];
```

Burada a bu dizinin tamamını temsil etmektedir ve int[8] türündendir. Ancak C standartlarına göre biz bir dizi ismini bir ifade kullanıldığımızda o dizi ismi otomatik olarak o dizinin ilk elemanın adresine dönüştürülür. Yani bir dizinin ismi aslında o dizinin tamamını belirtmekle birlikte bir ifade kullanıldığında onun başlangıç adresini belirtiyor durumda olur. Örneğin yukarıdaki bildirimde a ifadesi int[8] türündendir. Ancak biz onu bir ifade kullanıldığımızda o artık int * türü olarak işleme girer. Bildiğiniz gibi dizi isimleri bir ifade kullanıldığında nesne belirtmemektedir.

C'de bir dizinin adresi alınabilir. Bu durumda bu adres bir dizi göstericisine atanabilir. Dizi göstericileri (pointers to array) gösterdikleri yerde bütün bir dizi olan göstericilerdir. Dizi gösterici bildirimlerinin genel biçimi şöyledir:

```
<tür> (*<gösterici ismi>)[<sabit ifadesi>];
```

Örneğin:

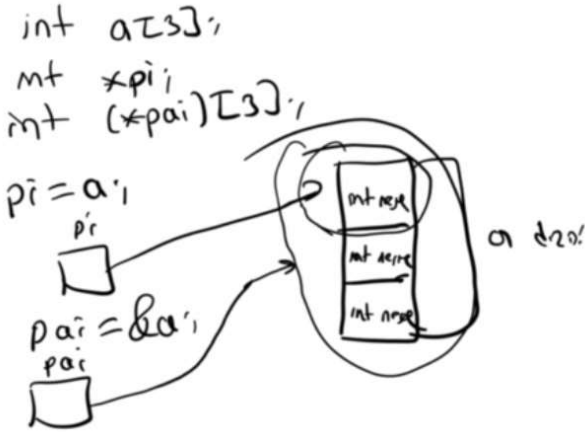
```
int (*pai)[3];
```

Burada pai göstericisi 3 elemanlı int bir diziyi göstermektedir. Bir dizi nesnesinin adresi aynı türden ve aynı uzunlukta bir dizi göstericisine atanır. Örneğin:

```
int a[3];
int (*pai)[3];
int *pi;

pi = a; /* geçerli */
pai = &a; /* geçerli */
```

Burada pi'nin gösterdiği yerde tek bir int nesne vardır. Halbuki pai'nin gösterdiği yerde 3 elemanlı bir int dizi vardır.



Dizi göstericilerini bildirirken parantezin neden gerektiğine dikkat ediniz:

```
int *api[3];
int (*pai)[3];
```

api her elemanı int * türünden olan bir gösterici dizisidir. Halbuki pai int[3] türünden diziyi gösteren bir dizi göstericidir.

Mademki bir dizi nesnesini bir ifade kullandığımızda artık o ifade dizinin adresini belirtiyor, o halde bir dizi göstericisini de * ya da [] operatörüyle kullandığımızda o ifade de o göstericinin gösterdiği yerdeki dizinin adresi anlamına gelmelidir. Biz bir dizi göstericisini * ya da [] operatörüyle kullandığımızda elde edilen ifade nesne belirtmemektedir. Örneğin:

```
int a[3];
int(*pai)[3];
```

```
pai = &a;
```

Burada artık *pai ile a tamamen aynı anlamdadır. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[3] = { 1, 2, 3 };
    int(*pai)[3];

    pai = &a;

    printf("%d\n", **pai);           /* 1 */
    printf("%d\n", (*pai)[1]);      /* 2 */
    printf("%d\n", (*pai)[2]);      /* 3 */

    return 0;
}
```

Burada (*pai)[n] ifadesine dikkat ediniz. Eğer *pai paranteze alınmasaydı [] operatörünün önceliği olduğu için ifade tamamen başka bir anlama gelirdi.

Dizi göstericilerindeki köşeli parantez içerisindeki uzunluk tür bilgisine dahildir. Yani biz bir dizinin adresini uzunluk bilgisi aynı olan bir dizi göstericisine atayabiliriz. Dizi göstericilerindeki uzunluk ifadeleri sabit ifadeleri biçiminde olmak zorundadır. Örneğin:

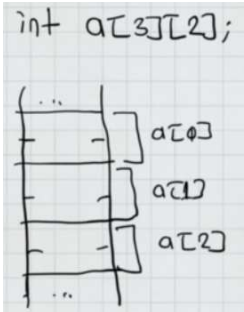
```
int a[3];
int(*pai)[2];

pai = a; /* geçersiz */
```

C'de aslında çok boyutlu dizileri "dizi dizileri" olarak düşünebiliriz. Örneğin bir matrisi aslında dizilerden oluşan bir dizi gibi değerlendirebiliriz. Örneğin:

```
int a[3][2];
```

Burada a matrisi her elemanı int[2] türünden (yani 2 elemanlı int bir dizi türünden) olan 3 elemanlı bir dizi dizisidir.



Çok boyutlu dizi bildirimindeki ilk köşeli parantez asıl diziyi diğerleri o dizinin elemanlarına ilişkin dizileri belirtmektedir. Örneğin:

elemen olan dizi
int a[3][2];
asıl dizi

a 3 elemanlı, her elemanı int[2]
türünden olan bir dizedir.

Peki C'de biz çok boyutlu bir dizinin adresini hangi türden bir göstericiye yerleştirebiliriz? Örneğin:

```
int a[3][2];
```

Burada a'nın adresini hangi türden bir göstericiye yerleştirebiliriz? İşte mademki çok boyutlu diziler bir dizi dizisidir, o halde yukarıdaki a matrisinin ismi de bu dizi dizisinin ilk elemanın adresi olur. Bu dizi dizisi her biri int[2] olan dizileri tutan 3 elemanlı bir dizidir. Bu dizi dizisinin ilk elemanın adresi aslında bir dizi adresidir. O halde matrisin ismi yani başlangıç adresi bir dizi göstericisine atanmalıdır. Örneğin:

```
int a[3][2] = { {1, 2}, {3, 4}, {5, 6} };  
int (*pai)[2];
```

```
pai = a;      /* geçerli */
```

Görüldüğü gibi bir matrisin adresi C'de bir göstericiye ya da göstericiyi gösteren göstericiye atanmamaktadır. Bir dizi göstericisine atanabilmektedir.

Çok boyutlu dizilerde her zaman ilk köşeli parantezin asıl diziyi sonraki köşeli parantezlerin eleman olan diziyi belirttiğine dikkat ediniz. Örneğin:

```
int a[2][3][4];
```

Burada aslında bir dizi dizisi dizisi söz konusudur. Yani a 2 elemanlı bir dizidir. a'nın her elemanı 3 elemanlı her elemanı 4 elemandan oluşan bir dizi dizisidir. Böyle bir çok boyutlu dizinin adresi (yani ismi) int (*)[3][4] türünden bir dizi göstericisine atanabilir.

```
int a[2][3][4];  
int (*pai)[3][4];
```

```
pai = a;      /* geçerli */
```

T bir tür N de tamsayı türlerinden birine ilişkin bir sabit ifadesi belirtmek üzere bir dizi göstericisinin türü sembolik olarak T (*)[N] biçiminde gösterilmektedir. Buna göre:

```
int a[2];
```

Dizi tanımlamasında &a ifadesi int (*)[2] türündendir.

Bir matrisin ismini bir tane [] operatörüne soktuğumuzda elde ettiğimiz ifade ilgili matrisin ilgili dizisidir. Diziler de ifade içerisinde kullanıldığında nesne belirtmezler. Örneğin:

```
int a[2][3];
```

Burada a[0] bu matrisin ilk dizisini a[1] ise sonraki dizisini belirtmektedir. a[0] ve a[1] nesne belirtmezler. a[i][k] ifadesi a matrisinin i'inci dizisinin k'ıncı elemanı anlamındadır.

Bir dizi göstericisini bir artırdığımızda gösterici içerisindeki adres o dizi göstericisinin gösterdiği dizinin uzunluğu kadar artar. Örneğin:

```
int a[3][2];  
int (*pa)[2];
```

```
pa = a;      /* Burada pa dizi dizisinin 0'inci indeksli dizisini göstermektedir */
++pa;      /* Artık burada pa dizi dizisinin 1'inci indeksli dizisini göstermektedir */
```

Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a[4][3] = {
        { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, {10, 11, 12}
    };
    int (*pa)[3];
    int val;
    int *pi;

    pa = a;

    val = **pa;
    printf("%d\n", val);      /* 1 */

    val = (*pa)[1];
    printf("%d\n", val);      /* 2 */

    val = pa[1][1];
    printf("%d\n", val);      /* eşdeğeri (*(pa + 1))[1] */
    /* 5 */

    pi = pa[1];
    val = pi[2];
    printf("%d\n", val);      /* 6 */

    pa += 2;
    val = (*pa)[1];
    printf("%d\n", val);      /* 8 */

    return 0;
}
```

Şimdi bir diziye adres operatörü ile adresini alarak fonksiyona geçirelim:

Örneğin:

```
#include <stdio.h>

void foo(int(*pa)[3])
{
    int i;

    for (i = 0; i < 3; ++i)
        printf("%d ", (*pa)[i]);
    printf("\n");
}

int main(void)
{
    int a[3] = { 1, 2, 3 };

    foo(&a);    /* geçerli */

    return 0;
}
```

Tabii genellikle biz bir diziye dizi nesnesinin adresi yoluyla değil onun ilk elemanının adresi yoluyla aktarırız.

Bir matrisi (çok boyutlu diziyi) adres yoluyla fonksiyona aktarmak istediğimizde fonksiyonun parametre değişkeni bir dizi göstericisi olmalıdır. Örneğin:

```
#include <stdio.h>

void foo(int(*pa)[2])
{
    int i, k;

    for (i = 0; i < 3; ++i) {
        for (k = 0; k < 2; ++k)
            printf("%d ", pa[i][k]);
        printf("\n");
    }
}

int main(void)
{
    int a[3][2] = { {1, 2}, {3, 4}, {5, 6} };

    foo(a);          /* geçerli */

    return 0;
}
```

Bir matrisin fonksiyona aktarıldığı durumda fonksiyonun parametre değişkeninin normal bir gösterici ya da göstericiyi gösteren gösteri olmadığına dikkat ediniz. Tabii C'nin kurallarına göre çok boyutlu diziler satır temelinde tek boyutlu olarak saklanmaktadır. Uygun bir dönüştürmesi ile bir matrisin adresi normal bir göstericiye de atanabilir. Bazı programcılar böyle çalışmanın daha kolay olduğu gerekçesiyle bunu tercih etmektedir. Örneğin:

```
#include <stdio.h>

void foo(int *pi, int rowSize, int colSize)
{
    int i, k;

    for (i = 0; i < rowSize; ++i) {
        for (k = 0; k < colSize; ++k)
            printf("%d ", pi[i * colSize + k]);
        printf("\n");
    }
}

int main(void)
{
    int a[3][2] = { {1, 2}, {3, 4}, {5, 6} };

    foo((int *)a, 3, 2);          /* geçerli */

    return 0;
}
```

Bu biçimdeki aktarımda matrisin sütun uzunluğunun tür bilgisi içerisinde yer almadığına dikkat ediniz. Yani biz bu biçimle herhangi bir satır ve sütun uzunluğuna sahip matrisi fonksiyona aktarabiliriz.

Fonksiyon parametrelerinde çok boyutlu dizi sentaksı kullanılırsa bu durum aslında dizi göstericisi anlamına gelmektedir. Örneğin:

```
void foo(int a[][5])          /* int (*a)[5] */
{
    /* ... */
}
```

```
int main(void)
{
    int a[5];

    foo(&a);

    return 0;
}
```

Tabii aşağıdaki gibi bir bildirim mümkün değildir:

```
void foo(int a[][])          /* geçersiz! */
{
    /* ... */
}
```

Şüphesiz iki fazla boyutlu dizi göstericisi söz konusu olabilir. Bu durumda gösterici dizinin ilk boyutu dışındaki tüm boyutlarını köşeli parantezler içerisinde almalıdır. Örneğin:

```
void foo(int (*pa)[3][9])
{
    /* ... */
}

int main(void)
{
    int a[5][3][9];

    foo(a);

    return 0;
}
```

Bir fonksiyon bir dizi nesnesinin ya da çok boyutlu bir dizinin adresi ile de geri dönebilir. Bu durumda yine parantezlerden faydalanılır. Örneğin:

```
#include <stdio.h>

int (*foo(void))[5]
{
    static int a[5] = {1, 2, 3, 4, 5};

    return &a;
}

int main(void)
{
    int(*pai)[5];
    int i;

    pai = foo();
    for (i = 0; i < 5; ++i)
        printf("%d\n", (*pai)[i]);

    return 0;
}
```

Burada foo fonksiyonunun geri dönüş değeri iki elemanlı bir dizi nesnesinin adresidir.

Dizi göstericilerine de typedef uygulayabiliriz. Örneğin:

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

```

typedef int (*PA)[5];

PA foo(void)
{
    static int a[5] = {1, 2, 3, 4, 5};

    return &a;
}

int main(void)
{
    PA pa;
    int i;

    pa = foo();
    for (i = 0; i < 5; ++i)
        printf("%d ", (*pa)[i]);
    printf("\n");

    return 0;
}

```

Burada PA int türünden 5 elemanlı dizi adresini temsil etmektedir.

5. Yol İfadeleri (Path Names) ve Proseslerin Çalışma Dizinleri (Current Working Directory)

Bir dosyanın yerini belirten yazısal ifadeye yol ifadesi (path name) denilmektedir. Yol ifadeleri mutlak (absolute) ve görelî (relative) olmak üzere ikiye ayrılırlar. Mutlak yol ifadeleri kök dizinden itibaren yer belirtirken, görelî yol ifadeleri prosesin çalışma dizininden itibaren yer belirtmektedir.

Her prosesin bir çalışma dizini (current working directory) vardır. Proseslerin çalışma dizinleri "Proses Kontrol Bloğun"da saklanmaktadır.

Bir yol ifadesinin ilk karakteri Windows'ta '\', UNIX/Linux sistemlerinde '/' ise böyle yol ifadeleri mutlaktır, değilse görelidir. Örneğin:

```

"a.dat"           /* Windows, görelî */
"\a.dat"         /* Windows, mutlak */
"a\b\c.dat"      /* Windows, görelî */
"\a\b\c.dat"     /* Windows, mutlak */

"a.dat"           /* UNIX/Linux, görelî */
"/a.dat"         /* UNIX/Linux, mutlak */
"a/b/c.dat"      /* UNIX/Linux, görelî */
"/a/b/c.dat"     /* UNIX/Linux, mutlak */

```

Windows sistemleri UNIX/Linux uyumunu korumak için dizin geçişlerinde '/' karakterini de kabul etmektedir.

Windows sistemlerinde ayrıca sürücü (drive) kavramı da vardır. Her sürücünün ayrı bir kökü bulunur. UNIX/Linux sistemlerinde ise sürücü kavramı yoktur. Dolayısıyla bu sistemlerde toplamda tek bir kök dizin vardır.

Peki Windows'ta mutlak yol ifadesi hangi sürücüye ilişkindir? Windows'ta sürücüyü de içeren yol ifadelerine tam yol ifadeleri (full path names) denilmektedir. Sürücü bir harf ve ':' karakterinden oluşmaktadır. Örneğin:

```

"c:\a\b\c.dat"   /* tam yol ifadesi */
"e:\a\b\c.dat"   /* tam yol ifadesi */

```

Windows'ta prosesin çalışma dizini Proses Kontrol Bloğunda tam yol ifadesi biçiminde tutulur. İşte eğer biz mutlak bir yol ifadesinde sürücü belirtmemişsek, işletim sistemi prosesin çalışma dizini hangi sürücüye ilişkinse o mutlak yol ifadesinin

o sürücüye ilişkin olduğunu düşünmektedir. Örneğin, prosesimizin çalışma dizini "e:\temp" olsun. Biz de "\a\b\c.dat" biçiminde bir mutlak yol ifadesi vermiş olalım. Buradaki kök e sürücüsünün köküdür.

Windows'ta ilginç bir durum daha vardır. Göreli bir yol ifadesi bir sürücü de içerebilir. Örneğin prosesimizin çalışma dizini "d:\temp" olsun:

```
"c:a\b\c.dat"  
"e:c.dat"
```

Buradaki göreli yol ifadeleri nereden itibaren yer belirtmektedir? İşte Windows burada bazı çevre değişkenlerine (environment variables) başvurmaktadır. Bu çevre değişkenleri yoksa Windows yine bu sürücülerin kök dizinlerinden itibaren yolu belirler. Yani söz konusu bu çevre değişkenleri tanımlanmamışsa (pek çok sistemde tanımlanmamıştır) yukarıdaki yol ifadeleri aşağıdakilerle eşdeğer olur:

```
"c:\a\b\c.dat"  
"e:\c.dat"
```

Proseslerin çevre değişkenleri ileride ele alınacaktır.

Biz kendi prosesimizin çalışma dizinini elde edebiliriz ve istersek onu değiştirebiliriz. Proseslerin çalışma dizinleri Windows'ta GetCurrentDirectory isimli API fonksiyonuyla elde edilmektedir:

```
DWORD GetCurrentDirectory(  
    DWORD nBufferLength,  
    LPTSTR lpBuffer  
);
```

Fonksiyonun birinci parametresi çalışma dizininin yerleştirileceği dizinin uzunluğunu, ikinci parametresi de bu dizinin adresini alır. Fonksiyon prosesin çalışma dizinini bu diziye yerleştirir, sonuna da null karakteri ekler. Fonksiyon normal olarak diziye yerleştirdiği karakter sayısı ile (null karakter dahil değil) geri dönmektedir. Eğer birinci parametre için girilen uzunluk yetersizse fonksiyon diziye hiç yerleştirme yapmaz fakat yol ifadesinin yerleştirileceği dizi için gereken uzunluğu (null karakterle birlikte) bize geri dönüş değeri olarak verir. (Yani biz istersek birinci parametreyi 0 geçerek yol ifadesi için gereken uzunluğu elde edebiliriz. Tabii bu durumda ikinci parametre için de NULL adres girebiliriz.) Örneğin:

```
#include <stdio.h>  
#include <Windows.h>  
  
int main(void)  
{  
    char cwd[1024];  
  
    GetCurrentDirectory(1024, cwd);  
    printf("%s\n", cwd);  
  
    return 0;  
}
```

UNIX/Linux sistemlerinde prosesin çalışma dizini getcwd isimli POSIX fonksiyonuyla elde edilir:

```
#include <unistd.h>  
  
char *getcwd(char *buf, size_t size);
```

Fonksiyonun birinci parametresi yol ifadesinin yerleştirileceği dizinin adresini, ikinci parametresi bunun uzunluğunu alır. Fonksiyon birinci parametreyle verilen adresin aynısına geri döner. Eğer ikinci parametreyle belirtilen uzunluk null karakter dahil olmak üzere yol ifadesinin uzunluğundan küçükse fonksiyon başarısız olur ve NULL adresine geri döner. Örneğin:

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

```

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

int main(void)
{
    char cwd[1024];

    if (getcwd(cwd, 1024) == NULL) {
        perror("getcwd");
        exit(EXIT_FAILURE);
    }
    puts(cwd);

    return 0;
}

```

Windows'ta Microsoft'un C derleyicilerinde ayrıca prototipi <direct.h> dosyası içerisinde olan `_getcwd` isimli POSIX fonksiyonuyla aynı biçimde çalışan eklenti (extension) bir kütüphane fonksiyonu da vardır. (Tabii aslında bu fonksiyon da kendi içerisinde `GetCurrentDirectory` API fonksiyonunu çağırılmaktadır.)

Prosesin çalışma dizini Windows'ta `SetCurrentDirectory` API fonksiyonuyla değiştirilebilir:

```

BOOL SetCurrentDirectory(LPCTSTR lpPathName);

```

Fonksiyon yeni çalışma dizininin bulunduğu dizinin adresini parametre olarak alır. Geri dönüş değeri işlemin başarısını belirtmektedir. Örneğin:

```

#include <stdio.h>
#include <Windows.h>

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    char cwd[1024];

    if (!SetCurrentDirectory("c:\\windows"))
        ExitSys("SetCurrentDirectory");

    GetCurrentDirectory(1024, cwd);
    printf("%s\n", cwd);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

UNIX/Linux sistemlerinde ise prosesin çalışma dizini `chdir` isimli POSIX fonksiyonuyla değiştirilir:

```

#include <unistd.h>

int chdir(const char *path);

```

Fonksiyon parametre olarak belirlenecek yeni çalışma dizinin yol ifadesini alır. Başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner. Örneğin:

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

void exit_sys(const char *msg);

int main(void)
{
    char cwd[1024];

    if (chdir("/usr/include") == -1)
        exit_sys("chidr");

    if (getcwd(cwd, 1024) == NULL)
        exit_sys("getcwd");

    puts(cwd);

    return 0;
}

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

Komut satırı programları (bash gibi, cmd.exe gibi) aslında prosesin çalışma dizinini elde edip onu prompt'a yazdırmaktadır. Dolayısıyla bunlar da birer procestir ve aslında bir dizine oturmak gibi bir kavram yoktur. Kullanıcı komut satırında girdiği komutta bir yol ifadesi kullandığında bu programlar dosya fonksiyonlarını kullanıcının yazdığı yol ifadesiyle çağırırlar. Kullanıcının girdiği yol ifadeleri mutlak ya da görelili olabilir. Eğer bu yol ifadeleri mutlaksa kök dizinden itibaren, göreliyse prompt'ta belirtilen çalışma dizininden itibaren yer belirtecektir. Örneğin:

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

/* Symbolic Constants */

#define MAX_CMD_LINE 1024
#define MAX_PARAMS 32

/* Type Declarations */

typedef struct tagCMD{
    const char *cmdText;
    void(*proc)(void);
} CMD;

/* Function Prototypes */

void print_syserr(const char *msg);
void parse_cmdline(void);
void proc_cls(void);
void proc_exit(void);
void proc_cd(void);

/* Global Object Definitions */

char g_cmdLine[MAX_CMD_LINE];
```



```

CMD g_cmds[] = {
    { "cls", proc_cls },
    { "exit", proc_exit },
    { "cd", proc_cd },
    { NULL, NULL }
};
char g_cwd[MAX_PATH];

char *g_params[MAX_PARAMS];
int g_nparams;

/* Function Definitions */

int main(void)
{
    int i;

    for (;;) {
        GetCurrentDirectory(MAX_PATH, g_cwd);
        printf("%s>", g_cwd);
        gets(g_cmdLine);
        parse_cmdline();

        if (g_nparams == 0)
            continue;
        for (i = 0; g_cmds[i].cmdText != NULL; ++i)
            if (!strcmp(g_params[0], g_cmds[i].cmdText)) {
                g_cmds[i].proc();
                break;
            }
        if (g_cmds[i].cmdText == NULL)
            printf("command not found!..\n");
    }

    return 0;
}

void parse_cmdline(void)
{
    char *str;

    g_nparams = 0;
    for (str = strtok(g_cmdLine, " \t"); str != NULL; str = strtok(NULL, " \t"))
        g_params[g_nparams++] = str;
}

void proc_cd(void)
{
    if (g_nparams == 1) {
        puts(g_cwd);
        return;
    }
    if (g_nparams > 2) {
        printf("too many arguments!..\n");
        return;
    }
    if (!SetCurrentDirectory(g_params[1]))
        print_syserr("cd");
}

void proc_cls(void)
{
    system("cls");
}

void proc_exit(void)

```

```

{
    exit(EXIT_SUCCESS);
}

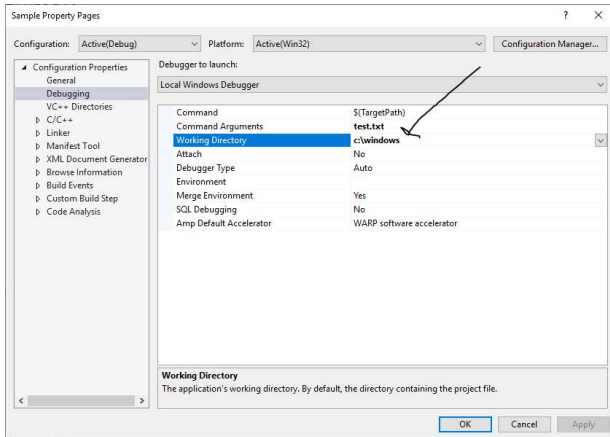
void print_syserr(const char *msg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", msg, lpszErr);
        LocalFree(lpszErr);
    }
}
}

```

5.1. Proses Çalışmaya Başladığında Onun Çalışma Dizini Neresidir?

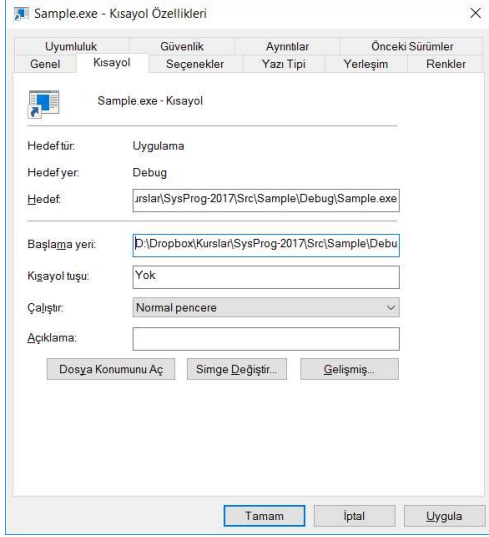
İşletim sistemlerinde her proses aslında bir sistem fonksiyonuyla başka bir proses tarafından yaratılmaktadır. Örneğin proses yaratmak için Windows'ta CreateProcess isimli API fonksiyonu, UNIX/Linux sistemlerinde ise fork isimli POSIX fonksiyonu kullanılmaktadır. Windows sistemlerinde procesi yaratmak için kullanılan CreateProcess API fonksiyonunun sekizinci parametresiyle yaratılacak proses için başlangıçtaki çalışma dizini belirlenebilmektedir. Eğer bu sekizinci parametrede yaratılacak proses için bir çalışma dizini belirtilmezse (yani bu parametre NULL geçilirse) yeni yaratılacak prosesin çalışma dizini üst procesten (yani onu yaratan procesten) alınır. (Yani bu durumda yeni yaratılan prosesin çalışma dizini üst prosesin çalışma dizini ile aynı olur.) Örneğin Visual Studio'da biz programı çalıştırmak için CTRL+F5 yaptığımızda aslında programı Visual Studio procesi CreateProcess API fonksiyonunu uygulayarak çalıştırmaktadır. İşte Visual Studio IDE'si de C/C++ projelerinde çalıştırdığı programın çalışma dizinini CreateProcess sırasında proje dizini olarak ayarlamaktadır. Tabii biz istersek Visual Studio'da proje ayarlarındaki "Debug" sekmesinden çalıştırılacak programın başlangıçtaki çalışma dizinini ayarlayabiliriz.



Diğer IDE'lerde de çalıştırılacak programın default çalışma dizini benzer biçimlerde belirlenebilmektedir.

Peki Windows'ta biz programı IDE'den değil de komut satırından ya da masaüstünden çalıştırdığımızda çalıştırılan programın default çalışma dizini neresi olmaktadır? İşte Windows sistemlerindeki komut satırı aslında "cmd.exe" isimli program tarafından oluşturulmaktadır. Bu komut satırı programıyla biz komut satırından yeni bir programı çalıştırdığımızda cmd.exe procesi her zaman çalıştırdığı prosesin çalışma dizinini işin başında kendi çalışma dizini olarak (yani prompt'ta gördüğümüz dizin olarak) ayarlar. Windows'ta grafik arayüz de aslında "explorer.exe" isimli bir procestir. Dolayısıyla biz bu sistemlerde bir dosyanın üzerine gelip çift tıklayarak bir programı çalıştırmak istediğimizde bunu "explorer.exe" CreateProcess uygulayarak çalıştırmaktadır. (Aslında "explorer.exe" programı ShellExecute isimli bir API fonksiyonu ile çalıştırır. Fakat ShellExecute fonksiyonu da CreateProcess fonksiyonunu çağırılmaktadır.) İşte "explorer.exe" default durumda çalıştırılan programın çalışma dizinini çalıştırılabilir (executable) dosyasının bulunduğu dizin olarak ayarlar. (Yani örneğin biz masaüstünden hareketle "C:\SysProg" dizinindeki "sample.exe" programını

çalıştırmak istersek bu programın çalışma dizini "C:\Sample" olacaktır.) Aslında biz kısa yok oluşturarak farenin sağ tuşuna basıp ilgili programın çalışma dizinini de belirleyebiliriz.



Bu diyalog penceresinde "başlama yeri" boş bırakılırsa bu durumda prosesin default çalışma dizini masaüstü (desktop) dizini olacaktır.

UNIX/Linux sistemlerinde prosesin çalışma dizini her zaman fork işlemi sırasında üst procesten alınmaktadır. Ancak fork işleminden sonra bu işlemi uygulayan üst proses isterse yarattığı prosesin çalışma dizinini chdir POSIX fonksiyonuyla değiştirebilir.

UNIX/Linux sistemlerinde en çok kullanılan komut satırı bash (/bin/bash) isimli programdır. Bu program yoluyla komut satırından bir program çalıştırıldığında her zaman çalıştırılan programın çalışma dizini bash prosesinin o anki çalışma dizini (yani komut satırında bulunduğumuz dizin) olacaktır.

5.2. Nokta ve Nokta Nokta Dizinleri

Hem Windows'ta hem de UNIX/Linux sistemlerinde yol ifadelerinde kullanılabilen iki özel dizin vardır: Bunlar "." ve ".." dizinleridir. Nokta dizini onun solundaki dizin ile aynı dizini, nokta nokta dizini ise onun solundaki dizinin üst dizinini belirtir. Eğer nokta ve nokta nokta dizinlerinin solunda bir şey yoksa sanki orada prosesin çalışma dizini varmış gibi işlem yapılmaktadır. Örneğin "sample.dat" yol ifadesiyle ".\sample.dat" yol ifadesi aynı anlamdadır. "..\..\sample.dat" yol ifadesi bulunulan dizinin iki üst dizinindeki "sample.dat" dosyası anlamına gelir. "\\a\b\c\d\..\..\e.dat" yol ifadesi "\\a\e.dat" yol ifadesi ile aynı anlamdadır.

Anahtar Notlar: Windows'ta ve UNIX/Linux sistemlerinde dizin geçişlerinde birden fazla '\' ya '/' karakteri kullanılabilir. Yani örneğin "c:\temp\ad.at" yol ifadesi ile "c:\temp\\\\\\\\\a.dat" yol ifadesi eşdeğerdir.

6. Özyineleme (Recursion), Özyinelemeli Algoritmalar ve Özyinelemeli Fonksiyonlar

Bir olgunun kendisine çok benzer bir olguyu içermesi durumuna özyineleme (recursion) denilmektedir. Özyineleme hem doğada hem de bilgisayar bilimlerinde sıklıkla karşılaştığımız bir olgudur. Örneğin dizin ağacını dolaşmak için dizin listesinde ilerlediğinizi düşünün. Dizin içerisindeki bir girişin bir dizin belirttiğini anladınız ve o dizine geçtiniz. İşte orada da benzer bir yapı karşınıza çıkacaktır. O halde dizin yapısı özyineleme içermektedir.

Özyineleme içeren algortmalara özyinelemeli algortmalar (recursive algorithms) denilmektedir. Yine dizin ağacını listeleme örneğine bakalım. Listeleme için kök dizinden girelim ve dosyaları buldukça yazdıralım. Pekiyi karşımıza bir dizin çıkarsa ne yaparız? Onun da içine geçerek aynı şeyi onun için de yaparız değil mi? İşte "eğer bir algoritmayı çözmek için ilerlerken bir noktaya geldiğimizde, bu geldiğimiz noktada başladığımız noktaya çok benzer bir durumla karşılaşıyorsak" muhtemelen bu algortima özyinelemeli bir algortmadır.

Özyinelemeli algoritmalar tipik olarak kendi kendini çağıran fonksiyonlarla (recursive functions) gerçekleştirilmektedir. Tabii özyinelemeli algortimaların başka biçimlerde de çözümleri olabilmektedir. (Örneğin fonksiyonun kendisini çağırması yapay bir stack ile döngüsel biçimde sağlanabilmektedir.)

Yazılımda karşımıza çıkan tipik özyinelemeli algortimalardan bazıları şunlardır:

- Dizin ağacının dolaşılması
- Ağaçlar ve graflar gibi veri yapılarının dolaşılması ve bu veri yapılarında arama yapılması
- Parsing algoritmaları
- Özel bazı problemler (Örneğin 8 vezir problemi)
- Bazı Sort işlemleri (quick sort, merge sort, heap sort) vs.
- Matematiksel bazı algortimalar
- Bazı optimizasyon algortimaları

Algoritmik problemleri özyineleme bakımından üç gruba ayırabiliriz:

- 1) Hem normal (özyinelemesiz) hem de özyinelemeli olarak gerçekleştirilebilecek problemler
- 2) Yalnızca özyinelemeli olarak gerçekleştirilebilecek problemler
- 3) Yalnızca normal (özyinelemesiz) olarak gerçekleştirilebilecek problemler

Bazı algoritmik problemler hem normal algoritmalarla hem de özyinelemeli algoritmalarla çözülebilmektedir. Bazı algoritmik problemler ise yalnızca özyinelemeli algoritmalar çözülebilmektedir. Gerçi bazı özyinelemeli algoritmaların normal (iteratif) çözümleri olabiliyorsa da bunlar çok verimsiz bir çözüm oluşturmaktadır.

6.1. Özyinelemeli Fonksiyonlar (Recursive Functions)

Bilindiği gibi fonksiyonların parametre değişkenleri ve yerel değişkenleri stack'te yaratılmaktadır. Çok prosesli fakat tek thread'li sistemlerde her prosesin ayrı bir stack'i bulunmaktadır. Çok thread'li sistemler de ise her thread ayrı bir stack'e sahiptir. Ayrıca pek çok sistemde bir fonksiyon çağrılırken CALL işlemi sırasında fonksiyonun geri dönebilmesi için geri dönüş adresi de prosesin (ya da thread'in) stack'inde saklanmaktadır. Ancak global ve static yerel değişkenlerin stack'te değil prosesin "data" ya da "bss" bölümlerinde yaratıldığını anımsayınız.

Aslında bir fonksiyonun kendini çağırmasıyla başka bir fonksiyonu çağırması arasında hiçbir farklılık yoktur.

Örneğin:

```
void bar(void)
{
    int a;
    ...
}

void foo(void)
{
    int a;
    ...
    bar();
    ...
}
```

Burada bar çağrıldığında bar'ın içerisinde yeni bir "a" nesnesi stack'te yaratılacaktır ve bar sona erdiğinde akış çağrılan yerden devam edecektir. Artık buradaki "a" nesnesi foo'daki "a" nesnesi olacaktır. Fonksiyonun kendi kendini çağırması da tamamen bu biçimde gerçekleşir. Örneğin:

```
void foo(void)
{
    int a;
```

```

...
foo();
...
}

```

Burada foo kendini her çağırıldığında yeni bir "a" nesnesi yaratılır. foo'nun çalışması sona erdiğinde akış bir önceki çağırmadan devam eder ve artık "a" bir önceki çağırmada yaratılan "a" olur.

Tabii fonksiyonun sürekli kendini çağırması sonsuz döngü oluşmasına yol açacaktır. O halde fonksiyon bir noktaya kadar kendini çağırılmalı bir noktadan sonra artık bundan vaz geçmelidir. Örneğin:

```

#include <stdio.h>

void foo(int n)
{
    printf("Giris:%d\n", n);

    if (n == 0)
        return;

    foo(n - 1);

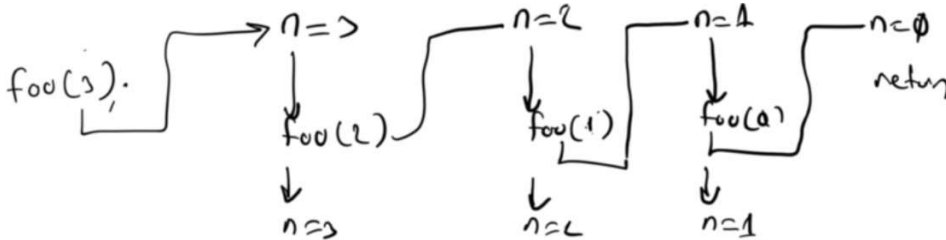
    printf("Cikis:%d\n", n);
}

int main(void)
{
    foo(3);

    return 0;
}

```

Burada fonksiyon kendini hep bir eksik değerle çağırıştır. Parametre değişkeni olan n 0 değerine geldiğinde artık kendini çağırma süreci sonlanmış ve çıkış süreci başlamıştır. Yukarıdaki çalışmayı şekilsel olarak aşağıdaki gibi gösterebiliriz:



Özyinelemeli çağırma şuana benzetilebiliriz: Bir suya akış dalıyoruz, dibe kadar gidiyoruz, sonra dipten yeniden yukarı çıkıyoruz. Eğer biz sürekli dibe inersek bu durum gerçek su dalışında "vurguna" programlamada ise "stack taşmasına (stack overflow)" yol açar.

6.2. Özyinelemeli Fonksiyonlara Örnekler

Yukarıda da belirtildiği gibi algoritmik problemlerin bir bölümü normal (özyinelemeli olmayan) biçimde, bir bölümü yalnızca özyinelemeli biçimde bir bölümü ise hem normal hem de özyinelemeli biçimde çözülebilmektedir. Özyineleme içermeyen çözümlere genellikle "iteratif çözüm" denilmektedir. Özyineleme hem karışık hem de debug edilmesi zor bir çözüm yöntemidir. Bu nedenle programcı gerekmediği durumlarda özyinelemeye başvurmamalıdır. Aşağıdaki örneklerin pek çoğunda algoritmik problem etkin bir biçimde iteratif de (yani özyinelemeli olmayan biçimde) çözülebilmektedir. Ancak biz bu bölümde özyineleme çalışması için bu örnekleri vermek istiyoruz.

6.2.1. Özyinelemeli Faktöriyel Hesabı

Faktöryel hesabının tavsiye edilen normal gerçekleştirimi şöyledir:

```
#include <stdio.h>

unsigned long factorial(unsigned n)
{
    unsigned long f = 1;

    for (; n > 1; --n)
        f *= n;

    return f;
}

int main(void)
{
    long result;

    result = factorial(10);
    printf("%lu\n", result);

    return 0;
}
```

Özyinelemeli versiyonu ise şöyle yazılabilir:

```
#include <stdio.h>

unsigned long factorial(unsigned n)
{
    unsigned long result;

    if (n == 0)
        return 1;

    result = n * factorial(n - 1);

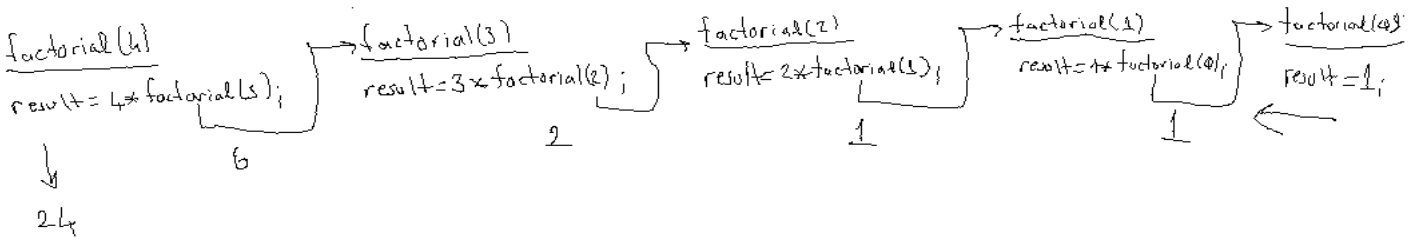
    return result;
}

int main(void)
{
    long result;

    result = factorial(4);
    printf("%lu\n", result);

    return 0;
}
```

Bu fonksiyon şöyle çalışmaktadır:



Fonksiyon daha basit olarak şöyle de yazılabilirdi:

```
unsigned long factorial(unsigned n)
{
```

```

    if (n == 0)
        return 1;

    return n * factorial(n - 1);
}

```

6.2.2. Yazının Tersten Yazdırılması

Özyineleme "düz olan bir şeyi ters çevirmek" ya da "ters olan bir şeyi düzeltmek" için sık kullanılmaktadır. Bu problemde de aslında özyinelemeye hiç gerek yoktur. Problemin klasik iteratif çözümü şöyle olabilir:

```

#include <stdio.h>

void putsrev(const char *str)
{
    int i;

    for (i = 0; str[i] != '\0'; ++i)
        ;
    for (--i; i >= 0; --i)
        putchar(str[i]);
}

int main(void)
{
    putsrev("ankara");
    putchar('\n');

    return 0;
}

```

Burada i'nin işaretsiz (örneğin size_t) bir türden olmasının soruna yol açacağına dikkat ediniz. Bu durumda kodu daha değişik düzenlemek gerekir. Örneğin:

```

void putsrev(const char *str)
{
    size_t i;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    while (i-- > 0)
        putchar(str[i]);
}

```

Fonksiyon şöyle de yazılabilirdi:

```

void putsrev(const char *str)
{
    const char *temp = str;

    while (*str != '\0')
        ++str;
    while (*str-- != temp)
        putchar(*str);
}

```

Fonksiyonun özyinelemeli gerçekleştirimi de şöyle olabilir:

```

#include <stdio.h>

void putsrev(const char *str)
{
    if (*str == '\0')

```

```

        return;

    putsrev(str + 1);
    putchar(*str);
}

int main(void)
{
    putsrev("ankara");
    putchar('\n');

    return 0;
}

```

6.2.3. Bir Yazıyı Ters Yüz etme (strrev)

Fonksiyonun klasik özyinelemeli olmayan biçimi şöyle yazılabilir:

```

#include <stdio.h>

void mystrev(char *str)
{
    size_t n, i;
    char temp;

    for (n = 0; str[n] != '\0'; ++n)
        ;
    for (i = 0; i < n / 2; ++i) {
        temp = str[i];
        str[i] = str[n - i - 1];
        str[n - i - 1] = temp;
    }
}

int main(void)
{
    char s[] = "ankara";

    mystrev(s);
    puts(s);

    return 0;
}

```

Asıl işi yapan fonksiyonu çağıran fonksiyonlara sarma fonksiyonlar (wrapper functions) denilmektedir. Bazen özyineleme için özyinelenen fonksiyonların ilave parametrelere sahip olması gerekebilmektedir. İşte bu tür durumlarda sarma fonksiyonlardan faydalanılır. Sarma fonksiyonlar parametreleri oluşturarak asıl özyinelemeli fonksiyonları çağırabilirler. Aşağıdaki örnekte mystrev bir sarma fonksiyondur. Bu sarma fonksiyon mystrev_recur isimli özyinelemeli fonksiyonu çağırılmaktadır.

```

#include <stdio.h>

void mystrev_recur(char *str, size_t left, size_t right)
{
    char temp;

    if (left >= right)
        return;

    temp = str[left];
    str[left] = str[right];
    str[right] = temp;

    mystrev_recur(str, left + 1, right - 1);
}

```



```

}

char *mystrrev(char *str)
{
    size_t i;

    for (i = 0; str[i] != '\0'; ++i)
        ;

    mystrrev_recur(str, 0, i - 1);    /* mustrev_recur(str, 0, strlen(str) - 1); */

    return str;
}

int main(void)
{
    char s[] = "ankara";

    mystrrev(s);
    puts(s);

    return 0;
}

```

6.2.4. Bir Sayıyı İkilik Sistemde Yazdıran Fonksiyon

Bu fonksiyonun özyineleme içermeyen klasik iteratif biçimi şöyle yazılabilir:

```

#include <stdio.h>

void putbits(unsigned n)
{
    int i;

    for (i = sizeof(n) * 8 - 1; i >= 0; --i)
        putchar(((n >> i) & 1) + '0');

    putchar('\n');
}

int main(void)
{
    unsigned val;

    printf("Bir sayı giriniz:");
    scanf("%u", &val);

    putbits(val);
    putchar('\n');

    return 0;
}

```

Fonksiyonun çalışması şöyle özetlenebilir (n = 10 olsun):



Fonksiyonun özinelemeli biçimi şöyle yazılabilir:

```
#include <stdio.h>

void putbits(unsigned val)
{
    if (val == 0)
        return;

    putbits(val >> 1);
    putchar((val & 1) + '0');
}

int main(void)
{
    unsigned val;

    printf("Bir sayi giriniz:");
    scanf("%u", &val);

    putbits(val);
    putchar('\n');

    return 0;
}
```

Bu gerçekleştirimde sayının başındaki sıfırlar yazdırılmamaktadır. Ayrıca fonksiyon yazma işleminin sonunda imleci aşağı satırın başına geçirmemektedir. İstersek burada da bir sarma fonksiyondan faydalanabiliriz:

```
#include <stdio.h>

void putbits_recur(unsigned val, unsigned n)
{
    if (n == 0)
        return;

    putbits_recur(val >> 1, n - 1);
    putchar((val & 1) + '0');
}

void putbits(unsigned val)
{
    putbits_recur(val, sizeof(val) * 8 - 1);
    putchar('\n');
}

int main(void)
{
    unsigned val;

    printf("Bir sayi giriniz:");
    scanf("%u", &val);

    putbits(val);

    return 0;
}
```

6.2.5. Sayıların Yalnızca putchar Fonksiyonu Kullanılarak Yazdırılması

Aslında bilgisayar sistemlerinde ekrana sayı yazdırmak diye birşey yoktur. Yalnızca ekrana karakterler yazdırılabilir. Bu durumda örneğin aslında printf gibi bir fonksiyon int türden bir tamsayıyı yazdırırken sayıyı basamaklarına ayırıştırıp onlara karşı gelen karakterleri bastırmaktadır. Yani ekrana (stdout dosyasına) basım işleri aslında putchar gibi bir fonksiyonla yapılmaktadır. O halde yalnızca putchar kullanarak bir sayının yazdırılması sistem programlama için

önemlidir. Bu işlem tipik olarak özyinelemeli biçimde gerçekleştirilir. Bu problemin özyinelemeli olmayan çözümü özyinelemeli çözümünden çok daha kötüdür. Fonksiyonun özyinelemeli olmayan biçimi şöyle yazılabilir:

```
#include <stdio.h>

void putnum(int n)
{
    char s[16];
    int i, sign;

    if (n < 0) {
        sign = -1;
        n = -n;
    }
    else
        sign = 1;

    for (i = 0; n; ++i) {
        s[i] = n % 10 + '0';
        n /= 10;
    }

    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';

    for (--i; i >= 0; --i)
        putchar(s[i]);
    putchar('\n');
}

int main(void)
{
    int n = -1235678;

    putnum(n);

    return 0;
}
```

Tipik özyinelemeli çözüm ise şöyle gerçekleştirilebilir:

```
#include <stdio.h>

void putnum(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n == 0)
        return;

    putnum(n / 10);

    putchar(n % 10 + '0');
}

int main(void)
{
    int n = -1235678;

    putnum(n);
    putchar('\n');
}
```

```
    return 0;
}
```

Özyinelemeli putnum aşağıdaki gibi daha etkin biçimde de yazılabilir:

```
void putnum(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }

    if (n / 10)
        putnum(n / 10);
    putchar(n % 10 + '0');
}
```

Burada 0 değeri için bir çağrının yapılmadığına dikkat ediniz.

Şimdi de sayıyı herhangi bir tabanda putchar kullanarak ekrana yazdıralım:

```
#include <stdio.h>

void putnum(int n, int base)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }

    if (n / base)
        putnum(n / base, base);
    putchar(n % base >= 10 ? n % base - 10 + 'A' : n % base + '0');

    /* putchar((n % base >= 10 ? -10 + 'A' : '0') + n % base); */
}

int main(void)
{
    int n = 126;

    putnum(n, 16);
    putchar('\n');

    return 0;
}
```

10'luk sistemin yukarıdaki sistemlerde 10'dan büyük sayılar A, B, C, ... biçiminde karakterlerle temsil edilmektedir. Karakter tablolarında sayısal karakterlerin bir sıra izlemesi standartlarda garanti altına alınmıştır. Ancak sayısal karakterlerden hemen sonra alfabetik karakterlerin geleceğine yönelik bir garanti bulunmamaktadır. (Örneğin ASCII tablosunda '9' karakterinin numarası 57, 'A' karakterinin numarası 65'tir.)

6.2.6. Kapalı Şeklin İçinin Boyanması

Bu algoritmaya su basması (flood fill) denilmektedir. Tipik olarak kapalı şeklin içerisinde bir nokta alınır. O nokta boyanır. Sonra fonksiyon 4 yönde kendini çağırarak ilerler. Tabii kapalı bölgenin sınırına geldiği zaman ya da daha önce boyanan bir yere geldiği zaman durulur. Örneğin g_bitmap isimli char türden bir matriste '#' karakteri ile oluşturulmuş kapalı bir resim olsun. Resmin içinin boyanması aşağıdaki gibi özyinelemeli bir fonksiyonla yapılabilir:

```
void flood_fill(int row, int col, char ch)
{
    if (g_bitmap[row][col] == ch || g_bitmap[row][col] == '#')
        return;
```

```

g_bitmap[row][col] = ch;

flood_fill(row + 1, col, ch);
flood_fill(row, col + 1, ch);
flood_fill(row - 1, col, ch);
flood_fill(row, col - 1, ch);
}

```

Örneğin aşağıdaki gibi 10x20 uzunluğunda bir bitmap.txt dosyasının içerisinde kapalı bir şekil bulunuyor olsun:

```

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

```

Bu şekli dosyadan okuyup flood_fill fonksiyonu ile içini boyayan fonksiyon şöyle olabilir:

```

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

char g_bitmap[10][20 + 2];

void flood_fill(int row, int col, char ch)
{
    if (g_bitmap[row][col] == ch || g_bitmap[row][col] == '#')
        return;

    g_bitmap[row][col] = ch;

    flood_fill(row + 1, col, ch);
    flood_fill(row, col + 1, ch);
    flood_fill(row - 1, col, ch);
    flood_fill(row, col - 1, ch);
}

void disp_matrix(void)
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%s", g_bitmap[i]);
}

int main(void)
{
    FILE *f;
    int i;

    if ((f = fopen("bitmap.txt", "r")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; fgets(g_bitmap[i], 1000, f) != NULL; ++i)
        ;

    flood_fill(6, 5, '.');
}

```

```

disp_matrix();

return 0;
}

```

6.2.7. Seçerek Sıralama Yönteminin Özyinelemeli Olarak Gerçekleştirilmesi

Bilindiği gibi seçerek sıralama (selection sort) yöntemi dizideki en küçük (ya da en büyük) elemanın bulunup dizinin ilk elemanı ile yer değiştirilmesi, sonra dizinin daraltılarak daraltılmış dizi için de aynı şeylerin yapılması biçiminde gerçekleştirilir. Seçerek sıralama algoritmasının normal (yani öz yinelemeli olmayan) biçimi özyinelemeli biçiminden daha etkindir. Tipik gerçekleştirimi şöyledir:

```

#include <stdio.h>

void ssort(int *pi, size_t size)
{
    int i, k;
    int min, minIndex;

    for (i = 0; i < size - 1; ++i) {
        min = pi[i];
        minIndex = i;
        for (k = i + 1; k < size; ++k)
            if (pi[k] < min) {
                min = pi[k];
                minIndex = k;
            }
        pi[minIndex] = pi[i];
        pi[i] = min;
    }
}

void disp(const int *pi, size_t size)
{
    size_t i;

    for (i = 0; i < size; ++i)
        printf("%d ", pi[i]);
    printf("\n");
}

int main(void)
{
    int a[10] = { 12, 34, 22, 45, 67, 23, 11, 56, 43, 21 };

    ssort(a, 10);
    disp(a, 10);
}

```

Algoritmanın özyinelemeli biçiminde dizinin en büyük elemanı bulunup dizinin sonuna atılıp fonksiyonun 1 eksik uzunlukla kendini çağırması sağlanabilir. Örneğin:

```

#include <stdio.h>

void ssort(int *pi, size_t size)
{
    int maxIndex;
    int i, temp;

    if (size == 1)
        return;
}

```

```

maxIndex = 0;
for (i = 1; i < size; ++i)
    if (pi[i] > pi[maxIndex])
        maxIndex = i;
temp = pi[maxIndex];
pi[maxIndex] = pi[size - 1];
pi[size - 1] = temp;

ssort(pi, size - 1);
}

void disp(const int *pi, size_t size)
{
    size_t i;

    for (i = 0; i < size; ++i)
        printf("%d ", pi[i]);
    printf("\n");
}

int main(void)
{
    int a[10] = { 34, 23, 12, -4, 56, 99, 38, -50, 72, 41 };

    disp(a, 10);
    ssort(a, 10);
    disp(a, 10);

    return 0;
}

```

6.2.8. 8 Vezir Probleminin Özyinelemeli Olarak Çözülmesi

Satranç tahtasına birbirlerini yemeyen 8 vezirin yerleştirilmesine "8 Vezir Problemi" denilmektedir. Bu problemde tahtaya birbirlerini yemeyen 8 vezir 92 farklı biçimde yerleştirilebilmektedir. Çeşitli optimizasyonlar yapılabilmekle birlikte problemin düz mantık (brute force) çözümü şöyle gerçekleştirilebilir: Fonksiyonda bir tahta alınır. Tahtanın ilk boş yerine Vezir yerleştirilip, fonksiyonun kendisini çağırması sağlanır. Böylece fonksiyon her kendini çağırdıkça ilk boşa yere yine vezir yerleştirecektir. Böyle böyle 8 vezire gelindiğinde tahtanın durumu yazdırılır. Buradaki sorunlardan biri artık vezir yerleştiremeyince tahtanın durumunun ne olacağıdır. Tahta stack'te yerel olarak yaratılırsa zaten fonksiyon çıkışında eski durumuna gelecektir. Tabii tahta statik biçimde (yani global olarak) de yaratılabilir. Bu durumda özyinelemeli fonksiyon sonlandığında tahtanın yeniden bir önceki durumuna çekilmesi gerekir. Örnek bir çözüm şöyle olabilir:

```

#include <stdio.h>

#define SIZE      8

int g_qcount;
int g_count;
char g_board[SIZE][SIZE];

void init_board(void)
{
    int r, c;

    for (r = 0; r < SIZE; ++r)
        for (c = 0; c < SIZE; ++c)
            g_board[r][c] = '.';
}

void print_board(void)
{
    int r, c;

```

```

printf("%d\n", g_count);

for (r = 0; r < SIZE; ++r) {
    for (c = 0; c < SIZE; ++c)
        printf("%c", g_board[r][c]);
    printf("\n");
}
printf("\n");
}

void locate_queen(int row, int col)
{
    int r, c;

    g_board[row][col] = 'V';

    r = row;
    for (c = col + 1; c < SIZE; ++c)
        g_board[r][c] = 'o';
    for (c = col - 1; c >= 0; --c)
        g_board[r][c] = 'o';
    c = col;
    for (r = row - 1; r >= 0; --r)
        g_board[r][c] = 'o';
    for (r = row + 1; r < SIZE; ++r)
        g_board[r][c] = 'o';
    for (r = row - 1, c = col - 1; r >= 0 && c >= 0; --r, --c)
        g_board[r][c] = 'o';
    for (r = row - 1, c = col + 1; r >= 0 && c < SIZE; --r, ++c)
        g_board[r][c] = 'o';
    for (r = row + 1, c = col - 1; r < SIZE && c >= 0; ++r, --c)
        g_board[r][c] = 'o';
    for (r = row + 1, c = col + 1; r < SIZE && c < SIZE; ++r, ++c)
        g_board[r][c] = 'o';
}

void queen8(int row, int col)
{
    char board[SIZE][SIZE];
    int r, c;

    for (; row < SIZE; ++row) {
        for (; col < SIZE; ++col) {
            if (g_board[row][col] == '.') {
                for (r = 0; r < SIZE; ++r)
                    for (c = 0; c < SIZE; ++c)
                        board[r][c] = g_board[r][c];

                ++g_qcount;
                locate_queen(row, col);

                if (g_qcount == SIZE) {
                    ++g_count;
                    print_board();
                }

                queen8(row, col);
                --g_qcount;

                for (r = 0; r < SIZE; ++r)
                    for (c = 0; c < SIZE; ++c)
                        g_board[r][c] = board[r][c];
            }
        }
    }
    col = 0;
}

```



```

    }
}

int main(void)
{
    init_board();
    queen8(0, 0);

    return 0;
}

```

Bu örnekte biz global bir tahta oluşturup o anda vezirlerin yemediği kareleri '.' ile yediği kareleri 'o' ile temsil ettik. Sonra vezirleri bir döngü içerisinde boş olan karelere yerleştirip fonksiyonun kendisini çağırmasını sağladık. Programımızda global tahta her defasında yerel bir tahtaya kopyalanarak işlemler orada yürütülmüştür.

Diğer bazı önemli özyinelemeli algoritmalar kursumuzda başka konuların içerisinde ele alınacaktır.

6.2.9. Dizin Ağacının Özyinelemeli Biçimde Dolaşılması

Dizin ağacının dolaşımı tipik olarak özyinelemeli bir fonksiyonla yapılmaktadır. Böyle bir fonksiyonun genel tasarımı şöyle olabilir: Fonksiyonun bir parametresi vardır, o da içi listelenecek dizinin yol ifadesidir. Fonksiyon o dizinin içerisine geçer (yani çalışma dizinini o dizin olarak ayarlar) ve çalışma dizinindeki dosyaları listelemeye başlar. Listeleme sırasında bir dizin gördüğünde onu argüman yaparak kendisini çağırır. Böylece aynı işlemler o alt dizin için de yapılacaktır. Tabii dizin listesi bittiğinde fonksiyon sonlanmadan önce üst dizine geri dönmelidir.

Anahtar Notlar: Windows "Windows Explorer"da dosya ve dizinleri görüntülerken doğal sırada görüntülememektedir. Yani FindFirstFile ve FindNextFile fonksiyonlarıyla elde ettiğimiz sıra ile Windows'un bize "Windows Explorer"da gösterdiği sıra aynı değildir.

Windows'ta dizin dolaşma işlemi şöyle yapılabilir:

```

#include <stdio.h>
#include <string.h>
#include <Windows.h>

void WalkDir(const char *path)
{
    HANDLE hFF;
    WIN32_FIND_DATA finfo;

    if (!SetCurrentDirectory(path))
        return;

    hFF = FindFirstFile("*.*", &finfo);
    if (hFF == INVALID_HANDLE_VALUE)
        return;

    do {
        if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))
            continue;
        printf("%s\n", finfo.cFileName);
        if (finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
            WalkDir(finfo.cFileName);
            if (!SetCurrentDirectory(".."))
                break;
        }
    } while (FindNextFile(hFF, &finfo));

    FindClose(hFF);
}

int main(void)

```

```

{
    WalkDir("c:\\");

    return 0;
}

```

Tabii biz üst dizine geçme işlemini fonksiyonun sonuna da bırakabilirdik. Örneğin:

```

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

void WalkDir(const char *path)
{
    HANDLE hFF;
    WIN32_FIND_DATA finfo;

    if (!SetCurrentDirectory(path))
        goto EXIT;

    hFF = FindFirstFile("*.*", &finfo);
    if (hFF == INVALID_HANDLE_VALUE)
        return;
    do {
        if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))
            continue;
        printf("%s\n", finfo.cFileName);
        if (finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
            WalkDir(finfo.cFileName);
    } while (FindNextFile(hFF, &finfo));

    FindClose(hFF);
EXIT:
    SetCurrentDirectory("..");
}

int main(void)
{
    WalkDir("c:\\");

    return 0;
}

```

Windows'ta bazı dizinlerin listelenmesi yetki sorunu yüzünden başarısızlıkla sonuçlanabilir. Örneğin biz bazı dizinlere hiç geçemeyebiliriz dolayısıyla onun listesini de alamayabiliriz. İşte eğer istersek listeleyemediğimiz dizinler için bir hata mesajını da aşağıdaki gibi yazdırabiliriz:

```

#include <stdio.h>
#include <string.h>
#include <Windows.h>

void PutSysError(const char *msg);

void WalkDir(const char *path)
{
    HANDLE hFF;
    WIN32_FIND_DATA finfo;

    if (!SetCurrentDirectory(path)) {
        PutSysError("SetCurrentDirectory");
        return;
    }

    hFF = FindFirstFile("*.*", &finfo);

```

```

if (hFF == INVALID_HANDLE_VALUE)
    return;

do {
    if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))
        continue;
    printf("%s\n", finfo.cFileName);
    if (finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
        WalkDir(finfo.cFileName);
        if (!SetCurrentDirectory(".."))
            return;
    }
} while (FindNextFile(hFF, &finfo));

FindClose(hFF);
}

void PutSysError(const char *msg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", msg, lpszErr);
        LocalFree(lpszErr);
    }
}

int main(void)
{
    WalkDir("c:\\");

    return 0;
}

```

Yukarıdaki programların küçük bir sorunu vardır. WalkDir burada prosesin çalışma dizinini değiştirip onu öyle bırakmaktadır. Çalışma dizinini yine işlemin başlatıldığı orijinal dizinde bırakmak için sarma fonksiyondan faydalanılabilir:

```

#include <stdio.h>
#include <string.h>
#include <Windows.h>

void WalkDirRecur(const char *path)
{
    HANDLE hFF;
    WIN32_FIND_DATA finfo;

    if (!SetCurrentDirectory(path))
        return;

    hFF = FindFirstFile("*.*", &finfo);
    if (hFF == INVALID_HANDLE_VALUE)
        return;

    do {
        if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))
            continue;
        printf("%s\n", finfo.cFileName);
        if (finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
            WalkDirRecur(finfo.cFileName);
            if (!SetCurrentDirectory(".."))
                break;
        }
    } while (FindNextFile(hFF, &finfo));
}

```

```

    FindClose(hFF);
}

void WalkDir(const char *path)
{
    char cwd[MAX_PATH];

    GetCurrentDirectory(MAX_PATH, cwd);
    WalkDirRecur(path);
    if (!SetCurrentDirectory(cwd)) {
        fprintf(stderr, "Cannot set directory!..\n");
        exit(EXIT_FAILURE);
    }
}

int main(void)
{
    WalkDir("c:\\");

    return 0;
}

```

Anahtar Notlar: printf fonksiyonunda alan belirten değer de argüman olarak girilebilir. Bunun için %* sentaksı kullanılır. Örneğin "%*d" format karakterleri için iki argüman girilmelidir. Birincisi * için uzunluk belirten argüman, ikincisi d için değer bizzat kendisi. Örneğin:

```

char buf[] = "ankara";
int n;

scanf("%d", &n);
printf("%-*sxxx\n", n, buf);

```

Dizin ağacını tab'larla kademeli olarak da yazdırabiliriz. Örneğin:

```

#include <stdio.h>
#include <string.h>
#include <Windows.h>

#define TABSIZE 4

void WalkDirRecur(const char *path, int tabCount)
{
    HANDLE hFF;
    WIN32_FIND_DATA finfo;

    if (!SetCurrentDirectory(path))
        return;

    hFF = FindFirstFile("*.*", &finfo);
    if (hFF == INVALID_HANDLE_VALUE)
        return;

    do {
        if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))
            continue;
        printf("%*s", tabCount * TABSIZE, "");
        if (finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
            printf("%s <DIR>\n", finfo.cFileName);
            WalkDirRecur(finfo.cFileName, tabCount + 1);
            if (!SetCurrentDirectory(".."))
                break;
        }
        else
            printf("%s\n", finfo.cFileName);
    } while (FindNextFile(hFF, &finfo));

    FindClose(hFF);
}

```

```

void WalkDir(const char *path)
{
    char cwd[MAX_PATH];

    GetCurrentDirectory(MAX_PATH, cwd);
    WalkDirRecur(path, 0);
    if (!SetCurrentDirectory(cwd)) {
        fprintf(stderr, "Cannot set directory!..\n");
        exit(EXIT_FAILURE);
    }
}

int main(void)
{
    WalkDir("c:\\");

    return 0;
}

```

Dizin ağacını dolaşan fonksiyon fonksiyon göstericileriyle genelleştirilebilir. Şöyle ki: WalkDir fonksiyonu bizden ayrıca bir fonksiyon adresi de alır ve dizin ağacında her dosyayı buldukça o fonksiyonu çağırır. Böylece biz o fonksiyonun içerisinde ister dosyayı yazdırır istersek başka birşey yaparız. Örneğin:

```

#include <stdio.h>
#include <string.h>
#include <Windows.h>

BOOL WalkDirRecur(const char *path, BOOL(*Proc)(const WIN32_FIND_DATA *, int), int level)
{
    HANDLE hFF;
    WIN32_FIND_DATA finfo;
    BOOL bResult;

    if (!SetCurrentDirectory(path))
        return TRUE;

    hFF = FindFirstFile("*.*", &finfo);
    if (hFF == INVALID_HANDLE_VALUE)
        return TRUE;

    bResult = TRUE;
    do {
        if (!strcmp(finfo.cFileName, ".") || !strcmp(finfo.cFileName, ".."))
            continue;
        if (!Proc(&finfo, level)) {
            bResult = FALSE;
            break;
        }
        if (finfo.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) {
            if (!WalkDirRecur(finfo.cFileName, Proc, level + 1)) {
                bResult = FALSE;
                break;
            }
        }
        if (!SetCurrentDirectory("..")) {
            bResult = FALSE;
            break;
        }
    } while (FindNextFile(hFF, &finfo));

    FindClose(hFF);

    return bResult;
}

```

```

void WalkDir(const char *path, BOOL(*Proc)(const WIN32_FIND_DATA *, int))
{
    char cwd[MAX_PATH];

    GetCurrentDirectory(MAX_PATH, cwd);
    WalkDirRecur(path, Proc, 0);
    if (!SetCurrentDirectory(cwd)) {
        fprintf(stderr, "Cannot set directory!..\n");
        exit(EXIT_FAILURE);
    }
}

BOOL DispTree(const WIN32_FIND_DATA *fd, int level)
{
    printf("%*s\n", level * 4 + strlen(fd->cFileName), fd->cFileName);

    if (!strcmp(fd->cFileName, "sample.c"))
        return FALSE;

    return TRUE;
}

int main(void)
{
    WalkDir("d:\\Dropbox\\Kurslar", DispTree);

    return 0;
}

```

Buradaki geri çağrılan fonksiyonun (call-back function) geri dönüş değerinin BOOL türden olduğuna dikkat ediniz. Bu fonksiyon 0 ile geri döndüğünde artık özyineleme sonlandırılmaktadır. Kodda özyinelemenin sonlandırılma biçimini dikkatlice inceleyiniz.

Fonksiyon göstericileri sayesinde kodun genelleştirilebildiğine dikkat ediniz. Örneğin biz yukarıdaki programda geri çağrılan fonksiyonu dizin ağacında belli bir dosyanın yerlerini bulacak biçimde de yazabilirdik:

```

BOOL DispTree(const WIN32_FIND_DATA *fd, int level)
{
    if (!strcmp(fd->cFileName, "sample.c")) {
        char path[4096];
        if (!GetCurrentDirectory(4096, path)) {
            fprintf(stderr, "cannot get current directory!\n");
            exit(EXIT_FAILURE);
        }
        strcat(path, "\\");
        strcat(path, fd->cFileName);
        puts(path);
    }

    return TRUE;
}

```

UNIX/Linux sistemlerinde dizin ağacını dolaşan fonksiyon şöyle yazılabilir:

```

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

#define TABSIZE    4

```

```

void exit_sys(const char *msg);

void walkdir(const char *path, int level)
{
    DIR *dir;
    struct dirent *ent;
    struct stat finfo;

    if (chdir(path) == -1)
        return;

    if ((dir = opendir(".")) == NULL)
        return;

    while ((ent = readdir(dir)) != NULL) {
        if (!strcmp(ent->d_name, ".") || !strcmp(ent->d_name, ".."))
            continue;
        if (lstat(ent->d_name, &finfo) == -1)
            break;
        printf("%*s\n", level * 4, ent->d_name);
        if (S_ISDIR(finfo.st_mode))
            walkdir(ent->d_name, level + 1);
    }

    chdir("../");
    closedir(dir);
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    walkdir(argv[1], 0);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

UNIX/Linux sistemleri için de fonksiyon göstericisi olarak işlem yapan genel dolaşım fonksiyonunu şöyle yazabiliriz:

```

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

#define FALSE    0
#define TRUE     1

typedef int bool_t;

bool_t walkdir_recur(const char *path, bool_t (*proc)(const char *, struct stat *, int), int level)
{
    DIR *dir;
    struct dirent *dire;

```

```

struct stat finfo;
bool_t result;

if (chdir(path) < 0)
    return TRUE;

if ((dir = opendir(".")) == NULL)
    return TRUE;

result = TRUE;

while ((dire = readdir(dir)) != NULL) {
    if (!strcmp(dire->d_name, ".") || !strcmp(dire->d_name, ".."))
        continue;

    if (lstat(dire->d_name, &finfo) < 0)
        continue;

    if (!proc(dire->d_name, &finfo, level)) {
        result = FALSE;
        break;
    }

    if (S_ISDIR(finfo.st_mode)) {
        if (!walkdir_recur(dire->d_name, proc, level + 1)) {
            result = FALSE;
            break;
        }
    }
}
closedir(dir);
chdir("..");

return result;
}

bool_t walkdir(const char *path, bool_t(*proc)(const char *, struct stat *, int))
{
    char cwd[4096];
    bool_t result;

    if (getcwd(cwd, 4096) == NULL)
        return FALSE;

    result = walkdir_recur(path, proc, 0);

    if (chdir(cwd) < 0)
        return FALSE;

    return result;
}

bool_t disp(const char *path, struct stat *finfo, int level)
{
    printf("%*s\n", level * 4 + (int)strlen(path), path);

    return TRUE;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }
    walkdir(argv[1], disp);
}

```



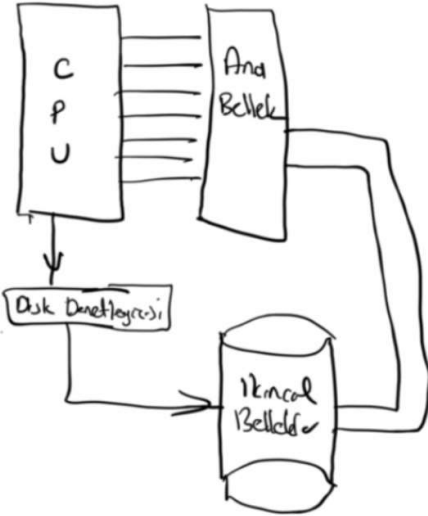
```
    return 0;
}
```

Yukarıdaki dizin ağacını dolaşan örneklerde dolaşımın "depth-first" biçimde yapıldığına dikkat ediniz. Yani biz bir dizinde ilerlerken o dizindeki ilk alt dizinin içerisine dalmaktayız. Böyle bir dolaşımında dolaştığımız kökteki son dosya en son bulunacaktır. Fakat bazen arama işlemlerinde bu istenmez. Bu durumda yukarıdaki dolaşımında küçük bir değişiklik yapmak gerekebilir. Şöyle ki: Her dizinin içeriği toplamda iki kez elde edilebilir. İlk elde etmede bu dizin baştan sona dolaşmış olur. İkinci elde etmede ise depth-first işlem uygulanır. Böylece kök dizindeki son dosya ilk önce bulunabilecektir.

7. Bellek Sistemleri

Bilgileri tutmakta ve geri almakta kullanılan birimlere bellek (memory) denilmektedir. Bilgisayar sistemlerindeki bellekler "Birincil Bellekler (Primary Memory)" ve "İkincil Bellekler (Secondary Memory)" olmak üzere ikiye ayrılır. Birincil belleklere "Ana Bellekler (Main Memeory)" de denilmektedir. Birincil bellekler ya da ana Bellekler CPU ile elektriksel olarak bağlantı halindedir. Bunlar genel olarak bilgisayarın güç kaynağı kesildiğinde bilgileri tutamazlar. İkincil bellekler güç kaynağı kesildiğinde bilgileri tutabilen belleklerdir. Tipik bir çalışmada bilgisayar kapatılmadan önce bilgiler ikincil belleklerde dosyalar biçiminde saklanır. İkincil bellekler genellikle dizinler biçiminde organize edilmiştir. İşletim sisteminin ikincil belleği organize eden alt sistemine "dosya sistemi (file system)" denilmektedir. İkincil bellekler tipik olarak hard-diskler, SSD'ler, flash EPROM'lar, CD/DVD ROM'lar biçiminde bulunmaktadır. Eskiden ikincil bellek olarak disketler ve teyp bantları da kullanılıyordu. Daha sonra hard-diskler ve CD/DVD ROM'lar yaygınlaştı. Disketler ve teyp bantları büyük ölçüde teknoloji dışı kaldılar. Artık bugünlerde SSD'lerin de hard disklerin yerini almaya başladığı söylenebilir.

Tipik olarak bir bilgisayar sisteminin bellek mimarisi şöyle özetlenebilir:

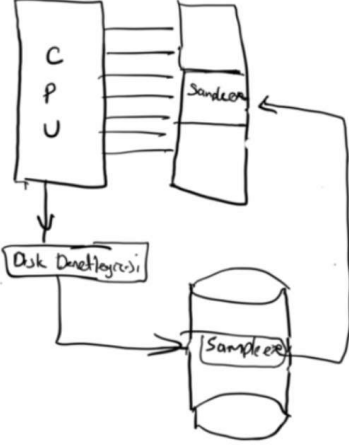


CPU çalışırken sürekli olarak ana belleğe erişmektedir. Örneğin bir ifade çalıştırılırken CPU değişkenlerin değerlerini ana bellekten alır, işleme sokar ve sonucu yeniden ana bellekteki değişkene aktarır. Örneğin aşağıdaki gibi bir ifade olsun:

```
a = b + c;
```

Burada a, b, ve c birincil bellektedir. CPU b ve c'yi birincil bellekten kendi içerisine çeker. Toplama işlemini kendi içerisindeki elektrik devreleriyle gerçekleştirir. Sonucu yeniden ana bellekteki a'ya aktarır. Programın makine komutları da yine ana belleklerde bulunmaktadır. Yani CPU aynı zamanda çalıştıracağı komutları da ana bellekten almaktadır. Yani program çalışırken programın kodu da değişkenler (nesnelere) de ana bellekte bulunmaktadır. İşletim sistemlerinin yüklü olduğu bir bilgisayar sisteminde çalıştırılabilen (executable) program bir dosya biçiminde (örneğin "sample.exe") ikincil belleklerde bulunur. Bu program çalıştırılmak istendiğinde işletim sistemi tarafından önce ana belleğe yüklenir. İşletim sisteminin bu işi yapan kısmına "yükleyici (loader)" denilmektedir. Ana belleğe yüklenen programın kodu CPU tarafından

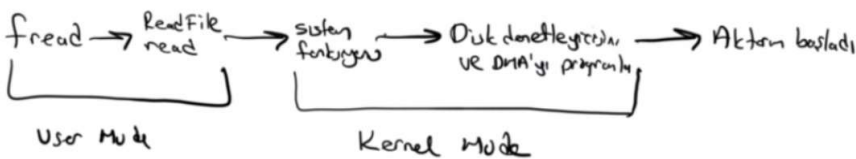
bellekten alınarak çalıştırılır. Bu çalıştırma sırasında aynı zamanda değişkenler (nesneler) de bellekten alınarak işleme sokulup güncellenmektedir.



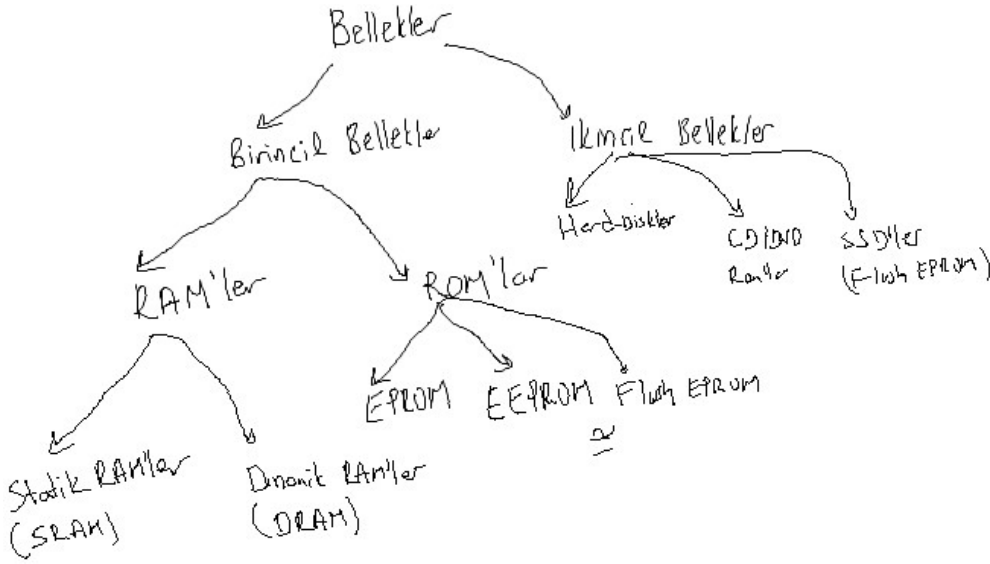
Çalıştırılabilen dosyaların içerisinde program kodlarının ve verilerinin nerede olduğu bilinmektedir. Genellikle bu dosyaların bir başlık kısımları (header) bulunmaktadır. Bu başlık kısımlarında dosya içerisindeki bölümler (sections) hakkında bilgiler bulunmaktadır. Tipik bir çalıştırılabilen dosyanın içeriği aşağıdaki gibidir:



Pek çok bilgisayar mimarisinde ikincil belleklerle ana bellekler arasında bir aktarım yolu vardır ve ikincil belleklerden birincil belleklere aktarım CPU tarafından değil başka bir birim tarafından (genellikle DMA olarak isimlendirilirler) gerçekleştirilmektedir. İkincil belleklerden ana belleklerden aktarım tipik olarak şöyle yapılmaktadır: CPU ikincil belleği kontrol eden birime (disk controller) ve aktarımı yapacak birime (DMA) komutlar yollayarak aktarım işlemini başlatır. Bundan sonra artık süreci izlemez. Kontrol birimi disk üzerindeki işlemcileri programlar, diskin kafalarını hareket ettirir, diskten okumayı yapar. DMA da okunan bilgileri birincil belleğe belirlenen adresten itibaren aktarır. İşlem bitince CPU bir donanım kesmesiyle haberdar edilmektedir. Yani diskteki bir bilginin RAM'e aktarılması işlemi CPU tarafından başlatılmakta ancak aktarımın kendisi CPU tarafından yapılmamaktadır. Başka bir deyişle CPU yalnızca aktarımın yapılabilmesi için çevre işlemcilerini programlamaktadır. Gerçek aktarım CPU'dan bağımsız biçimde arka planda yürütülmektedir. Tabii aktarımı başlatan CPU aslında bu işlemi çalıştırdığı kodla yapmaktadır. Yani bütün işlemler aslında daha önce yazılmış olan kodların işletilmesi yoluyla yapılmaktadır. Örneğin aşağı seviyeli olarak biz fread standart C fonksiyonu ile diskten bir grup veriyi belleğe transfer etmek isteyelim. Bu fread fonksiyonu aslında arka planda Windows sistemlerinde ReadFile isimli API fonksiyonunu, Linux sistemlerinde de read isimli POSIX fonksiyonunu çağırır. Bu fonksiyonlar da işletim sisteminin çekirdeği içerisinde bulunan sistem fonksiyonlarını çağırırlar. İşte disk denetleyicisini ve DMA'yı programlayan kodlar bu sistem fonksiyonlarının içerisinde yer almaktadır. Dolayısıyla aslında tüm aktarım makine komutlarının çalıştırılması yoluyla gerçekleşmektedir.



Birincil bellekler ve ikincil bellekler de kendi aralarında yapısal farklılıklara göre ayrıca sınıflandırılmaktadır:



Birincil Bellekler RAM'ler ve ROM'lar biçiminde ikiye ayrılmaktadır. RAM sözcüğü "Random Access Memory" sözcüklerinden kısaltılmıştır. (Buradaki "Random" sözcüğü hız belirtmektedir.) RAM'ler read/write belleklerdir. Bunlar teknolojik olarak statik RAM ve dinamik RAM olmak üzere iki biçimde üretilmektedir. Statik RAM'ler daha hızlıdır (tipik olarak 1 nanosaniyenin altında), dinamik RAM'lar daha yavaştır (tipik olarak 10 nanosaniye civarında). SRAM'ler daha büyük yer kaplarlar, DRAM'lar ise daha küçük yer kaplamaktadır. SRAM'ler tipik olarak flip-flop devresiyle (SR latch) gerçekleştirilmektedir. Dolayısıyla bunların 1 biti genellikle 4 transistörle yapılmaktadır. Halbuki DRAM'ların 1 biti bir transistör ve bir kapasitif elemanla gerçekleştirilebilmektedir. Ayrıca DRAM'larda tazeleme problemi de vardır. Çünkü kapasitif elemandaki yük zamanla düşer. Bunun düşmemesi için belli periyotlarda RAM'in tazelenmesi gerekmektedir. (Aslında tazeleme işlemini de bizzat CPU'unun kendisi yapmaktadır. CPU'nun bir ucu RAM'in bir ucuna bu amaçla bağlanmaktadır.) Oysa SRAM'ların tazeleme problemleri yoktur. Toplamda SRAM'ler DRAM'lardan daha pahalıdır. Bu yüzden ana bellekler halen SRAM'lerle değil DRAM'larla yapılmaktadır. SRAM'ler daha çok CPU içerisindeki cache sistemlerinde tercih edilmektedir.

ROM sözcüğü "Read Only Memory" sözcüklerinden kısaltılmıştır. Bu sözcüğün artık bugün için teknolojik bir anlamı kalmamıştır. Eskiden ROM'ların içerisine bir kez üretici firma tarafından bilgiler yerleştiriliyor ve bir daha da bu bilgiler değiştirilemiyordu. ROM'ların en önemli özelliği bilgisayarın güç kaynağı kesildiğinde de içerisindeki bilgileri tutabilmesidir. Halbuki RAM'ler güç kaynağı kesildiğinde içerindeki bilgiyi kaybetmektedir. ROM'lardan sonra EPROM isimli (Erasable Programmable ROM) silinibilen ve programlanabilen ROM'lar geliştirildi. EPROM'lar EPROM silici denilen aygıtla (ultraviyole ışınlarıyla silme yapılıyordu) silinebiliyordu ve EPROM programlayıcı denilen aygıtla da yeniden onlara bilgiler aktarılabilirdi. Bugün artık EPROM'lar da teknoloji dışı kalmak üzeredir. Yeni teknolojiye artık CPU tarafından doğrudan yazılabilen EEPROM (Electrically Erasable ROM) ve Flash EPROM'lar kullanılmaktadır. Bunlar yine güç kaynağı kesildiğinde bilgiyi tutmaya devam ederler. Ancak hiç bilgisayardan çıkartılmadan bunların içerisine yeniden yeni bilgiler yazılabilmektedir.

Pekiye ROM'lara neden gereksinim duyulmaktadır? Aslında tüm bilgisayar sistemlerinde en azından boot işlemi için bir kısım ROM belleğin bulunması gerekmektedir. CPU'lar reset edildiğinde belli bir adresten çalışmaya başlarlar. Buna CPU'ların reset vektörü denilmektedir. İşte biz bilgisayarımızı açtığımızda CPU'muz reset edilir ve reset vektöründe hazır bir programın bulunması gerekmektedir. Böylece bilgisayarımızı reset ettiğimizde çalışma ROM'daki bir adresten başlamış olacaktır. Buradaki kod da birtakım ilk işlemleri yaparak diske başvurur ve işletim sistemini diskten RAM'e yükler. Böylece boot işlemi gerçekleştirilmiş olur. Tabii eskiden reset vektöründeki ROM bellekler EPROM'lardan oluşuyordu. Artık belli bir zamandır bu tarz ROM bellekler EEPROM teknolojileri ile gerçekleştirilmektedir. Böylece bilgisayarın sahibi isterse ROM'daki yazılımı güncelleyebilmektedir. Bugün kullandığımız PC'lerin ROM alanında bulunan kodlara BIOS (Basic Input Output System) denilmektedir. Genel olarak aygıtların ROM alanlarındaki temel yazılımlar ise "firmware" olarak isimlendirilmektedir.

O halde bir bilgisayar sisteminde ana bellek olarak RAM'ler ve ROM'lar bulunmaktadır. RAM'ler ve ROM'lar ayrı entegre devreler biçiminde üretilmiş olsalar da mantık devreleri sayesinde sanki aynı fiziksel adres alanında bulunuyorlarmış gibi bir durum oluşturulmaktadır.



İkincil bellekler için hala en yoğun kullanılan teknoloji hard disk teknolojisidir. Ancak bu teknoloji yerini yavaş yavaş SSD (Solid State Disk)'lere bırakmaktadır. Hard diskler elektro mekanik aygıtlardır. Yani hard disklerin içerisinde hem mekanik aksamlar hem de elektronik aksamlar bulunmaktadır. Hard disklerin fiyatları çok makuldür. Fakat güç harcamaları yüksektir ve çok hızlı değildir. Ayrıca hard diskler sarsıntıya karşı çok korunaklı da değildir. SSD'ler aslında sıkça kullandığımız flash EPROM ya da EEPROM teknolojileri ile imal edilmektedir. Yani SSD'ler entegre devre biçiminde imal edilirler ve herhangi bir mekanik parçaları yoktur. Bunların güç harcaması daha düşüktür ve disklere göre de oldukça hızlıdır. Flash EPROM ve EEPROM tarzı belleklerden okuma çok hızlıdır (nano saniye mertebesinde) fakat bunlara yazma görelilik olarak yavaştır (mili saniyeler mertebesinde). Ayrıca bu teknolojilerde bir Flash EPROM bloğuna belli kez yazma yapılabilmektedir. Gerçi bu sayı gitgide yükseltiyor olsa da hala bu teknolojinin handikaplarından biridir.

Önbellek (Cache) Sistemleri

Bilgisayar sistemlerinde genellikle hızlı ve yavaş bellekler söz konusudur. Yavaş bellekler bol ve ucuzdur, hızlı bellek ise az ve pahalıdır. İşte yavaş belleğin bir kısmının hızlı bellekte tutulup yavaş belleğe erişim oranını azaltmayı hedefleyen sistemlere "önbellek (cache) sistemleri" denilmektedir. Bir önbellek sisteminde yavaş belleğin belli bir bölümü hızlı bellekte tutulur. Böylece bilgiye erişilmek istendiğinde önce bilgi hızlı bellekte aranır. Eğer bilgi hızlı bellekte bulunursa hemen oradan alınarak kullanılır. Eğer bilgi hızlı bellekte bulunamazsa bu kez yavaş belleğe başvurulur. Bu sistemde yavaş belleğin bir bölümünü saklamakta kullanılan hızlı belleğe "önbellek (cache)" denilmektedir.

Önbellek sistemlerinde çeşitli terimler sıklıkla kullanılmaktadır. Bu sistemlerde talep edilen bilginin hızlı bellekte bulunması durumuna İngilizce "cache hit", bulunamaması durumuna ise "cache miss" denilmektedir. Bir önbellek sisteminin performansı "cache hit oranı (cache hit ratio)" ile ölçülür. Cache hit oranı n tane erişimin yüzde kaçının önbellekten karşılandığını belirtir. Örneğin cache hit oranı %70 demek, 100 erişimin 70'inin hızlı bellekten karşılanıyor olması demektir.

Çok karşılaştığımız önemli önbellek sistemlerinin bazıları şunlardır:

- Modern işletim sistemleri diske erişimi azaltmak için son erişilen disk bloklarını RAM'de tutarlar. Bu önbellek sistemlerine "disk cache" ya da "buffer cache" denilmektedir.
- Mikroişlemcilerin içerisinde SRAM'lerle oluşturulmuş önbellekler vardır. DRAM görelilik olarak yavaş olduğu için işlemci onun belli bölümlerini kendi içerisindeki SRAM'lerden oluşan bu önbelleğe çeker. Böylece CPU RAM'e erişeceği zaman önce bu önbelleğe başvurur. Bilgiyi önbellekte bulursa hızlı bir biçimde elde eder. Eğer bilgiyi önbellekte bulamazsa bu kez DRAM erişimi yapar. İşlemcilerin içerisinde bulunan bu önbellek sistemine "internal cache" ya da "L1 (Level 1 cache)" denilmektedir.
- Web tarayıcıları son erişilen web sayfalarının içeriğini yerel makinenin diskinde tutuyor olabilir. Böylece aynı sayfa talep edildiğinde bunu hızlı bir biçimde getirebilir. Burada hızlı bellek yerel diski yavaş bellek ise sunucudaki diski temsil ediyor durumdadır.
- İşlemci sayfa tablolarındaki sayfa girişlerini kendi içerisinde küçük bir tampon alanda tutar. Böylece sayfa tablosuna erişimi azaltır. Bu önbellek sistemine "TLB (Translation Lookaside Buffer)" denilmektedir.

- İşletim sistemleri son gezilen dizin girişlerini bir önbellekte sisteminde toplamaktadır. Buna da "directory entry cache" denilmektedir.

- Hard diskler de kendi içerisinde küçük bir önbellek kullanmaktadır. Hard diskten belli bir blok okunmak istendiğinde hard disk eğer bu blok zaten önbellekte varsa hemen önbellekteki bilgiyi teslim eder. Eğer okunmak istenen blok önbellekte yoksa disk kafalarını harekete geçirerek talep edilen bloğu okumaktadır.

- Pek çok dilde dosya işlemi yapan fonksiyonlar ve sınıflar "user modda" dosyanın son okunulan bölgelerini bir önbellek sisteminde tutmaktadır.

Anahtar Notlar: Buffer (tampon) sözcüğüyle cache (önbellek) sözcüğü bazen birbirlerine karıştırılmaktadır. Buffer bir meşguliyet yüzünden gelen bilgilerin bekletildiği bölgelere denilmektedir. Buffer sisteminin amacı bilgilerin uygun zamanda işlenmek üzere bekletilmesidir. Halbuki cache sisteminin amacı hız kazancı sağlamaktır. Yani buffer sisteminin ana amacı bilgilerin kaybedilmemesi, uygun zamanda işlenmek üzere bekletilmesidir.

Önbellek Terminolojisi

Bir önbellek sistemi "read-only" olabilir ya da "read-write" olabilir. Eğer önbelleğe yalnızca okuma yaparken erişiliyorsa böyle önbellek sistemlerine "read-only" önbellek sistemleri denir. Eğer önbelleğe hem okuma hem de yazma amaçlı erişiliyorsa böyle önbellek sistemlerine de "read-write" önbellek sistemleri denilmektedir. Read-only önbellek sistemlerinde okuma için önce önbelleğe başvurulur. Fakat yazma her zaman yavaş belleğe yapılır. Halbuki read-write önbellek sistemlerinde hem okuma hem de yazma sırasında önbellek kullanılmaktadır. Read-write cache sistemleri genel olarak daha hızlıdır. Ancak bazı durumlarda (elektirik kesilmesi gibi) bilginin bütünlüğü bozulabilir.

Bazı cache sistemlerinde yavaş belleğin tek bir ardışıl bloğu önbellekte tutulmaktadır. Bazı sistemlerde ise yavaş belleğin birden fazla küçük blokları önbellekte tutulabilmektedir. Yavaş belleğin önbellekteki kısımlarını tutan önbellek bölgelerine İngilizce "cache line" denir. Tabii böyle bir sistemde ilgili bilgi aranırken etkin bir biçimde onun önbellekte olup olmadığının belirlenmesi gerekir.

Önbellek için ayrılan cache line'ların tıka basa dolu olduğunu düşünelim. Yeni bir bilgi önbelleğe alınmak istendiğinde ne olacaktır? Bu durumda hangi cache line'ı önbellekten çıkartmak gerekir? Burada kullanılan algoritmalara önbellek yer değiştirme politikaları (cache replacement policy) denilmektedir. Tipik olarak üç tür yer değiştirme politikası vardır:

1) Least Frequently Used (LFU): Bu algortimada her cache line için bir sayaç tutulur. O cache line'a erişildikçe sayaç artırılır. Sonra Önbellekten bir line çıkartılacağı zaman sayacı en az olan çıkartılır. Burada o zamana kadar az kullanılmış bir önbellek bloğunun ileride de az kullanılacağı varsayımı yapılmaktadır.

2) Least Recently Used(LRU): Burada son zamanlarda en az erişilen cache line'lar önbellekten çıkartılma yoluna gidilir. Yani bu algortimada "son zamanlarda erişilmiş olma" değerli bir durumdur. Örneğin işletim sistemlerinin önbellek sistemlerinin çoğunda bu model tercih edilmektedir. Bu sistemin tipik gerçekleştirimi şöyle yapılmaktadır: Bir bağlı listede cache line'lar tutulur (numaraları da tutulabilir). Cache line kullanıldıkça bağlı listenin başına alınır. Böylece kullanılmayanlar zaten sonda kalacaktır. Önbellekten line çıkartılacağı zaman listede sonda olan çıkartılır.

3) Most Frequently Used (MFU): Bazı sistemlerde line'lara toplam erişim sayısı önceden öngörülelebilmektedir. Örneğin her bir bloğa toplamda 1000 civarında erişileceği biliniyor olabilir. Böyle bir sistemde tam tersine çok erişilmiş olan cache line'ların önbellekten atılması daha makuldür. Fakat bu algoritmanın uygun olabileceği sistemler çok azdır.

Uygulamada en fazla kullanılan önbellek yer değiştirme politikası LRU (Least Recently Used)'dur.

Önbellek Sistemleri Nasıl Gerçekleştirilir?

Önbellek sistemleri donanımsal ya da yazılımsal olarak gerçekleştirilebilmektedir. Donanımsaldan kastededilen tamamen elektrik devreleriyle önbellek sisteminin oluşturulmasıdır. Örneğin mikroişlemcilerin içerisindeki önbellek sistemi tamamen donanımsal olarak yönetilmektedir. Yazılımsal gerçekleştirmede önbellek sistemi bir program tarafından yönetilir. Örneğin işletim sistemlerinin disk önbellek sistemleri işletim sistemlerinin kernel kodları tarafından (dosya

sistemi tarafından) oluşturulmaktadır. Şüphesiz biz bu kursta daha çok yazılımsal olarak gerçekleştirilen önbellek sistemleri üzerinde duracağız.

Önbellek sistemleri yazılımsal olarak gerçekleştirilirken sistemi yönetmek için bir veri yapısına gereksinim duyulur. (Genellikle önbellek üzerinde arama yapmak için önbellek blokları bir hash tablosu biçiminde organize edilmektedir.) Aşağıda UNIX/Linux sistemleri için bir dosyanın "cache line"lar kullanılarak önbelleklenmesine ilişkin örnek bir arayüz verilmektedir. Bu örnek yalnızca fikir vermek için oluşturulmuştur:

```
typedef struct tagCACHE_LINE {
    char buf[LINE_SIZE];
    int blockno;
    ....
} CACHE_LINE;

typedef struct tagCACHE {
    int fd;
    CACHE clines[NCACHE_LINES];
    ....
} CACHE, *HCACHE;

HCACHE open_file(const char *path, int flags);
int read_file(HCACHE hc, int blockno, void *buf);
int write_file(HCACHE hc, int blockno, const void *buf);
int close_file(HCACHE hc);
```

Yukarıda verilen sistemde cache line'lar CACHE_LINE isimli bir yapıyla temsil edilmiştir. Bir CACHE_LINE nesnesi dosyanın önbelleklenmiş bir bloğu hakkında bilgi tutmaktadır. Bu yapının buf elemanı dosyanın ilgili kısmının içeriğini, offset elemanı ise dosyanın o kısmının blok numarasını tutar. Böylece sistem erişeceği yerin önbellekte olup olmadığını anlayabilecektir. Tüm cache line'ların CACHE içerisinde bir dizide saklandığına dikkat ediniz. Bu sistemi kullanan programcı önce open_file fonksiyonu ile dosyayı önbellekli biçimde açar. Buradan bir handle değeri elde eder. Bu handle değeri aslında önbellek bilgilerinin tutulduğu CACHE_LINE isimli yapı nesnesinin adresidir. Daha sonra programcı read_file ve write_file fonksiyonlarıyla önbellek sisteminden okuma ve yazma yapar. Bu fonksiyonlar önce okunacak ya da yazılacak yerin önbelleğin herhangi bir cache line'ında olup olmadığına bakacaklardır. Eğer söz konusu yer önbellekteyse işlemi önbelleği kullanarak yaparlar. Eğer söz konusu yer önbellekte değilse bu kez gerçek dosyadan işlemi yaparlar. CACHE_LINE yapısı içerisinde dosyaya erişmek için dosya betimleyicisi tutulmuştur. Bu örnekte read_file ve write_file fonksiyonlarının okunacak ya da yazılacak byte miktarını almadığına dikkat ediniz. Bu fonksiyonlar okunacak ya da yazılacak bloğun numarasını almaktadır. (Yani bu sistemde okuma ve yazma fonksiyonlarının zaten LINE_SIZE kadar bilgiyi okuyup yazdığını düşünebilirsiniz.)

Aslında ileride de görüleceği gibi normal dosyalar üzerinde programcının bir önbellekleme yapmasına gerek yoktur. Çünkü bu önbellekleme (buarada "tamponlama" terimi de kullanılmaktadır) zaten ilgili dilin ya da framework'ün kütüphanesindeki dosya fonksiyonları ya da sınıfları tarafından yapılmaktadır. Yani örneğin C'nin prototipleri <stdio.h> içerisinde olan dosya fonksiyonları zaten kendi içlerinde böyle bir önbellek sistemi kullanmaktadır. Bu konu izleyen bölümlerde ayrıntılarıyla ele alınmaktadır.

Bir önbellek sisteminde yavaş belleğin erişilmek istenen yerinin önbellekte olup olmadığının hızlı biçimde belirlenmesi gerekir. Örneğin yukarıdaki gibi bir sistemde toplam 1000 tane cache line buluyor olsun. Biz de dosyanın belli bir yerine erişmek isteyelim. O yerin önbellekte olup olmadığını anlamak için 1000 tane cache line'ı sırasıyla gözden geçirmek (sıralı arama) zaman kaybına yol açar. İşte yavaş belleğin ilgi bölümünün önbellekte bulunup bulunmadığını anlayabilmek için algoritmik arama yöntemleri kullanılmaktadır. Bu konuda en çok kullanılan yöntem "hash tabloları"dır. Hash tabloları kursumuzun temel veri yapılarının açıklandığı bölümünde ele alınmaktadır.

Bir önbellek sisteminin performansını etkileyen unsurlar şunlardır:

- 1) Kullanılan önbellek yer değiştirme algoritması: Yani yavaş belleğin neresinin önbellekte tutulacağını, önbellekten gerektiğinde hangi bloğun çıkartılacağını belirleyen algoritmalar. Bu algoritmalara ilgili sistem analiz edilerek karar verilir.
- 2) Önbellek miktarı: Şüphesiz önbellek ne kadar büyütülürse performans o kadar artar.

3) Önbelleğin belleğin hızı: Hızlı cache kullanmak şüphesiz performansı artırır.

4) Önbelleğin read-only ya da read-write olması.

Yukarıdaki 4 performans unsurundan en önemlisi "önbellek yer değiştirme algoritması"dır. Büyük ve hızlı bir önbellek yanlış bir algoritma ile kullanılırsa performans umulduğu gibi artmaz. Önbelleğin hızı konusunda genellikle tasarımcı çok belirleyici olamamaktadır. Önbellek miktarları üzerinde de zaten belli sınırlar vardır. Önbelleğin read-only mi yoksa read-write mı olacağı da bazı unsurlara bakılarak belirlenmektedir.

Aşağıda LFU algoritması kullanılarak "read only" bir önbellek sistemi örneği verilmiştir. Bu örnekte toplam 10 tane cache line vardır. Bir cache line önbellekten atılacağı zaman yeni okunan bilgi için cache line count değeri ortalama bir değerden başlatılmaktadır. Ancak bu örnekte "cache hit" durumu cache line'ler tek tek incelenerek sıralı bir aramayla belirlenmeye çalışılmıştır. Cache line'ların fazla olması durumunda sıralı arama önbellek performansını düşürür. Bu durumda aramanın algoritmik yöntemlerle (örneğin hash tabloları kullanılarak) yapılması uygun olur. Yine aşağıdaki örnekte en az kullanılan cache line'ın belirlenmesi de sıralı aramayla gerçekleştirilmiştir. Uygulamada burada da daha iyi yöntemlerin kullanılması söz konusu olmaktadır.

```
/* filecache.h */

#ifndef FILECACHE_H_
#define FILECACHE_H_

/* Symbolic Constants */

#define LINE_SIZE          32
#define NCACHE_LINES      10
#define INITIAL_COUNT      2

#ifdef DEBUG
#define DEBUG_PRINT(fmt, ...) fprintf(stderr, fmt, ## __VA_ARGS__)
#else
#define DEBUG_PRINT(fmt, ...)
#endif

/* Type Definitions */

typedef struct tagCACHELINE {
    char buf[LINE_SIZE];
    int blockno;
    size_t count;
} CACHE_LINE;

typedef struct tagCACHE {
    int fd;
    CACHE_LINE clines[NCACHE_LINES];
    size_t tcount;
} CACHE, *HCACHE;

/* Function Prototypes */

HCACHE open_file(const char *path, int flags);
int read_file(HCACHE hc, int blockno, void *buf);
int close_file(HCACHE hc);

#endif

/* filecahe.c */

#define DEBUG

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

```

#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include "filecache.h"

#ifdef DEBUG
static void print_cline_counts(HCACHE hc);
#endif

static size_t select_line(HCACHE hc);

HCACHE open_file(const char *path, int flags)
{
    HCACHE hc;
    int i;

    if ((hc = (HCACHE)malloc(sizeof(CACHE))) == NULL)
        return NULL;

    if ((hc->fd = open(path, flags)) == -1) {
        free(hc);
        return NULL;
    }

    for (i = 0; i < NCACHE_LINES; ++i) {
        hc->clines[i].blockno = -1;
        hc->clines[i].count = 0;
    }
    hc->tcount = 0;

    return hc;
}

int read_file(HCACHE hc, int blockno, void *buf)
{
    int i;
    int rline;

    for (i = 0; i < NCACHE_LINES; ++i)
        if (hc->clines[i].blockno == blockno) {
            DEBUG_PRINT("Cache hit block %d, used cache line %d\n", blockno, i);

            memcpy(buf, hc->clines[i].buf, LINE_SIZE);
            ++hc->clines[i].count;
            ++hc->tcount;

#ifdef DEBUG
            print_cline_counts(hc);
#endif

            return 0;
        }

    rline = select_line(hc);

    if (lseek(hc->fd, (off_t)blockno * LINE_SIZE, SEEK_SET) == -1)
        return -1;
    if (read(hc->fd, hc->clines[rline].buf, LINE_SIZE) == -1)
        return -1;
    hc->clines[rline].blockno = blockno;

    hc->clines[rline].count = hc->tcount / NCACHE_LINES + 1;
    hc->tcount += hc->clines[rline].count;

    memcpy(buf, hc->clines[rline].buf, LINE_SIZE);
}

```



```

    DEBUG_PRINT("Cache miss block %d, used cache line %d\n", blockno, rline);

#ifdef DEBUG
    print_cline_counts(hc);
#endif

    return 0;
}

int close_file(HCACHE hc)
{
    close(hc->fd);

    DEBUG_PRINT("Cache closed!\n");

    free(hc);
}

static size_t select_line(HCACHE hc)
{
    size_t min_count;
    size_t min_index;
    int i;

    min_count = hc->clines[0].count;
    min_index = 0;

    for (i = 1; i < NCACHE_LINES; ++i) {
        if (hc->clines[i].count < min_count) {
            min_count = hc->clines[i].count;
            min_index = i;
        }
    }

    return min_index;
}

#ifdef DEBUG

static void print_cline_counts(HCACHE hc)
{
    int i;

    putchar('\n');
    printf("Total count: %llu\n", (unsigned long long)hc->tcount);
    for (i = 0; i < NCACHE_LINES; ++i)
        printf("Cache Line %d --> Block: %d, Count: %llu\n", i, hc->clines[i].blockno, (unsigned long
long)hc->clines[i].count);

    putchar('\n');
}

#endif

/* filecache-test.c */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include "filecache.h"

int create_test_file(const char *path, int nblocks);

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

```

```

{
    HCACHE hc;
    char buf[32 + 1];
    int blockno;

#ifdef CREATE
    if (create_test_file("test.dat", 100) == -1) {
        perror("create_test_file");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
#endif

    if ((hc = open_file("test.dat", O_RDONLY)) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    for (;;) {
        printf("Block No:");
        scanf("%d", &blockno);
        putchar('\n');
        if (blockno == -1)
            break;

        if (read_file(hc, blockno, buf) == -1) {
            perror("read_file");
            exit(EXIT_FAILURE);
        }
        buf[32] = '\0';
        printf("Block content: %s\n\n", buf);
    }

    close_file(hc);

    return 0;
}

int create_test_file(const char *path, int nblocks)
{
    FILE *f;
    int i, k;
    char buf[LINE_SIZE];

    srand(time(NULL));
    if ((f = fopen(path, "wb")) == NULL)
        return -1;

    for (i = 0; i < nblocks; ++i) {
        sprintf(buf, "%04d ", i);
        for (k = 5; k < 32; ++k)
            buf[k] = rand() % 26 + 'A';

        if (fwrite(buf, LINE_SIZE, 1, f) != 1)
            return -1;
    }

    fclose(f);
}

```

Daha önceden de belirttiğimiz gibi read-write önbellek sistemlerinde yazma doğrudan önbelleğe yapılmaktadır. Pekiyi önbelleğe yazılan bilgiler ne zaman yavaş belleğe aktarılacaktır? İşte bu bilgiler birkaç durumda yavaş belleğe aktarılırlar. Örneğin bir cache line önbellekten çıkartılacağı zaman o cache line'in güncellenip güncellenmediğine bakılır. Eğer o

cache line güncellenmişse o cache line önbellekten atılmadan önce yavaş belleğe yazılır. Bunun için programcılar cache line yapılarının içerisinde genellikle bir "dirty flag" tutarlar. Örneğin:

```
typedef struct tagCACHE_LINE {
    char buf[LINE_SIZE];
    int blockno;
    int dirty_flag;
    ....
} CACHE_LINE;
```

Cache line'a bir yazma yapıldığında bu dirty_flag TRUE yapılır. Yavaş bellekten çekilen blok cache line'a yerleştirilirken de bu flag FALSE yapılmaktadır. Eğer kendisine yazma yapılan bir cache line önbellekten hiç atılmamışsa en kötü olasılıkla önbellek sistemi CloseXXX gibi bir fonksiyonla kapatılırken tüm kirlenmiş cache line'lar yavaş belleğe yazılmaktadır. Genellikle önbellek sistemlerinde ismi FlushXXX biçiminde olan fonksiyonlar da bulundurulur. Bu fonksiyonlar çağrıldıkları anda tüm kirlenmiş cache line'ları yavaş belleğe yazarlar.

İşlemcilerin Koruma Mekanizması (Protection mechanisms)

Çok prosesli işletim sistemlerinde çalışmakta olan programlar aynı RAM üzerinde bulunurlar. İşletim sisteminin kendisi de yine RAM'de bulunmaktadır. Bu sistemlerde bir programın bilerek ya da yanlışlıkla başka programların bellek alanlarına erişmesi istenmez. Çünkü oradaki bilgiler değerli olabilir, oradaki bilgilerdeki bozulma o programın, belki de tüm sistemin çökmesine yol açabilir.



Öte yandan bazı makine komutları da sistemin tümünden çökmesine yol açabilmektedir. Örneğin CLI gibi, OUT gibi makine komutları sistem güvenliği bakımından tehlikelidir. İşte modern sistemler bu tür olumsuzluklardan başka proseslerin ve tüm sistemin etkilenmesini engellemek için koruma mekanizmasına sahiptir.

Koruma mekanizmasının iki yönü vardır: Bellek koruması ve komut koruması. Bir programın kendi alanının dışına erişimi engellenmektedir. Buna bellek koruması denir. Benzer biçimde bir programın sistemi çökertecek makine komutlarını kullanması da engellenmektedir. Buna da komut koruması denilmektedir.

İşte modern büyük kapasiteli işlemciler koruma mekanizmasına sahip olarak tasarlanırlar (Örneğin Intel 80X86, ARM modelleri, MIPS, Itanium, PowerPC vs.) Bu sistemlerde bellek koruması ya da komut koruması ihlal edildiğinde bunu birinci elden işlemci tespit eder ve işletim sistemine bildirir. İşletim sistemi de o prosesi cezalandırarak sonlandırır. Örneğin aşağıdaki gibi bir C programını çalıştırmayı deneyiniz:

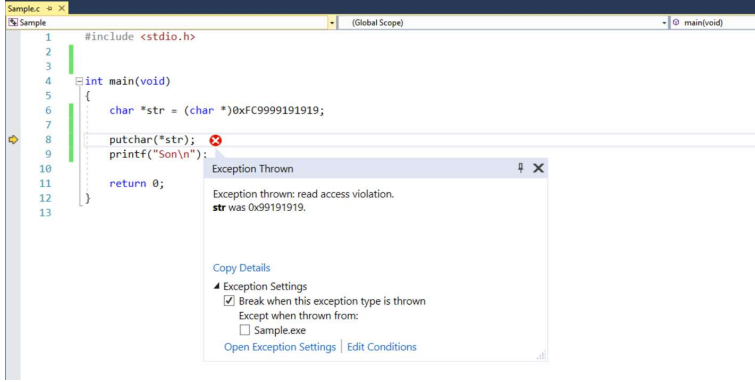
```
#include <stdio.h>

int main(void)
{
    char *str = (char *)0xFC9999191919;

    putchar(*str);
    printf("Son\n");

    return 0;
}
```

Bu programda str göstericisine rastgele bir adres yerleştirilip o adresteki bilgiye erişilmeye çalışılmıştır. İşte bu erişimin yapıldığı noktada işlemci prosesin kendi bellek alanı dışına erişim yaptığını belirleyecek ve bunu işletim sistemine bildirecektir.



```
1 #include <stdio.h>
2
3
4 int main(void)
5 {
6     char *str = (char *)0xFC9999191919;
7
8     putchar(*str);
9     printf("Son\n");
10
11     return 0;
12 }
13
```

Exception Thrown
Exception thrown: read access violation.
str was 0x99191919.

Copy Details
Exception Settings
[x] Break when this exception type is thrown
Except when thrown from:
[] Sample.exe
Open Exception Settings | Edit Conditions

Benzer biçimde örneğin:



```
1 #include <stdio.h>
2
3
4 int main(void)
5 {
6     _asm cli
7
8     return 0;
9 }
10
```

Exception Unhandled
Unhandled exception at 0x011F171E in Sample.exe: 0xC0000096:
Privileged instruction.

Copy Details
Exception Settings

Burada CLI isimli makine komutu sistemin çökmesine yol açabilmektedir. Bu makine komutuyla karşılaşan işlemci durumu işletim sistemine bildirmiş işletim sistemi de prosesi sonlandırmıştır.

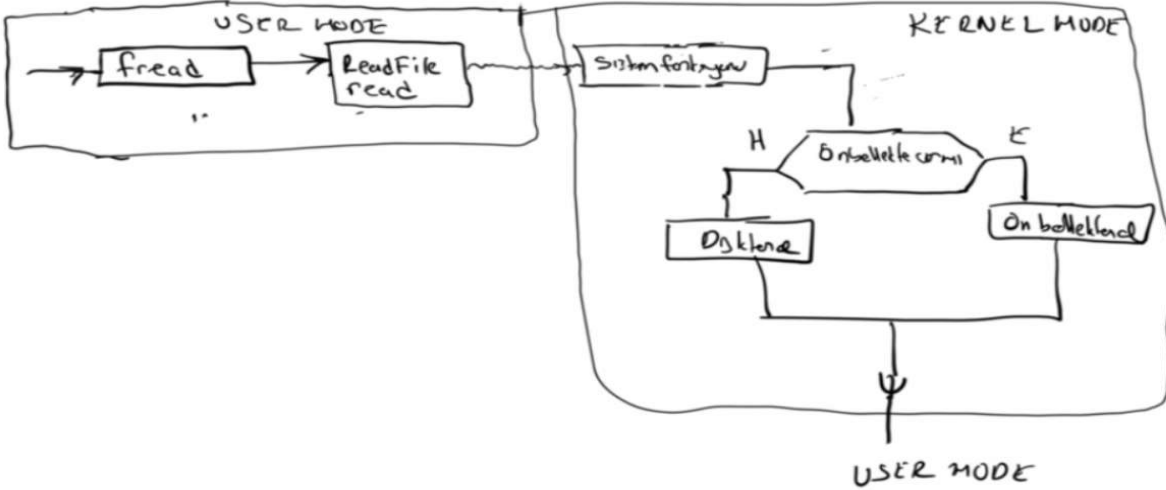
Peki koruma mekanizması her prosese mi uygulanmaktadır? Bu mekanizma tarafından denetlenmeyen kodlar yok mudur? İşte aslında işletim sistemlerinin çekirdek kodları ve aygıt sürücüler gerektiğinde tüm bellek alanına erişip tüm makine komutlarını kullanmaktadır. Aksi takdirde işletim sistemleri ve aygıt sürücüler pek çok işi yapamazlardı. İşte koruma mekanizmasına sahip işlemcilerde çalışmakta olan kodun bir modu vardır: (İntel 4 mod kullanmasına karşın yalnızca iki mod işletim sistemi yazarlar tarafından kullanılmıştır. Diğer işlemcilerin çoğu iki moda sahiptir) "kernel mode" ve "user mode". Kernel kodları, kernel modülleri ve aygıt sürücüler kernel modda çalışırlar. Kernel moddaki kodlar koruma mekanizmasına takılmazlar. Yani bu kodlar belleğin her yerine erişebilirler ve tüm makine komutlarını kullanabilirler. User mod kodlar ise koruma mekanizması tarafından denetlenirler. Bizim ya da başkalarının Windows'ta ve Linux'ta yazmış olduğu normal programların hepsi user modda çalışmaktadır.

Peki madem işletim sisteminin kodları kernel alanı içerisindeki verilere erişiyor ve özel komutları kullanabiliyor, o zaman biz bir sistem fonksiyonunu çağırdığımızda ne olacaktır? Eğer bizim user moddaki akışımız o sistem fonksiyonunu çağırmışsa oradaki kodlar koruma engeline takılmaz mı? İşte bu tür sistemlerde ismine kapı (gate) denilen bir mekanizmayla bu soruna çözüm getirilmiştir. User mod bir proses işletim sisteminin bir sistem fonksiyonunu çağırdığında otomatik olarak kapı mekanizması sayesinde kernel moda geçiş yapar. Böylece işletim sisteminin sistem fonksiyonu kernel modda çalıştırılmış olur. Sistem fonksiyonunun çalışması bittiğinde de proses yeniden user moda döner. Buna prosesin "user moddan kernel moda geçmesi (user mode to kernel mode transition)" denilmektedir. Tabii user moddan kernel moda geçişi sağlayan kapı mekanizması yalnızca kernel mod prosesler tarafından (tipik olarak işletim sistemi tarafından) yerleştirilebilmektedir. Böylece işletim sistemi yalnızca kendi sistem fonksiyonları çalıştırılırken user mod programın kernel moda geçiş yapmasına izin vermektedir. O halde bir proses tüm ömrünü user modda geçirmez. Arada kernel moda da geçebilmektedir.

Prosesin user moddan kernel moda geçmesinin bir zaman maliyeti vardır. Çünkü geçiş sırasında binlerce makine komutu çalışabilmektedir. Örneğin geçiş sırasında user mod stack'teki bilgiler daha korunaklı kernel mod stack'e taşınmaktadır (stack switch). Geçiş işlemi gerçekleştiikten sonra da birtakım işlemlerin yapılması gerekebilir. Benzer biçimde kernel moddan user moda dönüşün de bir maliyeti vardır.

Peki biz kernel modda çalışacak bir program yazamaz mıyız? Evet yazabiliriz. Bu tür programlara "kernel modülleri" ve "aygıt sürücüler" denilmektedir. Kernel modülleri ve aygıt sürücüler kernel alanına yüklenerek sanki kernel'in bir parçasıymış gibi çalışırlar. Kernel modüllerinin ve aygıt sürücülerinin içerisindeki kodlar user mod programlar tarafından çağrılabilir. Bu durumda yine kapı mekanizması yoluyla kernel moda geçilir. Her işletim sisteminin bir aygıt sürücü mimarisi vardır. Aygıt sürücüler o işletim sistemine özgü (hatta o versiyona özgü) bir biçimde yazılırlar. Çünkü aygıt sürücüler yalnızca kernel'daki fonksiyonları kullanırlar.

Şimdi örneğin C'deki fread gibi bir fonksiyonun arka planda aşama aşama nasıl işletildiğine bakalım:



Bu şekilden de görüldüğü gibi fread fonksiyonu Windows'ta ReadFile API fonksiyonunu UNIX/Linux sistemlerinde ise read POSIX fonksiyonunu çağırır. Bu fonksiyonlar da gerçek dosya işlemleri için kernel moda geçerek ilgili sistem fonksiyonlarını çağırırlar. Tabii aslında fread fonksiyonu user modda oluşturulmuş stdio önbelleğine de bakmaktadır. Bu konu izleyen bölümde ele alınmaktadır.

Bir de son olarak olası bir yanlış anlaşılma üzerinde duralım. Biz Windows sistemlerinde bir programı "Run as administrator" biçiminde çalıştırdığımızda ya da UNIX/Linux sistemlerinde "sudo" ile çalıştırdığımızda bu programlar kernel modda çalıştırılmamaktadır. Bu çalıştırma biçimlerinin kernel modla hiçbir ilgisi yoktur. Bu çalıştırma biçimleri yalnızca dosyalara ve birtakım kaynaklara ayrıcalıklı erişim sunar. Yoksa işlemcinin koruma mekanizması bağlamında bir muafiyet sunmamaktadır.

C'nin Standart Dosya Fonksiyonlarının Kullandığı Önbellek Mekanizması

C'nin prototipleri <stdio.h> içerisinde olan standart dosya fonksiyonları işletim sisteminin sistem fonksiyonlarını daha az çağırarak kendi içlerinde bir önbellek sistemi kullanmaktadır. Burada genellikle "önbellek (cache)" terimi yerine "tampon (buffer)" terimi tercih edilmektedir. Bu nedenle C'nin dosya fonksiyonlarına "buffered IO" fonksiyonları da denilmektedir.

Örneğin C'nin fopen fonksiyonu işletim sisteminin sistem fonksiyonlarıyla (Linux'ta open POSIX fonksiyonu sys_open sistem fonksiyonunu çağırır, Windows'ta CreateFile API fonksiyonu da bir sistem fonksiyonunu çağırır) dosyayı açar. Sonra o dosya için bir önbellek (tampon) oluşturur. Böylece okuma yazma işlemlerinde bu önbellek kullanılır. Yani örneğin biz işin başında fgetc fonksiyonu ile dosyadan bir byte okumak istediğimizde, fgetc bir byte değil işletim sisteminin aşağı seviyeli API fonksiyonuyla (Linux'ta read, Windows'ta ReadFile) daha fazla bilgiyi okuyarak önbelleğe çeker. Sonraki okumalarda bize bilgiyi önbellekten verir. Buradaki önbellek sistemi tamamen bizim "user mode" prosesimizin içerisinde yani onun bellek alanında oluşturulmaktadır. Oluşturulan bu önbelleğin (t-ya da

tamponun) kernel ile bir ilgisi yoktur. Standart stdio.h fonksiyonlarının oluşturduğu bu önbellek sistemi read-write bir sistemdir.

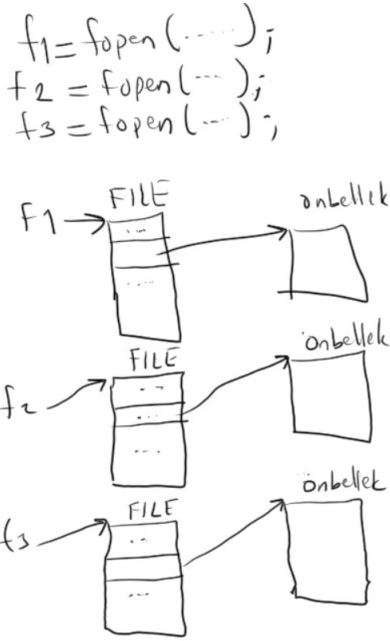
Anahtar Notlar: Standart C fonksiyonları için bu bağlamda önbellek yerine "tampon" sözcüğü kullanılmaktadır. Aslında tampon sözcüğü buradaki durumu iyi yansıtmamaktadır. Çünkü tampon bilgilerin daha sonra işlenmek üzere saklandığı bellek bölgelerini belirtir. Halbuki önbellek hızlandırma amacıyla kullanılan bir sistemi betimlemektedir. Fakat C'de bu kavram hep "tampon (buffer)" biçiminde ifade edildiği için biz de aşağıdaki anlatımlarda bazen "önbellek (cache)" terimini bazen de "tampon (buffer)" terimini kullanacağız.

Tabii her ne kadar biz burada C'nin dosya fonksiyonları üzerinde duruyorsak da aslında diğer dillerin kütüphanelerindeki mekanizmalar da benzerdir. Örneğin C++'ın <iostream> kütüphanesi, .NET'in ve Java'nın dosya işlemlerini yapan stream sınıfları da benzer bir önbellek sistemi kullanmaktadır.

C'nin standart dosya fonksiyonlarının kullandığı önbellek sisteminin şu özellikleri vardır:

- Önbellek read/write biçimindedir.
- Hemen her zaman tek önbellek kullanılır. Yani önbellekte dosyanın tek bir ardışıl bölümü tutulmaktadır.
- Açılmış olan her dosya için ayrı bir önbellek oluşturulur.

Peki açılan her dosyanın önbellek nerede oluşturulmaktadır? İşte fopen fonksiyonu dosyayı açtığında önbelleği de oluşturur. Önbellek bilgilerini (örneğin önbellek alanının adresini, uzunluğunu vs.) ise FILE yapısının içerisinde tutar. Yani fopen fonksiyonunun bize verdiği FILE yapısının içerisinde önbellek bilgileri de bulunmaktadır. Bu durumu şekilsel olarak aşağıdaki gibi özetleyebiliriz:



Standart dosya fonksiyonlarının kullandığı önbellek üç moda çalışabilmektedir: Tam tamponlamalı mod (full buffered mode), satır tamponlamalı mod (line buffered mode) ve sıfır tamponlamalı mod (no buffered mode). Aşağıdaki anlatımlarda daha çok önbellek yerine tampon terimini kullanacağız.

Tam tamponlamalı modda tampon tam kapasiteyle kullanılır. Yani okuma ve yazma tampondan yapılır. Yazılan bilgiler tampona yazılır. Tampon dolunca ya da tampona dosyanın başka bir kısmı çekileceği zaman tampondaki bilgi -eğer güncellenmişse- dosyaya yazılmaktadır. Aynı zamanda fflush fonksiyonu da tampondaki bilgileri o anda dosyaya aktarmak için kullanılır. Şüphesiz dosya fclose ile kapatıldığında da fflush işlemi yapılmaktadır.

Satır tamponlamalı modda, tamponda yalnızca bir satırlık bilgi (yani '\n' görülüne kadarki ('\n' de dahil) bilgi tamponda tutulur. Benzer biçimde tampona '\n' karakteri yazıldığında ya da fflush yapıldığında dosyaya aktarılmaktadır.

Sıfır tamponlamalı modda tampon devre dışı kalmaktadır. Yani her yazma ve okuma işleminde doğrudan işletim sisteminin API fonksiyonları dolayısıyla da sistem fonksiyonları çağrılır ve aktarım tampon kullanılmadan hep doğrudan yapılır.

Peki bir dosya açıldığında onun tamponlama modu default durumda nedir? İşte C standartlarında stdin, stdout ve stderr dosyalarının default tamponlama modları hakkında bazı şeyler söylenmiştir. (Bu konu ileride ele alınacak). Fakat normal dosyaların default tamponlama modlarının ne olacağı konusunda birşey söylenmemiştir. Bu durum normal dosyaların default durumda herhangi bir tamponlama stratejisine sahip olabileceğini belirtir. Ancak tabii sistemlerin hepsinde normal dosyalar default durumda "tam tamponlamalı mod"da açılmaktadır.

Bir dosyanın tamponlama modu setbuf ve setvbuf isimli standart C fonksiyonlarıyla değiştirilebilmektedir. Aslında setvbuf fonksiyonu zaten setbuf fonksiyonunu işlevsel olarak kapsar. setbuf yetersiz olduğu için setvbuf fonksiyonu da standartlara dâhil edilmiştir. Ancak dosyanın tamponlama modu değiştirilecekse bu işlemin fopen fonksiyonundan hemen sonra (yani o dosya için başka bir dosya fonksiyonu henüz çağrılmadan) yapılması gerekir. Eğer bu kurala uyulmazsa tanımsız davranış (undefined behavior) söz konusu olur.

setbuf fonksiyonunun prototipi şöyledir:

```
#include <stdio.h>

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

Fonksiyonun birinci parametresi tamponlama ile ilgili işlem yapılacak dosya bilgi göstericisini, ikinci parametresi ise yeni tamponun adresini belirtir. İkinci parametre NULL geçilirse sıfır tamponlama söz konusu olur. Buradaki yeni tampon <stdio.h> içerisindeki BUFSIZ isimli sembolik sabitinin belirttiği uzunlukta olmalıdır. Yani setbuf fonksiyonu ile biz dosyanın kullandığı tamponun yerini değiştirebiliriz. Ayrıca istersek dosyayı sıfır tamponlamalı moda da çekebiliriz. Ancak bu fonksiyonla tamponun boyutunu değiştiremeyiz ve dosyayı satır tamponlamalı moda geçiremeyiz.

Örneğin:

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

int main(void)
{
    FILE *f;
    char buf[BUFSIZ];
    char ch;

    if ((f = fopen("test.txt", "r+")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }
    setbuf(f, buf);    /* artık tampon olarak buf kullanılacak */
    ch = fgetc(f);
    putchar(ch);

    fclose(f);

    return 0;
}
```

setvbuf fonksiyonunun parametrik yapısı da şöyledir:

```
#include <stdio.h>

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

Fonksiyonun birinci parametresi tamponlama stratejisi değiştirilecek dosyanın dosya bilgi göstericisini belirtir. İkinci parametre dosyanın yeni tamponunu belirtmektedir. Bu parametre NULL geçilebilir. Bu durum tamponun fonksiyonun kendisi tarafından tahsis edileceği anlamına gelir. Üçüncü parametre yeni tamponlama modunun ne olacağını belirtmektedir. Bu parametre aşağıdaki sembolik sabitlerden biri olarak girilmelidir:

```
_IOFBF (tam tamponlama için)
_IOLBF (satır tamponlaması için)
_IONBF (sıfır tamponlama için)
```

Son parametre yeni tamponun uzunluğunu belirtmektedir. Tabii fonksiyonun üçüncü parametresi _IONBF olarak girilirse ikinci ve dördüncü parametrenin ne girildiğinin bir önemi yoktur. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda sıfır dışı bir değerle geri döner. Örneğin:

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

int main(void)
{
    FILE *f;
    char buf[BUFSIZ];

    if ((f = fopen("test.txt", "r+")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }
    if (setvbuf(f, buf, _IOLBF, BUFSIZ) != 0) {
        fprintf(stderr, "setvbuf failed!..\n");
        exit(EXIT_FAILURE);
    }

    /* ...*/

    fclose(f);

    return 0;
}
```

Dosyadan Byte Byte Okumalarda fgetc ve getc Fonksiyonları

Bir dosyadan byte byte okuma yapmak istediğimizi düşünelim. Bunun için fgetc ve getc fonksiyonlarının kullanıldığını biliyorsunuz. Pekiyi her ne kadar bir önbellek sistemi kullanılıyor olsa da byte byte okuma sırasında her defasında bir fonksiyonun çağrılmış olması zaman kaybına yol açmaz mı?

```
while ((ch = fgetc(f)) != EOF) {
    ...
}
```

İşte bu biçimde dosyadan byte byte okuma yapılacağına fgetc yerine getc fonksiyonu tercih edilmelidir. C standartlarına göre fgetc ile getc arasındaki tek fark getc'in bir makro olarak yazılabileceğidir. Gerçekten de genellikle getc bir makro olarak yazılmaktadır. getc makrosu bir fonksiyon çağrısına yol açmadığı için ve doğrudan tampondan bilgiyi aldığı için daha hızlı olma eğilimindedir.

fileno ve fdopen Fonksiyonları

UNIX/Linux sistemlerinde open POSIX fonksiyonuyla açtığımız bir dosyayı tamponlama mekanizması ile kullanmak isteyebiliriz. İşte bu sistemlerde open fonksiyonundan elde ettiğimiz dosya betimleyicisini alarak bize dosya bilgi göstericisini (stream) veren fdopen isimli bir POSIX fonksiyonu bulunmaktadır. Fonksiyonun prototipi şöyledir:

```
FILE *fdopen(int fd, const char *mode);
```


Fonksiyonun birinci parametresi open fonksiyonundan elde edilmiş olan dosya betimleyicisini, ikinci parametresi dosyanın açış modunu belirtmektedir. Şüphesiz buradaki mod open fonksiyonuyla açılmış olan dosyanın moduna uygun olmalıdır. Fonksiyon başarı durumunda dosya bilgi göstericisine, başarısızlık durumunda NULL adrese geri dönmektedir. Bu dosya bilgi gösterici kullanılarak dosya fclose fonksiyonuyla kapatıldığında dosyanın close POSIX fonksiyonuyla kapatılacağına dikkat ediniz. Örneğin:

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fd;
    ssize_t result;
    FILE *f;
    char buf[10 + 1];
    int ch;

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

    if ((result = read(fd, buf, 10)) == -1)
        exit_sys("read");

    buf[result] = '\0';
    printf("read ile okunan byte'lar: %s\n", buf);

    if ((f = fdopen(fd, "r")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    while ((ch = fgetc(f)) != EOF)
        putchar(ch);

    if (ferror(f)) {
        fprintf(stderr, "cannot read file!..\n");
        exit(EXIT_FAILURE);
    }

    fclose(f);

    return 0;
}

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

    exit(EXIT_FAILURE);
}
```

fdopen fonksiyonu Windows sistemlerinde de _fdopen ismiyle bulunmaktadır. Fonksiyonun prototipi şöyledir:

```
#include <stdio.h>

FILE *_fdopen(int fd, const char *mode);
```

Windows sistemlerinde CreateFile fonksiyonundan elde edilen handle değerinin void * anlamına gelen HANDLE türünden olduğunu anımsayınız. Dolayısıyla bu HANDLE değerini _fdopen fonksiyonuna tür dönüştürmesi yaparak vermelisiniz.

fdopen fonksiyonun mantıksal olarak ters işlemini ise fileno isimli POSIX fonksiyonu yapmaktadır. Yani biz fopen fonksiyonuyla dosyayı açtıktan sonra FILE yapısı içerisinde dosya betimleyicisini almak isteyebiliriz. fileno fonksiyonun prototipi şöyledir:

```
#include <stdio.h>

int fileno(FILE *stream);
```

Fonksiyon parametre olarak bizden dosya bilgi göstericisini alır geri dönüş değeri olarak dosya betimleyicisini verir. Aynı fonksiyon _fileno ismi ile Windows sistemlerinde de bulunmaktadır:

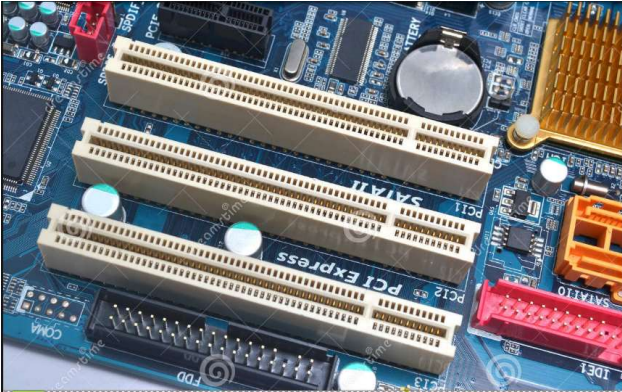
```
#include <stdio.h>

int _fileno(FILE *stream);
```

Windows sistemlerinde her ne kadar fonksiyonun geri dönüş değeri int türündense de aslında dosyanın HANDLE değerini belirtmektedir.

Aygıt Sürücü (Device Driver) Nedir ve Nasıl Kullanılır?

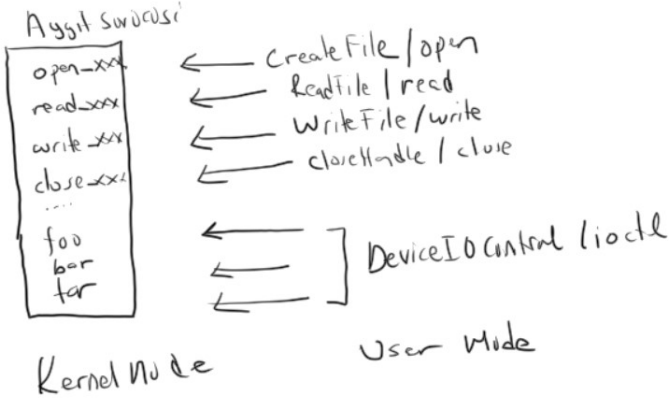
Bilindiği gibi masaüstü bilgisayarlar genişleme yuvalarına kartlar takılarak donanımsal olarak genişletilebilmektedir. Örneğin bugün PC'lerde kullandığımız kart yuvalarına PCI denilmektedir. Biz de bir donanım birimini bu PCI soketleri destekleyen bir kart halinde oluşturup bilgisayarımıza takabiliriz.



Donanıma bu biçimde eklemeler yapıldığında eklenen kartların üzerindeki işlemcilerin programlanması gerekmektedir. Genellikle merkezi işlemcilerde (CPU) bu tür yardımcı işlemcileri programlamak için IN, OUT tarzı IO komutları bulunmaktadır. Ancak bu komutlar yalnızca kernel modda kullanılabilir. İşte bu tür aygıtların programlanabilmesi için kernel modda çalışacak programlara gereksinim duyulur. Bunlara da genel olarak "aygıt sürücüler (device drivers)" denilmektedir. Aygıt sürücülerin aslında doğrudan bir donanım aygıtını programlaması da gerekmez. Her ne kadar bu terimin ortaya çıkış noktası buysa da aygıt sürücülerini kernel modda çalışan ve işletim sisteminin kernel'ına eklenen kod parçaları olarak düşünebiliriz. Bazı sistemlerde (örneğin Linux) IO işlemi yapmayan aygıt sürücülerine "kernel modülleri (kernel modules)" de denilmektedir. Yani bu sistemlerde yalnızca IO işlemi yapan kernel modüllerine aygıt sürücüsü denir. Biz nasıl bilgisayarımıza kart takıp onu donanımın bir parçası haline getiriyorsak kernel'a da aygıt sürücüsü ekleyip onu kernel'ın bir parçası haline getiririz.

Mikrokernel denilen kernel mimarisinde kernel kod olarak çok küçük tutulur. Bu tür mimarilerde aygıt sürücüler de kernel modda değil user modda çalışmaktadır. Ancak Windows ve Linux gibi işletim sistemleri mikrokernel mimarisine sahip değildir. Dolayısıyla bu sistemlerde aygıt sürücüler genel olarak kernel modda çalışmaktadır.

Genel olarak işletim sistemlerinde aygıt sürücüler sanki birer dosyaymış gibi kullanılmaktadır. Yani bir aygıt sürücüsü Windows'ta dosya açan CreateFile API fonksiyonuyla, UNIX/Linux sistemlerinde open POSIX fonksiyonuyla açılır. Aygıt sürücülerinden sanki bir dosyaymış gibi okuma yazma işlemi yapılabilir. Ve en sonunda aygıt sürücülerini yine bir dosyaymış gibi kapatılırlar. Bir aygıt sürücüsünden okuma yaptığımızda o aygıt sürücüsünün içerisindeki bizim tarafımızdan yazılmış olan okuma fonksiyonu çağrılır, yazma yaptığımızda da yine aygıt sürücüsünün içerisindeki bizim tarafımızdan yazılmış olan yazma fonksiyonu çağrılmaktadır. Ayrıca biz aygıt sürücüler okuma yazma fonksiyonları dışında istenildiği kadar çok fonksiyon da barındırabilmektedir. Biz aygıt sürücüsü içerisinde bulunan bir fonksiyonu kernel moda geçerek çalıştırabiliriz. Bu işlem Windows'ta DeviceIOControl, UNIX/Linux sistemlerinde ioctl isimli fonksiyonla yapılmaktadır.

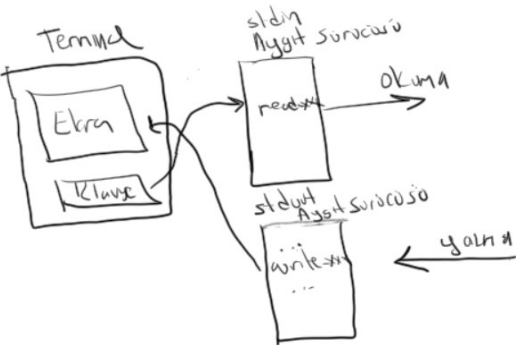


Şüphesiz bilgisayar sistemlerinde en sık karşılaştığımız aygıt sürücülerini terminal sürücüleridir. Ekran ve klavyeye terminal denilmektedir. Bu aygıtlar dosya gibi kullanılan aygıt sürücülerle yönetilirler. İşte C'de stdin dosyası (standard input) standart girdi için kullanılan (tipik olarak klavye), stdout (standard output) dosyası da çıktı için kullanılan (tipik olarak ekran) aygıt sürücülerini temsil etmektedir. Yani örneğin biz stdout dosyasına dosya fonksiyonlarıyla yazma yaptığımızda bu yazılacak bilgiler aygıt sürücüsüne gider. Bu aygıt sürücüsü de bizim yazmak istediklerimiz şeyleri ekran denilen donanım aygıtına aktarır.

Peki aygıt sürücülerini nasıl yazılmaktadır? İşte her işletim sisteminin bir aygıt sürücü mimarisi vardır. Aygıt sürücüler işletim sistemine özgü hatta onların versiyonlarına özgü biçimde yazılmaktadır. Derneğimizde aygıt sürücülerinin yazımları "Windows Sistem Programlama" ve "UNIX/Linux Sistem Programlama" kurslarında ayrıntılı olarak, "Sistem Programlama ve İleri C Uygulamaları – II" kursunda ise da temel düzeyde ele alınmaktadır.

stdin, stdout ve stderr Dosyaları

C'de stdin, stdout ve stderr dosyaları program başladığında açıldığı kabul edilen aygıt sürücü dosyalarıdır. stdin ve stdout terminal aygıt sürücüsüne yönlendirilmiştir. stdin dosyasının "read-only" moda stdout dosyasının ise "write-only" moda açıldığı kabul edilir. stdin dosyasından okuma yaptığımızda terminal aygıt sürücüsü klavyeden alınanları bize verir. stdout dosyasına yazma yapmak istediğimizde de terminal aygıt sürücüsü yazdırılmak istenen bilgileri ekrana yazdırılmaktadır.



stderr programın error mesajlarının yazdırılacağı hedefi belirtmektedir. Default olarak sistemlerde stderr dosyası da "write-only" açılmıştır ve terminal aygıt sürücüsüne yönlendirilmiştir. stdin, stdout ve stderr dosyaları programcı tarafından açılmazlar. Zaten C programları bu dosyalar açık olarak başlatılmaktadır. Yani bunların açılması derleyicilerin başlangıç kodlarında (startup code) yapılmaktadır. Benzer biçimde bu dosyalar da yine program sonlandığında exit fonksiyonu tarafından kapatılmaktadır.

Örneğin biz C'de stdout dosyasına aşağıdaki gibi fprintf fonksiyonuyla birşey yazmak isteyelim:

```
fprintf(stdout, "this is a test\n");
```

ileride ele alınacağı üzere bu bilgi önce diğer dosyalar gibi kütüphanenin oluşturduğu önbelleğe (tampona) yazılır. Buradan hedefe (yani aygıt sürücüye) işletim sisteminin sistem fonksiyonlarıyla (Windows'ta WriteFile, UNIX/Linux sistemlerinde write) aktarılmaktadır.

C'de stdin, stdout ve stderr FILE * türünden birer dosya bilgi gösterici belirtmektedir. Yani biz C'nin standart dosya fonksiyonlarıyla bu stdin, stdout ve stderr dosyalarını kullanabiliriz.

C'nin printf, scanf, getchar, putchar gibi fonksiyonları da aslında gizli birer dosya fonksiyonudur. printf fonksiyonu aslında fprintf fonksiyonunun default olarak stdout dosyasına yazan biçimidir. Benzer biçimde getchar aslında fgetc fonksiyonunun stdin dosyasından okuma yapan biçimidir. Başka bir deyişle:

```
printf(args);
```

çağrısı ile:

```
fprintf(stdout, args);
```

çağrısı eşdeğerdir. Benzer biçimde:

```
ch = fgetc(f);
```

çağrısı ile:

```
ch = getchar();
```

çağrısı da eşdeğerdir.

Dosya Yönlendirmeleri (IO Redirection)

Dosyaların hedefleri yönlendirilebilmektedir. Yani örneğin açılmış bir dosyanın hedefi "x.txt" olsun. Bu hedef "y.txt" gibi bir dosyaya yönlendirilebilir. Bu durumda ilgili dosyaya yazma yapıldığında aslında yazılanlar "x.txt" dosyasına değil "y.txt" dosyasına aktarılacaktır. Yönlendirme okuma amaçlı ya da yazma amaçlı olarak yapılabilmektedir.

Uygulamada stdout, stdin ve stderr dosyalarının yönlendirilmesi ile çok sık karşılaşılmaktadır. Bu sayede örneğin bir programın ekrana yazdıkları (stdout dosyasına yazdıkları) bir dosyaya yönlendirilerek programın ekran yerine bir dosyaya yazma yapması sağlanabilmektedir. Benzer biçimde stdin dosyası da başka bir dosyaya yönlendirilebilir. Bu durumda getchar gibi scanf gibi fonksiyonlar klavye yerine o dosyadan okuma yaparlar.

Hem Windows'ta hem UNIX/Linux ve Mac OS X sistemlerinde komut satırında programı çalıştırırken '>' işareti stdout dosyasının, '<' işareti de stdin dosyasının yönlendirileceği anlamına gelir. Örneğin:

```
./sample > x.txt
```

Burada UNIX/Linux sistemlerinde sample programı çalıştırılacak, programın stdout dosyası "x.txt" dosyasına yönlendirilecektir. Bu durumda programda ekrana yazılan her şey dosyaya yazılmış olacaktır. '>' işareti ile

yönlendirmede hedef dosya zaten varsa sıfırlanıp yeniden açılmaktadır. Benzer biçimde stdin dosyası da şöyle yönlendirilebilir:

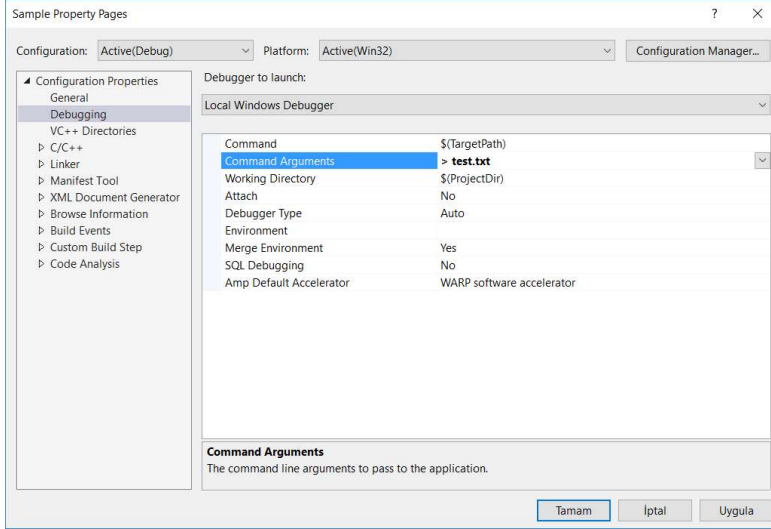
```
./sample < x.txt
```

Bireden fazla yönlendirme beraber de yapılabilir. Örneğin:

```
./sample > x.txt < y.txt
```

Burada sample programının stdout dosyası "x.txt" dosyasına stdin dosyası da "y.txt"ye yönlendirilmektedir.

Anahtar Notlar: IDE'lerde genel olarak yönlendirme işlemi IDE'lerin menüleriyle de yapılabilmektedir. Örneğin Visual Studio'da proje ayarlarında "Debugging" kısmında "Command Arguments" seçeneğinde yönlendirme ifadelerini girebiliriz. Örneğin:



Şüphesiz programın aynı zamanda komut satırı argümanları da girilebilir:

```
./sample ali veli selami > x.txt < y.txt
```

Burada "ali", "veli", "selami" programın komut satırı argümanlarıdır.

Kabuk üzerinde '>' ile yönlendirme yapılırken hedef dosya varsa bile içeriği sıfırlanmaktadır. İşte komut satırında ayrıca '>>' karakterleri ile de yönlendirme yapılabilmektedir. Bu durumda yönlendirmenin yapıldığı dosya varsa bile sıfırlanmaz, yazılanlar onun sonuna eklenir. Örneğin:

```
./sample >> test.txt
```

stderr Dosyasının Anlamı

stderr dosyası programın hata mesajlarının yazdırılması için kullanılan dosyadır. Fakat önceden de belirtildiği gibi stderr dosyası da default durumda pek çok sistemde terminale yönlendirilmiştir. Ancak istenirse yönlendirme sayesinde bunlar birbirlerinden ayrılabilir. Windows ve UNIX/Linux sistemlerinde '2>' sembolü stderr dosyasının yönlendirileceği anlamına gelir. Yani:

```
./sample
```

Böyle bir çalışmada hem programın normal mesajları hem de hata mesajları ekranda görünecektir. Fakat:

```
./sample 2> x.txt
```

Burada yalnızca programın normal mesajları ekranda görünecek, hata mesajları "x.txt" dosyasına yazdırılacaktır. UNIX/Linux sistemlerinde /dev/null isimli dosya bir aygıt sürücü dosyasıdır. Bu aygıt sürücünün yazma fonksiyonunun içi boştur. Böylece biz istersek aşağıdaki gibi programın hata mesajlarının kafa karıştırmalarını önleyebiliriz:

```
./sample 2> /dev/null
```

Bu durumda bizim programın hata mesajlarını stderr dosyasına yazmamız iyi bir tekniktir. Çünkü bu durumda programı çalıştıracak kişiler programın hata mesajlarını yönlendirme şansına sahip olacaklardır.

UNIX/Linux sistemlerindeki bash kabuğunda &> ile hem stdout hem de stderr dosyaları birlikte yönlendirilebilmektedir. Örneğin:

```
./sample &> test.txt
```

Burada sample programının stdout ve stderr dosyalarına yazdıkları test.txt dosyasına aktarılacaktır.

C'de Yönlendirme İşlemlerinin Standard C Fonksiyonları Yoluyla Yapılması

Biz yönlendirme işlemini C'de programlama yoluyla yapabiliriz. Bunun için freopen fonksiyonu kullanılmaktadır.

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

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Fonksiyonun birinci parametresi yönlendirmenin yapılacağı hedef dosyayı belirtir. İkinci parametre bu dosyanın açış modunu belirtmektedir. Son parametre de yönlendirmenin kaynağını belirten dosya bilgi göstericisidir. Fonksiyon başarı durumunda birinci parametreyle belirtilmiş olan hedef dosyanın bilgi göstericisine geri döner. Başarısızlık durumunda NULL adrese geri dönmektedir. Fonksiyon aynı zamanda üçüncü parametresiyle belirtilmiş olan dosyayı da kapatmaktadır. Örneğin:

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

```
int main(void)
{
    int i;
    FILE *f;

    if ((f = freopen("test.txt", "w+", stdout)) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        printf("%d\n", i);

    fclose(f);

    return 0;
}
```

Bu biçimde yapılan yönlendirme yeniden geri alınmaz. Dolayısıyla program sonlanana kadar yönlendirme etkisi devam edecektir. Şimdi de freopen ile stdin dosyasını yönlendirelim:

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

```
int main(void)
{
    FILE *f;
```

```

int i;
int val;

if ((f = freopen("test.txt", "r", stdin)) == NULL) {
    fprintf(stderr, "Cannot open file!..\n");
    exit(EXIT_FAILURE);
}

while (scanf("%d", &val) == 1)
    printf("%d\n", val);

fclose(f);

return 0;
}

```

stdin Dosyalarında EOF Etkisinin Yaratılması

Yukarıda da açıklandığı gibi aslında biz stdin dosyasından okuma yapmak istediğimizde bu dosya bir aygıt sürücüyü yönlendirildiği için aslında biz terminal aygıt sürücüsünden okuma yaparız. Terminal aygıt sürücüsü de okuma sırasında klavyeden alınanları bize vermektedir. Pekiyi klavye gerçek bir dosya olmadığına göre EOF etkisi nasıl oluşmaktadır? Yani örneğin aşağıdaki döngüde biz döngüden nasıl çıkarız?

```

while ((ch = fgetc(stdin)) != EOF) {
    ...
}

```

İşte stdin dosyasında EOF etkisi yaratmak için özel tuş kombinasyonları kullanılmaktadır. UNIX/Linux sistemlerinde Ctrl + D, Windows sistemlerinde Ctrl + Z + ENTER tuşları EOF etkisi yaratır. Tabii bu tuş kombinasyonları aslında aygıt sürücüyü kapatmaz. Sadece EOF etkisi yaratmaktadır. Biz birden fazla kez bu tuş kombinasyonlarına basabiliriz. (Başka bir deyişle bu tuş kombinasyonlarına bastıktan sonra stdin'den okuma yapmaya devam edebiliriz.)

Komut Satırında Boru (Pipe) İşlemleri

Boru aslında bir prosesler arası haberleşme yöntemidir. Bu konu ileride ele alınacaktır. Biz bu bölümde yalnızca komut satırında boru işlemi ile neyin yapılmaya çalışıldığını ele alacağız.

Windows ve UNIX/Linux komut satırında '|' sembolü boru işlemi anlamına gelir. '|' karakterinin solunda ve sağında çalıştırılabilen dosyaların yol ifadeleri bulunur. Örneğin:

```
a | b
```

Burada a ve b birer programdır. Shell bir boru oluşturduktan sonra a ve b programlarını çalıştırır. Ancak a'nın stdout dosyasını ve b'nin de stdin dosyasını bu boruya yönlendirir. İşte bu yönlendirmeden sonra artık a programının stdout dosyasına yazdıkları b programının stdin dosyasından okunacaktır. Yani sani a programın ekrana yazdıkları b tarafından klavyeden giriliyormuş etkisi yaratacaktır. Örneğin:

```
csd@csd-vm:~/Study/SysProg-2019$ ls -l | wc
10      83     417
```

Burada wc (word count) bir POSIX shell komutudur (bu sistemlerde neredeyse her komut bir programdır). wc bir dosyayı komut satırı argümanı olarak alır. O dosya içerisinde kaç satır, kaç sözcük ve kaç byte olduğunu stdout dosyasına yazdırır. Eğer wc'de dosya ismi verilmezse wc stdin dosyasından okuma yapmaktadır. Böylece yukarıdaki örnekte ls -l'nin ekrana yazdıklarını wc sanki stdin dosyasından okuyormuş gibi bir durum oluşur. Eğer böyle bir meknizma olmasaydı biz bunu aşağıdaki gibi üç aşamada yapmak zorunda kalırdık:

```
ls -l > temp.txt
wc temp.txt
rm temp.txt
```


Genel olarak UNIX/Linux sistemlerinde programlar komut satırı argümanı almamışlarsa okumayı hep stdin dosyasından yaparlar. Bu onların borularla kullanılmasını mümkün hale getirmek için yapılmıştır. Örneğin:

```
ps -e | grep "tty"
```

Burada ps -e prses listesini (yani çalışmakta olan programları) satır stair stdout dosyasına (ekrana) yazdırmaktadır. grep ise belli bir kalıbı (regex kalıbını) dosya içerisinde bulan ve bulunduğu satırın tamamını yazdıran standart bir POSIX komutudur. Böylece yukarıdaki örnekte proses listesinde "tty" geçen satırlar ekrana yazdırılmak istenmiştir. Örneğin:

```
csd@csd-vm:~/Study/SysProg-2019$ ps -e | grep "tty"
991 tty7      00:00:20 Xorg
999 tty1      00:00:00 agetty
```

Biz de komut satırından bir dosya alan programlarımızı boruyla kullanmak istiyorsak komut satırı argümanı verilmediğinde stdin dosyasından okuyacak biçimde yazmalıyız. Örneğin:

```
/* lcount.c */

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

int main(int argc, char *argv[])
{
    FILE *f;
    int ch;
    int count = 0;

    if (argc == 1)
        f = stdin;
    else
        if ((f = fopen(argv[1], "r")) == NULL) {
            fprintf(stderr, "cannot open file!..\n");
            exit(EXIT_FAILURE);
        }

    while ((ch = fgetc(f)) != EOF)
        if (ch == '\n')
            ++count;

    printf("%d\n", count);

    return 0;
}
```

Burada lcount.c programı hiçbir komut satırı argümanı girilmemişse okumayı stdin dosyasından yapmaktadır. Şimdi örneğin UNIX/Linux sistemlerinde biz bunu aşağıdaki gibi kullanabiliriz:

```
ps -e | ./lcount
```

Borular kombine edilebilir. Yani aşağıdaki işlem geçerlidir:

```
a | b | c
```

Burada a'nın stdout dosyasına yazdığını b stdin'den okur, b'nin stdout dosyasına yazdıklarını ise c stdin'den okur. Benzer biçimde UNIX/Linux sistemlerindeki cat programı da bazı ayrıntıları ihmal edilerek şöyle yazılabilir:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *f;
```



```

int i;
int ch;

i = 1;
if (argc == 1) {
    f = stdin;
    goto ONLY_STDIN;
}

for (; i < argc; ++i) {
    if ((f = fopen(argv[i], "r")) == NULL) {
        fprintf(stderr, "file not found: %s\n", argv[i]);
        continue;
    }
ONLY_STDIN:
    while ((ch = fgetc(f)) != EOF)
        putchar(ch);
    if (ferror(f)) {
        fprintf(stderr, "could not read file: %s\n", argv[i]);
        fclose(f);
        continue;
    }
    fclose(f);
}

return 0;
}

```

C'de stdin, stdout ve stderr Dosyalarının Tamponlaması

stdin, stdout ve stderr dosyaları da normal diğer stdio dosyaları gibi tamponlama mekanizmasına dahildir. Örneğin biz printf fonksiyonuyla stdout dosyasına yazma yapmak istediğimizde printf bunu önce tampona yazabilir. Tampon dolunca aygıt sürücüyü aktarabilir. Benzer biçimde biz getch ile bir karakter bile okuyacak olsak fonksiyon aygıt sürücüdün birden fazla karakteri okuyup bir tampona yerleştirir ve bize o tampondakiler bitene kadar oradan verir. Pekiyi bu standart dosyaların default tamponlama modu nasıldır? Standartlar bu konuda şunları söylemektedir: Eğer stdin ve stdout karşılıklı etkileşimli (interaktif) bir aygıtı yönlendirilmişse (terminal karşılıklı etkileşimli bir aygıttır) kesinlikle tam tamponlamalı olamaz. Fakat satır ya da sıfır tamponlamalı olabilir. Eğer stdout ve stdin karşılıklı etkileşimli bir aygıtı yönlendirilmemişse kesinlikle tam tamponlamalı moddadır. stderr ise başlangıçta hiçbir durumda tam tamponlamalı modda olamaz. Fakat satır ya da sıfır tamponlamalı modda olabilir. Tabii bu başlangıçtaki default durumdur. Yoksa daha sonra bu dosyaların tamponlama modları değiştirilebilir (tabii henüz bir işlem yapılmadan). Bu anlatımdan şu sonuçlar çıkar:

1) stdin ve stdout terminale yönlendirilmişse (tabii bu normal durumdur) başlangıçta default olarak tam tamponlamalı modda olamaz. Fakat sıfır ya da satır tamponlamalı modda olabilir. Bu tamamen derleyiciyi yazanların isteğine kalmıştır. Örneğin Linux libc kütüphanesinde stdout dosyasının default tamponlama modu satır tamponlamalıdır fakat Windows'taki Microsoft derleyicilerinde sıfır tamponlamalıdır. Örneğin aşağıdaki kodu çalıştıracak olalım:

```

#include <stdio.h>

int main(void)
{
    printf("ankara");
    for (;;)
        ;

    return 0;
}

```

Burada Windows'ta "ankara" yazısı görülecektir fakat Linux'ta görülmeyecektir. Tabii program aşağıdaki gibi olsaydı her iki sistemde de yazı görülecekti.

```
#include <stdio.h>

int main(void)
{
    printf("ankara\n");
    for (;;)
        ;

    return 0;
}
```

Tabii bu durumda imleç aşağıdaki satırın başına da geçecektir. Fakat bu istenmiyorsa yazının gözükmesini garanti etmek için fflush fonksiyonu da kullanılabilir:

```
#include <stdio.h>

int main(void)
{
    printf("ankara");
    fflush(stdout);

    for (;;)
        ;

    return 0;
}
```

Ya da stdout dosyasının tamponlama modunu sıfır tamponlamalı moda çekebiliriz:

```
#include <stdio.h>

int main(void)
{
    setvbuf(stdout, NULL, _IONBF, 0);

    printf("ankara");

    for (;;)
        ;

    return 0;
}
```

2) Biz stdin ve stdout dosyalarını normal disk dosyasına yönlendirirsek kesinlikle tam tamponlamalı mod kullanılır. Örneğin programı aşağıdaki çalıştırmış olalım:

```
./sample > x.txt
```

Burada '\n' karakteri görülünce değil tampon dolunca dosyaya yazma yapılacaktır.

3) stderr dosyası ister karşılıklı etkileşimli aygıtı yönlendirilmiş olsun isterse olmasın hiçbir zaman işin başında tam tamponlamalı modda olamaz. Yani biz stderr dosyasına yazma yaparken yazının sonuna '\n' karakterini eklersek her zaman yazı aygıtı transfer edilecektir.

Pekiyi Windows ve Linux'ta stdin dosyasının default tamponlama modu nasıldır? Sistemlerin hemen hepsinde eğer stdin terminale yönlendirilmişse (yani klavyeyi temsil ediyorsa) bu dosyanın default tamponlaması satır tamponlamalı moddur. Yani biz getchar gibi bir fonksiyonla stdin'den bir karakter okumak istediğimizde getchar aygıt sürücüsünden bir satır talep eder. Biz de ENTER tuşuna basana kadar pek çok karakter girebiliriz. Bastığımız ENTER '\n' anlamına gelir. Bu '\n' karakteriyle birlikte klavyeden girilen tüm karakterler stdin dosyasının tamponuna çekilir. getchar bize onun ilkini verir. Sonra getchar çağırılmaya devam edersek fonksiyon bize tampondakileri sırasıyla verecektir. En son getchar bize

tampondaki '\n' karakterini verir. Artık tampon boşalmıştır. Bir daha getchar çağırırsak o yine tamponu bir satırla doldurmak ister. Örneğin aşağıdaki programda yalnızca 'a' tuşuna ve sonra ENTER tuşuna basmış olalım. Ne olacaktır?

```
#include <stdio.h>

int main(void)
{
    int ch1, ch2;

    ch1 = getchar();
    ch2 = getchar();

    printf("ch1 = %d, ch2 = %d\n", ch1, ch2);

    return 0;
}
```

Burada tamponda "a\n" karakterleri bulunur. Birinci getchar tampondan 'a' yı alır. İkinci getchar tamponda bilgi olduğundan klavyeden giriş beklemez. O da '\n' yi alır. Bir daha getchar kullansaydık yeniden giriş istenecekti. Pekiye aşağıdaki örnekte program nasıl çalışır?

```
#include <stdio.h>

int main(void)
{
    int ch;

    while ((ch = getchar()) != EOF)
        putchar(ch);

    return 0;
}
```

Burada birinci getchar dolayısıyla bizden bir satır istenir. Artık diğer getchar'lar girdiğimiz yazının karakterlerini alarak yazdırır. Tabii tamponun sonundaki '\n' de yazdırılmaktadır. Bu sırada imleç aşağıdaki satırın başına geçmiş olur. Sonraki getchar yine bizden bir satır isteyecektir. Döngüden çıkmak için EOF tuş kombinasyonları kullanılır.

stdin Tamponunun Boşaltılması

Maalesef C'de standart olarak stdin tamponunu boşaltan bir fonksiyon yoktur. Bazı derleyicilerde fflush(stdin) bunu yapıyor olmakla birlikte, bu kullanım uygunsuzdur. Çünkü her şeyden önce fflush read-only dosyalarda kullanılamaz ve stdin de read-only bir dosya kabul edilmektedir. Tamponu boşaltmak için küçük bir fonksiyon ya da makro yazılabilir:

```
#include <stdio.h>

void empty_stdin(void)
{
    int ch;

    while ((ch = getchar()) != '\n' && ch != EOF)
        ;
}

int main(void)
{
    int ch1, ch2;

    ch1 = getchar();
    empty_stdin();
    ch2 = getchar();

    printf("ch1 = %d, ch2 = %d\n", ch1, ch2);
}
```

```
    return 0;
}
```

Tamponu boşaltırken EOF kontrolü de yapılmıştır. Aksi takdirde yönlendirme durumunda sonsuz döngü oluşabilir. Aynı fonksiyonu makro olarak da yazabiliriz:

```
#define empty_stdin() \
do { \
    int ch; \
    while ((ch = getchar()) != '\n' && ch != EOF) \
        ; \
} \
while (0)
```

stdin Dosyasını Kullanan Standart C Fonksiyonlarının Davranışı

Yukarıda zaten getchar fonksiyonu açıklandı. Burada diğer fonksiyonlar üzerinde duracağız.

gets Fonksiyonu

gets fonksiyonu C standartlarından 2011 yılında kaldırılmıştır. Bunun yerine isteğe bağlı olarak derleyicilerin destekleyebileceği bir gets_s fonksiyonu önerilmiştir. gets fonksiyonundaki sorun okuma işleminin yapılacağı dizinin ne kadar uzunlukta olması gerektiği ile ilgilidir. Teorik olarak kullanıcı daha fazla giriş yaparak programı çökertebilir.

gets fonksiyonun prototipi şöyledir:

```
char *gets(char *str);
```

Fonksiyon '\n' görene kadar ('\n' dahil olmak üzere) ya da EOF görene kadar stdin dosyasından karakterleri okur ve aldığı adresten itibaren bunları söz konusu diziye yerleştirir. Fakat gets '\n' karakterini okuduğunda bunu diziye yerleştirmez. Bunun yerine diziye '\0' karakterini yerleştirir. gets normal durumda parametresiyle aldığı adresin aynısına geri döner. Fakat stdin dosyasından hiç okuma yapmadan EOF görürse NULL adrese geri dönmektedir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int ch;
    char s[100];

    ch = getchar();
    gets(s);

    printf("%c-%s\n", ch, s);

    return 0;
}
```

Burada getchar ile biz "ankara" yazısını girip ENTER tuşuna basmış olalım. Girdiğimiz yazının ilk karakterini getchar alır, diğerlerini gets alır. gets tamponun sonundaki '\n' yi de okur fakat onun yerine diziye '\0' yerleştirir. Pekiyi çağrılar ters sırada olsaydı ne olurdu?

```
#include <stdio.h>

int main(void)
{
    int ch;
    char s[100];

    gets(s);
```

```

    ch = getchar();

    printf("%c-%s\n", ch, s);

    return 0;
}

```

Burada gets tüm tamponun bir satırla dolmasına yol açar fakat onların hepsini okur. Dolayısıyla getchar tamponu boş göreceğinden bizden yeniden giriş istenecektir. Aşağıdaki programı stdin dosyasını bir dosyaya yönlendirerek test ediniz:

```

#include <stdio.h>

int main(void)
{
    char buf[4096];

    while (gets(buf) != NULL)
        puts(buf);

    return 0;
}

```

Burada dosyanın sonuna gelindiğinde artık gets hiçbir karakter okuyamadan EOF ile karşılaşır ve NULL adrese geri döner. Böylece döngüden çıkmış olur.

Aslında gets fonksiyonu getchar kullanılarak yazılabilir (zaten çoğunlukla da böyle yazılmaktadır). Örneğin:

```

#include <stdio.h>

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

    i = 0;
    while ((ch = getchar()) != '\n' && ch != EOF)
        buf[i++] = ch;

    if (i == 0 && ch == EOF)
        return NULL;

    buf[i] = '\0';

    return buf;
}

int main(void)
{
    char buf[4096];

    while (mygets(buf) != NULL)
        puts(buf);

    return 0;
}

```

Yukarıda da belirtildiği gibi aslında gets fonksiyonun tasarımında bir hata vardır. Çünkü gets fonksiyonunda fonksiyona verilen dizi ne kadar büyük olursa olsun her zaman ondan daha fazla karakter girilerek program çökertilebilir. (Tabii pek çok sistemde terminal aygıt sürücülerinin belli bir okuma limiti vardır. Fakat hem bu limit belli değildir hem de dosyaya yönlendirme yapıldığında aynı sorun yine oluşabilir.) Bu nedenle ld bağlayıcıları gets kullanıldığında aşağıdaki gibi bir uyarı da vermektedir:

```

csd@csd-virtual-machine ~/Study/SysProg-2017 $ gcc -o sample sample.c
sample.c: In function 'main':
sample.c:7:5: warning: implicit declaration of function 'gets' [-Wimplicit-function-declaration]
     gets(buf);
     ^
/tmp/cc9CgyM9.o: In function `main':
sample.c:(.text+0x2a): uyarı: the `gets' function is dangerous and should not be used.

```

Yukarıda da belirtildiği gibi C'nin son sürümü olan C11'de (ISO/IEC 9899: 2011) artık gets tamamen kütüphaneden kaldırılmıştır. Onun yerine isteğe bağlı olarak derleyicilerin bulundurabileceği gets_s (safe gets) isimli fonksiyon kütüphaneye dahil edilmiştir. (Zaten pek çok C kütüphanesi uzun süredir gets_s fonksiyonunu da destekliyordu.) gets_s fonksiyonunun prototipi şöyledir:

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

```
char *gets_s(char *buffer, size_t count);
```

Fonksiyon en fazla count - 1 tane karakteri stdin dosyasından okur. Yani fonksiyon çağrılırken ikinci parametreye dizinin uzunluğunu geçmek gerekir. Fonksiyonun diğer davranışları gets ile aynıdır. gets_s fonksiyonu da getchar kullanılarak şöyle yazılabilir:

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

```

char *mygets_s(char *buf, size_t count)
{
    int ch;
    size_t i;

    for (i = 0; i < count - 1; ++i) {
        if ((ch = getchar()) == '\n' || ch == EOF)
            break;
        buf[i] = ch;
    }

    if (i == 0 && ch == EOF)
        return NULL;

    buf[i] = '\0';

    return buf;
}

int main(void)
{
    char buf[3];

    mygets_s(buf, 3);
    puts(buf);

    return 0;
}

```

Maalesef Microsoft'un gets_s fonksiyonunun semantiği standartlardakinden farklıdır. Microsoft bazı fonksiyonlarda "parametre denetimi (parameter validation)" kullanmaktadır. gets_s fonksiyonu da bu biçimde parametre denetimine sokulmaktadır. gcc derleyicilerinin standart C kütüphanelerine ise henüz gets_s eklenmemiştir.

Aslında gets_s fonksiyonun yerine fgets fonksiyonu da kullanılabilir. Ancak fgets fonksiyonu gets ve gets_s fonksiyonlarından farklı olarak '\n' karakterini de parametresiyle aldığı adrese yerleştirmektedir. fgets fonksiyonun prototipi şöyledir:

```
char *fgets(char *s, int size, FILE *stream);
```

Fonksiyon yine en fazla ikinci parametresiyle belirtilen miktarda karakteri birinci parametresiyle belirtilen adrese yerleştirir. Ancak '\n' karakterini gördüğünde (eğer görürse) bu karakteri de fgets verilen adrese yerleştirmektedir. Dolayısıyla fgets fonksiyonunu gets_s gibi kullanabilmek için eğer fonksiyona geçirilen dizinin sonunda '\n' karakteri varsa bunun yok edilmesi gerekir. Tipik kullanım şöyledir:

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

int main(void)
{
    char s[10];
    char *str;

    if ((str = fgets(s, 10, stdin)) != NULL)
        if ((str = strchr(s, '\n')) != NULL)
            *str = '\0';

    if (str != NULL)
        puts(s);

    return 0;
}
```

gets, gets_s ya da fgets (stdin ile) fonksiyonlarından önce stdin tamponunun boşaltılması gerekebilir. Örneğin stdin dosyasından getchar fonksiyonuyla önce bir karakter sonra da gets fonksiyonuyla bir isim almak isteyelim:

```
#include <stdio.h>

int main(void)
{
    int ch;
    char name[512];

    printf("Bir karakter giriniz:");
    fflush(stdout);

    ch = getchar();

    printf("Bir isim giriniz:");
    fflush(stdout);

    gets(name);

    printf("'%'c' - \"%s\"\n", ch, name);

    return 0;
}
```

Burada getchar fonksiyonu ile bir stailik giriş alındığında '\n' karakteri de stdin tamponuna yerleştirilecektir. Dolayısıyla gets fonksiyonu bu '\n' karakterini alarak hemen sonlanacaktır. Bu tür durumlarda yukarıda da ele aldığımız gibi stdin tamponunun boşaltılması gerekir:

```
#include <stdio.h>

void empty_stdin(void)
{
    int ch;

    while ((ch = getchar()) != '\n' && ch != EOF)
```

```

    ;
}

int main(void)
{
    int ch;
    char name[512];

    printf("Bir karakter giriniz:");
    fflush(stdout);

    ch = getchar();
    empty_stdin();

    printf("Bir isim giriniz:");
    fflush(stdout);

    gets(name);

    printf("'%c' - \"%s\"\n", ch, name);

    return 0;
}

```

scanf Fonksiyonu

scanf fonksiyonu formatlı okuma yapmak için kullanılmaktadır. Format karakterleri girdinin biçimini ve türünü belirtir. Fonksiyonun prototipi şöyledir:

```
int scanf(const char *format, ...);
```

Fonksiyon yerleştirme yapılan nesne sayısına geri döner. Örneğin:

```
result = scanf("%d%d", &a, &b);
```

Burada scanf önce boşluk karakterlerini atar sonra okumayı yaparak ilk nesneye yerleştirir. Yani scanf baştaki boşluk karakterlerini (leading space) okur fakat sondakileri (trailing space) okumaz, sondakiler stdin tamponunda kalmaktadır. scanf stdin dosyasından bilgileri karakter karakter okur, okuduğu karakterin format karakterine uygun olmadığını gördüğünde onu stdin tamponuna geri bırakır (ungetc standard C fonksiyonuna bakınız) ve işlemini sonlandırır. Örneğin:

```

#include <stdio.h>

int main(void)
{
    int a = -1;
    int result;
    char s[100];

    result = scanf("%d", &a);
    gets(s);
    printf("a = %d, result = %d, s = \"%s\"\n", a, result, s);

    return 0;
}

```

Burada scanf için " ali " girişini yapmış olalım. Aşağıdaki gibi bir sonuç elde edilir:

```
a = -1, result = 0, s = "ali "
```

a'ya yerleştirilme yapılmadığı için verilen ilkdeğerin orada kaldığına dikkat ediniz. Burada scanf sıfır ile geri dönmüştür. Çünkü hiç yerleştirme yapamamıştır. gets ise tampondan geri kalanların hepsini okumuştur. Örneğin:


```
#include <stdio.h>

int main(void)
{
    int a = -1, b = -1;
    int result;
    char s[100];

    result = scanf("%d%d", &a, &b);
    gets(s);
    printf("a = %d, b = %d, result = %d, s = \"%s\"\n", a, b, result, s);

    return 0;
}
```

Burada klavyeden " c " girmiş olalım. Şöyle bir sonuç elde edilir:

```
a = 100, b = -1, result = 1, s = "ali "
```

Görüldüğü gibi a için başarılı yerleştirme yapılmıştır. b için scanf işlemi başlattığında önce "ali" nin solundaki boşluk karakterlerini atmıştır. Fakat 'a' yı beğenmemiş ve onu yeniden stdin tamponuna bırakmıştır. gets de geri kalanları almıştır.

scanf fonksiyonunda format karakterleri arasına başka bir karakter konulursa scanf okuma sırasında bu karakterlerin o pozisyonda bulunmasını ister. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int a = -1, b = -1;
    int result;
    char s[100];

    result = scanf("%d,%d", &a, &b);
    gets(s);
    printf("a = %d, b = %d, result = %d, s = \"%s\"\n", a, b, result, s);

    return 0;
}
```

Girişin "100 200" biçiminde yapıldığını düşünelim. scanf 100'den sonra ',' karakteri beklemektedir. Bu gelmediği için oradan okuduğu boşluk karakterini tampona geri bırakır ve işlemini sonlandırır. Şöyle bir çıkış elde edilmiştir:

```
a = 100, b = -1, result = 1, s = " 200"
```

Format karakterleri arasında hiç boşluk kullanılmazsa bu durum "önceki boşluk karakterlerinin (leading space) atılacağı anlamına" gelir. Fakat araya bir başka bir karakter getirilirse artık boşluklar atılmaz kesinlikle o karakter beklenir. Aşağıdaki iki çağrı arasındaki farkı inceleyiniz:

```
scanf("%d%d", &a, &b);
scanf("%d,%d", &a, &b);
```

Birinci çağrıda girişte iki sayı arasında isteğimiz kadar boşluk karakteri bırakabiliriz. İkinci girişte kesinlikle birinci sayıdan sonra hiç boşluk karakteri olmadan ',' karakteri bulunmak zorundadır. Fakat bu ',' karakterinden sonra yine istenildiği kadar boşluk karakteri bulundurulabilir.

Örneğin:

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

```

int main(void)
{
    int day = -1, month = -1, year = -1;

    scanf("%d/%d/%d", &day, &month, &year);
    printf("%d/%d/%d\n", day, month, year);

    return 0;
}

```

Format kısmında tek boşluk karakteri "bir ya da birden fazla boşluk karakterini at" anlamına gelir. Örneğin:

```
scanf("%d %d", &a, &b);
```

Burada ilk girişten sonraki boşluk karakterlerinin hepsi atılacaktır. Bunun boşluksuz durumdan işlevsel farklılığı yoktur. Fakat aşağıdaki gibi bir çağrıda ilginç bir durum oluşur:

```
scanf("%d%d\n", &a, &b);
```

Burada format karakterlerinden sonra bir boşluk karakteri (white space) görüldüğü için scanf bu noktada tüm boşluk karakterlerini atmak ister. Böylece biz boşluk girdiğimiz sürece bizden karakter istemeye devam edecektir. Yani boşluk karakteri girmeyene kadar scanf bizden karakter isteyecektir. Tabii örneğimizde '\n' yerine SPACE karakteri koysaydık da davranış yine aynı olacaktı.

scanf boşluk karakterlerini ayıraç kullandığı için %s ile okuma yaparken ilk boşluk karakterini gördüğünde okumayı tamamlar. Yani biz %s ile boşluklu bir yazı okuyamayız. Örneğin:

```

#include <stdio.h>

int main(void)
{
    char buf1[100], buf2[100];

    scanf("%s", buf1);
    gets(buf2);

    printf("buf1 = %s, buf2 = %s\n", buf1, buf2);

    return 0;
}

```

Burada "ali veli selami" yazısını girmiş olalım. Şöyle bir çıktı elde ederiz:

```
buf1 = ali, buf2 = veli selami
```

scanf eğer boşluk karakterlerini attıktan sonra (ya da atmadan önce) dosya sonuyla karşılaşarsa EOF değerine (-1) geri döner.

Aşağıdaki gibi bir a.txt dosyası olsun. Dosyanın içerisinde boşluklarla ayrılmış pek çok sayı vardır:

```

a.txt
-----

100 200
300
400
500 600
700 800
900
100

```

Şimdi biz bu değerleri komut satırından aşağıdaki gibi bir program çalıştırarak "IO yönlendirmesi (IO redirection)" ile okumak isteyelim:

```
./sample < a.txt
```

Aşağıdaki program bunu yapabilir:

```
#include <stdio.h>

int main(void)
{
    int val;

    while (scanf("%d", &val) != EOF)
        printf("%d\n", val);

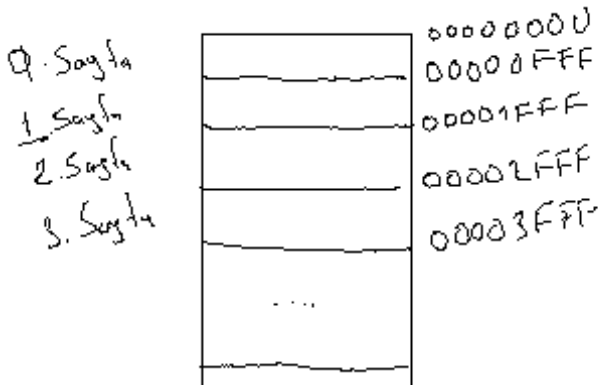
    return 0;
}
```

İşlemcilerin Sayfalama Mekanizmaları (Paging Mechanisms)

Modern ve güçlü işlemciler sayfalama (paging) denilen bir mekanizmaya sahiptir. Sayfalama mekanizmasında bellek sayfa (page) denilen ardışıl byte bloklarından oluşmaktadır. Sayfaların büyüklükleri işlemciye bağlı olmakla birlikte pek çok işlemci 4K uzunluğunda sayfa kullanmaktadır. (Bazı işlemciler çeşitli modlarda çeşitli uzunlukta sayfaları destekleyebilmektedir. Ancak pek çok işletim sistemi bu işlemcileri 4K'lık sayfa modunda çalıştırmaktadır.) Aşağıda yaygın bazı mikroişlemcilerin desteklediği sayfa uzunlukları tablo halinde verilmiştir:

İşlemci	En Küçük Sayfa Uzunluğu	Desteklenen Diğer Sayfa Uzunlukları
32 Bit X86	4 KB	2 MB, 4 MB
64 Bit X86 (X64)	4 KB	2 MB, 1 GB
Itanium (IA-64)	4 KB	8 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 256 MB
Power PC	4 KB	64 KB, 16 MB, 16 GB
SPARC	4 KB	256 KB, 16 MB
Ultra SPARC	8 KB	64 KB, 512 KB, 4 MB, 32 MB, 256 MB, 2 GB, 16 GB
ARM V7	4 KB	64 KB, 1 MB, 16 MB

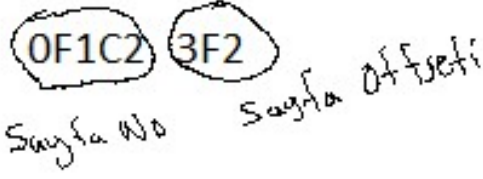
Bellekte her bir sayfaya 0'dan başlayarak bir sayfa numarası verilmiştir. Örneğin 4K'lık sayfa kullanan sistemde fiziksel belleğin haritalanması şöyledir:"



32 bit bir sistemde adresler 4 byte (8 hex digit) ile ifade edilir. Her fiziksel adres aslında bir sayfa içerisindedir ve o sayfada belli bir offset'tedir. (Belli bir noktadan görece uzaklığa "offset" denilmektedir.) Adresin hangi sayfada olduğunu bulmak için onu sayfa uzunluğuna bölmek gerekir. (Tabii 2'lik sistemde bir sayıyı 4096'ya bölmek demek 12 kez sağa ötelemek, başka bir deyişle sağdaki 12 biti atmak demektir.) Adresin o sayfanın hangi offset'inde olduğunu bulmak için

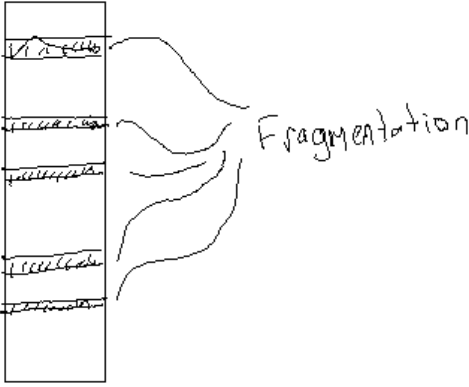
ise adresin sayfa uzunluđuna bölümünden elde edilen kalana bakılır (sayının 4096'ya bölümünden elde edilen kalan düşük anlamlı 12 bitidir.) Örneđin:

0F1C23F2 fizksel adres hangi sayfada ve o sayfanın hangi offset'tedir?



Sayfalama mekanizmasına sahip olan bir sistemde programlar fizksel RAM'e ardışıl yüklenmek zorunda değildir. Ardışıl yüklenme zorunluluđunun ortadan kaldırılması bölünme (fragmentation) denilen olgunun zararlı etkisini ortadan kaldırır ve sanal bellek (virtual memory) kullanımına olanak sağlar.

Anahtar Notlar: Bellek sistemlerinde en önemli olgulardan biri bölünme (fragmentation) durumudur. Bölünme arşılık tahsisat yönteminin yol açtığı bir sonuçtur. Eğer söz konusu belleđe (disk de olabilir RAM de olabilir) birtakım öğeler ardışıl yerleştiriliyorsa zamanla bunların yaratılıp yok edilmesi sonucunda küçük fakat çok sayıda boş alan oluşmaktadır. Bu alanlar küçük olduđu için pek bir işe yaramazlar. Fakat bellek kullanım oranını ciddi biçimde düşürürler. Bu olguya dışsal bölünme (external fragmentation) denilmektedir. Örneđin dışsal bölünmüş bir bellek şöyle görünümde olabilir (taranmış alanlar boş alanlar):



Örneđin heap bölgesi de aslında tahsis etme ve serbest bırakma işlemleri sonucunda bölünmeye maruz kalmaktadır. Öyle ki belli bir süre sonra çok sayıda fakat çok küçük boş bellek bölgeleri oluşmaktadır. Bunların toplam miktarı ciddi boyutlara vardığı halde küçük olduklarından dolayı kullanılacak durumda değildir. Örneđin toplam her biri 20 byte'tan küçük yüzlerce boş alanın toplama 50K olabilir. Durum böyle olduđu halde biz 500 byte'lık bir alan bile tahsis edemeyiz.

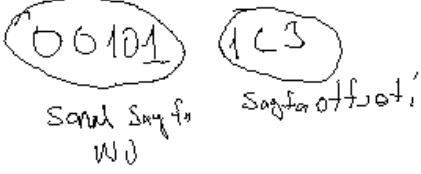
Peki bir program fizksel belleđe ardışıl yerleştirilmiyorsa nasıl çalışmaktadır? İşte program içerisindeki adreslere "dođrusal adresler (linear addresses), mantıksal adresler (logical addresses) ya da sanal adresler (virtual addresses)" denilmektedir. Biz burada "sanal adres" terimini kullanacađız. İşlemci bu sanal adresleri "sayfa tablosu (page table)" denilen bir tabloya bakarak fizksel adreslere dönüştürmektedir. İşlemci sayfa tablosunu belli bir yazmacın (register) gösterdiği yerde arar. (Örneđin Intel işlemcilerinde CR3 yazmacı sayfa tablosunun yerini belirtmektedir.) Prosesin sayfa tablosunu işletim sistemi oluşturur ve adresini de bu yazmaca yerleştirir. Böylece işlemci işletim sistemi tarafından oluşturulmuş olan sayfa tablosunu kullanır hale gelmektedir. Peki sayfa tablosu nasıldır? Bazı sistemlerde iki kademeli hatta üç kademeli sayfa tabloları kullanılmaktadır. Örneđin Intel'de iki kademeli sayfa tablosu kullanılır. Ancak biz burada kavramsal karmaşıklık oluşmasın diye tek kademeli bir sayfa tablosunu örnek vereceđiz. Sayfa tablosunun tipik biçimi şöyledir:

Sanal Sayfa No (Aslında Tabloda Yok)	Fizksel Sayfa No	Sayfa Özellikleri
...
00100	01FC3	RW, P, User, D
00101	2FC40	RW, P, User, D
00102	1C167	RW, P, User, D
...

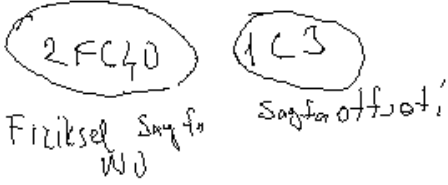
Tablonun ilk sütunu aslında bir indeks numarasıdır. Yani tabloda yoktur. Şöyle ki aslında 00100 sanal sayfasına ilişkin giriş tabloda tablonun başından itibaren bir sayfa girişinin uzunluğu ile 0x100'ün çarpımı kadar uzaklıktadır. Tablodaki her bir satıra "sayfa tablosu girişi (page table entry)" denilmektedir. İşlemci bir sanal adresle karşılaştığında önce o sanal adresi sanal sayfa numarasına ve sayfa offsetine ayırır. Sonra sanal sayfa numarasını sayfa tablosuna indeks yapar ve oradan fiziksel sayfa numarasını elde eder. Ona da sayfa offset'ini toplar ve nihai fiziksel adresi elde eder. Örneğin aşağıdaki gibi bir kod olsun:

```
MOV EAX, [001011C3]
```

İşlemci buradaki 001011C3 adresini aşağıdaki gibi sanal sayfa numarasına ve sayfa offset'ine ayırır:

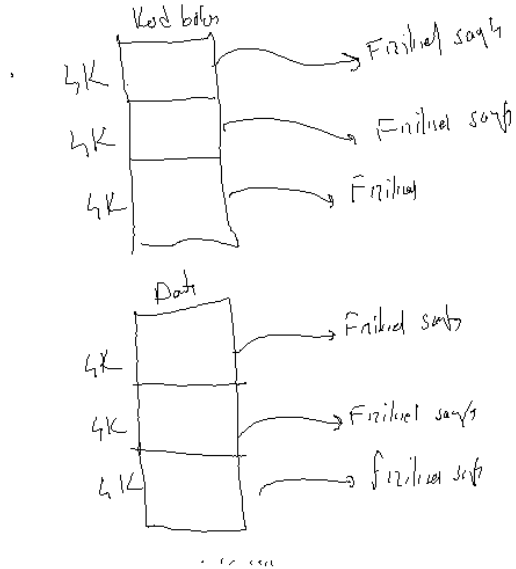


bundan sonra işlemci sayfa tablosuna bakarak 0x101 numaralı sanal sayfa numarasının hangi fiziksel sayfa ile eşleştirildiğine bakar. Yukarıdaki örnekte 0x101 numaralı sanal sayfa 2FC40 fiziksel sayfasıyla eşleştirilmiştir. Bu adresin 4096 ile çarpımına 1C3 eklenir ve şu değer bulunur:



Böylece erişimi fiziksel bellekteki 2FC401C3 adresinden yapar.

İşletim sistemi bir programı fiziksel belleğe sayfa sayfa (ardışıl olmayan sayfalar) yükler ve sayfa tablosundaki girişleri buna göre değiştirir. Böylece program işlemci tarafından sayfa tablosuna bakılarak çalıştırıldığı için kesiksiz çalışma devam eder.



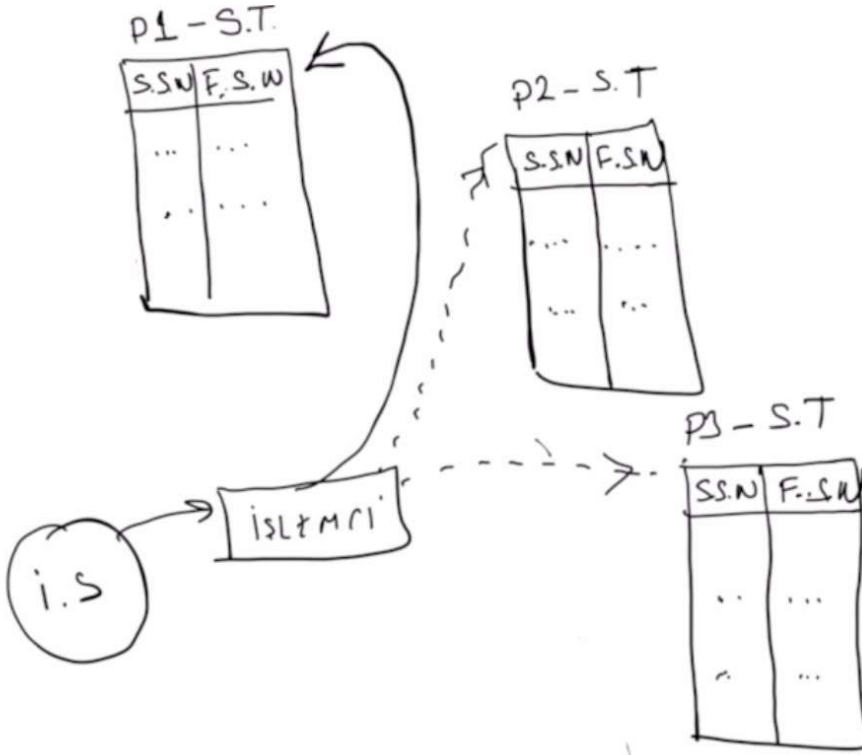
Sanal Bellek Mekanizması (Virtual Memory)

Sanal bellek bir programın tamamının değil belli kısımlarının fiziksel belleğe yüklenerek disk (ikincil bellekle) ile fiziksel bellek arasında yer değiştirmeli olarak çalıştırılmasına denir. Sanal bellek mekanizmasında bir programın yalnızca belirli kısmı fiziksel belleğe yüklenerek program çalıştırılır. Sonra program çalışırken kod ya da data bakımından fiziksel bellekte

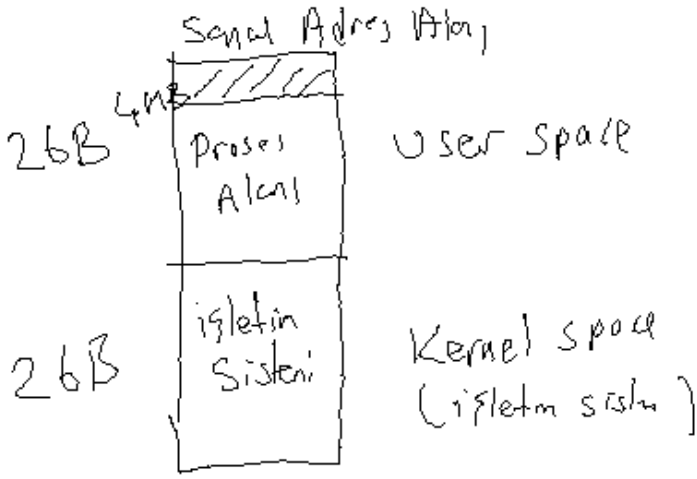
olmayan bir kısma erişildiğinde işlemci bunu anlar ve bir içsel kesme oluşturur. Bu içsel kesmeye Intel terminolojisinde "page fault" denilmektedir. Böylece işletim sistemi devreye girer. Prosesin bellekte olmayan sayfasını diskten bulur, onu fiziksel boş bir sayfaya yükler, prosesin sayfa tablosunu günceller ve kesmeden çıkar.

Derlenmiş bir kodda derleyicinin ürettiği adreslerin hepsi sanal adreslerdir. Yani örneğin derleyici 32 bit bir sistemde sanki program 4GB'lık fiziksel belleğe tek başına yüklenecekmiş gibi kod üretimi yapar. Sanki bütün programlar aynı adresten itibaren fiziksel bellekte tek başlarına çalışacakmış gibi bir koda sahiptir.

Windows, Linux, Mac OS X gibi sanal bellek kullanan çokişlemli (multiprocessing) sistemlerde her prosesin ayrı bir sayfa tablosu vardır. İşletim sistemi programı yükleyeceği zaman onu fiziksel sayfalara dağıtır ve o prosesin sayfa tablosunu o fiziksel sayfaları gösterecek biçimde düzenler. Böylece proseslerin sanal adres alanları aynı olmasına karşın onların parçaları farklı fiziksel adreslerde bulunurlar. Başka bir deyişle örneğin Windows'ta, Linux ve Mac OS X'te farklı proseslerdeki aynı sanal adresler aslında fiziksel bellekte aynı yere karşılık gelmezler. Aslında bu tür sistemlerde fiziksel belleğin bir önemi yoktur. İşletim sistemi her prosesler arası geçiş (task switch) olduğunda otomatik olarak işlemcinin kullandığı sayfa tablosunu da o prosesin sayfa tablosunu gösterecek biçimde değiştirir.



Sayfalama mekanizmasını kullanan işletim sistemlerinde proseslerin bellek alanları tam olarak izole edilmiştir. İşletim sistemi her prosesin sayfa tablosunda onların sanal sayfalarını farklı fiziksel sayfalara yönlendirdiği için bir proses başka bir prosesin alanına erişemez. Tabii işletim sistemi her zaman bellektedir. . Yani işletim sistemi bütün proseslerin sayfa tablosunda aynı yerde bulunmaktadır. Örneğin 32 Windows sistemlerinde tipik sanal bellek alanının kullanımı şöyledir:



Tipik olarak Windows sistemlerinde derleyici ve bağlayıcı sanki program 4 MB'tan itibaren fiziksel belleğe yüklenecekmiş gibi kod üretmektedir. Ancak bu yükleme adresi değiştirilebilmektedir. Yani Windows, Linux ve Mac OS X sistemlerinde programlar aynı sanal bellek alanına yüklenecekmiş gibi kod üretilmekle birlikte işletim sistemi programları aslında sayfa tabloları yoluyla farklı fiziksel sayfalara yüklemektedir.

Sayfa tablosu girişlerindeki sayfa özellikleri ilgili sayfaya erişim haklarını ve başka birtakım bilgileri içerir. Örneğin bir sayfa read-only ise o sayfaya yazma yapılmaya çalışıldığında exception oluşur. İşletim sistemi devreye girer ve prosesi cezalandırarak sonlandırır. Örneğin C ve C++'ta pek çok derleyici string'leri read-only sayfalara yerleştirmektedir. Böylece bir string update edilmek istendiğinde exception oluşur.

```
#include <stdio.h>

int main(void)
{
    char *s = "ankara";

    *s = 'x';      /* read-only sayfaya erişim! */

    return 0;
}
```

Tabii burada "string'leri gerçekte kimin read-only sayfalara yerleştirdiği sorusu aklınıza gelebilir. Aslında bu işe derleyici öncülük etmektedir. Derleyici ürettiği amaç kodda (object file) stringler'in read-only sayfalara yerleştirilmesi gerektiğini dosyanın içerisinde belirtir. Bağlayıcı da çalıştırılabilen dosyayı oluştururken bu bilgiyi çalıştırılabilen dosyanın içerisine yazar. İşletim sistemi de programı yüklediğinde sayfa tablosunu oluştururken string'lerin bulunduğu bu sayfaları çalıştırılabilen dosyada belirtildiği gibi read-only oluşturmaktadır.

Peki Windows, UNIX/Linux ve MAC OS X sistemlerinde proseslerin bellek alanlarının izole edildiğini gördük. Ancak işletim sisteminin kendi kod ve verileri yine aynı adres alanı içerisinde olduğuna göre bir proses işletim sisteminin alanına erişemez mi? İşte işletim sistemi kendini başka bir mekanizmayla korumaktadır. Her fiziksel sayfa "kernel" ya da "user" biçiminde önceliklendirilmiştir. Kernel mod özelliğine sahip sayfalara yalnızca kernel modda çalışan kodlar erişebilirler. Fakat user mod sayfalara hem kernel modda hem de user modda çalışan kodlar erişebilmektedir. Bu durumda örneğin biz kendi programımızda işletim sisteminin sayfalarına erişmek istersek bunlar kernel mod sayfalar olduğu için yine exception oluşacak ve prosesimiz sonlandırılacaktır. Sayfanın "user mod" sayfa mı "kernel mod" sayfa mı olduğu yine sayfa tablosu girişinde "sayfa özellikleri" kısmında belirtilmektedir. O halde sayfa tablosunda her sayfa için iki önemli sayfa özelliği de tutulmaktadır: Sayfa read-only i read/write biçimde mi ve sayfa kernel modda mı yoksa user modda mı? Aslında sayfa tablosunda başka sayfa özellikleri de bulunabilmektedir. Ancak kursumuzda bu ayrıntılardan bahsetmeyeceğiz. Bu konunun tüm ayrıntıları "80x86 ve ARM Sembolik Makine Dili" kursunda ele alınmaktadır.

Şüphesiz sayfalama mekanizmasını kullanan işletim sistemleri her sayfanın boş mu dolu mu olduğu bilgisini ve dolu olan fiziksel sayfaların hangi proses tarafından kullanılıyor olduğu bilgisini bir biçimde tutmak zorundadır. Bir prosesin sayfa tablosunun oluşturulması proses yaratılırken işletim sistemi tarafından yapılmaktadır.

Sanal bellek mekanizmasında aslında işletim sistemi prosesin sayfa tablosundaki tüm girişleri fiziksel sayfalarla eşleştirmez. Yalnızca bazı sayfaları fiziksel sayfalarla eşleştirir. Örneğin:

Prosesin Sayfa Tablosu

4GB'lik

Sanal Sayfa No	Fiziksel Sayfa No
...	...
101A	3BC7
101B	4714
101C	-
101D	481g
101E	-
...	...

Programın akışı kod ya da veri bakımından prosesin fiziksel bellekte olmayan bir kısmına geldiğinde (yani sayfa tablosunda sanal sayfa numarasının fiziksel sayfaya yönlendirilmemiş olduğu bir girişle karşılaşıldığında. Bu durum yukarıdaki şekilde '-' ile gösterilmiştir.) işlemci bir içsel kesme oluşturur (page fault). Bu kesme sonucunda işletim sistemi devreye girer. Hangi prosesin hangi sayfasına eriştiğini tespit eder. Bu sayfayı diskten alarak RAM'de boş bir fiziksel sayfaya yükler. Prosesin sayfa tablosunda fiziksel sayfa yönlendirmesini buna göre yapar ve kesmeden çıkar. Böylece akış sayfalama hatası (page fault) denilen kesmeye yol açan makine komutundan devam edecektir. Tabii artık bu makine komutu fiziksel sayfa yönlendirmesi olduğu için sorunsuz çalışacaktır. Bu mekanizmada görülen şudur: Bir prosesin küçük bir kısmı fiziksel RAM'e yüklenerek program başlatılabilir. Diğer kısımları yalnızca gerektiğinde yüklenebilir. Bu sisteme İngilizce "demand paging" denilmektedir.

Sanal bellek gerçekleştirimi adım adım şöyle yapılmaktadır:

1) Program kodunda bir sanal adrese erişilmiştir. İşlemci sayfa tablosuna bakar ve bunu fiziksel adrese dönüştürmeye çalışır. Eğer sanal sayfa sayfa tablosunda bir fiziksel sayfaya yönlendirilmemişse (sayfanın P (Present Bit) biti 0 ise) "sayfalama hatası (page fault)" denilen içsel kesmeyi oluşturur.

2) Sayfalama hatası kesmesi dolayısıyla işletim sisteminin kesme kodu (interrupt handler) devreye girer ve işletim sistemi bu hataya hangi prosesin hangi kısmının yol açtığını belirlemeye çalışır. Eğer erişilmek istenen yer proses tarafından tahsis edilmiş bir alan değilse işletim sistemi doğrudan prosesi sonlandırır. Eğer erişilmek istenen alan prosesin tahsis edilmiş bir parçası ise bu durumda işletim sistemi erişilmek istenen sanal sayfasının diskteki yerini bulur. Onu fiziksel RAM'e yüklemek ister. Tabii bunun için boş bir fiziksel sayfa bulmaya çalışır. Eğer bulursa diskten o sayfayı o fiziksel sayfaya yükler. Bu sürece İngilizce "swap-in" denilmektedir. Pekiyi ya fiziksel bellek tıka basa doluyorsa ne olacaktır? Bu durumda işletim sistemi, fiziksel bellekten başka bir prosese ait olan bir sayfayı çıkarmak ister. Tabii çıkaracağı sayfada bir güncelleme yapılmışsa onu son haliyle yeniden diske yazacaktır. Bu işleme ise İngilizce "swap-out" denilmektedir. Tabii burada işletim sistemi gelecekte en az kullanılacak bir sayfayı fiziksel RAM'den çıkarmak ister. Tıka basa dolu bir RAM'de her swap-in işleminde bir swap-out yapılması sistemi yavaşlatmaktadır. Bu probleme İngilizce "thrashing" denilmektedir.

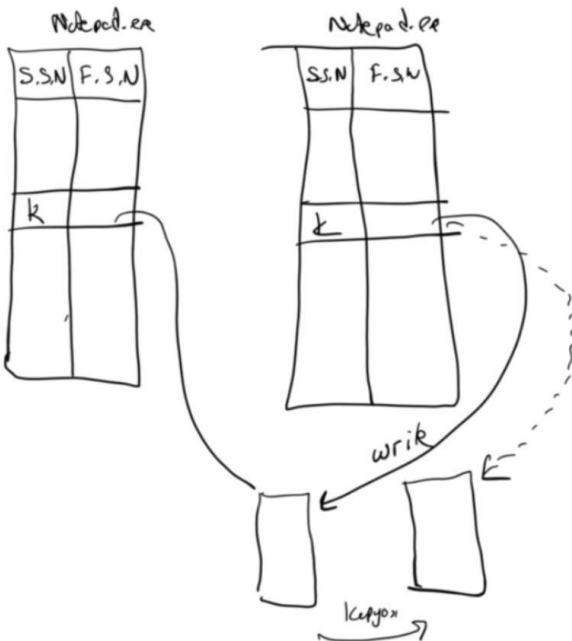
3) İşletim sistemi swap-in yaptıktan sonra prosesin sayfa tablosunu düzeltir ve kesmeden (fault'tan) çıkar. Böylece akış sayfalama hatasına (page fault) yol açan makine komutuyla devam edecektir. Tabii artık sayfalama hatası kesmesi oluşmayacaktır.

Yukarıdaki işlemlerin bazı ayrıntıları vardır. Örneğin işletim sistemi "swap-in" yapacağı zaman programın ilgili kısmını diskte nereden alınacaktır? Bu konuda işletim sistemlerinin değişik stratejileri vardır. Bazı işletim sistemleri her çalıştırılabilir (executable) dosyanın kendisini hem de bir "swap dosyasını" bu amaçla kullanır (örneğin Windows gibi). Bazı sistemlerde çalıştırılabilir (executable) dosya hiç kullanılmaz ve birden fazla swap dosyası kullanılır. Bazı sistemler swap dosyası yerine ayrı bir disk bölümünü (partition) kullanmaktadır. Pekiyi neden çalıştırılabilir dosya bunun için yeterli olmamaktadır? Eğer bir fiziksel sayfa üzerinde değişiklik yapılmışsa o sayfa swap-out yapılırken onu çalıştırılabilir dosyaya yazmak mümkün değildir. Fakat read-only sayfalar her zaman çalıştırılabilir dosyadan alınabilmektedir. Pekiyi bir sayfa fiziksel RAM'dan çıkartılacağı zaman her zaman o sayfanın diske geri yazılması gerekir mi? İşte sayfa tablo girişlerinde sayfa özellikleri kısmında bir D biti (Dirty bit) de bulunmaktadır. İşlemci ne zaman bir sayfaya yazma yapsa bu D bitini set eder. İşletim sistemi de sayfayı fiziksel bellekten çıkartacağı zaman bu D bitine bakar. Bu D biti 0 ise boşuna sayfayı swap dosyasına yazmaz. Böylece gereksiz swap-out işlemlerini elimine edilmiş olur.

Pekiyi proses hiç tahsis etmediği bir alana erişmek isterse ne olur? Çünkü alanın diskte de bir karşılığı yoktur. İşte işletim sistemi page fault oluştuğunda önce fault'a yol açan sanal adresin gerçekten legal bir adres olup olmadığına bakmaktadır. Yani erişilmek istenen adres tahsis edilmiş bir adres değilse prosesi hemen sonlandırır.

Pekiyi bir proses malloc gibi bir fonksiyonla dinamik bir tahsisat yaptığında ne olur? Bu durumda tipik olarak işletim sistemi tahsis edilen alan için sanal bellekte tahsisat yapar. Ancak onu fiziksel RAM'a taşımaz. Bu dinamik alan kullanıldıkça fiziksel RAM'e taşınacaktır.

Pekiyi aynı program ikinci kez yüklendiğinde onun sayfaları yeniden fiziksel belleğe yüklenir mi? İşte işletim sistemleri bir program ikinci kez çalıştırıldığında yeni yaratılan prosesin sayfa tablosunda sayfa girişlerini mümkün olduğunca çalışmakta olan programın fiziksel sayfalarına yönlendirir. Yani ikinci kez çalıştırılan program onun ilk kopyasıyla aynı fiziksel sayfaları görüyor durumuna getirilir. Pekiyi bu durumda proseslerden biri fiziksel sayfaya yazma yaptığında diğeri bundan etkilenmeyecek midir? İşte aynı fiziksel sayfayı gösteren proseslerde proseslerden biri ortak sayfaya yazma yapmaya çalıştığı anda işletim sistemi o noktada devreye girip o fiziksel sayfanın gerçek bir kopyasını çıkartıp fiziksel sayfaları birbirinden ayırmaktadır. Bu sürece una İngilizce "copy on write" denilmektedir. Böylece bir proses ikinci kez çalıştırıldığında önce yeni çalıştırılan eski çalıştırılmış olanla aynı sayfaları kullanır. Gerektiğinde bunlar birbirlerinden ayrılmaktadır. Gerçekten de bir programda aslında hiç değişiklik yapılmayan pek çok fiziksel sayfa bulunabilmektedir. Örneğin program kodlarının bulunduğu, string'lerin bulunduğu fiziksel sayfalar aslında hiç değiştirilmezler. Bu sayfaların farklı prosesler tarafından kullanılmasında bir sorun oluşmaz.



Peki işletim sistemi programın ne kadarını fiziksel belleğe yüklemektedir? Prosesin fiziksel bellekteki kısmına İngilizce "resident set" ya da "working set" denilmektedir. Bu da işletim sistemlerini yazanların kullandıkları algoritmaya bağlı olarak değişmektedir. İşletim sistemlerinin en zor gerçekleştirilen alt sistemlerinden biri "bellek yöneticisi (memory management)" bölümüdür.

Proseslerin Çevre Değişkenleri (Environment Variables)

Çevre değişkenleri kavramı modern işletim sistemlerinin hemen hepsinde bulunmaktadır. Çevre değişkeni aslında bir anahtar için bir değer karşı getirildiği bir çifti belirtir. Anahtar da değer de birer yazıdır. Çevre değişkeni denildiğinde genellikle bu anahtar kastedilmektedir. Örneğin anahtar (yani çevre değişkeni) "City" olsun. Buna karşı gelen değer "New York" olabilir. Ya da anahtar "Count" olsun. Buna karşı gelen değer "123" olabilir. Böylece her prosesin bir çevre değişken listesi vardır. Her ne kadar böyle bir veri yapısı tamamen manuel olarak oluşturulabilse de çevre değişkenlerinin organizasyonu ve kullanımı çekirdek düzeyinde yapılmaktadır ve taban bir kavramdır. Yani prosesin hiçbir şeyi yokken bile çevre değişkenleri kullanılabilir biçimde hazır bulunmaktadır. Ayrıca bazı çevre değişkenleri bizzat çekirdek tarafından da kullanılmaktadır (PATH çevre değişkeni gibi). Çekirdek tarafından kullanılan bazı çevre değişkenleriyle ilerleyen konularda karşılaşacağız.

Programcı çevre değişkenleriyle ilgili dört şeyi yapabilir:

- 1) Yeni bir çevre değişkenini değerle birlikte prosesin çevre değişken listesine ekleyebilir.
- 2) Bir çevre değişkenini prosesin çevre değişken listesinden silebilir.
- 3) Bir çevre değişkenine karşılık gelen değeri elde edebilir
- 4) Prosesin tüm çevre değişken listesini elde edebilir.

Çevre değişkenleri prosese özgüdür. Yani her prosesin ayrı bir çevre değişken listesi vardır. Genel olarak çevre değişkenleri pek çok işletim sisteminde üst procesten alt prosese aktarılmaktadır. Yani üst proses bir prosesi yarattığında üst prosesin çevre değişken listesi alt prosese aktarılmaktadır (tabii bu bir kopyalama gibidir. Bu kopyalamadan sonra üst ve alt proseslerin çevre değişken listesi bağımsız olur.) Yani üst proses alt prosesi yaratırken kendi çevre değişkenleri alt prosese aktarılır. Bundan sonra artık üst prosese yeni bir çevre değişkeni eklenirse bu alt proseste bulunmayacaktır. Benzer biçimde alt prosese yeni bir çevre değişkeni eklenirse bu da üst proseste bulunmayacaktır.

Peki prosesin çevre değişken listesi nerede tutulmaktadır? Bazı sistemlerde çevre değişkenleri Proses Kontrol Bloğunda tutulurken, modern sistemlerin çoğunda (örneğin Windows, Linux ve MAC OS X sistemlerinde) prosesin çevre değişkenleri prosesin bellek alanında tutulmaktadır.

Windows sistemlerinde çevre değişkenlerinin büyük-harf küçük harf duyarlılığı yoktur. Ancak UNIX/Linux ve MAC OS X sistemlerinde çevre değişkenlerinin büyük harf-küçük harf duyarlılığı vardır. Yani örneğin Windows sistemlerinde "city" çevre değişkeni ile "CITY" çevre değişkeni aynı çevre değişkenidir. Halbuki UNIX/Linux ve MAC OS X sistemlerinde bu çevre değişkenleri birbirlerinden farklıdır.

Prosesin çevre değişkeni verildiğinde onun değerini bize veren getenv isimli standart bir C fonksiyonu vardır:

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

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

Fonksiyon parametre olarak çevre değişkeninin ismini alır, geri dönüş değeri olarak bize onun değerinin bulunduğu yerin adresini verir. Bu adresteki yer statik olarak tahsis edilmiştir. Programcı bu alanı boşaltmaya çalışmamalıdır. getenv fonksiyonu başarısız olabilir. Bu durumda NULL adrese geri döner. Örneğin:

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

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

```

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

if ((value = getenv(argv[1])) == NULL) {
    fprintf(stderr, "environment variable not found!\n");
    exit(EXIT_FAILURE);
}

puts(value);

return 0;
}

```

Ayrıca getenv standart C fonksiyonunun dışında Windows sistemlerinde GetEnvironmentVariable isimli API fonksiyonu aynı işlemi yapmaktadır (zaten Windows altında getenv standart C fonksiyonu bunu çağırıyor):

```

DWORD WINAPI GetEnvironmentVariable(
    LPCTSTR lpName,
    LPTSTR lpBuffer,
    DWORD nSize
);

```

Fonksiyonun birinci parametresi çevre değişkenini (yani anahtarı) alır. İkinci parametresi onun değerinin yerleştirileceği dizinin adresini belirtir. Windows sistemlerinde bir çevre değişkeni null karakter dahil olmak üzere en fazla 32767 karakter olabilmektedir. Fonksiyonun üçüncü parametresi ikinci parametrede belirtilen dizinin uzunluğunu almaktadır. Yani bu parametre null karakter dahil olmak üzere çevre değişkeninin kaplayacağı toplam alanı belirtir. Başarı durumunda fonksiyon diziyeye yerleştirilen karakter sayısı (null karakter dahil değil) geri döner. Eğer üçüncü parametrede belirtilen değer çevre değişkeninin değerinin karakter uzunluğundan küçükse bu durumda fonksiyon null karakter dahil olmak üzere çevre değişkeninin değerinin karakter uzunluğuyla geri döner. Bu durumda fonksiyon ikinci parametresiyle verilen diziyeye bir yerleştirme yapmaz. Fonksiyonun üçüncü parametresi 0 girilebilir. Bu durumda ikinci parametre NULL adres olarak verilebilir. Böylece programcı isterse çevre değişkeninin değerinin yerleştirileceği alanın uzunluğunu elde edip onu dinamik biçimde tahsis edebilir. Örneğin:

```

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

int main(void)
{
    char *value;
    DWORD result;

    if ((result = GetEnvironmentVariable("PATH", NULL, 0)) == 0) {
        fprintf(stderr, "cannot get environment variable!..\n");
        exit(EXIT_FAILURE);
    }
    if ((value = (char *)malloc(result)) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    if ((result = GetEnvironmentVariable("PATH", value, result)) == 0) {
        fprintf(stderr, "cannot get environment variable!..\n");
        exit(EXIT_FAILURE);
    }
    puts(value);

    free(value);

    return 0;
}

```

UNIX/Linux ve MAC OS X sistemlerinde çevre değişkeninin değerini elde eden ek bir POSIX fonksiyonu yoktur. Bu sistemlerde zaten tüm standart C fonksiyonlarının aynı zamanda bir POSIX fonksiyonu kabul edilmektedir.

Prosesin çevre değişken listesine yeni bir ekleme yapmak için kullanılacak bir standart C fonksiyonu yoktur. Bunun için UNIX/Linux sistemlerinde setenv ve putenv POSIX fonksiyonları Windows sistemlerinde SetEnvironmentVariable API fonksiyonu kullanılmaktadır. setenv fonksiyonunun prototipi şöyledir:

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

```
int setenv(const char *name, const char *value, int overwrite);
```

Fonksiyonun birinci parametresi çevre değişkenini, ikinci parametresi ise bunun değerini belirtir. Üçüncü parametre sıfır ya da sıfır dışı bir değer olarak girilir. Eğer bu parametre sıfır girilirse bu çevre değişkeni zaten varsa çevre değişkeni içerisindeki değer değişmez. Eğer bu parametre sıfır dışı olarak girilirse eski değer silinir, yerine yenisi gelir. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner. Örneğin:

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

int main(void)
{
    char *value;

    if (setenv("CITY", "Istanbul", 1) < 0) {
        perror("setenv");
        exit(EXIT_FAILURE);
    }

    if ((value = getenv("CITY")) == NULL) {
        perror("getenv");
        exit(EXIT_FAILURE);
    }

    puts(value);

    return 0;
}
```

putenv fonksiyonunun prototipi de şöyledir:

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

```
int putenv(char *string)
```

Fonksiyonun parametresi "anahtar=değer" biçiminde tek bir yazı alır. İlgili çevre değişkeni varsa üstüne yazar. Başarı durumunda sıfır, başarısızlık durumunda -1 değerine geri döner.

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

int main(void)
{
    char *value;

    if (putenv("CITY=Istanbul") < 0) {
        perror("setenv");
        exit(EXIT_FAILURE);
    }

    if ((value = getenv("CITY")) == NULL) {
        perror("getenv");
        exit(EXIT_FAILURE);
    }
}
```

```

}

puts(value);

return 0;
}

```

Microsoft da putenv POSIX fonksiyonunu _putenv ismiyle desteklemektedir. (Tabii bu da aslında SetEnvironmentVariable API fonksiyonu çağırıyor).

Windows'ta SetEnvironmentVariable API fonksiyonunun prototipi de şöyledir:

```

BOOL WINAPI SetEnvironmentVariable(
    LPCTSTR lpName,
    LPCTSTR lpValue
);

```

Fonksiyonun birinci parametresi çevre değişkenini, ikinci parametresi bunun değerini belirtir. Çevre değişkeni zaten varsa silinerek yeni değer yerleştirilir.

Bir çevre değişkenini silmek bazı sistemlerde mümkünken bazı sistemlerde mümkün değildir.

Peki bir prosesin tüm çevre değişkenleri nasıl elde edilmektedir? Windows sistemlerinde GetEnvironmentString isimli API fonksiyonu bize prosesin tüm çevre değişkenlerini ve onların değerlerini verir:

```

LPTCH WINAPI GetEnvironmentStrings(void);

```

Fonksiyonun geri dönüş değeri char türden bir göstericidir. Verilen yazı şu biçimdedir:

```

Anahtar1=Değer1\0Anahtar2=Değer2\0Anahtar3=Değer3\0Anahtar4=Değer4\0\0

```

GetEnvironmentStrings fonksiyonu başarısızlık durumunda NULL adresle geri döneri. Ancak fonksiyonun başarız olması için önemli bir nedne yoktur. Geri dönüş değeri kontrol edilmeyebilir.

Örneğin:

```

#include <stdio.h>
#include <windows.h>

int main(void)
{
    char *env;

    env = GetEnvironmentStrings();

    while (*env != '\0') {
        puts(env);
        env += strlen(env) + 1;
    }

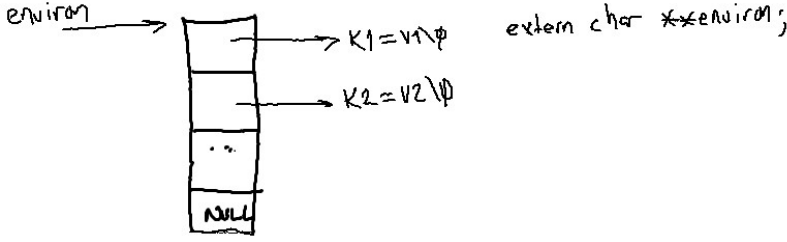
    return 0;
}

```

Windows'ta komut satırında "set" komutuna argüman verilmezse bu komut prosesin (cmd.exe'nin) tüm çevre değişken listesini yazdırmaktadır.

UNIX/Linux sistemlerinde kütüphane içerisinde tanımlanmış char ** türünden environ isimli bir global "göstericiyi gösteren gösterici" vardır. Bunun extern bildirimini maalesef hiçbir başlık dosyasında bulundurulmamaktadır. Bu nedenle

programcının extern bildirimini kendisinin yapması gerekmektedir. Bu göstericiyi gösteren gösterici anahtar değer çiftlerini tutan gösterici dizisinin başlangıç adresini göstermektedir. Dizinin sonunda NULL adres vardır:



Örneğin:

```
#include <stdio.h>

extern char **environ;

int main(void)
{
    int i;

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

    return 0;
}
```

UNIX/Linux sistemlerinde komut satırında "env" komutu kabuk prosesinin (/bin/bash prosesinin) o andaki çevre değişken listesini görüntüler.

Pekiyi prosesin çevre değişken listesi proses yaratıldığında nasıl oluşturulmaktadır? Windows'ta prosesler CreateProcess isimli API fonksiyonuyla yaratılır. İşte CreateProcess API fonksiyonunun 7'nci parametresi NULL geçilirse yaratılan alt prosese üst prosesin çevre değişkenleri aktarılmaktadır. Bu 7'inci parametreye NULL geçilmezse bu parametreyle belirtilen liste alt prosese aktarılır. Proseslerin yaratılması sonraki konuda ele alınmaktadır. CreateProcess API fonksiyonunun prototipi şöyledir:

```
BOOL WINAPI CreateProcess(
    LPCTSTR LpApplicationName,
    LPTSTR LpCommandLine,
    LPSECURITY_ATTRIBUTES LpProcessAttributes,
    LPSECURITY_ATTRIBUTES LpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID LpEnvironment,
    LPCTSTR LpCurrentDirectory,
    LPSTARTUPINFO LpStartupInfo,
    LPPROCESS_INFORMATION LpProcessInformation
);
```

Fonksiyonun bu 7'inci parametresi "anahtar=değer\0" çiftlerinden oluşan bir liste biçimindedir. Listenin sonunda ayrıca '\0' karakter olmalıdır (yani listenin sonunda iki tane '\0' bulunacaktır).

Aşağıda Windows sistemlerinde alt prosese çevre değişkenlerinin aktarımıyla ilgili bir örnek görüyorsunuz. Burada Proc1 ve Proc2 isimli iki ayrı program vardır. Proc1 Programı Proc2 programını çalıştırmaktadır. Tabii bu örneği Aşağıda Windows sistemlerinde alt prosese çevre değişkenlerinin aktarımıyla ilgili bir örnek görüyorsunuz. Bu örneği ileride daha iyi anlamlandıracaksınız:

```
/* Proc1.c */

#include <stdio.h>
```

```

#include <stdlib.h>
#include <Windows.h>

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    char cmdLine[] = "..\\Debug\\Proc2.exe";
    STARTUPINFO si = { sizeof(STARTUPINFO) };
    PROCESS_INFORMATION pi;
    char envs[] = "CITY=Eskisehir\0Country=Turkey\0";

    if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, envs, NULL, &si, &pi))
        ExitSys("CreateProcess");

    WaitForSingleObject(pi.hProcess, INFINITE);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

/* Proc2.c */

#include <stdio.h>
#include <Windows.h>

int main(void)
{
    char *envs;

    envs = GetEnvironmentStrings();

    while (*envs != '\0') {
        puts(envs);
        envs += strlen(envs) + 1;
    }

    return 0;
}

```

UNIX/Linux ve MAC OS X sistemlerinde fork işlemi sırasında üst prosesin çevre değişkenleri alt prosese aktarılmaktadır. exec işlemi sırasında da programcı isterse çalıştıracığı programın yeni bir çevre değişken listesi ile çalıştırılmasını sağlayabilir. Yani exec işlemi sırasında prosesin çevre değişken listesi atılarak yerine yenisi oluşturulabilmektedir. Bunun için exec fonksiyonlarının e'li versiyonları kullanılır. exec işlemi sonraki bölümde ele alınmaktadır.

O halde biz UNIX/Linux ve MAC OS X sistemlerinde kabuk üzerinden bir programı çalıştırdığımızda kabuğun çevre değişkenleri bizim programımıza aktarılır. Peki kabuğun çevre değişkenleri nasıl oluşturulmuştur? Bu konu ileride ele alınacaktır.

UNIX/Linux ve MAC OS X sistemlerinde kabuk üzerinde,

```
export Anahtar=Değer
```

yazılır ENTER tuşuna basılırsa "Anahtar" isimli çevre değişkeni yaratılır ve buna "Değer" r değeri atanır. (Eğer export komutu belirtilmezse bu değişken kabuk dilinin değişkeni olur. Fakat kabuk bunu prosesin çevre değişken listesine eklemez). Ayrıca değişkeni bir kez export etmek yeterlidir. Eğer değişkenin değeri değiştirilecekse artık yalnızca Anahtar=Değer sentaksını kullanabiliriz. Tabii bu biçimde kabuk prosesine (bash) eklenen çevre değişkeni o kabuktan çıkılınca yol olacaktır. Başka kabuk çalıştırmalarında da bu çevre değişkeni görülmeyecektir. Bunun nasıl kalıcı hale getirileceği ileride ele alınmaktadır.

UNIX/Linux ve Mac OS X sistemlerinde kabuk üzerinde \$<çevre değişken ismi> yazılırsa kabuk bize onun değerini verir. Başka bir deyişle \$<çevre değişkeni> ifadesi yerine kabuk onun değerini yerleştirmektedir. Örneğin:

```
export Name=Ali
echo $Name
```

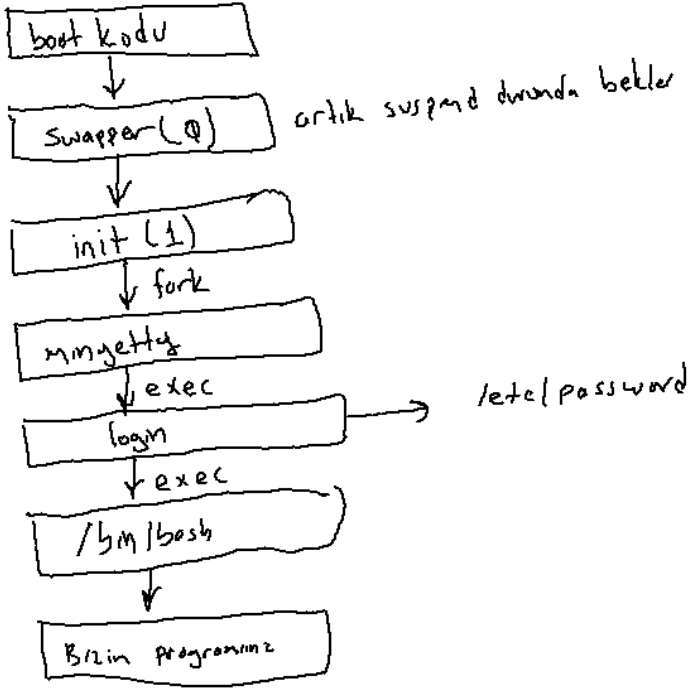
O halde biz kabuk programına bir çevre değişkeni ekleyip kendi programımızı çalıştırsak o değişken bizim programımıza da geçecektir.

Peki UNIX/Linux ve MAC OS X sistemlerindeki kabuk programlarının sahip olduğu çevre değişkenleri nereden gelmektedir? Örneğin kursun yürütüldüğü makinede Linux terminalinde "env" komutu uygulandığında elde edilen kabuğun çevre değişken listesinin bir bölümü şöyledir:

```
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
GNOME_TERMINAL_SERVICE=:1.65
XDG_SEAT=seat0
SHLVL=1
LANGUAGE=en_US
LC_TELEPHONE=tr_TR.UTF-8
GDMSESSION=cinnamon
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=csd
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
XDG_RUNTIME_DIR=/run/user/1000
XAUTHORITY=/home/csd/.Xauthority
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_CONFIG_DIRS=/etc/xdg/xdg-cinnamon:/etc/xdg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
LC_IDENTIFICATION=tr_TR.UTF-8
CINNAMON_VERSION=4.0.8
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
SESSION_MANAGER=local/csd-vm:@/tmp/.ICE-unix/1437,unix/csd-vm:/tmp/.ICE-unix/1437
LESSOPEN=| /usr/bin/lesspipe %s
_=/usr/bin/env
csd@csd-vm:~$
```

Tüm bu çevre değişkenleri kabuğa neredne gelmiştir? Kabuk bunları setenv ya da putenv ile kendisi mi yaratmıştır? İşte kabuk bu çevre değişkenlerinin bir bölümünü üset proseslerden almakta bir bölümünü de kendisi yaratmaktadır.

İşletim sistemleri boot edildiğinde genellikle tek bir proses ile çalışmaya başlar. Diğer prosesler proses yaratan sistem fonksiyonlarıyla oluşturulmaktadır. Yani sistemdeki prosesler aslında tek bir atadan gelmiş durumdadır. Tipik bir UNIX/Linux sisteminde boot işleminden sonra prosesler aşağıdaki gibi yaratılmaktadır:



İşte tipik bir UNIX/Linux sisteminde kabul prosesine gelene kadar yukarıda ismi geçen bazı prosesler yaratılmaktadır. O prosesler bazı çevre değişkenlerini setenv ya da putenv fonksiyonlarıyla oluştururlar bunlar da alt prosese katarılarak kümülatif bir birikim oluşur. Ayrıca yukarıdaki prosesler birtakım script dosyalarına ve konfigürasyon dosyalarına bakarak birtakım şeyleri de ayarlamaktadır. Sistem yöneticisi de bu dosyalara uygun şeyleri yazarak açılış sırasında sistemi konfigüre edebilir. Bu script dosyalarına aynı zamanda sistem yöneticisi birtakım çevre değişkenlerini de yazabilmektedir. Böylece bu çevre değişkenleri de ilgili prosese dolaylı biçimde eklenmiş olur.

Pekiye UNIX/Linux ve MAC OS X sistemlerinde sistem boot edilirken hangi proses hangi script dosyalarına bakmaktadır? İşte bu da sistemden sisteme, çeşitli dağıtımlara ve dağıtımlarda kullanılan paketlere göre değişebilmektedir. Genel olarak sistem reboot edildiğinde çalıştırılan dosyalara "startup files" ya da "statup scripts" denilmektedir. Siz de Internet üzerinden ya da birtakım kitaplardan hangi proseslerin hangi dosyaları çalıştırdığını öğrenebilirsiniz.

Linux dağıtımlarında pek çok paketin alternatif gerçekleştirmeleri vardır. Örneğin init programı için popüler iki paket kullanılmaktadır: sysvinit ve upstart. Örneğin Suse gibi sistemler klasik sysvinit paketini kullanırken, Ubuntu ve türevleri (örneğin Mint gibi) upstart paketini kullanmaktadır. sysvinit paketindeki init programı /etc/inittab dosyasına, upstart paketindeki init programı /etc/init/init.conf dosyasına bakmaktadır. Bu dosyaların formatlarına ilişkin bilgi man sayfalarından elde edilebilir.

UNIX/Linux sistemleri boot edilirken login prosesi kullanıcıyı sisteme sokmaktadır. Yani ekranda "user name" ve "password" yazısını çıkartan ve bu doğrulamayı yapan "login" isimli prosestir. Tipik bir UNIX/Linux sisteminde login prosesi kullanıcıdan "user name" ve "password" bilgileri istedikten sonra girilen parolayı /etc/passwd dosyasına (ya da /etc/shadow dosyasına) bakarak doğrular. Eğer girilen parola doğruysa login prosesi bu dosya içerisinde belirtilen programı çalıştıracaktır. /etc/passwd dosyası sistemdeki tüm kullanıcıların kayıtlarının bulunduğu bir text dosyadır. Bu dosyada her bir satır ':' lerle ayrılmış alanlardan oluşur. Örneğin:

```

csd:x:1000:1000:CSD,,,:/home/csd:/bin/bash
student:x:1001:1000:CSD,,,:/home/student:/bin/bash
usermetrics:x:115:124:User Metrics:/var/lib/usermetrics:/bin/false
clickpkg:x:116:125:/:nonexistent:/bin/false
  
```

Görüldüğü gibi her kullanıcı için 7 alan vardır. İlk alanda kullanıcının ismi, ikinci alanda parolası bulunmaktadır. Parola alanında 'x' karakteri varsa bu 'x' karakteri şifrenin /etc/shadow dosyasında olduğunu gösterir. Sonra user id, sonra da grup id değerleri gelmektedir. Son alanda login başarılıysa çalıştırılacak program bulunmaktadır. Sondan bir önceki alan ise ilgili program çalıştırıldığında onun başlangıç çalışma dizinini belirtmektedir.

login prosesi bazı çevre değişkenlerini prosese dahil etmektedir. Örneğin USER, HOME, PATH, SHELL gibi.

login programı kabuk programını çalıştırır (tipik olarak /bin/bash). Kabuk tarafından da pek çok çevre değişkeni listeye eklenmektedir. Örneğin: PWD, LINES, COLUMNS, LANG gibi...

Peki biz bir çevre değişkenlerinin kalıcılığını nasıl sağlayabiliriz? İşte UNIX/Linux ve MAC OS X sistemleri başlatılırken çalıştırılan çeşitli script dosyalarının içerisine çevre değişken yaratım komutlarını eklersek sistem açıldığında otomatik olarak o çevre değişkenleri proses ağacına dahil edilmiş olur. Örneğin çevre değişkenleri için tipik olarak /bin/bash kabuğunun çalıştırdığı script dosyaları tercih edilmektedir. bash isimli kabuk programı "interaktif login" biçiminde, "interaktif login olmayan" biçimde ya da "interaktif olmayan" biçimde çalıştırılabilir. (Buradaki "login" sözcüğü shell'in "user name" ve "password" sormasını belirtmektedir. Örneğin biz pencere yöneticisinden terminali açıyorsak bu "interaktif login olmayan" shell biçimindedir.) Eğer bash "interaktif login" biçiminde çalıştırılırsa bu durumda önce /etc/profile dosyasını çalıştırır. Sonra ~/.bash_profile, ~/.bash_login, ~/.profile dosyalarından bu sırada hangisi ilk varsa onu çalıştırır. Çıkarken de bash ~/.bash_logout dosyasını çalıştırmaktadır. Eğer bash "interaktif login olmayan" biçimde çalıştırılmışsa bu durumda yalnızca ~/.bashrc dosyasını çalıştırmaktadır. Ancak pek çok sistemde sistem tarafından oluşturulmuş olan ~/.bash_profile dosyası ~/.bashrc dosyasını da kendi içerisinde çalıştırmaktadır. Nihayet bash "interaktif olmayan" biçimde çalıştırılmışsa herhangi bir dosyayı çalıştırmamaktadır. (Buradaki '~' karakteri kullanıcının home dizini anlamına gelmektedir.)

Örneğin ~/.bashrc dosyasına aşağıdaki satırı yerleştirerek PATH çevre değişkenine ekleme yapabiliriz:

```
export PATH=$PATH:/home/csd/Study
```

UNIX/Linux sistemlerinde başı '.' karakteri ile başlayan dosya isimlerinin "gizli (hidden) dosya" anlamına geldiğini anımsayınız. Bu dosyalar normal ls komutunda ya da GUI dosya görüntüleme programları tarafından default durumda görüntülenmezler. Komut satırında gizli dosyaları görüntülemek için ls komutunda "-a (all)" seçeneği kullanılabilir.

Windows sistemlerinde de genel mantık aynıdır. Windows'ta masaüstünü "EXPLORER.EXE" isimli proses temsil eder. Windows'ta da bir login prosesi vardır. Bu proses login işlemini yapar. "EXPLORER.EXE" prosesi bu proses tarafından çalıştırılır. Biz "Denetim Masası/Sistem/Gelişmiş Sistem Ayarları/Ortam Değişkenleri" menüsünden yeni çevre değişkenleri ekleyip, mevcut olanları değiştirebiliriz. Tabii bu değişiklik o anda çalışmakta olan proseslere yansımaz.

Windows'ta komut satırında (cmd.exe) çevre değişkeni girmek için SET komutu kullanılmaktadır. Örneğin:

```
SET City=Ankara
```

Yine komut satırında çevre değişkenlerinin anahtarı verildiğinde değeri elde etmek için %Key% ifadesi kullanılır. Örneğin:

```
SET PATH=%PATH%;c:\Study
```

Ayrıca Windows sistemlerinde SET komutunun yanına bir şey yazılmazsa kabuk programının (cmd.exe) tüm çevre değişkenleri stdout dosyasına yazdırılır. Yani Windows'taki "set" komutu UNIX/Linux ve MAC OS X sistemlerindeki "env" komutu gibidir.

Çevre Değişkenlerine Neden Gereksinim Duyulmaktadır?

Çevre değişkenlerine işletim sistemi düzeyindeki global değişkenler gibi bakılabilir. Programlar birtakım dosyaları vs. bazı çevre değişkenlerinin belirttiği yerde arayabilirler. Örneğin bir veritabanı dosyası DATABASE isimli bir çevre değişkeninin belirttiği dizinde aranabilir. Bu durumda programcı önce getenv fonksiyonuyla bu çevre değişkeninin değerini alır ve dosyayı o dizinde arayabilir. Veritabanı dosyası başka dizine yerleştirmek istenirse tek yapılacak şey bu çevre değişkeninin değerini değiştirmektir. Örneğin pek çok C derleyicisi <...> biçiminde include edilmiş dosyaları INCLUDE isimli bir çevre değişkeninin belirttiği dizinde aramaktadır. Eğer bu çevre değişkeni set edilmemişse default bir dizin kullanılmaktadır. Ya da örneğin Java'da çeşitli jar dosyaları ve derlenmiş dosyalar CLASSPATH isimli bir çevre

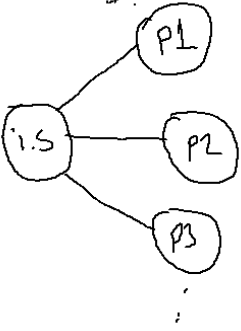
değişkenine bağlı olarak aranmaktadır. Bazı çevre değişkenlerini ise doğrudan işletim sisteminin kendisi de kullanmaktadır. (Örneğin PATH gibi, LD_LIBRARY_PATH gibi).

Bazı çevre değişkenleri işletim sisteminin içinde bulunduğu durum hakkında bilgi veriyor olabilir. Örneğin Windows hangi dizine yüklenmiştir? O anda kullanılan yerel ayarlar nelerdir? Kullanıcının ismi nedir? Home dizini hangisidir vs.

Çevre değişkenleri proseslerarası haberleşmede de bazen kullanılabilir. Örneğin, üst proses belli bir çevre değişkenini set ederek alt prosesi çalıştırır. Alt proses böylece o çevre değişkeninden bilgi alabilir. Böylesi basit bilgi aktarımı için herhangi bir proseslerarası haberleşme yönteminin kullanılmasına gerek yoktur.

İşletim Sistemlerinde Zaman Paylaşımı Çalışma

Eskiden thread'ler yokken her prosesin tek bir akışı vardı. İşte örneğin C programlama dilinde tipik olarak proses akışı main fonksiyonundan başlatılmaktadır. Pekiyi çok prosesli işletim sistemlerinde tek bir CPU olduğu durumda birden fazla program nasıl aynı anda çalışabilmektedir? Aslında programlar tek CPU'lu bir sistemde aynı anda çalışmamaktadır. Çalışma zaman paylaşımı (time sharing) bir biçimde yapılmaktadır. Zaman paylaşımı çalışmada bir proses CPU'ya atanır. Bir süre çalıştırılır. Sonra çalışmasına ara verilir. Diğeri yine bir süre çalıştırılır. Çalışma bu biçimde devam ettirilir. Prosesler kaldıkları yerden hep böyle çalışmaya devam ettirilirler. Dışarıdan bakıldığında sanki bunlar aynı anda çalışıyormuş gibi bir izlenim edinilmektedir. Fakat aslında "biraz ondan biraz bundan" biçiminde "zaman paylaşımı" bir çalışma söz konusudur.



Bir prosesin parçalı çalışma süresine quanta süresi ya da quantum denilmektedir. Örneğin bazı Windows sistemlerinde tipik quanta süresi 20 ms. civarındadır. UNIX/Linux sistemlerinde 60 ms. gibi quanta süreleri tercih edilmektedir. Quanta süreleri işletim sisteminden işletim sistemine hatta versiyondan versiyona değişebilir. Bu süre çekirdeğin çalışma biçimiyle uyumlu olması gerektiğinden genellikle sistem yöneticisi tarafından ayarlanamaz.

Bir prosesin quanta süresi dolduğunda çalışmasına ara verilip diğer prosese geçilmesi sürecine "proseslerarası geçiş (process switch/task switch)" ya da daha genel olarak "bağlam geçişi (context switch)" denilmektedir. Şüphesiz proseslerarası geçiş sırasında bir zaman kaybı oluşmaktadır.

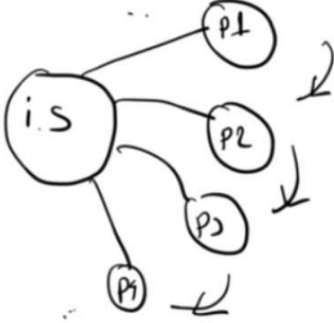
Pekiyi quanta süresi çok uzun seçilirse bunun etkileri ne olur? Quanta süresi çok uzun seçilirse interaktivite azalır. Fakat birim zamanda yapılan gerçek iş miktarı (throughput) yükselir. Quanta süresi çok kısa olursa bu durumda tersine interaktivite çok yüksek olur fakat birim zamanda yapılan iş miktarı (throughput) düşer. Çünkü proseslerarası geçiş işleminin de belli bir maliyeti vardır.

Windows ve UNIX/Linux sistemleri "preemptive" sistemlerdir. Bir prosesin quanta süresi dolduğunda akışın -o anda nerede bulunulursa bulunulsun- kopartılarak proseslerarası geçiş yapıldığı sistemlere "preemptive" sistemler denilmektedir. Fakat preemptive olmayan çok prosesli sistemler de vardır. Bu tür sistemlere İngilizce "non-preemptive" sistemler ya da "cooperative multitask" sistemler denilmektedir. Örneğin Windows 3.1, PalmOS gibi sistemler böyleydi. Bu sistemlerde proses kendi isteğiyle akışı bırakır ve proseslerarası geçiş oluşur. Bu tür sistemlerin gerçekleştirilmesi daha kolaydır fakat bunlarda bir proses sistemi tekeli altına alabilmektedir.

Peroseslerarası geçiş donanım kesmeleri yoluyla yapılmaktadır. Örneğin PC mimarisinde tipik olarak IRQ0'ı tetikleyen zamanlayıcı (Intel 8254) kurulur. Bu devre zaman dolduğunda CPU'ya sinyal gönderir. Böylece CPU kesme konumuna

geçer ve proseslerarası geçişi yapar. Artık pek çok mikroişlemcinin kendi içerisinde böyle zamanlayıcı devreleri zaten hazır bulunmaktadır. Örneğin Intel işlemcilerine böyle zamanlayıcı devreleri eklenmiştir.

İşletim sistemlerinin hangi prosesin CPU'ya ne zaman atanacağını belirleyen ve bu işlemi gerçekleştiren alt sistemlerine çizelgeleyici (scheduler) denilmektedir. Proseslerin hangilerinin ne zaman çalıştırılacağı konusunda kullanılan algoritmalara da "çizelgeleme algoritmaları (scheduling algorithms)" denir. En basit ve adil bir çizelgeleme algoritması "döngüsel çizelgeleme (round robin scheduling)" denilen algoritmadır. Bu algoritmada her proses sırasıyla belli bir süre çalıştırılır.

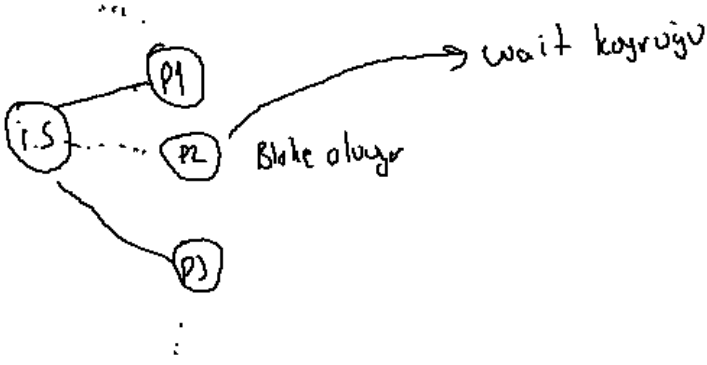


Proseslerin çizelgelenmesi birden fazla CPU ya da çekirdek olduğu durumda çok benzer yapılmaktadır. Bu durum birden fazla gişenin bulunduğu servis sistemlerine benzetilebilir. İşletim sistemleri birden fazla CPU ya da çekirdeğin bulunduğu durumlarda işlemciler ya da çekirdekler için genellikle ayrı kuyruklar oluştururlar. Yine her işlemci ya da çekirdek zaman paylaşımli biçimde çalışmaktadır. Tabii bu durum toplam performansı yükseltir. Örneğin 4 çekirdekli bir sistemde 20 proses bulunuyor olsun. Bu 20 proses 5'erli biçimde kendi aralarında döngüsel bir kuyruk oluşturabilirler. Tabii çizelgeleme alt sisteminin oldukça fazla ayrıntıları vardır. Örneğin buradaki gibi 4 çekirdekli bir sistemde 20 proses çalışırken bu çekirdeklerden birine ilişkin kuyruktaki proseslerin hepsi sonlansa ne olacaktır? Şüphesiz diğer kuyruklardaki bazı proseslerin bu boşalan çekirdeğin kuyruğına aktarılması söz konusu olabilir. (Örneğin siz de bir markette kasada beklediğınızı düşünün. Yandaki kasa boşalsa siz o kasada yine beklemeye devam eder misiniz?) Bazı işletim sistemleri çok işlemcili ya da çok çekirdekli sistemler için global tek bir kuyruk oluşturmaktadır. Her çekirdeğin quanta süresi dolduğunda kuyruktaki sıradaki proses ona atanmaktadır. Tabii genel olarak kuyruktaki sıradaki prosesin o anda quanta süresini bitirmiş olan CPU ya da çekirdeğe atanması her zaman iyi bir yöntem olmayabilir. Çünkü CPU ya da çekirdeklerin içsel cache'leri vardır. Bu cache'lerden maksimum faydalanabilmek için bir prosesin bir önceki quanta için atanmış CPU'ya atanması daha uygun olabilmektedir. Burada görüldüğü gibi çizelgeleme algoritmalarının göz önüne alınması gereken bazı optimizasyon problemleri vardır.

Anahtar Notlar: CPU'lar belli bir yıla kadar sürekli hızlandı (Moore yasası). Fakat belli bir düzeyden sonra artık fiziksel sorunlardan dolayı CPU'ların hızlandırılması çok zorlaştı. Bu durumda hızlanmayı sağlamak için çok çekirdekli işlemciler yapılmaya başlandı. İki çekirdekli bir CPU aslında iki ayrı CPU'nun tek entegre devre halinde üretilmesidir. Zaten eskiden birden fazla CPU'nun takılabildiği board'lar vardı. Çok çekirdekli sistemler önce "hyper threading" teknolojisi ile başlatılmıştır. Hyper threading'li bir CPU gerçek anlamda iki CPU değildir. Bunun bazı parçalarından iki tane vardır. Fakat bazı parçaları bir tanedir. İşletim sistemleri "hyper threading"li CPU'ları iki CPU gibi görmektedir. Ancak bu CPU'lar gerçek iki CPU (yani gerçek iki çekirdek) kadar performanslı değildir. Şimdilerde Intel hem birden fazla çekirdek hem de her çekirdekte hyper threading uygulayabilmektedir. Örneğin bu yazının yazıldığı bilgisayarda Intel'in i7-4790S numaralı işlemcisi kullanılmaktadır. Bu entegre devre 4 gerçek çekirdeğe (yani farklı CPU'ya) sahiptir ve her çekirdek ayrıca "hyper threading" yapılmıştır. Böylece Windows ve Linux bu entegre devreyi sanki 8 işlemcili bir sistem gibi görmektedir.

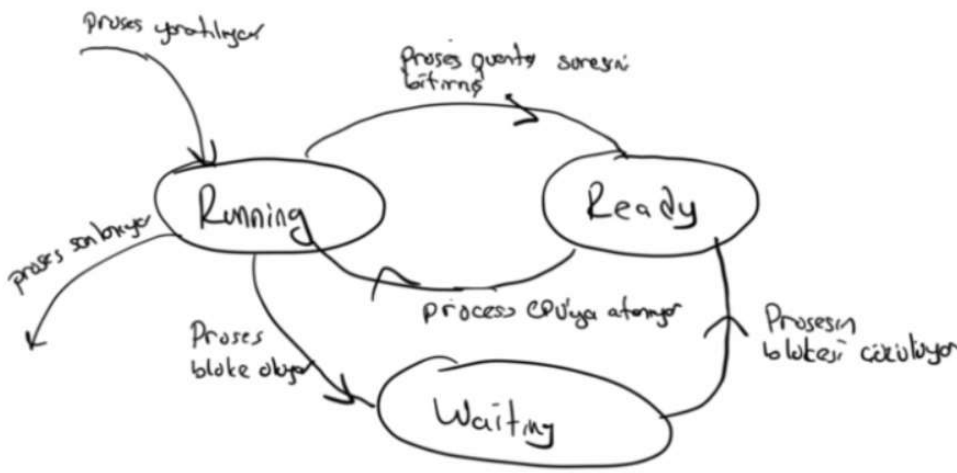
Bloke Kavramı

Bir proses çalışırken dışsal bir olayı başlattığında (örneğin disk işlemi, klavye okuması, soket okuması vs. gibi) işletim sistemi prosesi CPU zamanı harcanmasın diye geçici olarak çizelge dışına çıkartır ve olayı kendisi arka planda kesme (interrupt) tekniğıyle izler. Bu sırada sanki proses hiç çizelgede değilmiş gibi bekletilir. İşlem bittiğinde işletim sistemi yeniden prosesi çizelgeye yerleştirir. Sonuçta proses yine olay bitene kadar beklemiş olur fakat boşuna CPU zamanı harcanmamıştır. İşte bir prosesin bir işlem bitene kadar ya da gerçekleşene kadar çizelge dışına çıkartılarak bekletilmesine prosesin bloke olması (blocking) denilmektedir. Örneğin Sleep gibi fonksiyonlar da aslında meşgul bir döngüde sürekli bekleme yapmazlar. Bunlar da bloke edilerek bekletilirler. Böylece bir sistemde yüzlerce proses olabilir fakat bunların çoğu belli bir olayı bekler durumdadır. Yani çok az proses aktif olarak belli bir anda CPU'yu kullanma eğilimindedir.



Aslında prosesin bir quanta süresinin tamamını harcaması çok nadirdir. Örneğin quanta süresi 20 ms. olsun. Pek çok proses daha birkaç milisaniye içerisinde bir IO olayına girer ve uzun süre bekler.

Bir prosesin yaşam döngüsü tipik olarak şöyledir:



Burada Running prosesin o anda CPU'ya atanmış olduğunu gösteriyor. Proses quanta süresini normal olarak doldurduğunda çizelgede bekletilir. Bu durum şekilde "Ready" ile temsil edilmektedir. Proses çalışırken bloke olursa çizelgeden çıkarılır. Bu durum da şekilde "Waiting" ile belirtilmiştir.

Bir proses bloke olduğunda işletim sistemi onu ismine "wait kuyruğu (wait queue)" denilen bir kuyrukta bekletir. Sonra olay gerçekleşince oradan onu alarak yeniden çizelge kuyruğuna koyar. Genellikle işletim sistemleri her olay için ayrı birer wait kuyruğu oluşturmaktadır.

Peki bir proses zamanının ne kadarını CPU'da ne kadarını wait kuyruğunda harcamaktadır? Tabii bu prosesinden prosesine değişir. Genel olarak çok CPU kullanan fakat az IO işlemi yapan proseslere "CPU yoğun (CPU bound)" prosesler, çok IO işlemi yapıp az CPU kullanan proseslere de "IO yoğun (IO bound)" prosesler denilmektedir. Örneğin matematiksel hesaplamalar yapan bir proses CPU yoğun, veritabanı işlemi yapan bir proses IO yoğundur. Genellikle prosesler IO yoğun olma eğilimindedir. O halde bir sistemde yüzlerce proses olsa da aslında bunların çoğu uykuda yani wait kuyruğunda bekliyor durumdadır.

Bir prosesin CPU kullanım oranından bahsedilebilir. Peki bu oran nasıl hesaplanmaktadır? Değişik hesaplama yöntemleri söz konusu olabilmektedir. Örneğin bir procese verilen quanta süresinin o prosesin ortalama ne kadarını kullandığı iyi bir ölçüt olarak değerlendirilebilir. Örneğin prosesin toplam CPU'da harcadığı zaman ile wait kuyruklarında harcadığı zamanın toplamı ile de bir oran belirlenebilir. Bu durumda prosesin CPU kullanım oranı şöyle hesaplanabilir:

$$\frac{\text{CPU'da harcanan zaman}}{\text{CPU'da harcanan zaman} + \text{wait kuyruklarında harcanan zaman}}$$

Tabii ready durumunda geçen zaman (yani CPU'ya atanmamış fakat çizelgede bulunduğu zaman) dikkate alınmamıştır. Böylece örneğin bir proses çok az CPU zamanı kullanıp hemen uykuya dalıyorsa bu prosesin CPU kullanım oranı çok düşüktür. Bazen genel olarak CPU'nun kullanım oranından da bahsedildiği olur. Gerçekten de pek çok işletim sistemi böyle bir oranı kullanıcıya gösterebilmektedir:

Ad	%87 CPU	%52 Bellek	%0 Disk	%0 Ağ
023-ProcessTimeTest.exe (32 bit)	%14.1	0.5 MB	0 MB/sn	0 Mb/sn
023-ProcessTimeTest.exe (32 bit)	%14.1	0.5 MB	0 MB/sn	0 Mb/sn
023-ProcessTimeTest.exe (32 bit)	%14.1	0.5 MB	0 MB/sn	0 Mb/sn
023-ProcessTimeTest.exe (32 bit)	%14.0	0.5 MB	0 MB/sn	0 Mb/sn
Hizmetler ve Denetleyici uygula...	%1.5	2.9 MB	0 MB/sn	0 Mb/sn
VyprVPNService (32 bit)	%0.6	7.5 MB	0 MB/sn	0 Mb/sn
WMI Provider Host	%0.4	5.7 MB	0 MB/sn	0 Mb/sn
Task Manager	%0.3	14.9 MB	0 MB/sn	0 Mb/sn
Antimalware Service Executable	%0.1	55.7 MB	0.1 MB/sn	0 Mb/sn
Local Security Authority Process...	%0.1	3.6 MB	0 MB/sn	0 Mb/sn
Masaüstü Pencere Yöneticisi	%0.1	18.7 MB	0 MB/sn	0 Mb/sn
VMware Authorization Service (...)	%0.1	2.0 MB	0 MB/sn	0 Mb/sn
Microsoft Visual Studio 2015 (3...)	%0	107.4 MB	0.1 MB/sn	0 Mb/sn
System	%0	0.1 MB	0.1 MB/sn	0 Mb/sn

Genel CPU kullanım oranı CPU'nun boşa kaldığı (tüm proseslerin bloke olması dolayısıyla) zamana göre tespit edilebilir. Örneğin CPU zamanının 5'te birinde hiçbir şey yapmadan tüm prosesler bloke olduğundan dolayı bekliyorsa bu CPU'nun kullanım oranı %5 olacaktır. Yukarıdaki görsel 4 çekirdekli, 8 iş parçacıklı bir sisteme ilişkin olarak Windows işletim sistemi için verilmiştir. Windows burada zaman yüzdelerini tüm CPU'lar'ın ortalaması olarak hesaplamış olabilir. Ancak Windows'un Task Manager'daki bu hesaplamayı kesin olarak nasıl yaptığını herhangi bir dokümanda açıklamamıştır. Bu bilgiyi kaba bir fikir vermek amacıyla kullanıcıya göstermektedir. Aslında işletim sistemlerinde tüm prosesler bloke olsa bile işletim sisteminin bir proses (idle process) çalıştırılmaktadır. Bu da arka planda sistemi yormadan bazı küçük yardımcı işlemleri yapmaktadır.

Çok prosesli sistemlerde programın iki noktası arasında geçen zaman her çalıştırmada aynı olmayabilir. Prosesler zaman paylaşımli olarak çalıştırıldığına göre prosesin iki noktası arasında geçen zaman o anki sistem yüküne bağlı olarak değişebilmektedir. Örneğin aşağıdaki programda döngünün saniye cinsinden ne kadar sürede döndüğü ekrana yazdırılmıştır.

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

int main(void)
{
    long long int i;
    LARGE_INTEGER li1, li2, freq;
    __int64 result;

    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&li1);

    for (i = 0; i < 1000000000; ++i)
        ;
}
```



```

QueryPerformanceCounter(&li2);

result = li2.QuadPart - li1.QuadPart;
printf("%f\n", (double)result / freq.QuadPart);

return 0;
}

```

Bu programdan çok sayıda çalıştırıldığında sürenin uzadığını gözlemleyebilirsiniz. Çünkü bu durumda sistemin yükü artmakta ve belli bir prosese daha geç çalışma sırası gelmektedir.

Peki gerçekten her prosesin quanta süresi aynı mıdır? Yani sistemde belli haklara sahip olan daha öncelikli prosesler daha fazla quanta süresi kullanamazlar mı? İşte bu konu çizelgelemenin ayrıntılarıyla ilgilidir. Derneğimizde "UNIX/Linux Sistem Programlama" ve "Windows Sistem Programlama" kurslarında bu sistemlerdeki ayrıntılar üzerinde durulmaktadır. Ancak genel olarak sistemlerde proseslere birtakım öncelikler verilebilmekte bu da onlara daha fazla CPU kullanma hakkı verildiği anlamına gelmektedir.

Proseslerin Yaratılması

Bir prosesi yaratmak için işletim sistemlerinde sistem fonksiyonları bulunmaktadır. Windows sistemlerinde CreateProcess isimli API fonksiyonu, UNIX/Linux sistemlerinde fork fonksiyonu proses yaratmak için kullanılır.

Windows'ta CreateProcess API fonksiyonunun prototipi şöyledir:

```

BOOL WINAPI CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);

```

Fonksiyonun birinci parametresi çalıştırılacak program dosyasının yol ifadesini alır. Burada yol ifadesi mutlak ya da görel olarak verilebilir. Eğer dosyaya uzantı verilmemişse uzantısının ".exe" olduğu varsayılmaktadır. Fonksiyonun ikinci parametresi programın komut satırı argümanlarını belirtir. Komut satırı argümanları tek bir yazı olarak fonksiyona verilmektedir. (Sonra C derleyicilerinin başlangıç kodları (start up code) bu yazıyı boşluklardan parse ederek argv dizisini oluşturmaktadır.) İlk komut satırı argümanının programın yol ifadesi olması C'de zorunlu tutulmuştur. (Fakat işletim sistemlerinde genelinde böyle bir zorunluluk yoktur.) Kolaylık olsun diye şöyle bir seçenek de sunulmuştur: Eğer birinci parametre NULL geçilirse ikinci parametredeki ilk boşluksuz kısım sanki birinci parametredeki çalıştırılabilen dosyanın yol ifadesi gibiymiş gibi ele alınmaktadır. Dosya ismini bu yolla vermenin diğerinden bir farkı daha vardır. Bu yöntemde eğer dosya ismi hiç '\ ' karakteri içermiyorsa Windows onu sırasıyla bazı dizinlerde arar. Eğer buradaki dosya ismi '\ ' karakteri içeriyorsa Windows onu yol ifadesi ile belirtilen yerde arar fakat başka bir dizine bakmaz. Yine fonksiyonun birinci parametresi NULL geçilmezse Windows programı başka yerde aramamaktadır. Burada ikinci parametrenin const olmayan bir adres olduğuna dikkat ediniz. Biz ikinci parametreye bir string ifadesi vermemeliyiz. Çünkü CreateProcess bu adresteki bilgiyi saklayıp burayı tampon olarak kullanarak fonksiyon çıkışında yeniden orijinal yazıyı burada bırakır. Ama bu diziyi güncellemektedir. Halbuki string ifadeleri güncellenemez (C'de bir string ifadesinin güncellenmesinin tanımsız davranışa yol açtığını anımsayınız). Eğer birinci parametre NULL geçilirse ve ikinci parametredeki dosya ismi '\ ' içermiyorsa Windows dosyayı sırasıyla şu dizinlerde arar:

- 1) CreateProcess fonksiyonunu uygulayan programın .exe dosyasının bulunduğu dizin
- 2) CreateProcess fonksiyonunu uygulayan prosesin o andaki çalışma dizini
- 3) 32 Bit Windows System dizini (tipik olarak c:\windows\system32 dizini)
- 4) 16 bit Windows dizini (tipik olarak c:\windows\system)
- 5) Windows'un kendi dizini (tipik olarak c:\windows)

6) CreateProcess fonksiyonunu uygulayan prosesin PATH çevre değişkeni ile belirtilen dizinleri

Fonksiyonun ikinci parametresindeki çalıştırılabilen dosyanın yol ifadesi eğer boşluk karakteri içeriyorsa tüm yol ifadesi çift tırnaklar içerisine alınmalıdır. Windows'un komut satırı uygulaması olan cmd.exe programı komut satırından çalıştırılan programı nihayetinde CreateProcess uygulayarak onun birinci parametresine NULL geçip ikinci parametresine yazdığımız komut satırı yazısını geçirerek çalıştırmaktadır. Böylece komut satırından çalıştırılmak istenen program yukarıda belirtilen dizinlerde ve PATH çevre değişkeninde belirtilen dizinlerde sırasıyla aranmaktadır.

Fonksiyonun üçüncü ve dördüncü parametreleri prosese ve prosesin ana thread'ine ilişkin güvenlik parametreleridir. Bu parametreler NULL olarak geçilebilir. Bu durumda default güvenlik durumu anlaşılır. Fonksiyonun beşinci parametresi kernel nesnelere ilişkin alt proseslere geçirilebilmesine ilişkin ana şalter görevindedir. Bu parametre FALSE olarak geçilebilir. Altıncı parametre yaratılacak prosese ilişkin çeşitli belirlemeleri içermektedir. Bu parametre bazı bayrakların bit OR işlemine sokulmasıyla oluşturulur. Fakat istenirse bu parametre sıfır geçilebilir. Fonksiyonun yedinci parametresi yaratılacak prosesin çevre değişken listesini belirtir. Bu parametre NULL geçilirse yaratılacak prosesin çevre değişkenleri üst prosesten alınır. Fonksiyonun sekizinci parametresi yaratılacak prosesin çalışma dizinini belirtir. Eğer bu parametre NULL geçilirse prosesin çalışma dizini üst prosesten alınır. Fonksiyonun dokuzuncu parametresi yaratılacak prosese ilişkin bazı ayrıntıların belirlenmesini sağlar. Bu parametreye STARTUPINFO türünden bir yapının adresi geçirilmelidir. Bu yapının ilk elemanına yapının sizeof'u yazılmalıdır. Diğer elemanlar boş bırakılabilir. Çünkü yapının elemanları sıfır ise bu default durum anlamına gelir. Fonksiyonun son parametresi PROCESS_INFORMATION türünden bir yapının adresini alır. Fonksiyon bu yapının içeriğini bizim için doldurur. Bu yapıya fonksiyon yaratılacak proses nesnesinin HANDLE ve Id değerlerini, yaratılacak ana thread nesnesinin HANDLE ve id değerlerini yerleştirir. Prosesin HANDLE değeri Proses Kontrol Bloğuna erişmek için sistem tarafından kullanılmaktadır.

Fonksiyon başarı durumunda sıfır dışı herhangi bir değere, başarısızlık durumunda 0 değerine geri döner.

Aşağıda maksimum default değerler geçilerek bir prosesin yaratılmasına örnek verilmiştir:

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

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    char cmdLine[] = "c:\\windows\\system32\\notepad.exe";
    STARTUPINFO si = { sizeof(STARTUPINFO) };
    PROCESS_INFORMATION pi;

    if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
        ExitSys("CreateProcess");

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}
```

Anahtar Notlar: Windows API fonksiyonları hem ASCII hem de UNICODE karakter tablosuyla çalışabilmektedir. Visual Studio'da default durumda karakter tablosu UNICODE biçimindedir. Yukarıdaki örnekte bunun ASCII yapılması gerekir. Bunun için proje seçeneklerinden "Character Set" seçeneği "Not Set" seçilmelidir.

Sınıf Çalışması: Basit bir C programı yazınız. Bu cl.exe ile derleyen bir program yazınız ve onu çalıştırınız. Komut satırında cl.exe ile derleme işlemi şöyle yapılmaktadır:

```
cl.exe test.c
```

Visual Studio 2017 için cl.exe şurada bulunmaktadır:

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin\cl.exe
```

Visual Studio 2015'te ise cl.exe şuarada bulunmaktadır:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64\cl.exe
```

Çözüm:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <winapifamily.h>

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)
{
    char cmdLine[] = "\"C:\\Program Files (x86)\\Microsoft Visual Studio 12.0\\VC\\bin\\cl.exe\" "
/*
    "-I \"C:\\Program Files (x86)\\Microsoft Visual Studio 12.0\\VC\\include\" "
    "-I \"c:\\Program Files (x86)\\Windows Kits\\8.1\\Include\\um\" "
    "-I \"c:\\Program Files (x86)\\Windows Kits\\8.1\\Include\\shared\" "
    */
    " sample.c";
    STARTUPINFO si = { sizeof(STARTUPINFO) };
    PROCESS_INFORMATION pi;

    if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
        ExitSys("CreateProcess", EXIT_FAILURE);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}
```

cl.exe derleyicisinin gereksinim duyduğu bazı çevre değişkenleri (environment variable) de vardır. Bu çevre değişkenlerinin neler olduğu versiyondan versiyona değişebildiği için Microsoft bu çevre değişkenlerini set eden bir "vsvars32.bat" isimli bir batch script bulundurmaktadır.

Windows Sistemlerinde Proseslerin HANDLE ve ID Değerleri

Windows sistemlerinde CreateProcess API fonksiyonuyla bir proses yaratıldığında bu fonksiyon bize POCESS_INFORMATION isimli yapı yoluyla o proses için bir handle değeri ve bir de id değeri vermektedir. Yani Windows

sistemlerinde proseslerin hem handle değerleri hem de id değerleri vardır. CreateProcess API fonksiyonuna geçirdiğimiz PROCESS_INFORMATION yapısı şöyle bildirilmiştir:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Burada hProcess prosesin handle değerini dwProcessId ise prosesin id değerini belirtmektedir. Yapıdan da gördüğümüz gibi CreateProcess ayrıca bize prosesin ana thread'i için bir handle ve id değeri de vermektedir. Çünkü thread'lerin de Windos sistemlerinde handle ve id değerleri vardır. Thread'ler konusu ileride ayrı bir başlık halinde ele alınacaktır.

Proseslerin handle değerleri Windows sistemlerinde proses kontrol bloğuna erişmek için bir handle görevi görmektedir. Yani bu handle değerini alan API fonksiyonları prosesin kontrol bloğuna erişebilmektedir. Windows'ta prosesler üzerinde işlem yapan pek çok API fonksiyonu bizden üzerinde işlem yapılacak prosesin handle değerini ister. Ancak Windows sistemlerinde prosesin handle değeri o prosesi yaratan prosese özgüdür. Yani bu değer sistem genelinde tek (unique) değildir. Prosesin handle değeri prosese erişim konusunda bazı hakları da tanımlamaktadır. Dolayısıyla farklı prosesler bir X prosesine farklı haklarla erişebilmektedir. Oysa Windows sistemlerinde proseslerin Id değerleri sistem genelinde tektir (unique) ve bir tamsayı ile temsil edilmektedir.

Handle değerinin prosese özgü olması ne anlama gelmektedir? Biz X prosesinde Y prosesi için bir handle değerine sahipsek bu handle değerini başka biz Z prosesine gönderdiğimizde bu handle değeri Z prosesinde bir anlam ifade etmez. Çünkü X prosesi içerisinde Y prosesinin handle değeri yalnızca bu X prosesinde kullanılmak üzere oluşturulmuştur. Ancak prosesin Id değeri sistem genelinde tektir. Prosese özgü değildir. İşte Z prosesi Y prosesi üzerinde işlem yapacaksa Y prosesinin id değerini bilerek kendisine özgü bir handle değeri elde etmeye çalışır. Bu işlem OpenProses API fonksiyonuyla yapılmaktadır. OpenProses fonksiyonunun prototipi şöyledir:

```
HANDLE OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);
```

Fonksiyonun ilk iki parametresi burada açıklanmayacaktır. Konunun ayrıntıları "Windows Sistem Programlama" kurslarında ele alınmaktadır. Ancak fonksiyonun üçüncü parametresi handle değeri elde edilecek prosesin id değerini belirtmektedir. Fonksiyon bize geri dönüş değeri olarak prosesin handle değerini vermektedir. Konuyu şöyle özetleyebiliriz:

- Windows'ta proseslerin handle ve id değerleri vardır. Proses üzerinde işlem yapan API fonksiyonları bizden işlem yapılacak prosesin handle değerini ister.
- Proseslerin handle ve id değerleri zaten onu yaratan prosese doğrudan CreateProcess API fonksiyonunda verilmektedir.
- Başka bir proses kendi yaratmadığı bir proses üzerinde işlem yapacaksa o prosesin Id değerini bilmeli ve bu Id değerinden OpenProses fonksiyonuyla handle değeri elde etmelidir.
- Prosesin handle değeri prosese özgüdür. Biz bunu başka prosese iletsek bile orada bir işe yaramaz.

O anda çalışmakta olan prosesin (yani kendi prosesimizin) handle değeri GetCurrentProcess fonksiyonuyla elde edilebilir. GetCurrentProcess fonksiyonu parametre almaz, geri dönüş değeri prosesin handle değerini verir:

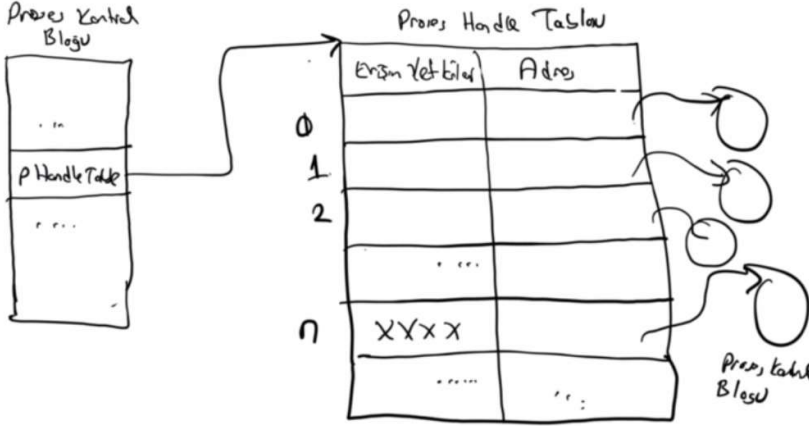
```
HANDLE GetCurrentProcess(void);
```

Benzer biçimde biz kendi prosesimizin Id değerini de GetCurrentProcessId fonksiyonuyla elde edebiliriz:

DWORD GetCurrentProcessId();

Burada dikkat edilmesi gereken nokta şudur: Bizim kendi prosesimizin handle ve id değerini işin başında biz bilmemekteyiz. Bu değerleri bizi çalıştıran üst proses (örneğin cmd.exe ya da explorer.exe) bilmektedir. Çünkü CreateProcess fonksiyonunu üst proses uygulayarak bizi yaratmıştır.

Windows'ta prosesin handle değerleri HANDLE türüyle temsil edilmiştir. Bu türünde void * biçiminde typedef edildiğini biliyorsunuz. Fakat aslında Windows sistemlerinde proseslerin handle değerleri bir adres görünümünde olsa da bir adres değildir. Şöyle ki: Windows'ta bir grup kernel veri yapısına "kernel nesnelere" denilmektedir. Yalnızca prosesler değil, thread'ler, dosyalar, semaphore'lar vs. birer kernel nesnesidir. İşte tüm kernel nesnelere gerçekte adresleri proses kontrol bloğunda "proses handle tablosu" denilen bir tabloda tutulmaktadır. Proses handle tablosu kernel nesnelere gerçekte adreslerini, onların erişim özelliklerini ve gerçekte adreslerini tutmaktadır. Genel yapısı aşağıdaki gibi benzer biçimdedir:



İşte aslında bir kernel nesnesinin handle değeri proses handle tablosunda bir indeks belirtmektedir. Asıl nesne adresi proses handle tablosunda bu indeksten elde edilmektedir. Yukarıdaki şekilde bir prosesin handle değerinin olduğunu varsayalım. Biz bu n değerini bir API fonksiyonuna geçirdiğimizde API fonksiyonu kernel moda geçerek o anda çalışmakta olan prosesin kontrol bloğuna ve oradan da handle tablosuna erişir. Bu n değerini proses handle tablosuna indeks yapar ve proses handle tablosundan nesnenin gerçekte adresini elde eder. Prosesler için kernel nesnesinin adresi demek ilgili prosesin kontrol bloğunun adresi demektir. Ancak proses handle tablosu yalnızca proseslerin handle değerlerini tutan bir tablo değildir. İsmine "kernel nesnesi" denilen bir grup nesnenin bilgilerini tutan bir tablodur. Örneğin dosyalar da bir kernel nesnesidir. Bu durumda biz CreateFile fonksiyonuyla bir dosyayı açtığımızda CreateFile fonksiyonunun bize verdiği handle değeri de proses handle tablosunda bir indeks belirtmektedir.

Windows sistemlerinde yalnızca proseslerin değil aynı zamanda tüm kernel nesnelere gerçekte adresleri o prosesle özgüdür. Çünkü bu handle değerleri o prosesin proses handle tablosunda bir indeks belirtmektedir. Bu indeks değeri biz başka prosesle iletsek artık o indeks o prosesin handle tablosunda indeks belirteceği için bir anlamı olmayacaktır. Aşağıdaki örnekte prosesle toplamda üç kernel nesnesi yaratılmıştır. (CreateProcess API fonksiyonunun ana thread için de bir kernel nesnesi yarattığını unutmayınız.) Ekran yazdırılan bu değerlerin küçük olduğunu ve bir indeks belirteceğine dikkat ediniz:

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

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    STARTUPINFO si = { sizeof(STARTUPINFO) };
    PROCESS_INFORMATION pi;
    char args[] = "notepad.exe";
    HANDLE hFile, hFile2, hFile3;
```

```

if (!CreateProcess(NULL, args, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
    ExitSys("CreateProcess");

if ((hFile = CreateFile("test1.txt", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, 0, NULL)) ==
INVALID_HANDLE_VALUE)
    ExitSys("CreateFile");

printf("%p, %p %p\n", pi.hProcess, pi.hThread, hFile);

CloseHandle(pi.hThread);
CloseHandle(pi.hProcess);
CloseHandle(hFile);

return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

```
00000000000000B0, 00000000000000AC 00000000000000B8
```

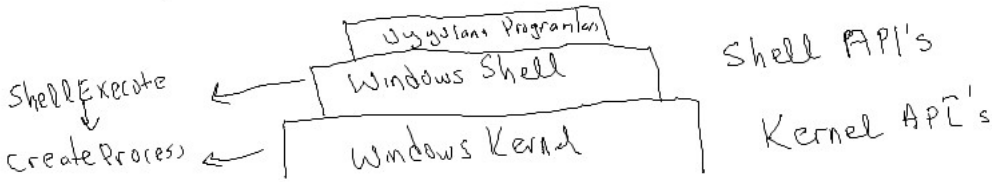
```
C:\Users\CSD\Dropbox\Kurslar\SysProg-1\Src\Sample\x64\Debug\Sample.exe (process 4240) exited with code 0
Press any key to close this window . . .
```

Windows'ta adresleri proses handle tablosunda saklanan tüm kernel nesneleri CloseHandle isimli API fonksiyonuyla kapatılmaktadır. Tabii programcı kapatımları yapmasa bile proses sonlandığında tüm kernel nesneleri sistem tarafından kapatılmaktadır. Bir kernel nesnesi CloseHandle fonksiyonuyla kapatıldığında proses handle tablosundaki ilgili satır boşaltılır. Artık bu satır (dolayısıyla bu indeks) yeni bir kernel nesnesi için kullanılabilir. Ayrıca bir proses ya da thread'i CloseHandle fonksiyonuyla kapatmış olmamızın o prosesin ya da thread'in sonlanacağı anlamına gelmediğine dikkat ediniz. Bir proses ya da thread'i CloseHandle fonksiyonuyla kapatığımızda o proses ya da thread sonlanmaz yalnızca bizim ona erişimimiz sonlandırılmış olur.

Anahtar Notlar: Windows'ta sistemde proseslerin (yani çalışmakta olan programların) listesini görmek Ctrl+Alt+Del ile "Task Manager"a başvurulabilir. Ancak Task Manager kullanıcılar için yüzeysel bilgiler vermektedir. Çok daha detaylı bilgileri elde etmek için "Process Explorer" isimli program önerilebilir. Bu programın kaynak kodları verilmemektedir. Ancak bu programa benzer "Process Hacker" isimli program açık kaynak kodludur. Dolayısıyla kaynak kodları da incelenebilmektedir.

Windows'ta ShellExecute Fonksiyonu İle Proseslerin Yaratılması

Windows'ta proses yaratan temel fonksiyon CreateProcess isimli API fonksiyonudur. Fakat ShellExecute isimli kabuk fonksiyonuyla da prosesler yaratılabilmektedir. ShellExecute bir sistem fonksiyonu değildir. Zaten kendi içerisinde CreateProcess fonksiyonunu çağırılmaktadır. ShellExecute bir kabuk (shell) fonksiyonudur. Windows'un kabuğuna "Windows Explorer" denilmektedir. Aslında masaüstü olarak gördüğümüz bu kabuk da "explorer.exe" isimli bir procestir. İstenirse Windows'ta da kabuk tamamen devre dışı bırakılabilir. Tabii bu durumda bu kabuk fonksiyonlarını kullanamayız. Fakat kabuk fonksiyonları da Windows'un bir parçası durumundadır.



Uzantısı .txt gibi, .doc gibi olan çalıştırılmayan dosyaları da biz ShellExecute fonksiyonuna verebiliriz. Bu durumda ShellExecute fonksiyonu "registry" denilen bir veritabanının kayıtlarına erişerek bu uzantılı dosyanın hangi çalıştırılabilen dosyayla ilişkilendirilmiş olduğunu belirler ve CreateProcess ile o çalıştırılabilen dosyayı çalıştırır. Sonra da bizim verdiğimiz dosyayı ona komut satırı argümanı olarak geçirir. Yani görüldüğü gibi ShellExecute fonksiyonu temel bir fonksiyon değildir. Kendi içerisinde CreateProcess fonksiyonunu kullanan daha yüksek seviyeli bir fonksiyondur. Aslında Windows'ta masaüstü (explorer.exe) ya da cmd.exe komut satırı prosesleri programları çalıştırmak için doğrudan CreateProcess fonksiyonunu değil ShellExecute fonksiyonunu kullanmaktadır. Böylece örneğin biz masaüstünde bir .txt dosyasına tıkladığımızda ya da komut satırında bir .txt dosyasının simini yazıp ENTER tuşuna bastığımızda bu .txt dosyasının ilişkilendirilmiş olduğu program (tipik olarak notepad.exe) çalıştırılacaktır.

Windows'ta dosya ilişkilendirmesi hangi uzantılı dosyaların hangi çalıştırılabilen programlarla açılacağını belirten bir kayıttır. Bu ilişkilendirme kayıtları Windows'un içerisindeki "registry" denilen dosyalarda tutulmaktadır. Dolayısıyla ShellExecute fonksiyonu da bu registry kayıtlarına bakmaktadır. Tabii registry yalnızca dosya ilişkilendirmelerini tutan bir veritabanı değildir. Windows tüm ayarları da bu registry veritabanında tutulmaktadır. Hatta isterse kendi program ayarlarını da bu registry veritabanında tutabilmektedir. Yani registry denilen bu veritabanına uygulama programcıları da erişebilmektedir. Ancak programcılar bu erişimi doğrudan yapmak yerine dolaylı olarak yapan ve ismine "registry API fonksiyonları" denilen bir grup API fonksiyonuyla yaparlar. Denerğimizde "Windows Sistem Programlama" kurslarında bu registry veritabanı hakkında ayrıntılı bilgiler verilmektedir.

Örneğin biz masaüstünde bir .txt dosyasına çift tıklamış olalım. Sırasıyla şu olaylar gerçekleşecektir:

- 1) Masaüstü "explorer.exe" isimli procestir. Dolayısıyla masaüstündeki farenin çift tıklanması da bu proses tarafından ele alınır.
- 2) Masaüstü prosesi (explorer.exe) çift tıklanan .txt dosyasını belirler. Ve bu dosya ile ShellExecute isimli fonksiyonu çağırır.
- 3) ShellExecute fonksiyonu registry kayıtlarına bakarak .txt dosyasının hangi çalıştırılabilen programla ilişkilendirildiğini belirler (default olarak notepad.exe).
- 4) ShellExecute CreateProcess API fonksiyonuyla .txt dosyasının ilişkilendirilmiş olduğu çalıştırılabilen dosyayı (notepad.exe) çalıştırır ve ShellExecute ta geçirilen .txt dosyasını da bu programa birinci komut satırı argümanı yapar.

ShellExecute fonksiyonun prototipi şöyledir:

```
#include <shellapi.h>

HINSTANCE ShellExecute(
    HWND hwnd,
    LPCTSTR lpOperation,
    LPCTSTR lpFile,
    LPCTSTR lpParameters,
    LPCTSTR lpDirectory,
    INT nShowCmd
);
```

Fonksiyonun birinci parametresi bir GUI penceresinin HANDLE değerini alır. Fonksiyon başarısızlık durumunda bir MessageBox çıkartabildiği için böyle bir üst pencereye gereksinim duymaktadır. Bu parametre NULL geçilebilir. Bu

durumda masaüstü penceresi anlaşılır. İkinci parametre yapılmak istenen eylemi belirtmektedir. Eğer bir program çalıştırılacaksa eylem "open" olmalıdır. "Open" dışında başka eylemler de vardır. Üçüncü parametre çalıştırılacak dosyanın yol ifadesini alır. Tabii bu dosya çalıştırılabilir bir dosya olmak zorunda değildir. Dördüncü parametre çalıştırılacak programa geçilecek komut satırı argümanlarını belirtmektedir. Eğer ikinci parametre çalıştırılabilir bir dosya değilse bu parametre NULL geçilmelidir. Son parametre eğer program bir GUI uygulaması ise onun nasıl açılacağını belirlemek için kullanılmaktadır. Örneğin bu parametre SW_MAXIMIZE olarak, SW_MINIMIZE olarak, SW_SHOWNORMAL olarak geçilebilir.

ShellExecute fonksiyonunun geri dönüş değeri eğer 32'den küçükse hataya işaret etmektedir. Eğer 32'den küçük değilse çalıştırılan programın sanal belleğe yüklenme adresine geri döner. Fonksiyonun geri dönüş değerinin HINSTANCE türünden (void *) olduğuna dikkat ediniz. Hata kontrolü için 32 ile karşılaştırma yaparken bu HINSTANCE değerini tamsayı türüne dönüştürmeniz gerekir.

Örnek bir ShellExecute çağırısı şöyle olabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <shellapi.h>

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    HINSTANCE hInstance;

    hInstance = ShellExecute(NULL, "open", "x.txt", NULL, NULL, SW_SHOWNORMAL);
    if ((int)hInstance < 32)
        ExitSys("ShellExecute");

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}
```

Windows'ta Proses Listesinin Elde Edilmesi

İşletim sistemlerinde bazen user mod programların çalışmakta olan programların (yani proseslerin) listesini elde etmesi gerekebilmektedir. Windows'ta bunun için iki grup API fonksiyonu kullanılmaktadır: ToolHelp API fonksiyonları ve PSAPI API fonksiyonları. ToolHelp API fonksiyonları daha eski, PSAPI API fonksiyonları ise daha modern bir tasarımdır. Bunlar birbirleri yerine kullanılabilirler. Bazı durumlarda biri diğerine avantaj sağlayabilmektedir. Biz burada PSAPI fonksiyonları için örnekler vereceğiz.

EnumProcess isimli PSAPI fonksiyonu sistemdeki tüm proseslerin Id değerlerini bize vermektedir. Biz de bu Id değerlerinden hareketle OpenProcess fonksiyonunu uygulayarak bu prosesleri açıp onlara ilişkin handle değerlerini elde edip onlar hakkında daha detaylı bilgiler edinebiliriz. EnumProcess fonksiyonunun prototipi şöyledir:

```
#include <psapi.h>
```

```

BOOL EnumProcesses(
    DWORD *lpidProcess,
    DWORD cb,
    LPDWORD lpcbNeeded
);

```

Fonksiyonun birinci parametresi proseslerin id değerlerinin yerleştirileceği DWORD türünden dizinin adresini almaktadır. İkinci parametre dizinin byte cinsinden uzunluğunu belirtir. Üçüncü parametre ise DWORD türünden bir nesnenin adresini alır. Fonksiyon buraya diziye yerleştirdiği byte sayısını yazar. Fonksiyon başarı durumunda sıfır dışı bir değere başarısızlık durumunda sıfır değerine geri döner. Örneğin:

```

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

#define MAX_PROCESS    1024

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    DWORD dwProcessIds[MAX_PROCESS];
    DWORD dwBytes;
    DWORD i;

    if (!EnumProcesses(dwProcessIds, sizeof(dwProcessIds), &dwBytes))
        ExitSys("EnumProcesses");

    printf("Number of processes: %lu\n", dwBytes / sizeof(DWORD));
    printf("-----\n");

    for (i = 0; i < dwBytes / sizeof(DWORD); ++i)
        printf("%lu ", dwProcessIds[i]);
    printf("\n");

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Yukarıda da açıklandığı gibi proseslerin id değerleri Windows'ta doğrudan işe yaramamaktadır. Bizim bu id değerlerini handle değerlerine dönüştürmemiz gerekir. İşte bu işlem OpenProcess API fonksiyonuyla yapılmaktadır. OpenProcess fonksiyonunun prototipi şöyledir:

```

HANDLE OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId
);

```


Fonksiyonun birinci parametresi açılacak prosesin arzu edilen erişim yetkilerini almaktadır. Bu parametre PROCESS_ALL_ACCESS geçilebilir. Bu yüksek bir açış yetkisi gerektirmektedir. İkinci parametre açılan prosesin handle değerinin alt proseslere geçirilip geçirilmeyeceğini belirtir. Üçüncü parametre prosesin id değerini almaktadır. Fonksiyon başarı durumunda prosesin handle değerine başarısızlık durumunda NULL değerine geri dönmektedir. Windows'ta kernel nesnelerinin güvenlik bilgileri vardır. Yani her kernel nesnesini her proses istediği erişim hakkıyla açamamaktadır. Eğer çok sayıda prosesi açabilmek isterseniz programınızı (ya da Visual Studio IDE'sini) "Run as administrator" biçiminde çalıştırmalısınız. Kernel nesnelerinin güvenlik parametreleri "Windows Sistem Programlam" kurslarında ele alınmaktadır.

Peki OpenProcess ile açılan bir prosesin bilgilerini nasıl elde edebiliriz. İşte artık handle değerini alarak bize proses hakkında bilgi veren pek çok API fonksiyonu vardır. Aşağıda proses listesini elde etmeye çalışan bir program görülmektedir:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <Psapi.h>

#define MAX_PROCESS          1024
#define MAX_PROCESS_NAME    512

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    DWORD dwProcessIds[MAX_PROCESS];
    DWORD dwBytes;
    DWORD i;
    HANDLE hProcess;
    HMODULE hModule;
    char processName[MAX_PROCESS_NAME];
    DWORD count;

    if (!EnumProcesses(dwProcessIds, sizeof(dwProcessIds), &dwBytes))
        ExitSys("EnumProcesses");

    printf("Number of processes: %lu\n", dwBytes / sizeof(DWORD));
    printf("-----\n");

    count = dwBytes / sizeof(DWORD);
    for (i = 0; i < count; ++i) {
        if ((hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, FALSE,
dwProcessIds[i])) == NULL) {
            fprintf(stderr, "Cannot open process: %lu\n", dwProcessIds[i]);
            continue;
        }

        if (!EnumProcessModules(hProcess, &hModule, sizeof(HMODULE), &dwBytes)) {
            fprintf(stderr, "Cannot get process module: %lu\n", dwProcessIds[i]);
            CloseHandle(hProcess);
            continue;
        }

        if (!GetModuleBaseName(hProcess, hModule, processName, sizeof(processName))) {
            fprintf(stderr, "Cannot get model base name: %lu\n", dwProcessIds[i]);
            continue;
        }

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

        CloseHandle(hProcess);
    }

    return 0;
}
```



```

}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Burada EnumProcessModules API fonksiyonu prosesin modüllerine ilişkin modül handle değerlerini elde eder. Windows'ta .exe ve .dll gibi yüklenebilen dosyalara modül (module) denilmektedir. Örneğimizde yalnızca prosesin ilk modülü elde edilmiştir. Bu da zaten .exe dosyasıdır. Modül handle değeri bilinen modülün ismi de GetModuleBaseName API fonksiyonuyla elde edilmektedir. Yukarıdaki programda çeşitli nedenlerden dolayı bazı proseslerin isimleri elde edilemeyebilir. Örneğin 64 bit Windows sistemlerinde 32 bit prosesler 64 bit prosesleri OpenProcess fonksiyonuyla açamamaktadır.

UNIX/Linux Sistemlerinde Proseslerin Yaratılması

UNIX/Linux sistemlerinde prosesler fork isimli bir POSIX fonksiyonuyla yaratılırlar. fork POSIX fonksiyonu Linux sistemlerinde doğrudan sys_fork isimli sistem fonksiyonunu çağırılmaktadır. Bu sistemlerde proses yaratmanın başka bir yolu yoktur. (fork fonksiyonunun vfork isimli bir versiyonu da vardır fakat bunun kullanım gerekçesi hepten ortadan kalkmıştır.) Fonksiyonun prototipi şöyledir:

```

#include <unistd.h>

pid_t fork(void);

```

fork bir prosesin özdeş bir kopyasından oluşturur. Yani yaratılan alt proses (child process) için yeni bir proses kontrol bloğu oluşturulur. Üst prosesin kontrol bloğundaki bilgiler alt procese kopyalanır. Yine alt prosesin sanal bellek alanı tamamen üst processten (parent process) kopyalanmaktadır. Böylece fork işleminden sonra aynı koda ve veriye sahip, geçmişleri aynı olan özdeş fakat farklı iki proses söz konusu olmaktadır.

UNIX/Linux sistemlerinde her prosesin sistem genelinde tek olan bir proses id değeri vardır. Bu sistemlerde Windows sistemlerinde olduğu gibi ayrıca proseslerin handle değerleri yoktur. UNIX/Linux sistemlerinde prosesin id değeri proses kontrol bloğuna erişmekte kullanılan bir handle gibi işlem görür. Prosesin id değeri tamsayısal bir değerdir ve pid_t türü ile temsil edilmektedir. POSIX sistemlerinde komut satırında "ps (process status)" komutu o anda sistemdeki prosesler hakkında bize bilgiler vermektedir. ps komutu default durumda yalnızca komutun uygulandığı terminale bağlı prosesleri listelemektedir. Ancak "-e" seçeneği sistemdeki tüm prosesleri listeler. Örneğin:

```

csd@csd-virtual-machine ~/Study/SysProg-2017 $ ps
  PID TTY          TIME CMD
  4089 pts/0        00:00:00 bash
 19848 pts/0        00:00:00 ps
csd@csd-virtual-machine ~/Study/SysProg-2017 $ █

```

fork fonksiyonuna bir proses girmekte (üst proses) fakat iki proses çıkmaktadır. Alt prosesin yaratımı fork fonksiyonu içerisinde yapılmaktadır. Üst prosesin sanal bellek alanının da kopyalandığına dikkat ediniz. Böylece fork fonksiyonundan çıkan iki proses de aynı kodu çalıştıracaklardır.

Peki her iki proses de fork fonksiyonundan çıktığına göre ve aynı kodu çalıştıracaklarına göre onları birbirlerinden nasıl ayırabiliriz? İşte fork fonksiyonundan üst proses alt prosesin id değeri ile, alt proses ise sıfır değeri ile çıkmaktadır.

Böylece fork işleminden sonra kodda üst proses ile alt proses birbirlerinden ayrılabilir. Tabii fork başarısız da olabilir. Bu durumda -1 değerine geri döner. O halde fork fonksiyonunun tipik uygulanma biçimi aşağıdaki gibidir:

```
pid_t pid;

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

if (pid != 0) { /* parent process */
    ...
}
else { /* child process */
    ...
}
```

Örneğin:

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

int main(void)
{
    pid_t pid;

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

    if (pid != 0) { /* parent process */
        printf("parent\n");
    }
    else { /* child process */
        printf("child\n");
    }

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

    return 0;
}
```

Pekiyi aşağıdaki programda ekrana kaç tane "ends" yazısı çıkar?

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

int main(void)
{
    int i;

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

    printf("ends\n");

    return 0;
}
```

Yanıt: 8 tane. Programda ne kadar proses yaratılmışsa o kadar "ends" yazısı çıkar. Burada toplam 8 proses yaratılmaktadır. Yukarıdaki programın eşdeğeri aslında aşağıdaki gibidir:

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

int main(void)
{
    int i;

    fork();
    fork();
    fork();

    printf("ends\n");

    return 0;
}
```

fork işlemi ile üst prosesin kontrol bloğunun alt prosesin kontrol bloğuna kopyalandığını belirtmiştir. Bunun anlamı şudur: Alır proses de üst proses ile aynı user id'ye, grup id'ye sahip olur. Alt prosesin de çalışma dizini aynı olacaktır. Yani üst prosesin sahip olduğu tüm özellikler (bazıları hariç) alt procese aktarılacaktır. Tabii UNIX/Linux sistemlerinde her prosesin ayrı bir id değeri vardır. Dolayısıyla üst prosesin id değeri ile alt prosesin id değerleri farklı olacaktır. fork fonksiyonundan üst prosesin alt prosesin id değeriyle alt prosesin ise 0 değeri ile çıktığını söylemiştik. Bu alt prosesin id değerinin 0 olduğu anlamına gelmemektedir. Yalnızca fork fonksiyonunun geri dönüş değeri bu biçimdedir. Yoksa yeni yaratılmış olan alt prosesin de anlamlı bir id değeri vardır.

fork işlemi sırasında üst prosesin tüm bellek alanının da alt procese kopyalandığını belirtmiştik. Siz bu işlemin zaman kaybına yol açacağını düşünebilirsiniz. Oysa sanal bellek mekanizması sayesinde aslında fork işlemi sırasında gerçek anlamda bir kopya oluşturulmamaktadır. Başlangıçta fork fonksiyonu alt prosesin sayfa tablosunu üst prosesle aynı olacak biçimde ayarlar. Böylece aslında kopya çıkartılmadan üst prosesle alt prosesin gerçek fiziksel sayfaları ortak kullanması sağlanmış olur. Tabii bu işlemde sonra alt proses bir fiziksel sayfaya yazma yaptığında artık o sayfanın o anda bir kopyası çıkartılıp sayfalar birbirlerinden ayrılmaktadır. Bu mekanizmaya "copy on write" denildiğini anımsayınız.

Eskiden thread'ler yoktu. Dolayısıyla bir işi birden fazla akışa yaptırabilmek için fork mekanizması kullanıyordu. Şöyle ki: Program fork yapıp yeni bir akış oluşturup işin bir kısmını bu akışa devredebiliyordu. Ancak tabii bu biçimdeki çalışma proseslerarası haberleşme gerektirmektedir. Thread'ler ortaya çıkınca artık bu biçimdeki çalışma da büyük ölçüde kullanım dışı kalmıştır.

fork ile biz başka bir programı çalıştıramayız. fork ancak bir prosesin özdeş kopyasını çalıştırır. Pekiyi başka bir program dosyasını nasıl çalıştırırız? İşte bu exec fonksiyonlarıyla yapılmaktadır.

exec Fonksiyonları

POSIX sistemlerinde ismi exec ile başlayan bir grup execxxx biçiminde fonksiyon vardır. Bu fonksiyonlar benzer işlemleri yapmaktadır. Biz bunlara exec fonksiyonları diyeceğiz. exec fonksiyonları bir prosesin başka bir kodla çalışmaya devam etmesini sağlamaktadır. exec işlemiyle mevcut prosesin çalıştırdığı kod bellekten atılır, onun yerine exec fonksiyonunda belirtilen dosya belleğe yüklenir ve o dosyadaki kod çalıştırılır. exec işlemiyle prosesin kontrol bloğu değişmez. Yani prosesin id'si, yetkileri, çalışma dizini, açtığı dosyalar vs. hep aynı kalır. exec yalnızca prosesin başka bir kodla çalışmaya devam etmesine yol açmaktadır.

Yukarıda da belirttiğimiz gibi exec aslında bir grup fonksiyondan oluşan bir ailedir. Bu ailede toplam exec ismiyle başlayan 7 fonksiyon vardır. Bu fonksiyonların yaptıkları şey aynı olmasına karşın yalnızca parametrik yapıları (yani arayüzleri) farklıdır. Bu 7 fonksiyonun prototipleri şöyledir:

```
#include <unistd.h>

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

```

int execlp(const char *file, const char *arg, ...);
int execl(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
int execve(const char *filename, char *const argv[], char *const envp[]);

```

Aslında asıl taban fonksiyon execve fonksiyonudur. Yani yalnızca execve bir sistem fonksiyonudur. Diğer fonksiyonlar kendi içlerinde execve fonksiyonunu çağırarak biçimde yazılmışlardır. Fonksiyonların sonlarındaki 'l' eki "list" sözcüğünden, 'p' eki "PATH" sözcüğünden, 'v' eki de "vector" sözcüğünden gelmektedir.

En çok kullanılan exec fonksiyonlarından biri execl fonksiyonudur.

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

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

Fonksiyonun birinci parametresi çalıştırılabilir (executable) dosyanın yol ifadesini alır. Diğer parametreler programın komut satırı argümanlarını belirtir. Listenin sonunun NULL adresle bitirilmesi gerekmektedir. execl başarılıysa geri dönmaz. Başarısızsa -1 değerine geri döner.

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

```
int main(void)
{
    printf("begins...\n");

    if (execl("/bin/ls", "/bin/ls", "-l", (char *)NULL) < 0) {
        perror("execl");
        exit(EXIT_FAILURE);
    }

    printf("ends...\n");    /* unreachable code */

    return 0;
}
```

Fonksiyonun son parametresi için NULL argümanı girilirken tür dönüştürmesi yapılmalıdır. Yani aşağıdaki çağırma biçimi sorunludur:

```
if (execl("/bin/ls", "/bin/ls", "-l", NULL) < 0) {
    perror("execl");
    exit(EXIT_FAILURE);
}
```

Çünkü "..." parametresine karşılık NULL makrosu girilirse bu NULL makrosunun nasıl define edilmiş olduğuna bağlı olarak sorunlar çıkabilir. Bilindiği gibi C standartlarına göre NULL makrosu iki biçimde define edilmiş olabilmektedir:

```
#define NULL 0
```

ya da,

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

İşte eğer NULL birinci biçimde olduğu gibi define edilmişse buna karşı gelen parametre gösterici olmadığı için derleyici argümanı stack'e int türünden sıfırmış gibi gönderir. Oysa 64 bit sistemlerde göstericiler 8 byte uzunluğunda olduğu için sorun çıkar. Fakat biz bu son parametreyi (char *)NULL biçiminde girersek bu durumda her halukarda stack'e NULL adres atılacaktır.

Diğer çok kullanılan exec fonksiyonlarından biri de execv fonksiyonudur (Buradaki 'v' "vector" sözcüğünden kısaltmadır.) execv fonksiyonunda komut satırı argümanları bir gösterici dizisine yerleştirilip geçirilir. Fonksiyonun prototipi şöyledir:

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

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

Örneğin:

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

int main(void)
{
    char *args[] = { "/bin/ls", "-l", NULL };

    if (execv("/bin/ls", args) < 0) {
        perror("execv");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Söz konusu gösterici dizisinin NULL adresle sonlandırılmış olması gerekir. (Burada NULL makrosununun dönüştürme yapılması gerekmektedir.)

Şimdi komut satırı argümanını ile aldığı programı çalıştıran bir program yazmak isteyelim. Örneğin programımız sample olsun o da ls'yi çalıştırsın:

```
./sample /bin/ls -l
```

execv fonksiyonu bunun çok için uygundur:

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

int main(int argc, char *argv[])
{
    if (argc == 1) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }

    if (execv(argv[1], &argv[1]) < 0) {
        perror("execv");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

exec fonksiyonlarının p'li versiyonları çalıştırılacak dosyayı exec uygulayan prosesin PATH çevre değişkeni ile belirtilen dizinlerde arar. Çevre değişkenleri konusu izleyen bölümde ele alınmaktadır. UNIX/Linux sistemlerinde PATH çevre değişkeni ':' karakterleriyle ayrılan dizinlerden oluşan bir yazı biçimindedir. (Windows sistemlerinde PATH çevre değişkeni içerisindeki dizinler ';' karakteri ile değil ';' karakteri ile ayrılmaktadır.) Örneğin:

```
echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

Ancak exec fonksiyonlarının p'li versiyonlarının PATH çevre değişkenine bakması için yol ifadesinde hiç bir '/' karakterinin bulunmaması gerekir. Aksi halde bu fonksiyonlar PATH çevre değişkenine bakmazlar ve tamamen p'siz versiyonlar gibi yol ifadesinin mutlak ya da görelili olması durumuna göre aramalarını yaparlar. Örneğin:

```
execvp("a/sample", "a/sample", (char *)NULL);
```

Burada execvp PATH çevre değişkenine bakmayacaktır. sample dosyasını prosesin çalışma dizininin altındaki a dizininin içerisinde arayacaktır. Ancak yol ifadesinde hiç '/' karakteri yoksa bu p'li versiyonlar yalnızca PATH çevre değişkeni ile belirtilen dizinlerde arama yaparlar. Ayrıca çalışma dizinine bakmazlar. Örneğin:

```
execvp("sample", "a/sample", (char *)NULL);
```

Burada sample programı bulunulan dizinde olsa bile oraya bakılmayacaktır. Yalnızca PATH çevre değişkeni ile belirtilen dizinlere bakılacaktır. Örneğin:

```
execvp("ls", "ls", "-l", (char *)NULL);
```

Burada dosya isminde hiçbir '/' karakteri geçmediği için execvp dosyası yalnızca PATH çevre değişkeni ile belirtilen dizinlerde arar. Örneğin çalışma dizinimizde "sample" isimli çalıştırılabilen bir dosya bulunuyor olsun. Aşağıdaki gibi bir exec çağrısı bu dosyayı çalıştırabilir mi?

```
execvp("sample", "sample", (char *)NULL);
```

Yanıt hayır. Çünkü burada exec execvp dosya isminde '/' karakteri olmadığı için onu yalnızca PATH çevre değişkeni ile belirtilen dizinlerde arayacaktır. Prosesin çalışma dizinine bakmayacaktır. Fakat exec işlemi şöyle yapılsaydı program çalıştırılabilirdi:

```
execvp("./sample", "./sample", (char *)NULL);
```

Ya da aşağıdaki bir çağrıda da sample programı çalıştırılabilirdi:

```
exec("sample", "./sample", (char *)NULL);
```

UNIX/Linux sistemlerindeki kabuk programları exec fonksiyonlarının p'li versiyonlarını kullanarak programları çalıştırmaktadır. İşte biz de bu yüzden programları çalıştırırken "./sample" biçiminde isimleri belirtiriz. Eğer böyle bir program "sample" biçiminde çalıştırılmaya çalışılırsa exec fonksiyonlarının p'li versiyonları onları yalnızca PATH çevre değişkeni ile belirtilen dizinlerde arayacak, dolayısıyla bulamayacaktır. "./sample" yol ifadesi aslında "sample" ile aynı anlama geliyor olsa bile işin içerisine bir '/' karakteri karıştırılmıştır ve bu '/' karakteri exec fonksiyonlarının p'li versiyonlarının artık PATH çevre değişkenine bakmamasına yol açar.

Ayrıca exec fonksiyonlarının bir de e'li versiyonları vardır. (Burada 'e' "environment" sözcüğünden gelmektedir.) Bu versiyonlar program çalıştırılırken çevre değişken takımının değiştirilmesine yol açmaktadır. Bu e'li versiyonların hepsi çevre değişkenlerini bir göstericisi dizisi biçiminde son parametrelerinde bizden almaktadır. Örneğin execle fonksiyonunu şöyle kullanabiliriz:

```
/* sample.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/wait.h>  
  
int main(void)  
{  
    pid_t pid;
```

```

char *env[] = { "City=Istanbul", "Name=Ali", NULL };

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

if (pid == 0 && execl("app", "app", (char *)NULL, env) < 0) {
    perror("execl");
    exit(EXIT_FAILURE);
}

wait(NULL);

return 0;
}

/* app.c */

#include <stdio.h>

extern char **environ;

int main(void)
{
    int i;

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

    return 0;
}

```

fork ve exec Fonksiyonlarının Bir Arada Kullanılması

Bilindiği gibi yalnızca fork fonksiyonu bir prosesin özdeş yeni bir kopyasını oluşturmaktadır. Yani bu durumda yeni prosesin çalıştırdığı kod eskisi ile aynı kod olmaktadır. Tabii biz üst prosesle alt prosesle fork çıkışında birbirinden ayırabilmekteyiz. Yalnız başına exec fonksiyonları ise proses yaratmayı mevcut prosesin başka bir kodla çalışmasına devam etmesini sağlar. Pekiyi hem bizim programımız çalışmaya devam ederken hem de başka bir programı nasıl çalıştırabiliriz? İşte bunun için fork ve exec birlikte kullanılmalıdır. Önce bir kez fork yapılır, alt prosesle exec uygulanır. Tipik kalıp şöyledir:

```

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

if (pid == 0)
    if (execl(...) < 0) {
        perror("execl");
        exit(EXIT_FAILURE);
    }

```

Burada ikinci if deyiminin else kısmına gerek yoktur. Çünkü zaten birinci if deyiminin doğruysa kısmında yeni yaratılan alt proses başka bir programı çalıştırdığı için o akış artık bu koddan devam etmeyecektir. İkinci if deyimi içerisindeki if deyiminin else kısmına da gerek yoktur. Çünkü exec başarılı olursa zaten ekış bu koddan devam etmeyecektir. Ayrıca exec işleminin başarısızlığı durumunda exit ile yalnızca alt prosesin sonlandırıldığına dikkat ediniz. Yukarıdaki işlem daha kompakt yazılabilir:

```

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

```

```

}

if (pid == 0 && execl(...) < 0) {
    perror("execl");
    exit(EXIT_FAILURE);
}

```

Örneğin:

```

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

int main(void)
{
    pid_t pid;

    printf("parent begins\n");

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

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

    printf("parent ends..\n");

    return 0;
}

```

Windows sistemleriyle UNIX/Linux sistemlerinin proses yaratma bakımından farklı olduğuna dikkat ediniz. Windows'ta fork ve exec benzeri bir mekanizma yoktur. Orada CreateProcess adeta buradaki fork ve exec'in birlikte kullanılmasına benzemektedir. Yani Windows sistemlerinde zaten yeni bir proses çalıştırılabilir bir program yüklenerek başlatılmaktadır.

UNIX/Linux Sistemlerinde Script Dosyalarının Çalıştırılması

Kabuk programları hem bir komut satırı sunarlar hem de bunlar bir yorumlayıcı (interpreter) da içermektedir. Kabuk üzerinde yazdığımız komutlar bir text dosyaya yazılırsa bunlar hızlı biçimde çalıştırılabilmektedir. İçerisinde komutların bulunduğu bu kodlara "script" denilmektedir. Script genel bir kavardır. Kabuk programlarının çalıştırdığı script'lere kabuk scriptleri (shell scripts) denilmektedir. Kabuk script dilleri if gibi, for gibi çeşitli deyimler de içermektedir. Her kabuk ortamının script dili farklı olabilmektedir. Örneğin UNIX/Linux dünyasında "C Shell (csh)" denilen kabuğun dili ile "Bourne Again Shell (bash)" denilen kabuğun dili az çok farklıdır. Bugün UNIX/Linux ve MAC OS X sistemlerinde en çok kullanılan kabuk "bash" isimli kabuktur. bash'in de bir script dili vardır. Bu dilin öğrenilmesi zor değildir. Bir kabuk script'i bir kaynak dosyadır ve genellikle uzantısı .sh biçimindedir. Kabuk script dosyaları kabuk programları tarafından yorumlanarak çalıştırılmaktadır. Kabuk script dosyalarını komut satırından çalıştırmanın iki yolu vardır:

1) İlgili kabuk programına script dosyasını komut satırı argümanı biçiminde vererek. Örneğin:

```
/bin/bash sample.sh
```

2) Bir script dosyası text bir dosya olduğu halde biz ona 'x' özelliği vererek exec fonksiyonlarıyla çalıştırabiliriz. Dosyaya 'x' hakkı pratik bir biçimde aşağıdaki gibi verilebilir:

```
chmod +x sample.sh
```

Artık biz dosyayı komut satırından normal bir program gibi şöyle çalıştırabiliriz:


```
./sample
```

Bu durumda dosyanın ilk satırının yorumlayıcı programın yol ifadesini içermesi gerekmektedir.

```
/* sample.sh Dosyası */  
  
#!/bin/bash  
...
```

Pekiye exec fonksiyonları nasıl oluyor da bir text dosyayı çalıştırabiliyor? İşte exec fonksiyonları dosyayı açtıklarında bu dosyanın çalıştırılabilir olup olmadığını onun başlık kısmına bakarak anlamaktadır. Eğer dosya ELF gibi çalıştırılabilir bir formata sahip değilse o dosyanın ilk satırına bakarlar. Bu ilk satırında #! karakterlerinden sonra belirtilen program dosyasını yükleyip çalıştırırlar. Söz konusu scrşpt dosyasını da bu programa komut satırı argümanı olarak verirler. Yani yukarıdaki gibi bir sample.sh dosyası tamamen exec fonksiyonlatı tarafından aşağıdaki gibi çalıştırılacaktır:

```
/bin/bash sample.sh
```

Tabii biz bu yöntemle aslında her türlü dosyayı çalıştırabiliriz. Örneğin perl, python dosyalarını vs. UNIX/Linux dünyasında script dosyalarının başındaki bu exec fonksiyonları için yazılan ve #! ile başlayan satıra "shebang" denilmektedir.

Windows Sistemlerinde de çok yaygın olmasa da script tarzı çalışma vardır. Windows'taki klasik script dosyalarına "batch script" denilmektedir. Bu dosyaların uzantıları .bat biçimindedir. Ancak Windows sonları ismine "Power Shell" denilen farklı bir kabul da kullanıma sokmuştur. Power Shell klasik Batch Shell'e göre çok daha yeteneklidir.

Proseslerin Sonlandırılması ve Exit Kodları

Prosesler aslında işletim sisteminin sistem fonksiyonlarıyla sonlandırılmaktadır. Windows sistemlerinde ExitProcess API fonksiyonu, POSIX sistemlerinde _exit (Linux'ta sys_exit'i çağırır) fonksiyonu prosesi sonlandırmakta kullanılır. C'nin standart exit fonksiyonu ise dolaylı olarak bu fonksiyonları çağırılmaktadır. Windows'taki ExitProcess API fonksiyonunun prototipi şöyledir:

```
void ExitProcess(UINT uExitCode);
```

Fonksiyon parametre olarak prosesin exit kodunu almaktadır. _exit isimli POSIX fonksiyonunun prototipi de benzerdir:

```
#include <unistd.h>  
  
void _exit(int status);
```

C'nin standart exit fonksiyonunun prototipini biliyorsunuz:

```
#include <stdlib.h>  
  
void exit(int status);
```

Yukarıda da belirttiğimiz gibi aslında C'nin standart exit fonksiyonu Windows sistemlerinde ExitProcess API fonksiyonunu, UNIX/Linux ve MAC OS X sistemlerinde ise _exit POSIX fonksiyonunu çağırılmaktadır.

Bir proses başka bir proses tarafından eğer yetki derecesi yeterliyse zorla da sonlandırılabilir. Örneğin Windows sistemlerinde TerminateProcess API fonksiyonu UNIX/Linux sistemlerinde kill isimli POSIX fonksiyonu bu amaçla kullanılabilir. Ancak ne olursa olsun bir prosesin başka bir prosesini ansızın bu biçimde sonlandırması iyi bir teknik değildir. Çünkü sonlandırılan proses önemli bir işlemin ortasında olabilir. Bu sonlandırmanın zararlı sonuçları oluşabilir. TerminateProcess API fonksiyonunun prototipi şöyledir:

```
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);
```

Fonksiyonun birinci parametresi sonlandırılacak prosesin handle değerini ikinci parametresi ise onun exit kodunu alır.

C'nin standart exit fonksiyonu işletim sisteminin sistem fonksiyonlarını çağırmadan önce standart kütüphaneye ilişkin çeşitli sonlandırma işlemlerini yapmaktadır. Dolayısıyla C'de çalışıyorsak programı işletim sisteminin API fonksiyonları ile ya da POSIX fonksiyonlarıyla değil exit standart C fonksiyonuyla sonlandırmamız daha uygun olur. Örneğin biz bir dosya açıp içine birşeyler yazmış olalım. Bu durumda prosesi işletim sisteminin sistem fonksiyonlarıyla (API ya da POSIX fonksiyonlarıyla) sonlandırmamız uygun olmaz. Çünkü stdio fonksiyonlarının oluşturduğu tampon fclose işlemi sırasında flush edilmektedir. Standart exit fonksiyonu kapatılmamış dosyalar için fclose işlemini yapmaktadır. Oysa işletim sisteminin sistem fonksiyonlarının bu tampondan haberi yoktur. Dolayısıyla onlar dosyayı doğrudan işletim sistemi düzeyinde kapatırlar. Fakat tabii eğer biz standart C kütüphanesi ile ilgili önemli işlemler yapmamışsak prosesi doğrudan işletim sisteminin API fonksiyonlarıyla ya da POSIX fonksiyonlarıyla da sonlandırabiliriz.

exit → Bazı geri alın işlemleri yap → ExitProsesi - exit → sonlanma

Bir proses sonlandığında işletim sistemine "exit kodu" denilen bir kod iletilir. Prosesin exit kodu C'de exit fonksiyonuna verilen argümandır. main fonksiyonunda return uygulanırsa bu da aynı anlama gelmektedir. C ve C++ standartlarına göre main fonksiyonun geri dönüş değeri (eğer geri dönerse) exit fonksiyonuna argüman yapılmaktadır. Yani C ve C++ standartlarına göre bir C programı şöyle çalıştırılır:

```
exit(main(...));
```

Ayrıca C'de (main fonksiyonu için istisna olarak) eğer main'de return uygulanmamışsa sanki ana bloğun sonunda 0 ile geri dönmüş gibi işlem uygulanır. Yani:

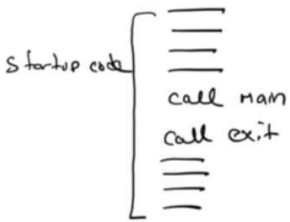
```
int main(void)
{
    ...
}
```

ile,

```
int main(void)
{
    ...
    return 0;
}
```

aynı anlamdadır.

Aslında bir C programında ilk çalışmaya başlayan kod main değildir. Program ismine "startup code" denilen derleyiciler tarafından yerleştirilmiş olan bir koddan çalışmaya başlar. main de aslında bu kod tarafından çağrılmaktadır. Dolayısıyla akış main fonksiyonunu bitirirse yeniden "startup-code"a gerid öner. İşte bu noktada exit uygulanmıştır.



Peki exit kodu ne işe yaramaktadır? İşletim sistemi için exit kodunun kaç olduğunun bir önemi yoktur. İşletim sistemi bu kodu alır, saklar. Eğer üst proses isterse ona verir. Böylece üst proses alt prosesin hangi exit koduyla (yani nasıl)

sonlandığını bilmiş olur. Duruma göre birşeyler yapabilir. Geleneksel olarak başarılı sonlanmalarda sıfır değeri, başarısız sonlanmalarda sıfır dışı değerler tercih edilmektedir.

C'nin <stdlib.h> başlık dosyasında okunabilirliği artırmak için sonlanmaya ilişkin iki sembolik sabit de bulundurulmuştur:

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

Yani biz exit fonksiyonunda bu sembolik sabitleri kullanabiliriz.

Proseslerin exit kodları nasıl alınabilir? Tabii exit kodunun alınabilmesi önce ilgili prosesin sonlanması gerekir. Windows'ta GetExitCodeProcess fonksiyonu prosesin exit kodunu almakta kullanılır:

```
BOOL WINAPI GetExitCodeProcess(
    HANDLE hProcess,
    LPDWORD lpExitCode
);
```

Fonksiyonun birinci parametresi exit kodunun alınacağı prosesin handle değeridir. İkinci parametresi exit kodunun yerleştirileceği DWORD türden nesnenin adresini alır. Eğer söz konusu proses henüz sonlanmamışsa exit kodu olarak 259 (STILL_ACTIVE) elde edilir.

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

```
void ExitSys(LPCSTR lpszMsg);
```

```
int main(void)
{
    char cmdLine[] = "Debug\\TestApp.exe";
    STARTUPINFO si = { sizeof(STARTUPINFO) };
    PROCESS_INFORMATION pi;
    DWORD dwExitCode;

    if (!CreateProcess(NULL, cmdLine, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi))
        ExitSys("CreateProcess", EXIT_FAILURE);

    WaitForSingleObject(pi.hProcess, INFINITE);

    if (!GetExitCodeProcess(pi.hProcess, &dwExitCode))
        ExitSys("GetExitCodeProcess", EXIT_FAILURE);

    if (dwExitCode == STILL_ACTIVE) {
        printf("Program is still running...\n");
        exit(EXIT_SUCCESS);
    }

    printf("%lu\n", dwExitCode);

    return 0;
}
```

```
void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }
}
```

```
    exit(EXIT_FAILURE);
}
```

UNIX/Linux sistemlerinde prosesin exit kodu waitxxx isimli POSIX fonksiyonlarıyla elde edilmektedir. Eskiden yalnızca wait fonksiyonu vardı. Ancak wait fonksiyonu yetersiz kaldığı için waitpid ve wait3 fonksiyonları da daha sonra POSIX sistemlerine dahil edildi. wait fonksiyonun prototipi şöyledir:

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

pid_t wait(int *status);
```

wait fonksiyonu hem prosesin sonlanmasını bekler hem de sonlandığında onun exit kodunu alır. Yani wait fonksiyonu uygulandığında eğer alt proses henüz sonlanmamışsa wait fonksiyonu üst prosesi blokede bekletir. Eğer wait fonksiyonu uygulandığında alt proses zaten sonlanmışsa bu durumda wait fonksiyonu hiç blokeye yol açmadan alt prosesin exit kodunu alarak hemen geri döner. wait fonksiyonu uygulandığında birden fazla alt proses çalışıyor olabilir. Bu durumda wait fonksiyonu ilk sonlanan alt prosesin exit kodunu alarak geri dönecektir. Eğer wait fonksiyonu çağrıldığında birden fazla alt proses sonlanmış durumdaysa wait bu alt proseslerin herhangi birinin exit kodunu alarak geri döner. POSIX standartları bu durumda ilk prosesin exit kodunun alınacağı yönünde bir garanti vermemektedir.

wait fonksiyonu exit kodunu aldığı alt prosesin proses id değeri ile geri dönmektedir. Fonksiyon sonlanan prosesin exit kod bilgisini parametresiyle aldığı int türden nesneye yerleştirmektedir. Başarısızlık durumunda ise -1 değerine geri dönmektedir. Yani wait fonksiyonu da başarısız olabilmektedir. (Örneğin beklenecek hiçbir alt proses yoksa wait fonksiyonu başarısız olur.)

Aslında fonksiyonun parametreye yerleştirdiği değer yalnızca exit kodu değildir. Bu int nesnenin bazı bitleri prosesin neden sonlandığına yönelik bilgi de içermektedir. İşte hangi bitlerin hangi amaçla kullanıldığı sistemden sisteme değişebildiği için standart birkaç makro bulundurulmuştur. WIFEXITED(status) makrosu eğer alt proses normal bir biçimde sonlanmışsa sıfır dışı değer verir. Yalnızca normal biçimde sonlanmış alt bir prosesin exit kodu elde edilebilir. Bir proses başka biçimde de (örneğin sinyal dolayısıyla) da sonlanmış olabilir. WEXITSTATUS(status) makrosu ise bize exit kodunu vermektedir. wait fonksiyonun parametresi NULLL adres de geçilebilir. Bu durumda exit kod elde edilmez. Donksiyon yalnızca ilk alt prosesin bitmesini bekler. (Yani wait fonksiyonu Windows'taki WaitForSingleObject ve GetExitCodeProcess fonksiyonlarının birleşimi gibidir.) Örneğin:

```
/* sample.c */

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

void exit_sys(const char *msg);

int main(void)
{
    pid_t pid;
    int status;

    if ((pid = fork()) < 0)
        exit_sys("fork");

    if (pid == 0 && execl("app", "app", (char *)NULL) < 0)
        exit_sys("exec1");

    if (wait(&status) < 0)
        exit_sys("wait");

    if (WIFEXITED(status))
        printf("Exit code: %d\n", WEXITSTATUS(status));
    else
        printf("child doesn't terminate normally\n");
}
```

```

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* app.c */

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

int main(void)
{
    int i;

    printf("app is running...\n");

    for (i = 0; i < 10; ++i) {
        printf(".");
        fflush(stdout);
        sleep(1);
    }
    printf("\n");

    return 100;
}

```

waitpid fonksiyonu wait fonksiyonunun daha gelişmiş bir biçimidir. Prototipi de şöyledir:

```

#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);

```

Fonksiyonun birinci parametresi beklenecek alt prosesin id değerini alır. İkinci parametre prosesin exit kodunun yerleştirileceği int nesnesinin adresini almaktadır. Üçüncü parametre ise bekleme işleminin nasıl yapılacağına ilişkin seçenekleri belirtir. Birinci parametre aslında dört farklı biçimde girilebilmektedir:

> 0: Bu durumda birinci parametre beklenecek prosesin id değerini belirtir. Fonksiyon yalnızca bu prosesi bekler.

-1: Bu durumda fonksiyon herhangi bir alt prosesi bekler. Bu seçenek fonksiyonun wait gibi davranmasına yol açar.

0: Bu durumda proses grup id'si fonksiyonu çağıran prosesin grup id'si ile aynı olan herhangi bir proses beklenir.

< -1: Bu durumda fonksiyon proses grup id'si burada belirtilen değer mutlak değeri olan proseslerin herhangi birini bekler.

Fonksiyonun son parametresi 0 geçilebilir. Fakat 0 yerine WNOHANG biçiminde de geçilebilir. Bu durumda ilgili alt proses sonlanmamışsa sonlanması beklenmez. waitpid fonksiyonunun aşağıda çağırmasına bakınız:

```
waitpid(-1, &status, 0)
```

Bu çağırım aşağıdaki wait çağırımı ile eşdeğerdir:

```
wait(&status);
```

Yine fonksiyonun ikinci parametresi wait fonksiyonunda olduğu gibi NULL geçilebilir.

waitpid fonksiyonu başarı durumunda beklenen alt prosesin id değeri ile, başarısızlık durumunda -1 değeri ile geri dönmektedir. Ancak fonksiyonun son parametresi WNOHANG geçilmişse ve henüz beklenen alt proses sonlanmamışsa bu durumda waitpid 0 değeri ile geri döner. Örneğin:

```
result = waitpid(pid, NULL, WNOHANG);

if (result == -1)
    exit_sys("waitpid");
if (result != 0)
    if (WIFEXITED(status))
        printf("Child exit code = %d\n", WEXITSTATUS(status));
    else
        printf("child doesn't terminate normally\n");
```

Anahtar Notlar: UNIX/Linux sistemlerinde kabul üzerinde son çalıştırılan programın exit kodu \$? ile kabuktan alınabilmektedir. Benzer biçimde Windows sistemlerinde de kabuk üzerinde son çalıştırılan prosesin exit kodu %errorlevel% ile elde edilebilir.

UNIX/Linux Sistemlerinde Hortlak (Zombie) Proses Kavramı

UNIX/Linux sistemlerinde proseslerin exit kodları Proses Kontrol Bloğuna yazılır ve onların üst prosesleri tarafından buradan alınır. Bu nedenle bir proses bittiği halde henüz onun exit kodunu wait fonksiyonlarıyla üst proses almamışsa prosesin kontrol bloğu işletim sistemi tarafından serbest bırakılmaz. İşte sonlandığı halde exit kodu alınmamasından dolayı kontrol bloğu boşaltılmamış proseslere hortlak (zombie) prosesler denilmektedir. wait fonksiyonlarıyla exit kod alındığında hortlaklık ortadan kalkar.

UNIX/Linux sistemlerinde eğer üst proses alt procesten daha önce sonlanmışsa bu durumdaki alt proseslere öksüz (orphan) prosesler denilmektedir. Sistem öksüz duruma düşen procese init prosesini (id'si 1 olan proses) üst proses olarak atar. init de başarılı bir biçimde alt proses sonlandığında onun exit kodunu alarak onu hortlak olmaktan kurtarır. O halde hortlaklık yalnızca "alt proses sonlandığı halde üst prosesin wait fonksiyonu uygulamadığı ve çalışmasına devam ettiği durumlarda söz konusu olmaktadır. Eğer alt proses sonlandıktan sonra üst proses de sonlanırsa yine init proses alt prosesin exit kodunu alarak onun hortlak duruma düşmesini engeller. Pekiyi hortlaklığın önemi nedir? Bazı programlar çok uzun süre (örneğin bir seneden fazla) çalışabilmektedir. Bunlar sürekli hortlak proses üretirse sistem kaynaklarını tüketebilir. Ayrıca hortlak proseslerin id değeri de başka bir procese verilememektedir. wait fonksiyonlarının dışında otomatik hortlaklığı engellemenin başka yolları da vardır. Fakat en normal durum fork işlemi yapan üst prosesin wait fonksiyonlarından birini uygulayarak alt prosesini hortlaklıktan kurtarmasıdır. Hortlak proses oluşturan bir örnek şöyle verilebilir:

```
/* app.c */

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

int main(void)
{
    pid_t pid;

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

    getchar();      /* child process is zombie state */

    return 0;
}
```

Hortlak prosesler ps komutunda proses durumu (process state) 'Z' biçiminde gösterilmektedir.

```
csd@csd-virtual-machine ~ $ ps -al
F S  UID      PID     PPID  C  PRI  NI ADDR  SZ  WCHAN  TTY          TIME CMD
0 S  1000    34425   4089  0   80   0  -   1089  wait_w pts/0        00:00:00 app
1 Z  1000    34426   34425  0   80   0  -     0  exit   pts/0        00:00:00 ap <defunct>
0 R  1000    34456   34434  0   80   0  -   7583  -      pts/1        00:00:00 ps
```

Burada çalıştırılan program "app" isimli programdır. Onun proses id'si 34425'tir. app prosesinin fork yoluyla yarattığı prosesin id'sinin 34426 olduğuna dikkat ediniz. 34426 id'sine sahip bu proses hortlak durumdadır. Programda biz ENTER tuşuna bastığımızda üst proses sonlanacak ve alt proses'e de init prosesi üst proses olarak atanacak ve alt proses hortlaklık durumundan çıkarılacaktır.

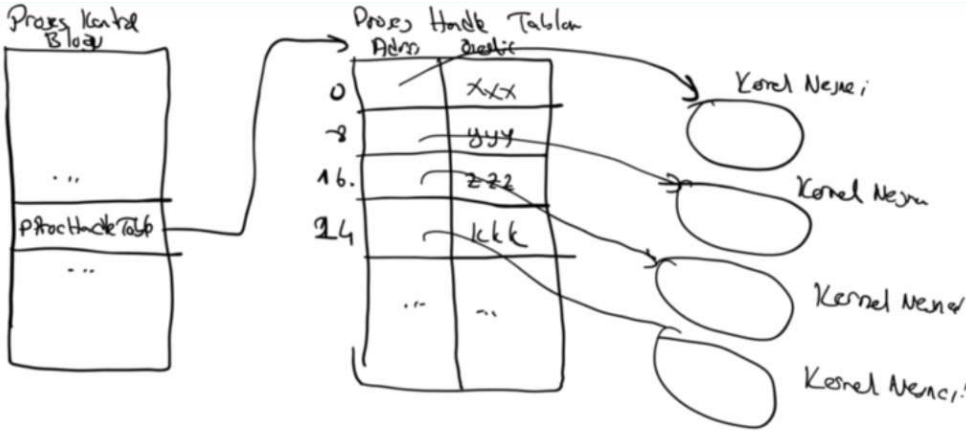
Peki UNIX/Linux sistemlerinde üst proses alt prosesi wait ile beklemek istemiyorsa bu hortlaklık durumu yine de engellenebilir mi? İşte bunun için iki temel yöntem vardır. Birincisinde üst proses SIGCLD isimli sinyali ele alır. Alt proses sonlanıp bu sinyal için set edilen fonksiyon çağrıldığında bu fonksiyon da hiç bekleme yapmadan wait uygulayabilir. İkinci yöntemde daha alt proses yaratılmadan önce üst proses alt prosesin exit kodunu almayacağını beyan eder. Böylece sistem alt proses sonlandığında hemen onun proses kontrol bloğunu yok edip onun hortlak duruma düşmesini engellemektedir. Bu işlemler "UNIX/Linux Sistem Programlama" kursunda ayrıntılı biçimde ele alınmaktadır.

Windows sistemlerinde hortlak proses bir kavram olarak kullanılmamaktadır. Bu sistemlerde üst proses alt prosesin handle değerini tuttuğu için üst proses sonlandığında alt prosesin handle alanı da sistem tarafından zaten boşaltılmaktadır. Windows sistemlerinde UNIX/Linux sistemlerinde olduğu gibi fork ve exec biçiminde iki işlemin olmadığını proseslerin zaten yeni bir kodla çalışmaya başladığını anımsayınız. Dolayısıyla bu sistemlerde alt proses yaratımı UNIX/Linux sistemleri kadar çok ve yoğun olmamaktadır. Windows sistemlerinde alt prosesi yaratan üst proses istediği zaman CloseHandle API fonksiyonuyla onun handle değerini serbest bırakabilmektedir. Bu durumda da alt proses sonlandığında zaten onun handle değerini tutan bir proses kalmayacağı için onun tuttuğu handle alanı da otomatik olarak boşaltılacaktır. Ayrıca Windows sistemlerinde prosesin exit kodunu alan API fonksiyonun UNIX/Linux sistemlerinde olduğu gibi bekleme yapmadığını anımsayınız. Tüm bunlar dikkate alındığında bu sistemlerde hortlak proses kavramının pek tehlikeli bir boyuta gelmeyeceği görülmektedir. Özetle Windows sistemlerinde üst proses eğer alt prosesin exit kodunu alıp kullanmayacaksa CreateProcess işleminden sonra CloseHandle API fonksiyonu ile tuttuğu handle değerini serbest bırakır. Böylece alt proses sonlandığında sistem alt prosesin Proses Kontrol Bloğunu muhafaza etmez. UNIX/Linux sistemlerinde hortlaklığın otomatik olarak ortadan kaldırılması için birkaç yöntem kullanılabilir. Bu yöntemler "UNIX/Linux Sistem Programlama" kurslarında ele alınmaktadır.

Windows Sistemlerinde Handle Değerlerinin Anlamı

Anımsanacağı gibi Windows sistemlerinde CreateFile API fonksiyonuyla bir dosya açıldığında fonksiyon HANDLE ismiyle temsil edilen bir handle değerini bize veriyordu. Daha sonra dosya işlemleri bu handle değeri ile yürütülüyordu. HANDLE türü void * olarak typedef edilmiştir. Windows'ta aslında yalnızca dosyaların handle değerleri değil ismine "Kernel Nesneleri (Kernel Object)" denilen tüm nesnelere benzer mekanizmaya sahiptir. Yani dosyalar için söz konusu olan mekanizma diğer kernel nesneleri için de aynı biçimde söz konusudur. Windows'ta kernel nesnelere denilen bir grup nesne aynı biçimde organize edilmektedir. Bu tasarım UNIX/Linux sistemlerinden biraz farklıdır. Ancak genel veri yapısı bir bakımdan UNIX/Linux sistemlerine de benzemektedir. Daha önceden de belirtildiği gibi Windows'ta her prosesin bir "Proses Handle Tablosu" vardır. Windows'taki Proses Handle Tablosu UNIX/Linux sistemlerindeki Dosya Betimleyici Tablosuna benzetilebilir. Ancak UNIX/Linux sistemlerinde Dosya Betimleyici Tablosu yalnızca dosya betimleyicileri için kullanılırken Windows'ta Proses Handle Tablosu tüm kernel nesnelere için kullanılmaktadır.

Aslında Windows sistemlerinde her ne kadar HANDLE türü void * olarak typedef edilmişse de Proses Handle Tablosunda bir indeks belirtmektedir. Yani bu handle değerleri aslında birer adres değil tıpkı UNIX/Linux sistemlerinde olduğu bir tamsayısal indeks (tablonun başlangıcına göre offset) belirtir. Windows'taki sistem şekilsel olarak şöyle betimlenebilir:

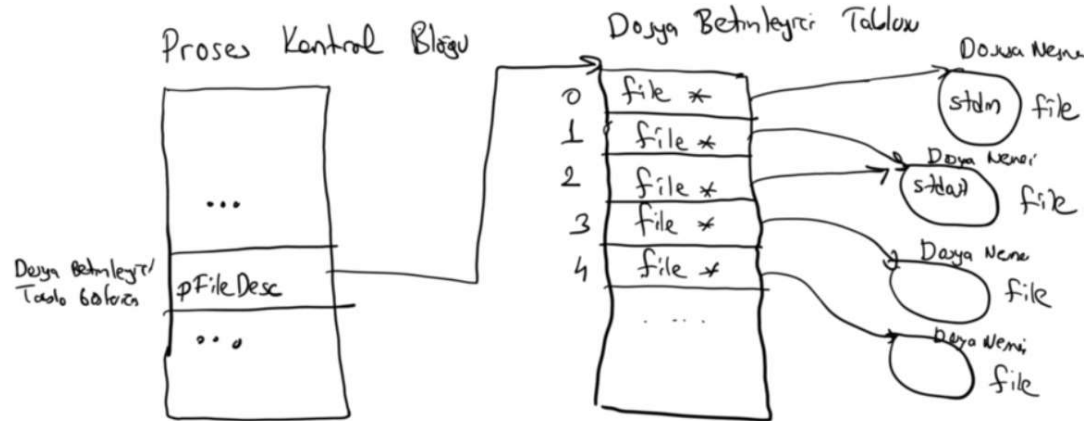


Windows sistemlerinde elde ettiğimiz handle değeri aslında Proses Handle Tablosundaki ilgili girişin tablonun başından itibaren byte uzunluğudur. Örneğin yukarıdaki şekilde biz CreateFile işleminden aslında 24 değerine sahip void * türünden bir handle elde etmiş olabiliriz. Bu sistemlerde her handle ayrıca bir özellik (mode) bilgisine de sahiptir. Böylece örneğin bu sistemlerde bir aynı kernel nesnesi değişik erişim haklarıyla birden fazla kez açılabilir.

UNIX/Linux Sistemlerinde Dosya Betimleyicilerinin Anlamı

Bir dosya open fonksiyonuyla açıldığında işletim sistemi bu dosyayla ilgili işlemleri yönetebilmek için kernel alanı içerisinde ismine "dosya nesnesi (file object)" denilen bir veri yapısı oluşturur. Açık her dosya için bir dosya nesnesi vardır. Linux kaynak kodlarında file isimli yapı bu dosya nesnesini temsil etmektedir. Bir prosesin açmış olduğu tüm dosyaların dosya nesneleri adres olarak dolaylı biçimde proses kontrol bloğunda tutulmaktadır.

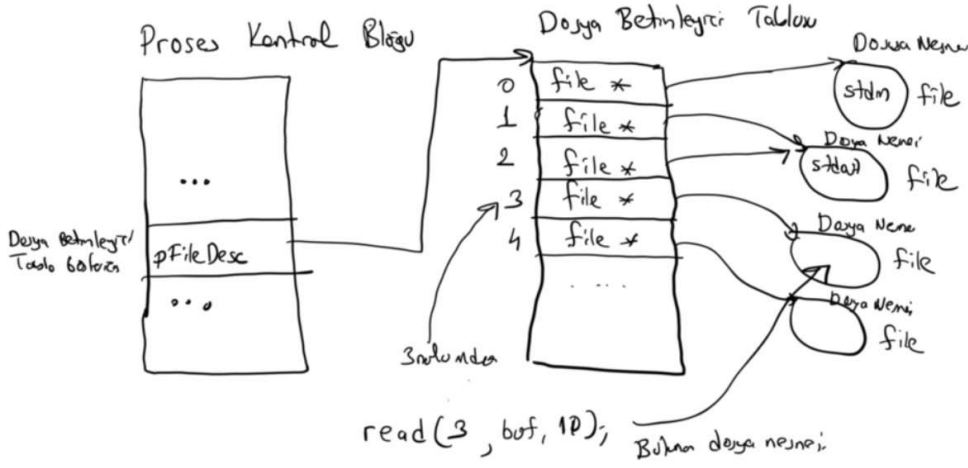
Proses kontrol bloğundaki bir gösterici ismine "Dosya Betimleyici Tablosu" denilen bir tabloyu göstermektedir. Dosya betimleyici tablosu dosya nesnelerinin adreslerini tutan bir gösterici dizisidir. (Örneğin Linux için bu dizinin her bir elemanı struct file * türündendir). Açılmış olan bir dosyanın tüm bilgileri bu dosya nesnesinin içerisinde yer almaktadır. İşletim sisteminin dosya sistemi dosya ile ilgili read/write gibi bir işlem yapacağı zaman bu dosya nesnesinin içerisindeki bilgileri kullanmak zorundadır. İşte open fonksiyonunun bize verdiği dosya betimleyicisi de aslında "Dosya Betimleyici Tablosunda" bir indeks belirtmektedir.



İşletim sistemi dosya betimleyicisini gördüğünde prosesin kontrol bloğuna erişir. Oradan prosesin Dosya Betimleyici Tablosunu bulur. Bu dosya betimleyicisini indeks yaparak dosya nesnesinin adresini elde eder. Örneğin 3 numaralı betimleyiciyi kullanarak read fonksiyonuyla bir okuma yapmak isteyelim:

```
read(3, buf, 10);
```

İşletim sistemi önce 3 numaralı betimleyiciye karşılık gelen dosya nesnesini bulmaya çalışacaktır.

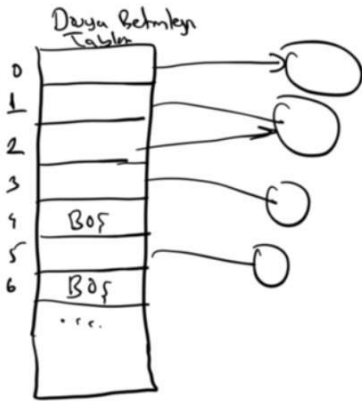


Dosya için gereken tüm bilgiler dosya nesnesinin içerisinde.

Burada en önemli noktalardan biri şudur: Dosya Betimleyici Tablosu toplamda bir tane değildir. Her prosesin ayrı bir Dosya Betimleyici Tablosu vardır. Dolayısıyla bir dosya betimleyicisi o proses için anlamlıdır. Örneğin 3 numaralı dosya betimleyicisi bir proste falanca dosyayı belirtirken diğer bir proste falanca dosyayı belirtiyor olabilir.

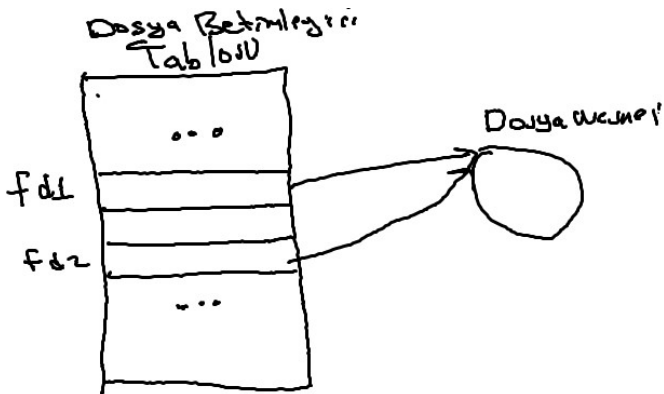
Bir dosya kapatıldığında eğer dosyayı gösteren başka bir betimleyici yoksa dosya nesnesi yok edilir. Dosya Betimleyici Tablosundaki ilgili giriş de boşaltılır. Yani belli bir anda Dosya Betimleyici Tablosu'nun bazı girişleri dolu bazıları boş (yani NULL değerinde) olabilmektedir.

POSIX standartlarında open fonksiyonunun tablodaki ilk boş betimleyiciyi vereceği garanti edilmiştir. Örneğin:



Burada open fonksiyonu uygulansa open başarı durumunda bize 4 numaralı betimleyiciyi verecektir.

Bazen aynı dosya nesnesini gösteren birden fazla betimleyici olmasını isteyebiliriz. Örneğin:

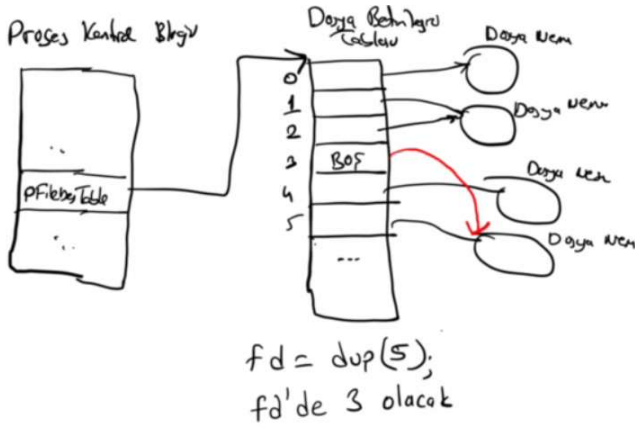


Bu şekildeki fd1 betimleyicisi ile fd2 betimleyicisi aynı dosya nesnesini gösteriyor durumdadır. Dolayısıyla programcı read/write gibi işlemlerde fd1 ya da fd2'yi kullanabilir. Böyle bir durumda programcının fd1 ya da fd2 betimleyicisini kullanması arasında hiçbir farklılık yoktur. İki farklı betimleyicinin aynı dosya nesnesini gösterir hale getirilmesine "dosya betimleyicinin çiftlenmesi" denilmektedir. Dosya betimleyicilerinin çiftlenmesiyle ilgili iki önemli POSIX fonksiyonu vardır. Bunlar dup ve dup2 fonksiyonlarıdır. Bu fonksiyonlar aynı dosya nesnesini gösteren ikinci bir betimleyiciyi bize verirler. dup fonksiyonunun prototipi şöyledir:

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

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

dup fonksiyonu parametre olarak bir dosya betimleyicisi alır. O dosya betimleyicisinin gösterdiği dosya nesnesini gösteren ikinci bir betimleyici oluşturur ve o betimleyiciyi geri dönüş değeri olarak verir. dup fonksiyonunun yine en düşük numaralı boş betimleyiciyi vermesi garanti edilmiştir. Tabii dup fonksiyonu başarısız da olabilir. Bu durumda -1 değerine geri döner.



Aşağıdaki gibi bir dup işleminin yapıldığını düşünelim:

```
fd2 = dup(fd1);
```

Artık dosya işlemlerinde fd1'i kullanmakla fd2'yi kullanmak arasında bir fark yoktur. Örneğin:

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

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

```
int main(void)
{
    int fd1, fd2;
    char buf[11];
    ssize_t result;

    if ((fd1 = open("sample.c", O_RDONLY)) == -1)
        exit_sys("open");

    if ((fd2 = dup(fd1)) == -1)
        exit_sys("dup");

    if ((result = read(fd1, buf, 10)) == -1)
        exit_sys("read");

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

```

if ((result = read(fd2, buf, 10)) == -1)
    exit_sys("read");

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

close(fd1);
close(fd2);

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Açılmış olan dosyaların erişim hakları, dosya göstericisinin konumu gibi bilgiler dosya nesnesinin içerisinde tutulmaktadır. Dolayısıyla yukarıdaki örnekte iki farklı betimleyici aynı dosya nesnesini gösterdiği için aynı dosya göstericisi konumunu kullanıyor olmaktadır.

dup2 fonksiyonun prototipi ise şöyledir:

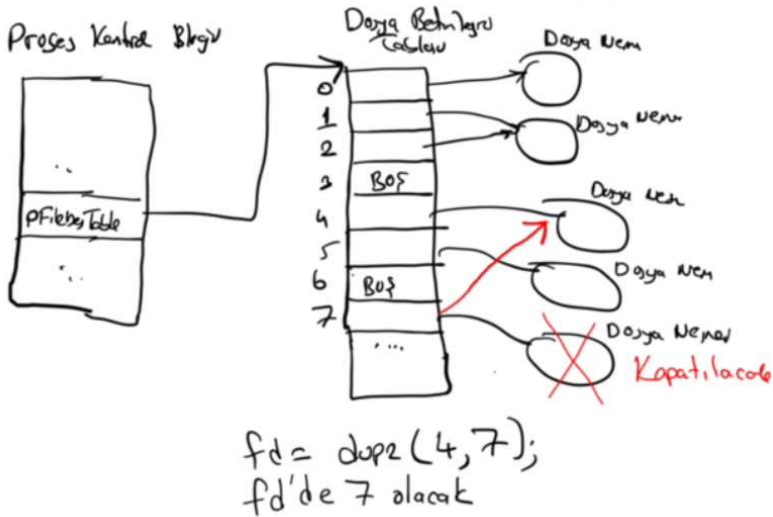
```

#include <unistd.h>

int dup2(int oldfd, int newfd);

```

Fonksiyon yine birinci parametresiyle belirtilmiş olan betimleyiciyi çiftler. Ancak en düşük numaralı boş betimleyici yerine ikinci parametreyle belirtilen betimleyiciyi bize verir. Eğer ikinci paramrede belirtilen betimleyici açık bir dosya nesnesini gösteriyorsa önce onu kapatır. Sonra bu betimleyicinin ilk paramtredeki betimleyici ile aynı dosya nesnesini göstermesini sağlar. Fonksiyon başarı durumunda ikinci parametresiyle belirtilen betimleyici değerine, başarısızlık durumunda ise -1 değerine geri dönmektedir. Örneğin:



dup2 fonksiyonunu aşağıdaki gibi kullandığımızı düşünelim:

```
fd3 = dup2(fd1, fd2);
```

Burada artık fd2 betimleyicisi fd1 ile aynı dosya nesnesini gösteriyor durumdadır. Fonksiyon fd2 ile aynı değere geri dönecektir. Yani fd3 ile fd2 çağrı sonunda (tabii dup2 başarılı ise) aynı değere sahip olacaktır.

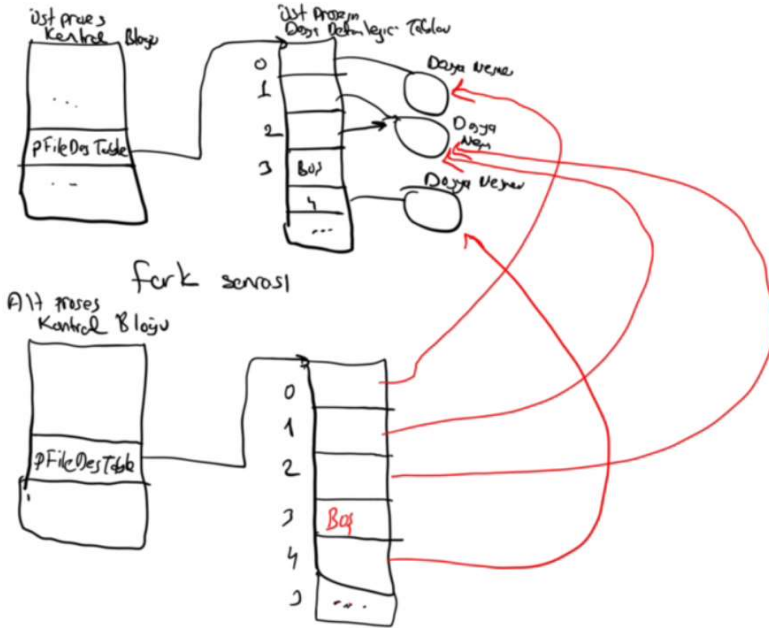
Dosya Betimleyici Tablosu'nun bir uzunluğu vardır. Örneğin Linux sistemlerinde default olarak her prosesin Dosya Betimleyici Tablosu 1024 eleman uzunluğundadır. Yani biz bu sistemlerde aynı anda en fazla 1024 dosyayı açık durumda tutabiliriz. Tabii root kullanıcısı isterse kendi ya da başka bir prosesin Dosya Betimleyici Tablosunu büyütebilir. Ancak bunu sıradan kullanıcılar yapamamaktadır.

Peki bir dosya kapatıldığında ne olmaktadır? İşte işletim sistemi önce Dosya Betimleyici Tablosundaki dosya betimleyicisi ile belirtilen indeksi boşaltır. Fakat hemen dosya nesnesini silmez. Çünkü kapatılan dosya betimleyicisinin gösterdiği dosya nesnesini başka betimleyiciler de gösteriyor olabilir. Dosya Nesnelerinin içerisinde bir referans sayacı da tutulmaktadır. Aynı dosya nesnesinin birden fazla betimleyici tarafından gösterildiği durumda bu betimleyicilerden biri close fonksiyonuyla dosyayı kapatırsa işletim sistemi Dosya Betimleyici Tablosunda bu betimleyiciyi boşaltır ve dosya nesnesindeki sayacı 1 eksiltir. Eğer sayacı 0'a düşmüşse dosya nesnesini siler. Her dup işlemi ve fork işlemi nesne sayacını 1 artırmaktadır. Linux sistemlerindeki dosya nesnesini temsil eden file yapısını inceleyiniz (Kernel 2.4.22):

```
struct file {
    struct list_head    f_list;
    struct dentry       *f_dentry;
    struct vfsmount     *f_vfsmnt;
    struct file_operations *f_op;
    atomic_t           f_count;
    unsigned int        f_flags;
    mode_t              f_mode;
    loff_t              f_pos;
    unsigned long       f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    struct fown_struct  f_owner;
    unsigned int        f_uid, f_gid;
    int                 f_error;
    unsigned long       f_version;
    void                *private_data;
    struct kiobuf       *f_iobuf;
    long                f_iobuf_lock;
};
```

Yapının f_count elemanı nesne sayacını, f_pos elemanı dosya göstericisinin konumunu, f_uid, f_gid elemanları dosyanın kullancısı ve grup id'lerini belirtmektedir.

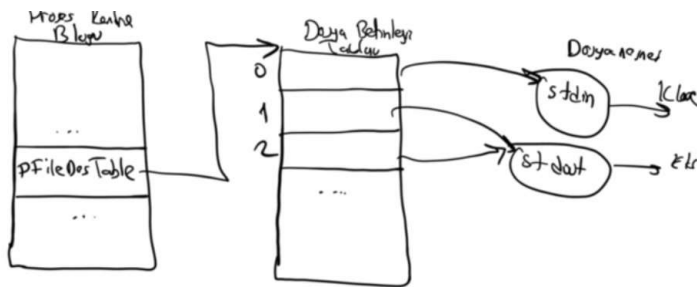
fork işlemi sırasında üst prosesin açmış olduğu dosyalar alt proseste de açık biçimde görülmektedir. fork fonksiyonu üst prosesin Dosya Betimleyici Tablosunu alt proseste kopyalamaktadır. Böylece alt prosesin Dosya Betimleyici Tablosu girişleri üst prosesin Dosya Betimleyici Tablosu girişleri ile eşdeğer duruma getirilir. Tabii bu işlem sırasında dosya nesnelerinin kopyaları çıkartılmaz. Yalnızca Dosya Betimleyici Tablosundaki adresler kopyalanır. Başka bir deyişle fork işleminden sonra üst ve alt proseste aynı numaralı betimleyiciler aynı dosya nesnesini gösteriyor durumda olur. Tabii dosya nesnelerinin sayacı da birer artırılmış olacaktır. Örneğin:



fork işleminden sonra exec işlemi yapıldığında artık yeni bir program çalışacağına göre çalışan program üst prosesin açmış olduğu dosyalarla ilgilenmeyecektir. (Tabii ilgilenecek biçimde de yazılmış olabilir.) İşte exec yapıldığında açık olan dosyaların otomatik kapatılması sağlanabilmektedir. İşletim sistemi açık olan her dosya için o prosese özgü bir "close on exec" bayrağı tutar. Bu bayrak default durumda "clear" edilmiştir. Yani exec işlemi yapıldığında dosya betimleyicisi otomatik olarak kapatılmaz. Fakat istenirse fcntl isimli POSIX fonksiyonu ile bu bayrak "set" edilebilir. Bu durumda exec işlemi sırasında açık olan betimleyici otomatik biçimde kapatılmaktadır.

UNIX/Linux Sistemlerinde Dosya Yönlendirmesi

Anımsanacağı gibi UNIX/Linux ve Windows sistemlerinde aygıt sürücüler de aslında birer dosya gibi organize edilmektedir. Kabuk üzerinden proses yaratıldığında üst prosesten 0, 1 ve 2 numaralı betimleyiciler dolu olarak gelir. 0 numaralı betimleyicinin gösterdiği dosya nesnesi "stdin" (tipik olarak klavye) aygıt sürücüsü ile, 1 numaralı betimleyicinin gösterdiği dosya nesnesi ise "stdout" (tipik olarak ekran) aygıt sürücüsü ile ilişkilendirilmiştir. 2 numaralı betimleyici dup işlemi yapılarak elde edilmiştir ve o da 1 numaralı betimleyici ile aynı dosya nesnesini gösterir. Bu 2 numaralı betimleyiciye "stderr" denilmektedir ve default durumda "terminal" aygıt sürücüne yönlendirilmiştir. Yani işin başında UNIX/Linux sistemlerindeki bir proses tipik olarak şöyle başlatılmaktadır:



Böylece bu sistemlerde ilk açtığımız dosya için bize 3 numaralı betimleyici verilir. Bu sistemlerde hangi dilden hangi fonksiyonu kullanacak olursak olalım stdin dosyasından okuma yapıldığında eninde sonunda 0 numaralı betimleyici kullanılarak read işlemi yapılmaktadır. Benzer biçimde stdout dosyasına bir şeyler yazılmak istendiğinde de eninde sonunda 1 numaralı betimleyici kullanılarak write işlemi yapılır. O halde yönlendirme basit bir biçimde şöyle yapılabilir:

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

```

void exit_sys(const char *msg);

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

    close(1);

    if ((fd = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) < 0)
        exit_sys("open");

    for (i = 0; i < 10; ++i)
        printf("%d\n", i);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

open fonksiyonunun ilk boş betimleyiciyi verdiğini anımsayınız. Yukarıdaki örnekte önce 1 numaralı betimleyici kapatılmış ve hemen arkasından open işlemi yapılmıştır. Dolayısıyla artık 1 numaralı betimleyici stdout dosya nesnesini değil, "test.txt" isimli dosyaya ilişkin dosya nesnesini gösteriyor durumda olur. Fakat yönlendirmede dup2 fonksiyonunu kullanmak daha güvenlidir. Çünkü çok thread'li uygulamalarda bu tür işlemler senkronizasyon sorununa yol açabilirler. O halde yönlendirme daha biçimsel olarak aşağıdaki gibi yapılmaktadır:

```

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

void exit_sys(const char *msg);

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

    if ((fd = open("test.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) <
0)
        exit_sys("open");

    if (dup2(fd, 1) < 0)
        exit_sys("dup2");

    for (i = 0; i < 10; ++i)
        printf("%d\n", i);

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Peki kabuk '>' ya da '<' işareti ile yönlendirmeyi nasıl yapmaktadır? İşte kabuk önce bir kez fork yapar. Yönlendirmeyi yaptıktan sonra exec yapar. Böylece çalışan program kodu artık bu yönlendirmeye göre çalışacaktır. Aşağıda kabuğun nasıl yönlendirme yaptığına ilişkin bir örnek veriyoruz:

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

void exit_sys(const char *msg);

/* redirect <dosya > <program> [argüman listesi] */

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

    if (argc < 3) {
        fprintf(stderr, "wrong number of arguments!..\nusage: redirect <dosya > <program> [argüman listesi]\n");
        exit(EXIT_FAILURE);
    }

    if ((pid = fork()) == -1)
        exit_sys("fork");

    if (pid == 0) {
        if ((fd = open(argv[1], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
            exit_sys("open");

        if (dup2(fd, 1) == -1)
            exit_sys("dup2");

        close(fd);

        if (execvp(argv[2], &argv[2]) == -1) {
            remove(argv[1]);
            exit_sys("execvp");
        }
    }

    if (wait(NULL) == -1)
        exit_sys("wait");

    return 0;
}

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

Bu örnekte kullanıcı programı çalıştırırken yönlendirmenin yapılacağı dosyayı ve çalıştıracığı programı komut satırı argümanlarıyla birlikte komut satırı argümanı olarak verir. Örneğin:

```
redirect test.txt ls -l
```

Böylece ls programını "-l" seçeneğiyle çalıştırılacak ancak bunun stdout dosyası test.txt dosyasına yönlendirilecektir. Programda exec fonksiyonu olarak execvp'nin kullanıldığına dikkat ediniz. argv dizisinin sonunda NULL adres olacağı için çalıştırılacak programın komut satırı argümanları kolay bir biçimde execvp fonksiyonuna aktarılmıştır.

Proseslerarası Haberleşme (Interprocess Communication - IPC)

Önceki konularda da gördüğümüz gibi modern işletim sistemlerindeki de sayfa tabloları yoluyla prosesler arasında tam bir izolasyon sağlanmıştır. Bu sistemlerde her proses sanki fiziksel belleğe tek başına yüklenip çalışıyormuş gibi bir illüzyon oluşturulmaktadır. Bir prosten diğerine belli miktarda byte gönderip alma sürecine "Proseslerarası Haberleşme" denilmektedir. Proseslerarası haberleşme kabaca ikiye ayrılır:



Proseslerarası haberleşme yöntemlerine pek çok durumda gereksinim duyulmaktadır. Örneğin bir "chessboard.exe" isimli satranç tahtası programını düşünelim. Bu program bir GUI ortamı oluşturarak taşların hareketlerini sağlıyor olsun. İşte bu programa hangi taşı nereye oynatacağını başka bir program proseslerarası haberleşme yöntemleriyle iletiyor olabilir. Ya da bir araba simülatoründeki bir proses arabanın o andaki konumunu proseslerarası haberleşme yöntemleriyle araba kabinini yöneten başka bir prostese iletebilir. İstemci-Sunucu (Client-Server) mimarisinde arka planda hep proseslerarası haberleşme yöntemleri kullanılmaktadır. İstemci-Sunucu mimarisinde asıl işi yapan programa "sunucu (server)" denilmektedir. "İstemci (client)" olarak çalışan program proseslerarası haberleşme yöntemleriyle isteğini sunucu programa iletir. Sunucu program da işlemini yapıp sonuçları yine proseslerarası haberleşme yöntemleriyle istemciye yollayabilir. İstemci-Sunucu mimarisi ileride ele alınmaktadır.

Aynı makinenin prosesleri arasındaki haberleşmelerde kullanılan tipik teknikler şunlardır:

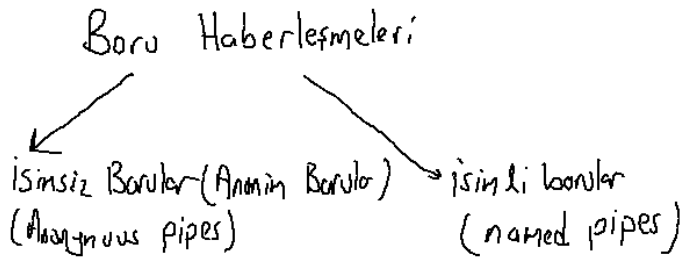
- Borular (Pipes)
- Paylaşılan Bellek Alanları (Shared Memory)
- Mesaj Kuyrukları (Message Queue)

Ağ altında farklı makinelerin prosesleri arasında haberleşmede bir protokol de devreye girmektedir. Bu amaçla oluşturulmuş çeşitli protokol aileleri vardır. Bugün en yaygın kullanılan protokol ailesi Internet'in de kullandığı IP protokol ailesidir. Kursumuzda IP protokol ailesi ile haberleşme ele alınacaktır.

Biz kursumuzda önce Windows ve UNIX/Linux sistemlerinde en yaygın olarak kullanılan "aynı makinenin prosesleri arasındaki" haberleşme yöntemleri üzerinde sonra da IP ailesi ile farklı makinelerin prosesleri arasında haberleşme yöntemleri üzerinde duracağız.

Boru Haberleşmeleri

Boru haberleşmeleri hem UNIX/Linux sistemlerinde hem de Windows sistemlerinde kullanılan yaygın bir tekniktir. Bu haberleşme yönteminde senkronizasyon da otomatik sağlanmaktadır. Boru haberleşmeleri hem UNIX/Linux hem de Windows sistemlerinde "İsimli (named)" ve "İsimsiz (unnamed/anonymous)" olmak üzere ikiye ayrılmaktadır.



Borular UNIX/Linux sistemlerinde tek kanallıdır. Yani bir taraf yazar, diğer taraf okur (half duplex). Fakat tek bir boruyla aynı anda iki tarafın yazıp okuması mümkün değildir. Windows sistemlerinde borular çift kanallıdır. Dolayısıyla bir boru yaratıldığında bir tarafın yazdığını diğer taraf okurken, diğer tarafın yazdığını da karşı taraf okuyabilir. Aynı durumu UNIX/Linux sistemlerinde sağlamak için iki boru yaratmak gerekir.

İsimsiz borular yalnızca üst ve alt prosesler arasında haberleşmede kullanılmaktadır. İsimli borular ise aynı makine içerisindeki herhangi iki proses arasındaki haberleşmede kullanılabilir.

Tipik bir boru haberleşmesi şu aşamalardan geçilerek gerçekleştirilir:

- 1) Önce boru yaratılır.
- 2) Proseslerden biri boruya bilgiyi yazar. Genellikle borular işletim sistemlerinde sanki birer dosyaymış gibi ele alınmaktadır. Bu nedenle yazma ve okuma işlemleri dosya fonksiyonlarıyla yapılır. (UNIX/Linux sistemlerinde read ve write POSIX fonksiyonları, Windows sistemlerinde ReadFile ve WriteFile API fonksiyonları).
- 3) Diğer proses bilgiyi okur. Borular birer FIFO kuyruk sistemi gibidir. Borularda stream tabanlı bir haberleşme söz konusudur. Stream tabanlı haberleşme demek okuyanın istediği kadar byte okuyabilmesi (yani bir bilgiyi birden fazla okuma eylemiyle ele elde edebilmesi) demektir.
- 4) Borunun belli bir uzunluğu vardır. Eğer yazan tarafın yazdığını, okuyan almazsa (bu durum okuyan tarafın yavaş kalması sonucunda da oluşabilir) boru dolar. Boru dolduğu zaman yazan taraf yazmaya çalıştığında bloke oluşur. (Bloke (blocking) ileride başka konularda ele alınacaktır. Bloke olmak ilgili akışın donarak bekletilmesi anlamına gelmektedir.) Ta ki karşı taraf okuyup boruda yer açılana kadar. Benzer biçimde okuyan taraf borudakilerin hepsini okumuşsa ve yeniden borudan okuma yapmak isterse yine bloke oluşur. Yani o da donarak bekler. Ta ki diğer taraf boruya birşeyler yazana kadar.
- 5) Boru haberleşmesi yazan tarafın boruyu kapatmasıyla sonlandırılmalıdır. Bu durumda okuyan taraf önce boruda kalanları okur. Sonra artık okuma fonksiyonu sıfır ile geri döner. Okuyan taraf da böylece karşı tarafın boruyu kapattığını anlar. O da boruyu kapatır. Önce okuyan tarafın boruyu kapatması normal bir durum değildir. Bu genellikle sistemlerde çökmeye yol açar.

Aslında boru haberleşmeleri blokeli (blocking) ya da blokesiz (nonblocking) modda yapılabilir. Biz burada default durum olan blokeli modda boru haberleşmeleri üzerinde duracağız.

UNIX/Linux Sistemlerinde İsimsiz Boru Haberleşmeleri

Yukarıda da belirtildiği gibi isimsiz boru haberleşmeleri üst ve alt prosesler arasında yapılmaktadır. Blokeli modda UNIX/Linux sistemlerinde isimsiz boru haberleşmesi şöyle yapılır:

- 1) Üst proses önce pipe isimli fonksiyonla isimsiz boruyu yaratır.

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

Anahtar Notlar: C'de fonksiyonun gösterici parametresi dizisel biçimde de belirtilebilir ve köşeli parantezlerin içerisine sabit ifadeleri yazılabilir. Bunun okunabilirlik dışında göstericiden hiçbir farkı yoktur. Dolayısıyla aşağıdaki prototiplerin hepsi eşdeğerdir:

```
void foo(int *a);
void foo(int a[]);
void foo(int a[2]);
void foo(int a[100]);
```

Dolayısıyla pipe fonksiyonunun prototipi şöyle de yazılabilirdi:

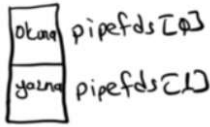
```
int pipe(int *pipefd);
```

pipe fonksiyonu başarı durumunda 0, başarısızlık durumunda -1 değerine geri döner. Fonksiyona iki elemanlı bir int dizinin adresi geçirilmelidir. Örneğin:

```
int pipefds[2];
...
if (pipe(pipefds) < 0) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
```

Fonksiyon verdiğimiz dizinin içerisine iki betimleyici yerleştirir. Dizinin ilk elemanına yerleştirilen betimleyici borudan okuma yapmak için, ikinci elemanına yerleştirilen betimleyici boruya yazma yapmak için kullanılmalıdır.

```
int pipefds[2];
if (pipe(pipefds) < 0) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
```



2) Artık üst proses fork fonksiyonuyla alt proses oluşturur. Böylece üst prosesin boru betimleyicileri alt procese aktarılmış olur. Bu durumda borudan okuma potansiyeli olan ve boruya yazma potansiyeli olan ikişer betimleyici vardır.

```
if ((pid = fork()) < 0) {
    perror("fork1");
    exit(EXIT_FAILURE);
}
```

UNIX/Linux sistemlerinde bir proses fork ile yaratıldığında üst prosesin dosya betimleyicileri alt procese aktarılmaktadır. Yani örneğin üst proses bir dosyayı açmışsa fork işleminden sonra alt proses de bu dosyayı açık görmektedir. İlgili betimleyici her iki procese de aynı dosyaya yazmak ve okumak amacıyla kullanılabilir. Bu konunun ayrıntıları "UNIX/Linux Sistem Programlama" kurslarında ele alınmaktadır. İsimli boru haberleşmesinde de üst proses boruyu yaratıp fork uyguladığında alt proses de aynı boruya erişmek için betimleyicilere sahip olmuş olur. Böylece fork işleminden sonra boruya yazma ve borudan okuma potansiyeline sahip ikişer betimleyici bulunuyor durumdadır.

3) Şimdi hangi prosesin boruya yazma yapacağı, hangisinin okuma yapacağı belirlenir. Artık yazan taraf okuma betimleyicisini, okuyan taraf da yazma betimleyicisini close fonksiyonuyla kapatmalıdır. Yani boruya yazma ve okuma potansiyelinde olan tek bir betimleyici kalmalıdır.

```
if ((pid = fork()) < 0) {
    perror("fork1");
    exit(EXIT_FAILURE);
}
```

```

}

if (pid != 0) {          /* üst proses boruya yazacak */
    close(pipefds[0]);
    ...
}
else {                  /* alt proses borudan okuyacak */
    close(pipefds[1]);
    ...
}

```

4) Artık yazan taraf write fonksiyonuyla boruya yazma yapar. Okuyan taraf da read fonksiyonuyla borudan okuma yapacaktır:

```

#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);

```

read ve write fonksiyonlarını daha önce görmüştük. Bu fonksiyonların birinci parametreleri borunun (dosyanın) handle yani betimleyici değeridir. İkinci parametreler okuma ya da yazma için gereken bellek transfer adresidir. Son parametreler okunacak ya da yazılacak byte miktarını belirtir. Fonksiyonlar başarılıysa okunan ya da yazılan byte sayısına, başarısızsa -1 değerine geri dönerler. read fonksiyonu 0 ile geri dönerse bu durum karşı tarafın boruyu kapatmış olduğu ve artık boruda okunacak hiçbirşeyin kalmadığı anlamına gelir. read boru boşsa, write ise boru doluyorsa zaten geri dönmeden beklerler. Blokeli modda çalışmanın önemli birkaç noktası vardır.

- Borudan n byte okunmak istendiğinde boruda n'den daha az byte varsa read fonksiyonu okuyabildiği kadar byte'ı okuyup okuyabildiği byte sayısına geri döner. Yani örneğin biz borudan tek bir read fonksiyonuyla 100 byte okumak isteyelim. Ancak boruda 80 byte bulunuyor olsun. Bu durumda read fonksiyonu 80 byte'ı okuyarak 80 değeri ile geri dönecektir. Ancak read ile borudan okuma yapılmak istendiğinde eğer boruda okunacak hiç byte yoksa bu durumda read boruya en az bir byte yazılana kadar blokede kalır.

- Birden fazla kaynak aynı boruya yazma yaparken normalde iç içe geçme oluşmaz. Ancak borunun bir uzunluğu vardır (bu uzunluk <limits.h> içerisindeki PIPE_BUF sembolik sabiti ile programcıya verilmektedir). Eğer boruya write fonksiyonu ile boru uzunluğundan daha fazla miktarda byte yazılmaya çalışılırsa bu durumda iç içe geçme oluşabilmektedir. Yazma sırasında da tüm byte'lar yazılana kadar bloke çözülmez. Programcı borunun uzunluğunu değiştiremez ancak PIPE_BUF sembolik sabiti yoluyla bu uzunluğu elde edebilir.

Örnek bir program şöyle yazılabilir:

```

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

void exit_sys(const char *msg);

int main(void)
{
    int pipefds[2];
    pid_t pid;
    int i, val;
    ssize_t result;

    if (pipe(pipefds) < 0)
        exit_sys("pipe");

    if ((pid = fork()) < 0)
        exit_sys("fork");

```

```

if (pid != 0) {
    close(pipefds[0]);
    for (i = 0; i < 100; ++i)
        if (write(pipefds[1], &i, sizeof(int)) < 0)
            exit_sys("write");
    close(pipefds[1]);
    if (wait(NULL) < 0)
        exit_sys("wait");
}
else {
    close(pipefds[1]);

    while ((result = read(pipefds[0], &val, sizeof(int))) > 0) {
        printf("%d ", val);
        fflush(stdout);
    }
    printf("\n");
    if (result < 0)
        exit_sys("read");

    close(pipefds[0]);
}

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Boruya yazma ve borudan okuma işlemlerini bir fonksiyona da devredebiliriz:

```

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

void exit_sys(const char *msg);
void write_pipe(int fd);
void read_pipe(int fd);

int main(void)
{
    int pipefds[2];
    pid_t pid;

    if (pipe(pipefds) < 0)
        exit_sys("pipe");

    if ((pid = fork()) < 0)
        exit_sys("fork");

    if (pid != 0) {
        close(pipefds[0]);
        write_pipe(pipefds[1]);
        close(pipefds[1]);

        if (wait(NULL) < 0)
            exit_sys("wait");
    }
    else {
        close(pipefds[1]);
        read_pipe(pipefds[0]);
    }
}

```

```

        close(pipefds[0]);
    }

    return 0;
}

void write_pipe(int fd)
{
    int i;

    for (i = 0; i < 100; ++i)
        if (write(fd, &i, sizeof(int)) < 0)
            exit_sys("write");
}

void read_pipe(int fd)
{
    ssize_t result;
    int val;

    while ((result = read(fd, &val, sizeof(int))) > 0) {
        printf("%d ", val);
        fflush(stdout);
    }
    printf("\n");
    if (result < 0)
        exit_sys("read");
}

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

    exit(EXIT_FAILURE);
}

```

Pekiye boruya yazmak ve borudan okuma yapmak için yalnızca birer betimleyicinin bırakılmasının anlamı nedir? Anımsanacağı gibi isimsiz boru haberleşmesinde boruya yazan taraf okuma betimleyicisini, okuyan taraf da yazma betimleyicisini kapatıyordu. Böylece toplamda boruya yazma yapan ve borudan okuma yapan birer betimleyici kalıyordu. İşte eğer boruya yazma potansiyelinde olan birden fazla betimleyici varsa ancak son betimleyici kapatıldığında okuyan taraf borunun kapatıldığını anlayabilmektedir. Başka bir deyişle read fonksiyonu eğer boruya yazma yapacak hiçbir betimleyici yoksa 0 değerine geri dönmektedir. Birden fazla betimleyicinin borudan okuma yapma potansiyelinde olmasında genel olarak bir sakınca yoktur. Ancak bu tür haberleşmelerde gereksiz betimleyicilerin kapatılması her zaman iyi bir tekniktir.

Borularla işlemler dosya işlemleri gibi yapılıyor olsa da aslında borular işletim sistemi tarafından kernel alanında bellek oluşturulmaktadır. Yani örneğin bir proses write fonksiyonu ile bir boruya yazma yaptığı zaman aslında bir disk işlemi yapılmamaktadır. Boru kernel alanı içerisinde tahsis edilmiş olan bir bellek bölgesidir. Bu durumda yazma ve okuma işlemleri aslında bu bellek bölgesinden yapılmaktadır. Yani borular dosyaymış gibi işleme sokulmakla birlikte aslında birincil bellekte tahsis edilmiş olan bölgelerdir.

Kabuk Boru İşlemini Nasıl Yapıyor?

Önceki konularda da görüldüğü gibi kabuk üzerinden,

```
a | b
```

gibi bir işlemde a'nın stdout dosyasına yazdıklarını b stdin dosyasından okumaktadır. Pekiye kabuk bu işlemi nasıl yapabilmektedir? Aslında bu işlem isimsiz bir boru yardımıyla özetle şöyle yapılmaktadır: Kabuk bir isimsiz boru oluşturur. Sonra alt prosesleri fork ile yaratır. Ancak exec yapmadan önce a prosesinin stdout dosyasını yazma amaçlı, b prosesinin de stdin dosyasını okuma amaçlı boruya yönlendirir. Sonra da exec işlemini yapar. Böylece a prosesinin

ekrana yazdıkları aslında boruya yazılmaktadır. b prosesinin de klavyeden okudukları ise aslında borudan okunmaktadır. Bu işlemin ayrıntılı adımları madde madde şöyle açıklanabilir:

1) Kabuk önce isimsiz boruyu yaratır.

2) Borular da birer dosya gibi dosya betimleyicisine sahiptir. Kabuk "a" için fork uygular ve "a"nın stdout dosyasını boruya yönlendirir. Sonra da "a" için exec uygular. Böylece "a" stdout dosyasına yazdığını sanırken aslında boruya yazacaktır.

3) Bu kez kabuk "b" için fork uygular ve "b"nin stdin dosyasını boruya yönlendirir. Sonra da "b" için exec uygular. Böylece "b" stdin dosyasından okuduğunu sanırken aslında borudan okuyacaktır.

5) Hem kabul hem de iki alt proses boruya ilişkin gereksiz betimleyicilerini kapatır.

4) Kabuk her iki prosesin de bitmesini wait fonksiyonları ile bekler. Artık "a" prosesinin stdout dosyasına yazdıklarını "b" prosesi "stdin" dosyasından okuyacaktır.

Böyle bir boru işlemi aşağıdaki gibi bir programla yapılabilir:

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

void exit_sys(const char *msg);

/* shellpipe <prog1> args "|" <prog2> args */

int main(int argc, char *argv[])
{
    int i, pindex;
    int pipefds[2];
    pid_t pidout, pidin;

    if (argc < 4) {
        fprintf(stderr, "wrong number old arguments!..\n");
        exit(EXIT_FAILURE);
    }
    pindex = 0;
    for (i = 1; i < argc; ++i)
        if (!strcmp(argv[i], "|")) {
            pindex = i;
            break;
        }

    if (!pindex || pindex == 1 || pindex == argc - 1) {
        fprintf(stderr, "| delimited two programs must be specified!..\n");
        exit(EXIT_FAILURE);
    }

    argv[pindex] = NULL;

    if (pipe(pipefds) == -1)
        exit_sys("pipe");

    if ((pidout = fork()) == -1)
        exit_sys("fork");
    if (pidout == 0) {
        if (dup2(pipefds[1], 1) == -1)
            exit_sys("dup2");
        close(pipefds[0]);
```

```

    close(pipefds[1]);
    if (execvp(argv[1], &argv[1]) == -1)
        exit_sys("execv");
    /* unreachable code!... */
}

if ((pidin = fork()) == -1)
    exit_sys("fork");
if (pidin == 0) {
    if (dup2(pipefds[0], 0) == -1)
        exit_sys("dup2");
    close(pipefds[0]);
    close(pipefds[1]);
    if (execvp(argv[pindex + 1], &argv[pindex + 1]) == -1)
        exit_sys("execv");
    /* unreachable code!... */
}

close(pipefds[0]);
close(pipefds[1]);

if (wait(NULL) == -1)
    exit_sys("wait");

if (wait(NULL) == -1)
    exit_sys("wait");

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Program şöyle çalıştırılabilir:

```
shellpipe ls -l "|" wc
```

Programı çalıştırırken | sembolünü tırnak içerisine almayı unutmayınız. Eğer bu sembolü tırnak içerisine almazsanız kabuk bunu komut satırı argümanı değil kendi boru sembolü olarak ele alacaktır.

Aynı işlemi bir fonksiyona da yaptırabilirdik. Aşağıdaki örnekte runpipe isimli fonksiyon tek bir string almaktadır. Dolayısıyla programın komut satırı argümanı da bir tane olmalıdır. Aralarında boşluk olan fakat tek bir argüman biçiminde işleme sokmak istediğiniz argümanları tırnak içerisine almayı unutmayınız.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <sys/wait.h>
#include <limits.h>

#define MAX_PARAM 10

void exit_sys(const char *msg);

/* runpipe("ls -l | grep "tty") */

```

```

int runpipe(char *cmd)
{
    int fds[2];
    char *ppos, *str;
    char *first[MAX_PARAM], *second[MAX_PARAM];
    int i;
    pid_t pidfirst = 0, pidsecond = 0;
    int result = 0;

    if ((ppos = strchr(cmd, '|')) == NULL)
        return -1;
    *ppos = '\\0';

    i = 0;
    for (str = strtok(cmd, "\\t"); str != NULL; str = strtok(NULL, "\\t"))
        first[i++] = str;
    first[i] = NULL;

    i = 0;
    for (str = strtok(ppos + 1, "\\t"); str != NULL; str = strtok(NULL, "\\t"))
        second[i++] = str;
    second[i] = NULL;

    if (pipe(fds) == -1)
        return -1;

    if ((pidfirst = fork()) == -1)
        goto EXIT;

    if (pidfirst == 0) {
        close(fds[0]);
        if (dup2(fds[1], 1) == -1)
            exit(EXIT_FAILURE);
        close(fds[1]);
        if (execvp(first[0], &first[0]) == -1)
            exit(EXIT_FAILURE);
    }

    if ((pidsecond = fork()) == -1)
        goto EXIT;

    if (pidsecond == 0) {
        close(fds[1]);
        if (dup2(fds[0], 0) == -1)
            exit(EXIT_FAILURE);
        close(fds[0]);
        if (execvp(second[0], &second[0]) == -1)
            exit(EXIT_FAILURE);
    }

EXIT:
    close(fds[0]);
    close(fds[1]);

    if (pidfirst != 0 && waitpid(pidfirst, NULL, 0) == -1)
        result = -1;

    if (pidsecond != 0 && waitpid(pidsecond, NULL, 0) == -1)
        result = -1;

    return result;
}

/* ./shellpipe "<prog1> args | <prog2> args" */

```



```

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "wrong number of arguments!..\n");
        exit(EXIT_FAILURE);
    }
    if (runpipe(argv[1]) == -1) {
        fprintf(stderr, "runpipe error");
        exit(EXIT_FAILURE);
    }

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

UNIX/Linux Sistemlerinde İsimli Boru Haberleşmesi

İsimsiz boru haberleşmesi yalnızca üst proseslerle alt prosesler arasındaki haberleşmelerde kullanılabilir. Halbuki isimli borularla biz aralarında üstlük-altlık ilişkisi olmayan herhangi iki prosesi haberleştirebiliriz. UNIX/Linux sistemlerinde isimli borular sanki birer dosyaymış gibi ele alınmaktadır. Bu boru dosyaları dosya sisteminde izin girişlerinde 'p' dosya özelliğiyle görüntülenirler. Tabii boru dosyaları gerçek anlamda diskte yer kaplamazlar. Onların yalnızca bir izin girişi vardır. Onlar işletim sistemi tarafından yine isimsiz borularda olduğu gibi ana bellekte oluşturulurlar. İsimli boru dosyaları dosya sisteminde silinmedikleri sürece kalıcıdır. Sistem kapatılıp yeniden açıldığında izin girişlerinde biz isimli boru dosyalarını 0 uzunlukta görürüz.

İsimli borular sırasıyla şu adımlarla oluşturulur:

1) UNIX/Linux sistemlerinde isimli borularla çalışmak için önce bir boru dosyasının oluşturulması gerekir. Boru dosyasını oluşturabilmek için mkfifo isimli POSIX fonksiyonu kullanılabilir. Ya da komut satırından mkfifo isimli komut zaten bu fonksiyonu çağırarak isimli boru dosyası oluşturmaktadır. mkfifo fonksiyonunun prototipi şöyledir:

```

#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);

```

Fonksiyonun birinci parametresi boru dosyasının yol ifadesini, ikinci parametresi boru dosyasının erişim haklarını belirtmektedir. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda -1 değerine geri döner. Eğer birinci parametreye belirtilen isimde izin girişi zaten varsa mkfifo fonksiyonu başarısız olmaktadır. Tabii isimli boru haberleşmesinde proseslerden yalnızca biri isimli boruyu yaratmaya çalışmalıdır. Ya da bunun yerine iki proses de isimli boru varmış gibi işlemlere başlayabilir. Yukarıda da belirttiğimiz gibi isimli boru dosyaları haberleşme başlamadan önce kullanıcı tarafından mkfifo komutuyla da yaratılabilir.

mkfifo komutu aşağıdaki gibi basit biçimde kullanılabilir:

```
mkfifo mypipe
```

Bu biçimdeki kullanımda boru dosyası rw-r--r- haklarıyla yaratılmaktadır. İsimli boruyu -m seçeneği ile istediğimiz erişim haklarını vererek de yaratabiliriz. Örneğin:

```

csd@csd-virtual-machine ~/Study/SysProg-2017 $ mkfifo -m 666 testpipe
csd@csd-virtual-machine ~/Study/SysProg-2017 $ ls -l testpipe
prw-rw-rw- 1 csd study 0 Ağu 19 11:13 testpipe _

```

mkfifo isimli kabuk komutu mkfifo POSIX fonksiyonu kullanılarak şöyle yazılabilir:

```

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int result, mflag, smode, eflag = 0;
    char *marg = "644";
    int i;
    mode_t modes[] = { S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH };
};
    mode_t dmode;

    if (argc == 1) {
        fprintf(stdout, "at least one file name must be specified!..\n");
        exit(EXIT_FAILURE);
    }

    if (umask(0) == -1)
        exit_sys("umask");

    opterr = 0;
    while ((result = getopt(argc, argv, "m:")) != -1) {
        switch (result) {
            case 'm':
                mflag = 1;
                marg = optarg;
                break;
            case '?':
                if (optopt == 'm')
                    fprintf(stderr, "-%c option given must be followed by a mode argument\n", optopt);
                else
                    fprintf(stderr, "invalid switch: %c\n", optopt);
                eflag = 1;
        }
    }

    if (eflag)
        exit(EXIT_FAILURE);

    sscanf(marg, "%o", &smode);

    dmode = 0;
    for (i = 8; i >= 0; --i)
        if ((smode >> i) & 1)
            dmode |= modes[8 - i];

    for (i = optind; i < argc; ++i)
        if (mkfifo(argv[i], dmode) == -1)
            fprintf(stderr, "cannot change mode: %s\n", argv[i]);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

2) Boru dosyası sanki normal bir dosyaymış gibi open POSIX fonksiyonuyla iki proses tarafından da açılır. (Ya da istenirse bunun için fopen standart C fonksiyonu da kullanılabilir. Ancak fopen kullanılacaksa dosya tamponlama stratejisinin "sıfır tamponlamalı moda" çekilmesi uygun olur.) open fonksiyonunda açış işlemini bir proses yalnızca okuma modunda diğeri de yalnızca yazma modunda yapmalıdır. Yani isimsiz borularda olduğu gibi boruya yazma ve borudan okuma potansiyeline sahip tek bir betimleyici olmalıdır:

```
if ((fd = open("mypipe", O_RDONLY)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
...
if ((fd = open("mypipe", O_WRONLY)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
...
```

Tabii aslında isimli boruyu proses O_RDWR modunda da açabilir. Bu yasak bir işlem olmamakla birlikte çoğu kez anlamsızdır.

Blokeli modda proseslerden biri boruyu okuma modunda açmak istediğinde, diğeri yazma modunda açana kadar open blokede kalmaktadır. Benzer biçimde bir proses boruyu yazma modunda açmak istediğinde diğeri okuma modunda açana kadar yine open blokede kalır.

3) Artık yazan taraf write fonksiyonuyla boruya yazma yapar. Okuyan taraf da read fonksiyonuyla borudan okur. Tabii yine boru dolduğunda yazan taraf, boru boşaldığında da okuyan taraf blokede kalır. Yani isimli borunun davranışı tamamen isimsiz boruda olduğu gibidir.

4) Yine önce yazan tarafın boruyu kapatması gerekir. Daha sonra karşı taraf read fonksiyonuyla sıfır byte okuduğunda o da boruyu kapatır.

Örnek bir program şöyle yazılabilir:

```
/* pipeproc1.c */

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

void exit_sys(const char *msg);

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

    if ((fd = open("testpipe", O_WRONLY)) < 0)
        exit_sys("open");

    for (i = 0; i < 100000; ++i)
        if (write(fd, &i, sizeof(int)) < 0)
            exit_sys("write");

    close(fd);

    return 0;
}

void exit_sys(const char *msg)
```

```

{
    perror(msg);

    exit(EXIT_FAILURE);
}

/* pipeproc.2 */

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    ssize_t result;
    int val;

    if ((fd = open("testpipe", O_RDONLY)) < 0)
        exit_sys("open");

    while ((result = read(fd, &val, sizeof(int))) > 0) {
        printf("%d ", val);
        fflush(stdout);
    }
    printf("\n");
    if (result < 0)
        exit_sys("read");

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

İsimli boru dosyası standart C fonksiyonlarıyla da açılıp, okuma yazma yapılabilir. Tabii yukarıda da belirtildiği gibi bu durumda tamponlama mekanizmasını kaldırmak uygun olur:

```

/* pipeproc1.c */

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

int main(void)
{
    FILE *f;
    int i;

    if ((f = fopen("mypipe", "w")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }
    setvbuf(f, NULL, 0, _IONBF);

```

```

for (i = 0; i < 100000; ++i)
    if (fwrite(&i, sizeof(int), 1, f) != 1)
        break;

if (ferror(f)) {
    fprintf(stderr, "cannot write pipe!\n");
    exit(EXIT_FAILURE);
}

fclose(f);

return 0;
}

/* pipeproc2.c */

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

void pipe_read_proc(FILE *f);

int main(void)
{
    FILE *f;
    int val;

    if ((f = fopen("mypipe", "r")) == NULL) {
        fprintf(stderr, "cannot open file!..\n");
        exit(EXIT_FAILURE);
    }
    setvbuf(f, NULL, 0, _IONBF);

    while (fread(&val, sizeof(int), 1, f) > 0)
        printf("%d ", val), fflush(stdout);

    if (ferror(f)) {
        perror("cannot read pipe!\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

Mademki isimli borular bir dosya gibi dizin girişlerine sahiptir. O halde biz isimli borulara yönlendirme yapabiliriz. Örneğin:

```

csd@csd-virtual-machine ~/Study/SysProg-2017/namedpipe $ ls -l > testpipe

csd@csd-virtual-machine ~/Study/SysProg-2017/namedpipe $ cat < testpipe
toplam 32
-rwxr-xr-x 1 csd study 8904 Ağu 19 11:45 process1
-rw-r--r-- 1 csd study 488 Ağu 19 11:42 process1.c
-rwxr-xr-x 1 csd study 9104 Ağu 19 11:45 process2
-rw-r--r-- 1 csd study 570 Ağu 19 11:45 process2.c
prw-r--r-- 1 csd study 0 Ağu 19 11:45 testpipe

```

Örneğin:

```

csd@csd-vm:~/Study/SysProg-2019$ cat testpipe
█

csd@csd-vm:~/Study/SysProg-2019$ cat > testpipe
ali
veli
selami

```

```
csd@csd-vm:~/Study/SysProg-2019$ cat testpipe
ali
veli
selami
```

UNIX/Linux Sistemlerinde Blokesiz Modda Boru Haberleşmeleri

Biz yukarıdaki örneklerde boru haberleşmelerini default blokeli modda yaptık. Bunun yanı sıra boru haberleşmeleri blokesiz modda da yapılabilmektedir. Blokesiz modda boru işlemleri için açış modunda O_NONBLOCK bayrağı kullanılmaktadır. Eğer boru isimli ise bu bayrak daha sonra fcntl fonksiyonuyla eklenebilir. Eğer boru isimli ise bu bayrak doğrudan open fonksiyonunun açış modu parametresine eklenir.

Blokesiz modun blokeli moddan farklılıkları şunlardır:

- Blokesiz modda read fonksiyonu boruda hiç bilgi yoksa, write fonksiyonu da boruda hiç boş yer yoksa blokeye yol açmaz. Bu durumda bu fonksiyonlar -1 ile geri döner ve errno değişkeni EAGAIN değeriyle set edilir. Böylece programcı blokede beklemek yerine o sırada başka işler yapmayı tercih edebilir.
- İsimli borularda blokesiz modda open fonksiyonu karşı taraf boruyu açana kadar bloke oluşturmaz. Open boruyu karşı tarafa bakılmaksızın başarılı biçimde açar.

Kursumuzda blokesiz modda boru örnekleri yapılmayacaktır. Bu konu "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır.

Windows Sistemlerinde İsimli Boru Haberleşmesi

Windows'ta isimli boru haberleşmeleri de UNIX/Linux sistemlerindeki gibi çok benzer yapılmaktadır. İsimli borular Windows'ta CreatePipe API fonksiyonuyla yaratılır:

```
BOOL CreatePipe(
    PHANDLE          hReadPipe,
    PHANDLE          hWritePipe,
    LPSECURITY_ATTRIBUTES lpPipeAttributes,
    DWORD            nSize
);
```

Fonksiyonun birinci ve ikinci parametreleri borudan okumak ve boruya yazmak için gereken handle nesnelere adreslerini alır. Fonksiyon buraya okuma ve yazma için gereken handle değerlerini yerleştirir. Üçüncü parametre kernel nesnelere güvenlik parametresidir. Bu parametre NULL geçilebilir. Son parametre ise yaratılacak borunun byte cinsinden uzunluğudur. Bu parametre 0 girilirse boru sistem tarafından belirlenen default uzunlukta yaratılır. Fonksiyon başarı durumunda sıfır dışı bir değere, başarısızlık durumunda sıfır değerine geri döner.

Boru yaratıldıktan sonra handle değerlerinin alt prosese geçirilebilmesi için CreateProcess fonksiyonunun beşinci parametresi TRUE geçirilmelidir. Bu konu "Windows Sistem Programlama" kursunda ele alınmaktadır. Alt prosese gerekmiyorsa diğer handle'ın (örneğin üst proses yazma yapıyorsa bu yazma handle'ının) geçirilmesine gerek yoktur. CreateProcess'teki beşinci parametre "ana şalter" görevindedir. Ancak bu parametre TRUE yapılsa bile default olarak kernel nesnelere handle değerleri alt prosese aktarılmamaktadır. Fakat biz istersek alt prosese belirli handle'ların aktarılmasını sağlayabiliriz. Bu aktarımın sağlanması yaratıcı fonksiyonların SECURITY_ATTRIBUTES parametresiyle yapılabileceği gibi SetHandleInformation API fonksiyonuyla daha sonra da yapılabilir. Yani Windows sistemlerinde bizim alt prosese bir handle değerini aktarabilmemiz için CreateProcess API fonksiyonunun beşinci parametresini TRUE yapmamız ve aynı zamanda da ilgili handle'ı aktarılabilir (inheritable) hale getirmemiz gerekir. Yukarıda da belirttiğimiz gibi bir handle'ın aktarılabilir hale getirilmesi handle sistemini yaratan fonksiyonların SECURITY_ATTRIBUTES parametresiyle yapılabileceği gibi daha sonra SetHandleInformation API fonksiyonuyla da yapılabilmektedir. SetHandleInformation API fonksiyonunun prototipi şöyledir:

```
BOOL SetHandleInformation(
    HANDLE hObject,
```

```
    DWORD dwMask,  
    DWORD dwFlags  
);
```

Fonksiyonun birinci parametresi aktarılabılır yapılacak kernek nesnesinin handle değerini alır. İkinci parametresi handle'a ilişkin değiştirilecek özelliği belirtmektedir. Bu parametre HANDLE_FLAG_INHERIT biçiminde girilmelidir. Üçüncü parametre eğer handle aktarılabılır olmaktan çıkartılacaksa 0 biçiminde aktarılabılır hale getirilecekse yine HANDLE_FLAG_INHERIT biçiminde girilmelidir. Fonksiyon başarısızlık durumunda 0 değerine, başarı durumunda 0 dışı bir değere geri döner.

Windows sistemlerinde isimsiz boru yaratıldıktan sonra yine CreateProcess fonksiyonuyla alt proses oluşturulur. Sonra yine UNIX/Linux sistemlerinde olduğu gibi okuyan taraf yazma borusunu, yazan taraf da okuma borusunu kapatır. Ondan sonra ReadFile ve WriteFile dosya fonksiyonlarıyla okuma yazma işlemleri gerçekleştirilir. Yine önce boruya yazma yapan tarafın boruyu kapatması gerekir. Okuyan taraf ReadFile fonksiyonu ile borudan sıfır byte okuduğu zaman borunun kapatılmış olduğunu anlar o da boruyu kapatır.

Windows'taki uygulamanın UNIX/Linux sistemlerinden küçük bir farkı daha vardır. Bilindiği gibi bu sistemlerde CreateProcess API fonksiyonu adeta UNIX/Linux sistemlerindeki fork/exec işlemine karşılık gelir. Bu durumda alt proses üst processten aktarılan boru handle değerini bilemez. O halde bu handle değerinin de alt processe aktarılması gerekir. Bu aktarım komut satırı argümanları yoluyla ya da çevre değişkenleri yoluyla yapılabilir.

Windows'ta da üst prosesin handle değerlerinin alt processe aktarılması benzer biçimdedir. Yukarıda belirtilen iki önemli nokta göz önüne alındıktan sonra aktarım yapılır. Aktarım sırasında üst prosesin Proses Handle Tablosundaki handle değerleri aynı numarada olacak biçimde alt processe geçirilmektedir.

Windows'ta borulardan okuma yapılırken eğer yazan taraf boruyu kapatmışsa ReadFile API fonksiyonu başarısızlıkla (0 değeri ile) geri dönmektedir. Bu durumda GetLastError değeri ERROR_BROKEN_PIPE olarak set edilir. Halbuki UNIX/Linux sistemlerinde yazan taraf boruyu kapattığı zaman okuyan tarafın read fonksiyonun 0 ile geri döndüğünü başarısızlık ile geri dönmediğini anımsayınız.

İsimsiz Boru Haberleşmesi yapan örnek bir Windows programı şöyle yazılabilir:

```
/* Process1.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <Windows.h>  
  
void ExitSys(LPCSTR lpszMsg);  
  
int main(void)  
{  
    HANDLE hReadPipe, hWritePipe;  
    char cmdLine[1024];  
    STARTUPINFO si = { sizeof(STARTUPINFO) };  
    PROCESS_INFORMATION pi;  
    DWORD dwWritten;  
    int i;  
  
    if (!CreatePipe(&hReadPipe, &hWritePipe, NULL, 4096))  
        ExitSys("CreatePipe");  
  
    if (!SetHandleInformation(hReadPipe, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT))  
        ExitSys("SetHandleInformation");  
  
    sprintf(cmdLine, "ChildProcess.exe %p", hReadPipe);  
  
    if (!CreateProcess(NULL, cmdLine, NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi))  
        ExitSys("CreateProcess");  
}
```

```

CloseHandle(hReadPipe);

for (i = 0; i < 100; ++i)
    if (!WriteFile(hWritePipe, &i, sizeof(int), &dwWritten, NULL))
        ExitSys("WriteFile");

CloseHandle(hWritePipe);

WaitForSingleObject(pi.hProcess, INFINITE);

return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

/* Process2.c */

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

void ExitSys(LPCSTR lpszMsg);

int main(int argc, char *argv[])
{
    HANDLE hReadPipe;
    DWORD dwRead;
    int val;
    BOOL result;

    sscanf(argv[1], "%p", hReadPipe);

    while (ReadFile(hReadPipe, &val, sizeof(int), &dwRead, NULL))
        printf("%d ", val), fflush(stdout);

    if (GetLastError() != ERROR_BROKEN_PIPE)
        ExitSys("ReadFile");

    printf("\n");

    CloseHandle(hReadPipe);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }
}

```



```

}
exit(EXIT_FAILURE);
}

```

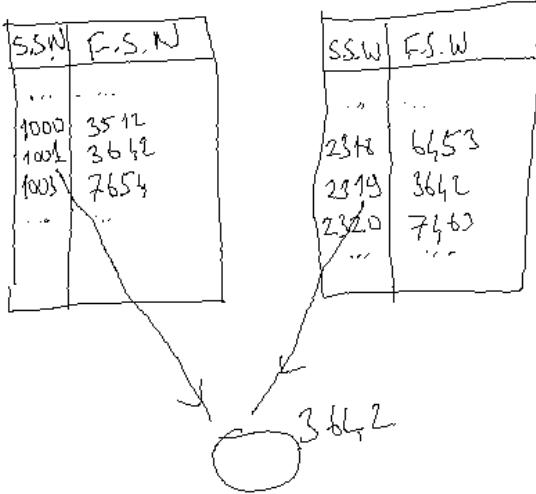
Yukarıdaki örnekte WaitForSingleObject isimli bir API fonksiyonu kullandık. Bu fonksiyon alt proses bitene kadar üst prosesi bloke yoluyla bekletmektedir. WaitForSingleObject fonksiyonu thread'ler konusunda ayrıntılarıyla ele alınmaktadır.

Windows Sistemlerinde İsimli Boru Haberleşmesi

Bu kursumuzda Windows sistemlerindeki isimli boru haberleşmeleri üzerinde durmayacağız. Çünkü Windows sistemlerinde isimli borular adeta client-server tarzda kullanılmaktadır. Kavramsal karmaşıklığı fazladır. Bu nedenle isimli borular Derneğimizde "Windows Sistem Programlama" kursunun konusu içerisinde değildir.

Paylaşılan Bellek Alanları (Shared Memory) Yöntemi İle Proseslerarası Haberleşme

Anımsanacağı gibi modern sistemlerde işlemcinin sağladığı bir sayfalama ve buna bağlı olarak bir sanal bellek mekanizması vardır. Bu mekanizma proseslerin bellek alanlarını tamamen birbirlerinden izole etmektedir. Paylaşılan bellek alanları yönteminde işletim sistemi iki ya da daha fazla prosesin farklı sanal sayfalarını aynı fiziksel sayfaya yönlendirir. Örneğin:



Burada soldaki prosesin 1001 numaralı sanal sayfası ile sağdaki prosesin 2319 numaralı sanal sayfası aynı fiziksel sayfaya yönlendirilmiştir. Yani birinci proses bu sanal sayfayı içeren bir adresi kullanarak yazma yaptığında diğer proses diğer sanal sayfayı içeren bir adresten bunları okuyabilir. Başka bir deyişle iki proses sanki farklı sanal adreslerle aynı fiziksel adrese erişiyor olmaktadır.

Paylaşılan bellek alanları yöntemi çok hızlı bir yöntemdir. Prosesler hiç kernel moda geçmeden haberleşebilirler. (Örneğin boru haberleşmesinde read ve write gibi POSIX fonksiyonları prosesi kernel moda geçirmektedir. Bu da zaman kaybına yol açmaktadır.) Ancak bu yöntem kendi içerisinde borularda olduğu gibi bir senkronizasyon içermez. Yani proseslerden biri paylaşılan bellek alanına yazdığında diğeri onun yazdığını ne zaman alacaktır? Diğeri yeni birşey yazarsa eskisi ezilmez mi? İşte senkronizasyon için üretici-tüketici problemi (producer-consumer problem) gibi bir yöntemin kullanılması gerekir.

Paylaşılan bellek alanları hem Windows hem de UNIX/Linux sistemlerinde var olan bir yöntemdir. Paylaşılan bellek alanları yöntemi aynı zamanda "bellek tabanlı dosyalar (memory mapped files)" konusunun da temelini oluşturmaktadır. Bellek tabanlı dosyalar konusu izleyen bölümlerde ele alınmaktadır.

Windows Sistemlerinde Paylaşılan Bellek Alanlarının Oluşturulması

Windows sistemlerinde paylaşılan bellek alanları şu adımlardan geçilerek oluşturulur:

1) Öncelikle her iki proste de bir "file mapping" nesnesinin yaratılması gerekir. Bu işlem CreateFileMapping fonksiyonuyla yapılır:

```
HANDLE WINAPI CreateFileMapping(  
    HANDLE hFile,  
    LPSECURITY_ATTRIBUTES lpAttributes,  
    DWORD flProtect,  
    DWORD dwMaximumSizeHigh,  
    DWORD dwMaximumSizeLow,  
    LPCTSTR lpName  
);
```

Fonksiyonun birinci parametresi eğer "bellek tabanlı dosya (memory mapped file)" kullanılacaksa o dosyanın handle değerini alır. Eğer yalnızca paylaşılan bellek alanı oluşturulacaksa bu parametreye INVALID_HANDLE_VALUE özel değeri geçilmelidir. Fonksiyonun ikinci parametresi file mapping nesnesinin güvenlik parametresidir. NULL geçilebilir. Üçüncü parametre oluşturulacak bellek alanının koruma özelliğini belirtir. Bu parametre PAGE_READWRITE girilirse bu bölgeye hem okuma hem de yazma yapılabilir. PAGE_READ yalnızca okuma için PAGE_WRITE yalnızca yazma için kullanılmaktadır. Fonksiyonun dördüncü ve beşinci parametreleri paylaşılacak bellek alanının uzunluğunu belirtir. Aslında bu iki parametre 8 byte'lık bir tamsayının düşük ve yüksek anlamlı 4'er byte'larıdır. Bu değerın sayfa katlarında olması anlamlıdır. Son parametre proseslerarası paylaşım için gereken isimdir. Bu isim herhangi bir biçimde verilebilir. İşletim sistemi birden fazla prosin aynı file mapping nesnesini ortaklaşa kullanacağını bu isme bakarak anlar. Dolayısıyla farklı prosesler aralarında haberleşmek için aynı ismi kullanmalıdır. Fonksiyon başarı durumunda mapping nesnesine ilişkin bir handle değeri ile, başarısızlık durumunda NULL değeri ile döner.

CreateFileMapping fonksiyonu haberleştirilecek iki proste de çağrılmalıdır. İlk çağırın proses nesneyi yaratmış olur. Artık diğer çağırındaki parametreler etkili olmaz. İkinci çağırın proses yalnızca nesneyi açmış olur. Dolayısıyla fonksiyonu sonradan çağırın kişi için üçüncü, dördüncü, beşinci parametrelerin bir önemi yoktur. Tabii iki proste de aynı değerler ve isimler kullanılırsa bu proseslerin hangisinin önce çalıştığının bir önemi kalmayacaktır. Eğer istenirse proseslerden biri CreateFileMapping fonksiyonuyla nesneyi yaratırken diğeri OpenFileMapping fonksiyonuyla onu açabilir. OpenFileMapping yalnızca olanı açma özelliğine sahiptir. Yaratma özelliğine sahip değildir. Başka bir deyişle CreateFileMapping fonksiyonu işlevsel olarak OpenFileMapping fonksiyonunu kapsamaktadır.

2) Bundan sonra MapViewOfFile isimli API fonksiyonuyla gerçek bellek adresi elde edilir. Yani işletim sistemi bu fonksiyonla paylaşılan fiziksel bellek alanını gören bir sanal sayfa numarasını sayfa tablosunda oluşturup bize oranın sanal adresini verecektir:

```
LPVOID WINAPI MapViewOfFile(  
    HANDLE hFileMappingObject,  
    DWORD dwDesiredAccess,  
    DWORD dwFileOffsetHigh,  
    DWORD dwFileOffsetLow,  
    SIZE_T dwNumberOfBytesToMap  
);
```

Fonksiyonun birinci parametresi mapping nesnesinden elde edilen handle değeridir. İkinci parametre sayfaya erişim özelliğini belirtir. Tabii buradaki parametre mapping nesnesi yaratılırken verilen parametreden daha geniş haklara sahip olamaz. Tipik olarak bu parametreye FILE_MAP_READ|FILE_MAP_WRITE değeri girilmektedir. Bu değer ilgili paylaşılan alana hem yazma yapılabileceğini hem de oradan okuma yapılabileceğini belirtir. İstersek bu parametreye yalnızca FILE_MAP_READ girebiliriz. Bu durumda paylaşılan bellek alanından yalnızca okuma yapabiliriz. Intel mimarisinde aslında yalnızca yazma yapma diye bir durum donanımsal olarak bulunmamaktadır. Ancak başka mimarilerde bu durum mümkün olabilmektedir. Fonksiyonun üçüncü parametreleri map edilecek kısmın offsetini alır. Tabii bu parametreler bellek tabanlı dosyalar için anlamlıdır. Eğer biz yalnızca paylaşılan bellek alanı oluşturuyorsak bu parametreye sıfır geçmeliyiz. Son parametre paylaşılacak bellek alanının büyüklüğüdür. Mapping nesnesinde belirtilen büyüklük maksimum büyüklüktür. Buradaki değer ondan küçük olabilir ya da ona eşit olabilir. Ancak ondan büyük olamaz. Fonksiyon başarı durumunda paylaşılan bellek alanına erişmekte kullanılan adrese, başarısızlık durumunda NULL adrese geri döner.

3) Kullanım bitince paylaşılan bellek alanının sisteme iade edilmesi gerekir. Bu `UnmapViewOfFile` fonksiyonuyla yapılır:

```
BOOL WINAPI UnmapViewOfFile(  
    LPCVOID lpBaseAddress  
);
```

Fonksiyon paylaşılan bellek alanının başlangıç adresini parametre olarak alır ve o alanı serbest bırakır.

4) Nihayet proses mapping nesnesini de `CloseHandle` fonksiyonuyla serbest bırakmalıdır. Tabii bittiğinde bu son iki aşama otomatik yapılır.

```
BOOL WINAPI CloseHandle(  
    HANDLE hObject  
);
```

Örneğin:

```
/* Process1.c */  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <Windows.h>  
  
void ExitSys(LPCSTR lpszMsg);  
  
int main(void)  
{  
    HANDLE hFileMapping;  
    char *memstr;  
  
    hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, 4096,  
    "ThisIsATest");  
    if (hFileMapping == NULL)  
        ExitSys("CreateFileMapping");  
  
    memstr = (char *)MapViewOfFile(hFileMapping, FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 4096);  
    if (memstr == NULL)  
        ExitSys("MapViewOfFile");  
  
    printf("Process-1: Press ENTER to write into shared memory\n");  
    getchar();  
    strcpy(memstr, "this is a test");  
    printf("Process1: Press ENTER to EXIT");  
    getchar();  
  
    UnmapViewOfFile(memstr);  
    CloseHandle(hFileMapping);  
  
    return 0;  
}  
  
void ExitSys(LPCSTR lpszMsg)  
{  
    DWORD dwLastError = GetLastError();  
    LPTSTR lpszErr;  
  
    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,  
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {  
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);  
        LocalFree(lpszErr);  
    }  
  
    exit(EXIT_FAILURE);  
}
```

```

}

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

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    HANDLE hFileMapping;
    char *memstr;

    hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0, 4096,
"ThisIsATest");
    if (hFileMapping == NULL)
        ExitSys("CreateFileMapping");

    memstr = (char *)MapViewOfFile(hFileMapping, FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 4096);
    if (memstr == NULL)
        ExitSys("MapViewOfFile");

    printf("Process2: Press ENTER to read from shared memory\n");
    getchar();
    puts(memstr);
    printf("Process2: Press ENTER to exit\n");
    getchar();

    UnmapViewOfFile(memstr);
    CloseHandle(hFileMapping);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Bellek Tabanlı Dosyalar

Bellek tabanlı dosyalar işletim sistemlerine 90'lı yıllarda girmeye başlamıştır. Bellek tabanlı dosya bir dosyanın belleğe taşınması ve orada işleme sokulması anlamına gelir. Dosya bellek tabanlı olarak açıldığında dosyanın ilgili kısmı belleğe çekilir ve artık o kısımla ilgili işlemler doğrudan bellek üzerinde yapılır. Pekiyi bellek tabanlı dosya ile bir dosyayı normal olarak açmanın ve onu Window sistemlerinde ReadFile, UNIX/Linux sistemlerinde read gibi bir fonksiyonla belleğe okumaktan farkı nedir? İşte dosya bellek tabanlı olarak açıldığında eğer yazma hakkı verilmişse aynı zamanda dosyaya yazma da yapılabilmektedir. Yazma işlemi aslında doğrudan belleğe yapılır. Ancak işletim sistemi yapılan değişiklikleri dosyaya kendisi yansıtmaktadır. Değişikliklerin dosyada görülmesi hemen gerçekleşebilmektedir. Yani başka bir proses aynı dosyayı açtığında artık o da bellekte yapılmış olan değişiklikleri o anda görür. Dosyanın tamamı yerine belli bir kısmı da bellek tabanlı olarak açılabilir. Özellikle dosyanın farklı yerlerinden çok fazla okumanın gerektiği durumlarda (örneğin bir dosya formatını analiz ederken) dosyanın bellek tabanlı açılması hem hız bakımından hem de kod bakımından daha etkindir.

Windows Sistemlerinde Bellek Tabanlı Dosyalar (Memory Mapped Files)

Windows'ta bellek tabanlı dosyalar sanki paylaşılan bellek alanları yaratılmış gibi açılmaktadır. Bunun için sırasıyla şu işlemler yapılmalıdır:

- 1) Dosya önce CreateFile API fonksiyonuyla açılır ve buradan bir handle değeri elde edilir.
- 2) Sonra paylaşılan bellek alanlarında olduğu gibi CreateFileMapping fonksiyonuyla bir "file mapping" nesnesi oluşturulur. Bu fonksiyonda bizden aynı zamanda ne kadar bir alan için "file mapping" işleminin yapılacağı sorulmaktadır. Fonksiyonun beşinci parametresi olan dwMaximumSizeLow için 0 geçilirse tüm dosya anlaşılır.
- 3) CreateFileMapping fonksiyonundan elde edilen handle kullanılarak MapViewOfFile fonksiyonu çağrılır ve buradan bir bellek adresi elde edilir. Bu fonksiyonda biz dosyanın hangi offsetleri arasının belleğe çekileceğini belirtiriz.
- 4) Dosya işlemleri bittiğinde UnmapViewOfFile fonksiyonu ile dosya için bellekte tahsis edilen alan serbest bırakılır. Sonra sırasıyla "file mapping" ve "file" nesnelere de CloseHandle fonksiyonuyla kapatılabilir.

Örneğin:

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

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)
{
    HANDLE hFile, hFileMapping;
    char *memstr;
    DWORD fileSize;
    int i;

    if ((hFile = CreateFile("test.txt", GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, 0, NULL)) ==
        INVALID_HANDLE_VALUE)
        ExitSys("Cannot open file", EXIT_FAILURE);

    hFileMapping = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
    if (hFileMapping == NULL)
        ExitSys("CreateFileMapping", EXIT_FAILURE);

    memstr = (char *)MapViewOfFile(hFileMapping, FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);
    if (memstr == NULL)
        ExitSys("MapViewOfFile", EXIT_FAILURE);

    fileSize = GetFileSize(hFile, NULL);

    for (i = 0; i < fileSize; ++i)
        putchar(memstr[i]);

    memcpy(memstr, "Ankara", 6);        /* Bu noktada dosyada değişiklik yapılmıştır */

    UnmapViewOfFile(memstr);
    CloseHandle(hFileMapping);
    CloseHandle(hFile);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
```

```

LPTSTR lpszErr;

if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
    fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
    LocalFree(lpszErr);
}

exit(status);
}

```

Örnekten de görüldüğü gibi önce dosya önce normal olarak CreateFile fonksiyonuyla açılmıştır. Sonra CreateFile fonksiyonundan elde edilen handle değeri kullanılarak CreateFileMapping fonksiyonuyla bir "file mapping" nesnesi oluşturulmuştur. CreateFileMapping fonksiyonunda uzunluk parametresinin 0 girildiğine dikkat ediniz. Bu durum dosyanın tamamının map edileceği anlamına gelmektedir. Programımızda daha sonra MapViewOfFile fonksiyonuyla bellek tabanlı dosya için bellek adresi elde edilmiştir. Yine burada da offset'in ve uzunluğun 0 olarak girildiğine dikkat ediniz. Bu özel durum tüm dosyanın belleğe çekileceği (map edileceği) anlamına gelmektedir. Programda daha sonra dosyanın içerisindeki tüm byte'lar tek tek ekrana yazdırılmıştır. Son olarak dosyada bir güncelleme denemesi yapılmıştır. Bu güncellemeyle dosyanın başına "Ankara" yazısı yazdırılmaktadır.

Peki bellek tabanlı dosyalara neden gereksinim duyulmaktadır? En önemli gerekçelerden biri işlem kolaylığıdır. Örneğin bir dosyadan çok sayıda okuma işlemi yapacağımızı düşünelim. Her okuma hem bir zaman kaybına yol açar hem de bunun yazımsal yükü vardır. (Örneğin okunacak yer için dosya göstericisinin konumlandırılması ve her okumanın başarısının kontrol edilmesi gerekir.) Halbuki bu tür uygulamalarda dosya bellek tabanlı olarak açılırsa artık okuma işlemi göstericilerle belleğe erişme işleminden farksız biçimde yapılır. Bu da özellikle dosya formatları üzerinde işlem yapan programların çok daha basit gerçekleştirilmesini sağlamaktadır. Örneğin bir bmp dosyasının üzerinde değişiklikler yapmak isteyelim. Dosyayı bellek tabanlı olarak açıp doğrudan bellek üzerinde değişiklikler yapabiliriz.

UNIX/Linux Sistemlerinde Paylaşılan Bellek Alanlarının Oluşturulması

UNIX/Linux sistemlerinde paylaşılan bellek alanlarını oluşturmak için iki grup fonksiyon vardır. Bunlardan biri en eskiden beri var olan "System 5 shared memory" fonksiyonları denilen gruptur. İkincisi ise 90'lı yıllarda UNIX türevi sistemlere sokulmuş olan modern fonksiyonlardır. Bunlara "POSIX shared memory" fonksiyonları denilmektedir. Aslında her iki fonksiyon grubu da POSIX standartlarında tanımlıdır. Ancak biz burada daha sonra çıkmış olan bu modern paylaşılan bellek alanı fonksiyonlarını göreceğiz. Klasik System 5 paylaşılan bellek alanı fonksiyonları "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır.

POSIX paylaşılan bellek alanı fonksiyonları standart C ve diğer POSIX fonksiyonlarının çoğunun bulunduğu "libc" kütüphanesinde değildir. Bu fonksiyonlar "librt" isimli kütüphaneye yerleştirilmiştir. Bu kütüphanenin de "libc" gibi statik (librt.a) ve dinamik (librt.so) versiyonları vardır. Dolayısıyla POSIX paylaşılan bellek alanı fonksiyonlarını kullanan bir programı derlerken link aşaması için "-lrt" seçeneğinin komut satırı argümanlarının sonunda bulundurulması gerekmektedir.

UNIX/Linux sistemlerinde paylaşılan bellek alanı oluşturma işlemi şu adımlardan geçilerek yapılır:

1) Paylaşılan bellek alanı nesnesi shm_open fonksiyonuyla yaratılır. shm_open hem nesneyi ilk kez yaratmak için hem de yaratılmış olanı açmak için kullanılır. shm_open fonksiyonu işlev olarak Windows'taki CreateFileMapping fonksiyonuna benzetebiliriz:

```

#include <sys/mman.h>
#include <sys/stat.h>          /* For mode constants */
#include <fcntl.h>            /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);

```

Fonksiyonun birinci parametresi paylaşılan bellek alanının proseslerarası kullanımı için gereken isimdir. Bu ismin kök dizinde sanki bir dosya ismiymiş gibi verilmesi gerekmektedir. (Tabii aslında böyle bir dosya yoktur. Dizin girişinde de

görülmecektir.) İkinci parametre açış modunu belirtir. Açış modu open fonksiyonundakine benzerdir. O_CREAT "yoksa yarat, varsa olanı aç" anlamına gelir. Bununla birlikte O_RDONLY, ya da O_RDWR bayrakları OR'lanmalıdır. Ancak O_WRONLY seçeneği açış modunda kullanılamaz. O_EXCL "nesne varsa onu açma, fonksiyon başarısızlıkla sonlansın" anlamına gelmektedir. Bu bayrağı da açış moduna ekleyebiliriz. Üçüncü parametre nesnenin erişim haklarını belirtir. Bu üçüncü parametre open fonksiyonundaki gibi nesne yaratılırken kullanılmaktadır. Eğer zaten yaratılmış olan paylaşılan bellek alanı nesnesi açılacaksa bu argüman dikkate alınmaz. Fonksiyon başarı durumunda paylaşılan bellek alanını temsil eden betimleyiciye, başarısızlık durumunda -1 değerine geri dönmektedir. Geri dönüş değerine ilişkin bu betimleyici diğer fonksiyonlara argüman olarak geçirilmektedir.

2) Paylaşılan bellek alanına ilişkin dosya için (aslında burada gerçek anlamda bir dosya söz konusu değildir) ftruncate fonksiyonuyla bir alanın ayrılması gerekir. ftruncate fonksiyonu aslında bir dosyayı küçültmek ya da büyütme için kullanılan bir POSIX fonksiyonudur. Örneğin bizim elimizde 1000 byte'lık bir dosya olsun. Biz ftruncate fonksiyonuyla bu dosyayı 500 byte'a düşürebiliriz. Bu durumda sonraki 500 byte dosyadan atılacaktır. Benzer biçimde biz 1000 byte'lık dosyasımızı 1500 byte'a uzatabiliriz. Bu durumda yeni bir 500 byte'lık içi 0 olan bir alan dosyanın eklenecektir. İşte ftruncate fonksiyonu aynı zamanda paylaşılan bellek alanının uzunluğunu belirlemek için de kullanılmaktadır. Siz shm_open ile yeni yaratılan paylaşılan bellek alanı dosyasının başlangıçta 0 uzunlukta olduğunu düşünebilirsiniz. İşte ftruncate fonksiyonu ile ona istenilen uzunluk verilmektedir.

ftruncate fonksiyonunun prototipi şöyledir:

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

```
int ftruncate(int fd, off_t length);
```

Fonksiyonun birinci parametresi paylaşılan bellek alanı dosyasının betimleyicisini, ikinci parametresi oluşturulacak paylaşılan alanın uzunluğunu almaktadır. Paylaşılan bellek alanına ftruncate uygulanabilmesi için açış modunun O_RDWR olması gerekmektedir. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda -1 değerine geri döner. Bu fonksiyonun her iki proste de aynı uzunluk değeriyle uygulanması bir soruna yol açmaz. Paylaşılan bellek alanlarının sayfa katlarında (4K) bir uzunluğa sahip olması en normal olan durumdur.

3) Daha sonra paylaşılan bellek alanı nesnesi için mmap fonksiyonuyla bir sanal adres elde edilir. Bu fonksiyonu Windows'taki MapViewOfFile fonksiyonuna benzetebiliriz. İşletim sistemi bu sanal adrese ilişkin sayfayı ortak bir fiziksel sayfaya yönlendirecektir:

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

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Fonksiyonun birinci parametresi mapping için önerilen adresi belirtir. Programcı buraya bir adres değeri girerek işletim sisteminin paylaşılan bellek alanını o sanal adreste oluşturmasını isteyebilir. Ancak buraya girdiği adres uygunsuzsa ya da zaten kullanılıyor durumdaysa fonksiyon başarısız olacaktır. Bu parametre NULL geçilirse sistem kendisinin uygun gördüğü bir sanal adreste paylaşılan bellek alanını oluşturacaktır. İkinci parametre paylaşılan bellek alanı için ayrılacak sanal belleğin uzunluğunu belirtmektedir. Yani biz ftruncate fonksiyonu ile geniş bir mapping nesnesi oluşturup onun bir kısmını belleğe çekebiliriz. Üçüncü parametre ise erişim bilgilerini belirtir. Bu parametre aşağıdaki bir ya da birden fazla bayrağın bit düzeyinde OR'lanmasıyla oluşturulabilir:

PROT_EXEC	Paylaşılan alana yerleştirilen kod çalıştırılabilir.
PROT_READ	Paylaşılan alan okunabilir
PROT_WRITE	Paylaşılan alana yazma yapılabilir
PROT_NONE	Paylaşılan alana bir şey yapılamaz

Dördüncü parametre ya MAP_SHARED ya da MAP_PRIVATE geçilmelidir. Paylaşılan bellek alanı için MAP_SHARED kullanılmalıdır. MAP_PRIVATE başka uygulamalarda "copy on write" özelliği sağlamak için tercih edilebilmektedir. Beşinci parametre shm_open fonksiyonundan elde edilen betimleyicidir. Son parametrede ayrılan alanın neresinden itibaren map edileceğini belirtir. Buradaki offset tipik olarak 0 geçilmektedir. Fonksiyon başarı durumunda paylaşılan bellek alanının sanal adresiyle, başarısızlık durumunda MAP_FAILED değeriyle geri döner.

4) Paylaşılan bellek alanı munmap fonksiyonuyla geri bırakılır:

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

int munmap(void *addr, size_t length);
```

Fonksiyonun birinci parametresi paylaşılan alanın sanal bellek adresini ikinci parametresi ise ne kadar alanın serbest bırakılacağını belirtir. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda -1 değerine geri döner.

5) Paylaşılan bellek alanı nesnesi close fonksiyonuyla yok edilir.

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

6) shm_open ile O_CREAT bayrağı verilerek yaratılmış olan POSIX paylaşılan bellek alanı nesnesi sistem kapatılana kadar (reboot edilene kadar) kalmaktadır. Ancak sistem kapatıldığında bu nesne otomatik biçimde yok edilir. Yani bu nesne bir dosya gibi sistem kapatıldıktan sonra kalıcı değildir. İşte biz sistem kapatılmadan önce de bu nesnenin yok edilmesini isteyebiliriz. Bunun için shm_unlink fonksiyonu kullanılmaktadır:

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

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

Fonksiyon parametre olarak paylaşılan bellek alanı nesnesine ilişkin yol ifadesini alır. Başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönmektedir.

Örnek bir paylaşılan bellek alanı uygulaması şöyle yapılabilir:

```
/* process1.c */

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    char *addr;
    char buf[4096 - 1];

    if ((fd = shm_open("/ThisIsATeest", O_CREAT|O_RDWR,
        S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) < 0)
        exit_sys("shm_open");

    if (ftruncate(fd, 4096) < 0)
        exit_sys("ftruncate");

    if ((addr = (char *)mmap(NULL, 4096, PROT_WRITE|PROT_READ, MAP_SHARED, fd, 0)) == MAP_FAILED)
        exit_sys("mmap");

    printf("Enter text:");
    gets(buf);
    strcpy(addr, buf);
    printf("Press ENTER to exit!\n");
    getchar();
}
```



```

    munmap(addr, 4096);
    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

/* process2.c */

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    char *addr;

    if ((fd = shm_open("/ThisIsATeest", O_CREAT | O_RDWR,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) < 0)
        exit_sys("shm_open");

    if (ftruncate(fd, 4096) < 0)
        exit_sys("ftruncate");

    if ((addr = (char *)mmap(NULL, 4096, PROT_READ, MAP_SHARED, fd, 0)) == MAP_FAILED)
        exit_sys("ftruncate");

    printf("Press ENTER to read from shared memory!\n");
    getchar();
    puts(addr);
    printf("Press ENTER to exit!\n");
    getchar();

    munmap(addr, 4096);
    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Anahtar Notlar: Özellikle gömülü ortamlarda bazı fiziksel bellek bölgeleri çeşitli IO uçlarına yönlendirilmiş olabilmektedir. Yani biz adeta bellekte o yere veri aktardığımızda aslında o IO uçlarına elektriksel işaret (örneğin 5V) göndermiş oluruz. Benzer biçimde belleğin o bölgesini okuduğumuzda o uçlardaki gerilim değerini okumuş olabiliriz. Bu tekniğe "bellek tabanlı IO tekniği (memory mapped IO)" denilmektedir. Örneğin Raspberry PI bu tekniği kullanmaktadır. Raspberry PI'da fiziksel belleğin 512 megabyte'ından itibaren IO bölgesi başlar.

UNIX/Linux Sistemlerinde Bellek Tabanlı Dosyalar

Bellek tabanlı dosyalar belli bir tarihten sonra POSIX sistemlerine de sokulmuştur. Bu sistemlerde de Windows sistemlerinde olduğu gibi bellek tabanlı dosyalar paylaşılan bellek alanları yaratılmış gibi oluşturulmaktadır. Bellek tabanlı dosya işlemleri için sırasıyla şunlar yapılır:

1) Dosya open fonksiyonuyla normal olarak açılır. Örneğin:

```
int fd;

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

2) Açılan dosya kullanılarak mmap fonksiyonu çağrılır. Yani mmap fonksiyonunda biz shm_open fonksiyonundan elde edilen betimleyici yerine open fonksiyonundan elde edilen betimleyiciyi kullanırsak bellek tabanlı dosya oluşturmuş oluruz. Örneğin:

```
if ((ptr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) == NULL)
    exit_sys("mmap");
```

3) UNIX/Linux sistemlerinde bellek tabanlı dosyalara yazma yapıldıktan sonra gerçek dosyanın bundan etkilenmesi Windows sistemlerindeki gibi garanti altına alınmamıştır. Bellekte yapılan güncellenmelerin gerçek dosyaya aktarılması için bu sistemlerde msync çağırması gerekmektedir. Eğer bu çağrı yapılmazsa bellekteki değişikliklerin gerçek dosyaya aktarılması belli bir zaman sonra yapılabileceği gibi, hiç yapılmayabilir de. msync fonksiyonun prototipi şöyledir:

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

int msync(void *addr, size_t length, int flags);
```

Fonksiyonun birinci parametresi senkronize edilecek alanın başlangıç adresini, ikinci parametresi ise senkronize edilecek alanın uzunluğunu belirtir. Üçüncü parametre şunlardan biri olabilir:

MS_ASYNC: Bu durumda senkronizasyon için başlatma işlemi yapılır. Ancak fonksiyon hemen geri döner. Senkronizasyon asenkron olarak arka planda yapılır.

MS_SYNC: Bu durumda senkronizasyon bitine kadar fonksiyon bloke bekletilir.

Bu bayraklardan biriyle birlikte istenirse MS_INVALIDATE bayrağı da bitsel OR işlemine sokulabilir. Bu durumda eğer dosyayı başka proses de bellek tabanlı olarak açmışsa onlarda da tazeleme yapılır.

msync fonksiyonu başarı durumunda 0 değeri ile, başarısızlık durumunda -1 değeriyle geri dönmektedir.

4) Yine bellek tabanlı dosyalar da munmap fonksiyonuyla tahsis edilen bellek alanı prosesin bellek alanından boşaltılır.

5) Nihayet open fonksiyonuyla açılmış olan gerçek dosya close ile kapatılır.

Örneğin:

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

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    int fd;
    char *ptr;
```

```

off_t length;
int i;

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

length = lseek(fd, 0, SEEK_END);

if ((ptr = (char *)mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) == NULL)
    exit_sys("mmap");

for (i = 0; i < 100; ++i)
    putchar(ptr[i]);

for (int i = 0; i < 10; ++i)
    ptr[i] = 'x';

if (msync(addr, len, MS_SYNC) == -1)
    exit_sys("mssync");

if (munmap(ptr, length) == -1)
    exit_sys("munmap");

close(fd);

return 0;
}

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

    exit(EXIT_FAILURE);
}

```

Mesaj Kuyukları (MessageQueue) Yöntemi

Bu proseslerarası haberleşme yönteminde ismine mesaj kuyruğu (message queue) denilen bir kuyruk sistemi yaratılır. Bir proses bu kuyruğa yazma yaparken diğeri okuma yapar. Kuyruk sistemi FIFO biçimdedir. Bu nedenle haberleşme biçim bakımından boru haberleşmesine benzemektedir. Ancak mesaj kuyuklarında paket tarzı bir aktarım vardır. Bir taraf mesaj adı altında bir paket bilgi gönderir. Diğeri bunu alır. Alan taraf için mesajın bir kısmını almak sonra kalanını almak gibi bir durum söz konusu değildir. Alan taraf gönderilen paketin tamamını tek hamlede almak zorundadır. Boru haberleşmesi "stream" tabanlı bir haberleşme sunduğu halde mesaj kuyukları paket tabanlı (datagram) bir haberleşme sunmaktadır. Yani boru haberleşmesi bu bakımdan IP ailesindeki "tcp" protokolüne benzetilirse, mesaj kuyukları "udp" protokolüne benzetilebilir.

Mesaj kuyukları Windows işletim sisteminde GUI alt sistemindeki mesajlar yoluyla kullanılabilir. Zaten Windows'un GUI alt sistemi tamamen mesaj tabanlı bir sistemdir. Mesaj kuyukları sistemi daha çok Windows'tan ziyade UNIX/Linux sistemlerinde daha popülerdir.

UNIX/Linux Sistemlerinde Mesaj Kuyuklarının Kullanımı

Mesaj kuyukları için de tıpkı paylaşılan bellek alanlarında olduğu gibi iki farklı fonksiyon grubu kullanılabilir. Bunlardan biri klasik Sistem 5 mesaj kuyuklarıdır. Diğeri de daha modern POSIX mesaj kuyuklarıdır. (Yukarıda da belirtildiği gibi her ne kadar isimleri böyle anılıyorsa da aslında her iki fonksiyon grubu da POSIX standartlarında tanımlanmışlardır.) POSIX mesaj kuyuklarının API tasarımı bakımından daha modern olduğu söylenebilir. Ancak klasik pek çok uygulama yüksek oranda taşınabilirlik için ya da eskiden yazılmış oldukları için klasik Sistem 5 mesaj kuyuklarını kullanmaktadır. Kursumuzda modern POSIX mesaj kuyuklarını göreceğiz. Klasik Sistem 5 mesaj kuyukları "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır. POSIX mesaj kuyuklarını kullanan programları derlerken de yine link aşaması için "-lrt" seçeneğinin kullanılması gerekmektedir.

POSIX mesaj kuyruklarının kullanımı şöyledir:

1) Mesaj kuyruğu her iki proses tarafından mq_open isimli POSIX fonksiyonuyla açılır. mq_open fonksiyonunun prototipi şöyledir:

```
#include <mqqueue.h>
```

```
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

Fonksiyonun birinci parametresi proseslerarası haberleşmede kullanılacak ortak yol ifadesini belirtir. Bu yol ifadesinin yine mshm_open fonksiyonunda olduğu gibi yine kök dizinde bir dosya ismi gibi oluşturulması gerekmektedir. İkinci parametre açış modudur. Bu parametre yine O_RDONLY, O_WRONLY ya da O_RDWR girilebilir. Açış modu dosya yaratılacaksa O_CREAT ve duruma göre O_EXCL bayraklarını içerebilir. O_CREAT yine "mesaj kuyruğu yoksa yarat, varsa olanı aç" anlamına gelir. Üçüncü parametre mesaj kuyruğu nesnesinin erişim haklarını belirtmektedir. Programcının son parametrede yaratılan mesaj kuyruğunun bazı özelliklerini belirtmesi gerekmektedir. mq_attr yapısı şöyledir:

```
struct mq_attr {
    long mq_flags;          /* Flags (ignored for mq_open()) */
    long mq_maxmsg;        /* Max. # of messages on queue */
    long mq_msgsize;       /* Max. message size (bytes) */
    long mq_curmsgs;       /* # of messages currently in queue (ignored for mq_open()) */
};
```

Yapının mq_flags elemanına bazı bayraklar girilebilir. Eğer herhangi bir bayrak girilmeyecekse bu eleman 0 olarak girilmelidir. Yapının mq_maxmsg elemanı mesaj kuruğundaki maksimum mesaj sayısını belirtir. mq_msgsize mesaj kuyruğunun byte cinsinden maksimum uzunluğunu belirtmektedir. mq_curmsgs elemanı o anda kuyrukta kaç mesajın olduğunu belirtir. Programcı daha sonra bu elemana bakarak kuyruktaki mesaj sayısını öğrenebilir. mq_open fonksiyonunda set edilen kuyruk özellikleri her mq_open fonksiyonu çağrıldığında yeniden değiştirilmez. Çünkü POSIX mesaj kuyrukları eğer mq_unlink fonksiyonuyla silinmemişlerse sistem boot edilene kadar yaşamaktadır. mq_open fonksiyonunda verilen bu kuyruk özellikleri istenirse daha sonra mq_gettattr fonksiyonuyla elde edilip mq_settattr fonksiyonuyla yeniden değiştirilebilmektedir.

mq_open fonksiyonu eğer mesaj kuyruğu nesnesi sıfırdan yaratılmayacak var olan açılacaksa iki parametrelili de kullanılabilir. Başka bir deyişle (open fonksiyonundaki gibi) fonksiyonun ikinci parametresinde O_CREAT bayrağı kullanılmamışsa fonksiyonun üçüncü ve dördüncü parametresini girmeye gerek yoktur. Yani aslında bu fonksiyonun orijinal prototipi şöyledir:

```
#include <mqqueue.h>
```

```
mqd_t mq_open(const char *name, int oflag, ...);
```

mq_open fonksiyonu başarı durumunda kuyruğa ilişkin bir handle değeri ile başarısızlık durumunda -1 değeriyle geri döner. Örneğin:

```
mqd_t msgid;
struct mq_attr mattr = { 0, MSG_MAX, MSG_SIZE, 0 };
...
if ((msgid = mq_open("/sample_message_queue", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR, &mattr)) == -1)
    exit_sys("mq_open");
```

2) Mesaj kuyruğuna mesajı yazmak için mq_send, kuyruktan mesajı almak için mq_receive fonksiyonları kullanılır. mq_send fonksiyonunun prototipi şöyledir:

```
#include <mqqueue.h>
```

```
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);
```

Fonksiyonun birinci parametresi mesaj kuyruğunun handle değeridir. İkinci parametre mesajı oluşturan bilgilerin bulunduğu yerin bellek adresidir. Üçüncü parametre mesajın byte cinsinden uzunluğunu belirtir. Son parametre mesajın öncelik derecesini belirtmektedir. Fonksiyon başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönmektedir. Fonksiyonun ikinci parametresinin const char * türünden olması size yanlış bir izlenim vermesin. Biz göndereceğimiz mesaja yalnızca bir yazı değil herhangi bir binary veriyi kodlayabiliriz. Siz bu parametreyi const void * olarak yorumlayabilirsiniz.

mq_receive fonksiyonun prototipi ise şöyledir:

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
```

Fonksiyonun birinci parametresi mesaj kuyruğunun handle değerini alır. İkinci parametre mesaj bilgilerinin yerleştirileceği tamponun adresidir. Üçüncü parametre bu tamponun uzunluğunu belirtmektedir. Mesaj okunurken bu üçüncü parametreyle girilen uzunluğun kuyruk yaratılırken belirtilen uzunluğa eşit ya da daha büyük olması gerekir. Fonksiyonun son parametresi ise mesaj önceliğinin yerleştirileceği unsigned int türden nesnenin adresini almaktadır. Bu parametre NULL geçilebilir. Fonksiyon başarı durumunda alınan mesajdaki byte sayısına, başarısızlık durumunda -1 değerine geri dönmektedir.

3) mq_open fonksiyonuyla açılan mesaj kuyruğu mq_close fonksiyonuyla kapatılır. Prosesler mesaj kuyruğunu mq_close ile kapatsalar bile mesaj kuyruğu nesnesi yaşamaya devam etmektedir. Yani mesaj kuyruğu nesnelere silinmemişlerse onları yeniden mq_open fonksiyonu ile açıp kullanabiliriz.

4) Tıpkı paylaşılan bellek alanı nesnelere olduğu gibi mesaj kuyrukları da sistem kapatılana kadar yaratılmış biçimde kalmaktadır. Sistem reboot edildiğinde bu nesnelere silinmiş durumda olur. Ancak biz reboot işleminden önce de mesaj kuyruklarını mq_unlink fonksiyonuyla yok edebiliriz. Fonksiyonun prototipi şöyledir:

```
#include <mqqueue.h>
```

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

Fonksiyon parametre olarak mesaj kuyruğunun yol ifadesini alır. Başarı durumunda 0 değerine, başarısızlık durumunda -1 değerine geri döner.

Örnek bir mesaj kuyruğu uygulaması şöyle olabilir:

```
/* msg-send.c */
```

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

```
#define MSG_MAX    10
#define MSG_SIZE   4096
```

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

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

```
{
    mqd_t msgid;
    struct mq_attr mattr = { 0, MSG_MAX, MSG_SIZE, 0 };
    char msg[MSG_SIZE];

    if ((msgid = mq_open("/sample_message_queue", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR, &mattr)) == -
1)
        exit_sys("mq_open");

    for (;;) {
```

```

    printf("Bir yazı giriniz:");
    gets(msg);
    if (mq_send(msgid, msg, strlen(msg) + 1, 1) == -1)
        exit_sys("mq_send");
    if (!strcmp(msg, "exit"))
        break;
}

close(msgid);

return 0;
}

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

    exit(EXIT_FAILURE);
}

/* msg-receive.c */

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

#define MSG_MAX    10
#define MSG_SIZE   4096

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    mqd_t msgid;
    struct mq_attr mattr = { 0, MSG_MAX, MSG_SIZE, 0 };
    char buf[MSG_SIZE];
    unsigned prio;
    ssize_t size;

    if ((msgid = mq_open("/sample_message_queue", O_RDONLY | O_CREAT, S_IRUSR | S_IWUSR, &mattr)) == -
1)
        exit_sys("mq_open");

    for (;;) {
        printf("waiting for message...\n");
        if ((size = mq_receive(msgid, buf, MSG_SIZE, &prio)) == -1)
            exit_sys("mq_receive");
        printf("%ld bytes received (priority = %d): %s\n", (long)size, prio, buf);
        if (!strcmp(buf, "exit"))
            break;
    }

    close(msgid);

    if (mq_unlink("/ThisIsMyTestMessageQueue") == -1)
        exit_sys("mq_unlink");

    return 0;
}

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

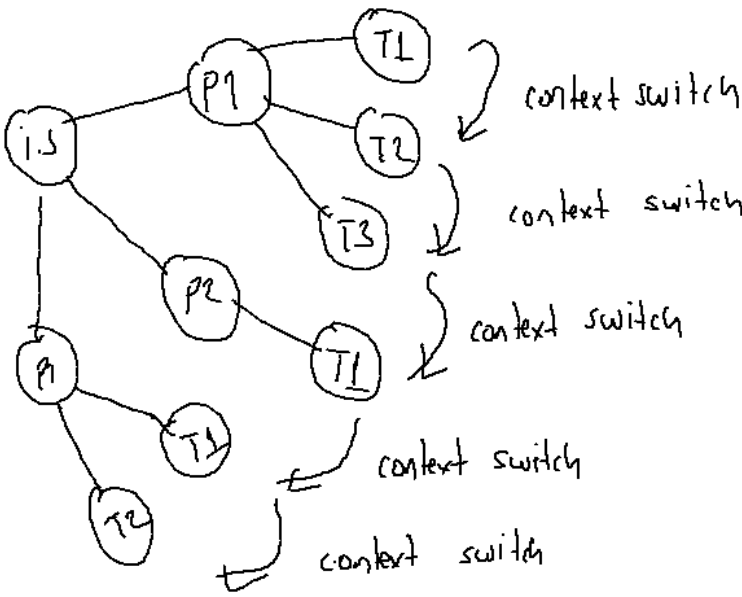
```

```
    exit(EXIT_FAILURE);  
}
```

Thread'lerle İşlemler

Thread sözcüğü etimolojik olarak "iplik" sözcüğünden gelmektedir. Akışlar ipliklere benzetilmesinden hareketle bu sözcük uydurulmuştur. Thread'ler bir prosesin bağımsız çizelgelenen akışlarını belirtir. Proses çalışmakta olan programın tamamını kavramsal olarak temsil etmektedir. Thread ise yalnızca bir akış belirtir. Dolayısıyla thread'ler proses kavramının içerisinde yer alır. Thread'lerin ilk ciddi denemeleri 80'li yıllarda yapılmıştır. Fakat 90'lı yıllarda işletim sistemlerine gerçek anlamda sokulmuştur. Örneğin DOS'ta thread yoktu. Windows 3.1 sistemleri de thread'li sistemler değildir. Microsoft'un ilk thread'li sistemi Windows NT (1993) ve sonra Windows 95 (1995)'tir. Linux'un ilk versiyonlarında thread'ler yoktu. 2.0'dan itibaren thread'li çalışma Linux sistemlerine sokulmuştur.

Çok thread'li işletim sistemlerinde çizelgelenen elemanlar artık prosesler değil thread'lerdir. Yani her quanta süresi dolduğunda bir thread'in çalışmasına ara verilir diğer bir thread çalıştırılır. Bu sistemlerde thread'ler bloke edilmektedir. Yani bir prosesin bir thread'i bloke olmuşken diğerleri çalışmaya devam edebilir.



Thread'li işletim sistemlerinde artık çizelgeleyici için prosesler değil, thread'ler çizelgenmektedir. Bu durumda thread'siz sistemler tek thread'li sistemler gibi düşünülebilir.

Çok thread'li işletim sistemlerinde proses çalışmaya bir thread'le başlar. Yani proses yaratıldığında bir thread de yaratılmış durumdadır. Buna prosesin ana thread'i (main thread) denir. Diğer thread'ler işletim sisteminin sistem fonksiyonlarıyla (yani Windows'ta API fonksiyonlarıyla, UNIX/Linux sistemlerinde POSIX fonksiyonlarıyla) yaratılırlar.

Anahtar Notlar: Birden fazla akışı çağrıştıran birkaç terim sıklıkla birbirlerine karıştırılabilmektedir. "Concurrency" terimi genel bir terimdir. Aynı anda birden fazla akışın bulunduğu durumları anlatmaktadır. Bu terim genelleşmiş olarak "concurrent computing" biçiminde de kullanılabilir. Buradaki birden fazla akışın iç içe geçmesi ve birlikteliği çeşitli biçimlerde sağlanabilmektedir. (Örneğin birden fazla proses oluşturarak, thread'ler yoluyla ya da kesme tekniği ile vs.) "Multi threading" terimi birden fazla thread ile iş yapmak anlamına gelir. Tabii "multi threading" aynı zamanda "concurrent computing" konusu içerisindedir. "Parallel Programming" işlerin thread'lere ayrılarak aynı makinede farklı CPU ya da çekirdeğe atanması yoluyla eş zamanlı yapıma gayreti için kullanılmaktadır. "Distributed Computing" ya da "Distributed Programming" terimi ise bir işin network içerisinde farklı makinelerle dağıtılarak birlikte yapılması anlamına gelmektedir.

Thread'lere Neden Gereksinim Duyulmaktadır?

Thread'lere neden gereksinim duyulmaktadır? Bu gereksinim birkaç maddeyle özetlenebilir:

1) Thread'ler arka plan olayları izlemek için iyi bir araç oluşturmaktadır. Örneğin hem bir işi yaparken hem de ekranın sağ üst köşesine saati basmak isteyelim. Saati ne zaman basacağız. Her işlemin arasında saati basmamız gerekir. Peki bu durumda klavye ya da disk işlemleri yapıldığında ne olacak? Ya da hem bir işi yaparken hem de arka planda dışsal bir olayı (örneğin seri portu, ya da bir termometreyi) izleyecek olalım. Eskiden bu tür işlemleri yapmak için tüm programın organizasyonunu değiştirmek gerekiyordu. Yani bu tür işlemler çok zor yapılabiliyordu. Halbuki thread'li sistemlerde bir thread yaratıp bu arka plan olayı bu thread' devredebiliriz. Böylece diğer thread'ler kendi işlemini yapabilir. Artık bu thread'ler bloke olsa bile arka plan olaylar izlenmeye devam edecektir.

2) Thread'ler bir programı hızlandırmak için kullanılabilir. Yani biz programımızı birden fazla thread ile organize edersek toplamda daha fazla CPU zamanı çekeriz.

3) Thread'ler blokeli IO işlemlerinde yoğun olarak kullanılmaktadır. Yani bir IO işlemi başlattığımızda (örneğin boru ya da soket gibi) belli bir süre bloke oluruz. Bu durumda gerekli olan başka şeyleri yapamayız. İşte IO işlemleri thread'lere yaptırılırsa blokeden yalnızca o thread etkilenir.

4) Thread'ler paralel programlama için mecburen kullanılmaktadır. Paralel programlama bir işi parçalara ayırarak onu aynı anda birden fazla işlemci ya da çekirdeğe atayarak gerçekleştirme sürecine denilmektedir.

5) Thread'ler GUI programlama modelinde bazen mecburen kullanılmak zorundadır. Örneğin bir mesaj geldiğinde bir işi uzatırsak kuyrukta sıradaki mesajları işleyemeyiz. İşte uzun sürebilecek işlemler thread'lere havale edilebilir.

Thread'lerin Yaratılması

Yukarıda da belirtildiği gibi proses çalışmasına tek bir thread'le başlar. Buna prosesin ana thread'i (main thread) denilmektedir. Daha sonra thread'ler işletim sisteminin sistem fonksiyonlarıyla ya da bunları çağıran kütüphane fonksiyonlarıyla yaratılır. Windows'ta thread işlemleri için API fonksiyonları UNIX/Linux sistemlerinde de POSIX fonksiyonları bulunmaktadır. POSIX'in thread fonksiyonlarına "pthread (posix thread)" fonksiyonları da denilmektedir. Bu fonksiyonların hepsinin ismi "pthread_xxx" biçimindedir. MAC OS X sistemlerinin bu anlamda POSIX uyumu olduğu için bu sistemlerde de thread'ler yine UNIX/Linux sistemlerinde olduğu gibi pthread kütüphanesi kullanılarak yaratılırlar. Java gibi, .NET gibi ortamlar ve diğer programlama dilleri kendi thread kütüphanelerine sahiptir. Tabii bu thread kütüphaneleri eninde sonunda işletim sistemlerinin yukarıda belirttiğimiz temel thread fonksiyonlarını çağırılmaktadır. Örneğin biz .NET'te ya da Java'da thread'leri bir sınıfın fonksiyonlarını kullanarak yaratıp kullanırız. Fakat bunlar arka planda Windows sistemlerinde API fonksiyonlarını, UNIX/Linux sistemlerinde de POSIX fonksiyonlarını çağırarak işlemlerini yaparlar.

Windows sistemlerinde thread'ler CreateThread API fonksiyonuyla yaratılmaktadır:

```
HANDLE WINAPI CreateThread(  
    LPSECURITY_ATTRIBUTES LpThreadAttributes,  
    SIZE_T dwStackSize,  
    LPTHREAD_START_ROUTINE LpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId  
);
```

Fonksiyonun birinci parametresi thread'in güvenlik bilgilerine ilişkindir. Bu parametre NULL geçilebilir. İkinci parametre thread'in stack uzunluğunu belirtmektedir. (Thread'lerin stack'leri sonraki başlıkta ele alınmaktadır.) Bu parametre 0 geçilirse bu durumda thread'in default stack uzunluğu kPE (Portable Executable Format) formatında belirtilen değerde alınır. Microsoft linker'ları bu değeri PE formatına default durumda (linker ayarlarıyla değiştirilebilir) 1 MB olarak yazmaktadır. Üçüncü parametre thread akışının başlatılacağı fonksiyonun adresini alır. LPTHREAD_START_ROUTINE türü aşağıdaki gibi typedef edilmiştir:

```
typedef DWORD (WINAPI *LPTHREAD_START_ROUTINE)(LPVOID lpThreadParameter);
```


Görüldüğü gibi thread fonksiyonunun geri dönüş değeri DWORD parametresi LPVOID (yani void *) türünden olmak zorundadır. Ayrıca thread fonksiyonları __stdcall çağırma biçimine sahip olmak zorundadır.

Yani örneğin aşağıdaki fonksiyon bir thread fonksiyonu olabilir:

```
DWORD __stdcall ThreadProc(void *param)
{
    ...
}
```

CreateThread fonksiyonunun dördüncü parametresi thread yaratıldığında thread'e geçirilecek argümanı belirtir. Beşinci parametre yaratıma ilişkin bazı özellikleri belirtmektedir. Bu parametre sıfır geçilebilir ya da CREATE_SUSPENDED biçiminde geçilebilir. Bu durumda thread yaratıldığında henüz çalışmıyor durumda olur. Onu çalıştırmak için ResumeThread API fonksiyonunun çağırılması gerekir. Fakat bu parametre sıfır geçilirse thread yaratılır yaratılmaz çalıştırılır. Fonksiyonun son parametresi thread id'sinin yerleştirileceği DWORD nesnenin adresini alır. Thread'lerin id'leri ve handle değerleri vardır. Fonksiyon başarı durumunda thread'in handle değerine, başarısızlık durumunda NULL değerine geri döner.

Windows sistemlerinde örnek bir thread yaratımı şöyle yapılabilir:

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc(LPVOID param);

int main(void)
{
    HANDLE hThread;
    DWORD dwThreadId;
    int i;

    if ((hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwThreadId)) == NULL)
        ExitSys("CreateThread");

    for (i = 0; i < 10; ++i) {
        printf("Main thread: %d\n", i);
        Sleep(1000);
    }

    return 0;
}

DWORD __stdcall ThreadProc(LPVOID param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("My thread: %d\n", i);
        Sleep(1000);
    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
```

```

    fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
    LocalFree(lpszErr);
}

exit(EXIT_FAILURE);
}

```

UNIX/Linux sistemlerinde thread'ler pthread_create isimli POSIX fonksiyonuyla yaratılmaktadır:

```

#include <pthread.h>

int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg
);

```

Fonksiyonun birinci parametresi thread'in id değerinin yerleştirileceği pthread_t türünden nesnenin adresini alır. UNIX/Linux sistemlerinde her thread'in proses genelinde tek olan (sistem genelinde tek değil) bir id değeri vardır. İkinci parametre thread özelliklerinin bulunduğu yapı nesnesinin adresini almaktadır. Bu parametre NULL geçilirse thread default özelliklerle yaratılır. Üçüncü parametre thread akışının başlatılacağı fonksiyonu belirtmektedir. Thread fonksiyonunun geri dönüş değeri ve parametresi void * türünden olmak zorundadır. Fonksiyonun son parametresi thread fonksiyonuna geçirilecek argümanı belirtir. Fonksiyon başarı durumunda sıfır değerine başarısızlık durumunda bizzat hata kodunun kendisine geri döner. POSIX'te thread fonksiyonları errno değerini set etmemektedir. Bu nedenle thread fonksiyonlarının bize verdiği hata kodlarını biz perror fonksiyonuyla doğrudan stderr dosyasına yazdırmayız. Biz thread fonksiyonlarının verdiği hata kodlarını alıp önce strerror fonksiyonu ile hata yazısına dönüştürüp o yazıyı stderr dosyasına yazdırmalıyız. Örneğin:

```

pthread_t tid;
int result;

if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0) {
    printf("pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

```

UNIX/Linux sistemlerinde thread yaratan örnek program şöyle olabilir:

```

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

void *thread_proc(void *param);

int main(void)
{
    pthread_t tid;
    int result;
    int i;

    if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0) {
        printf("pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i) {
        printf("Main thread: %d\n", i);
        sleep(1);
    }
}

```

```

    return 0;
}

void *thread_proc(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("New thread: %d\n", i);
        sleep(1);
    }

    return NULL;
}

```

UNIX/Linux sistemlerinde thread fonksiyonları libc kütüphanesinde değildir, libpthread isimli ayrı bir kütüphanededir. Linker default durumda yalnızca libc kütüphanesine baktığı için thread fonksiyonlarını bulamaz. Bu nedenle derlemeyi yaparken linker'ın pthread kütüphanesine bakmasını sağlamak gerekir. Bu işlem -lpthread seçeneğiyle yapılmaktadır. Yani thread kullanan bir program tipik olarak şöyle derlenerek link edilmelidir:

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

Yukarıda da belirtildiği gibi genel olarak Windows sistemlerinde ve UNIX/Linux sistemlerinde thread'ler arasında bir altlık-üstlük ilişkisi yoktur. Yani bir thread herhangi bir thread akışı tarafından yaratılabilir. Thread'in hangi thread akışı tarafından yaratıldığının bir önemi yoktur.

Windows sistemlerine CreateThread, UNIX/Linux sistemlerinde pthread_create fonksiyonu çağrıldıktan threda başarılı olarak yaratıldığında akış bu fonksiyonlardan çıkar. Artık söz konusu thread'ler bağımsız bir biçimde işletim sistemi tarafından çözelgenirler.

Thread'lerin Sonlanması

Bir thread çeşitli biçimlerde sonlanabilir. En normal sonlanma thread fonksiyonunun sonlanmasıyla gerçekleşen sonlanmadır. Thread fonksiyonu sonlandığında thread'in çalışması da biter. Thread'ler Windows sistemlerinde ExitThread API fonksiyonuyla UNIX/Linux sistemlerinde de pthread_exit fonksiyonuyla sonlandırılabilir. Bu fonksiyonlar kendi thread'lerini sonlandırmaktadır. Başka bir deyişle bu fonksiyonlar hangi thread akışı tarafından çağrılmışlarsa o thread akışını sonlandırmaktadır. Örneğin:

```

void Foo(void)
{
    for (int i = 0; i < 10; ++i) {
        printf("New Thread: %d\n", i);
        if (i == 5)
            ExitThread(0);
        Sleep(1000);
    }
}

DWORD __stdcall ThreadProc(void *param)
{
    Foo();

    return 0;
}

```

ExitThread fonksiyonun prototipi şöyledir:

```

VOID WINAPI ExitThread(
    DWORD dwExitCode

```

```
);
```

Fonksiyonun parametresi thread'in exit kodunu belirtir. Thread'lerin de tıpkı prosesler gibi exit kodları vardır. Thread fonksiyonunun geri dönüş değeri ve ExitThread fonksiyonuna girilen argüman bu exit kodunu belirtir. Windows sistemlerinde thread'lerin exit kodları DWORD türüyle, UNIX/Linux sistemlerinde de void * türüyle temsil edilmektedir. pthread_exit fonksiyonunun prototipi de şöyledir:

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Bir thread başka bir thread tarafından zorla da sonlandırılabilir. Bu işlem Windows'ta TerminateThread API fonksiyonuyla, UNIX/Linux sistemlerinde pthread_cancel fonksiyonuyla yapılır. TerminateThread fonksiyonu thread'i o anda ani olarak sonlandırır. Bir thread'i kritik bir işlemi yaparken ani olarak sonlandırmak yapılan iş dikkate alındığında sorunlara yol açabilir. Bu nedenle thread'lerin Windows sistemlerinde TerminateThread fonksiyonuyla sonlandırılması son çare olarak düşünülmelidir. Ancak POSIX sistemlerindeki pthread_cancel fonksiyonu sonlandırmayı hemen yapmaz. Sonlandırma thread akışı "sonlandırma noktası (cancellation point)" denilen bazı POSIX fonksiyonlarına girdiğinde o fonksiyonlarda yapılmaktadır. POSIX standartlarında sonlandırma noktası işlevi gören fonksiyonlar tek tek listelenmiştir. Ancak kabaca tüm dosya fonksiyonlarının ve sistem çağrısı yapan pek çok POSIX fonksiyonunun birer sonlandırma noktası belirttiğini söyleyebiliriz. TerminateThread API fonksiyonun prototipi şöyledir:

```
BOOL TerminateThread(
    HANDLE hThread,
    DWORD dwExitCode
);
```

Fonksiyonun birinci parametresi sonlandırılacak thread'in handle değerini ikinci parametresi de exit kodunu belirtir. pthread_cancel POSIX fonksiyonun da prototipi şöyledir:

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

Fonksiyonun parametresi sonlandırılacak thread'in id değerini alır. Geri dönüş değeri işlemin başarısı hakkında bilgi vermektedir.

Nihayet bir proses sonlandığında prosesin bütün thread'leri de sonlanır. Yani örneğin main fonksiyonu sonlandığında ya da exit fonksiyonu çağrıldığında o anda çalışmakta olan bütün thread'ler de sonlanacaktır. Örneğin sık yapılan bir hata şudur: Programcı bir grup thread yaratmıştır fakat ana thread'i bekletmemiştir. Böylece ana thread main'i bitirir. Proses biter, diğer thread'ler de sonlanmış olur.

Ayrıca hem Windows'ta hem de UNIX/Linux sistemlerinde ana thread'in diğer thread'lerden hiçbir farkı yoktur. Ana thread sonlandığı halde diğer thread'ler devam edebilir. Pekiyi prosesin ana thread'i sonlanmış olsun. Diğer thread'leri çalışıyor olsun. Bu durumda proses ne zaman sonlanacaktır? İşte çalışan thread'lerden hiçbiri exit fonksiyonu çağırılmamışsa işletim sistemi son thread de sonlandığında prosesi otomatik olarak sonlandırmaktadır. Örneğin:

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc(void *param);

int main(void)
{
    DWORD dwThreadId;
    HANDLE hThread;

    if ((hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwThreadId)) == NULL)
```

```

        ExitSys("CreateThread", EXIT_FAILURE);

ExitThread(0);    /* main thread sonlandıriliyor */

return 0;
}

DWORD __stdcall ThreadProc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("New Thread: %d\n", i);
        Sleep(1000);
    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Burada sorun şudur: Ana thread sonlanmış fakat exit fonksiyonu çağrılmamıştır (main fonksiyonu bitince exit fonksiyonunun çağrıldığını anımsayınız). Yaratılan thread de sonlanmıştır (Bir thread fonksiyonu sonlandığında arka planda exit çağrılmaz ExitThread ya da pthread_exit çağrılmaktadır). Bu durumda ortada hiçbir akış kalmadığı halde proses yaşayacak mıdır? İşte işletim sistemi eğer exit fonksiyonu çağrılmamışsa son thread sonlandığında prosesi otomatik olarak sonlandırır.

Thread'lerin Stack'leri

Thread'lerin stack'leri birbirlerinden ayrılmıştır. Yerel değişkenler ve parametre değişkenleri stack'te yaratıldığı için her thread bunların sanki farklı bir kopyasını kullanıyor gibidir. Örneğin iki farklı thread aşağıdaki aynı foo fonksiyonunda ilerliyor olsun:

```

void foo(void)
{
    int a = 10;
    ...
    ++a;
    ...
    ++a;
    ...
    ++a;
    ...
}

```

Burada a yerel bir değişkendir ve stack'te yaratılmaktadır. Fakat thread'lerin stack'leri ayrıdır. Bu nedenle her iki thread de a'yı kendi stack'lerinde farklı birer kopya olarak yaratırlar. Yani her thread a'nın aslında kendine özgü bir kopyasını kullanmaktadır. Eğer buradaki a global ya da static yerel olsaydı her thread aynı nesneyi artırıyor olurdu. Çünkü global ve static yerel nesnelere "data veya bss" bölümlerinde yaratılmaktadır. Bu bölümler prosese özgüdür. Tüm thread'ler aynı data ve bss bölümünü ortak kullanırlar. Benzer biçimde prosesin heap alanı da toplamda bir tanedir. Thread'lerin ayrı heap'leri yoktur. Heap alanı da tüm thread'ler tarafından ortak kullanılmaktadır. Buradan çıkartacağımız basit sonuç şöyle olabilir: Birden fazla thread aynı fonksiyon üzerinde ilerlerken o fonksiyonun yerel ve parametre değişkenlerinin

farklı kopyalarını kullanıyor durumdadır. Ancak global değişkenler ve static yerel değişkenlerin stack başına bir kopyası mevcut değildir. Bunların toplamda tek bir kopyası vardır. Bir thread bir global değişkende ya da static yerel değişkende değişiklik yaptığında diğer thread'ler onu hemen değişmiş olarak görürler.

Aşağıdaki örnekte iki thread (ana thread ve yeni yaratılan thread) aynı Foo fonksiyonunu çağırmıştır. Foo fonksiyonu şöyle tanımlanmıştır:

```
void Foo(const char *msg)
{
    /* static */ int i;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", msg, i);
        Sleep(1000);
    }
}
```

Burada i yerel değişkeni static olmadığında iki thread'in i'si farklı olur. Ancak i static yapıldığında iki thread aynı i'yi kullanıyor durumda olur:

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc(void *param);
void Foo(const char *msg);

int main(void)
{
    DWORD dwThreadId;
    HANDLE hThread;

    if ((hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwThreadId)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    Foo("Main thread");

    return 0;
}

DWORD __stdcall ThreadProc(void *param)
{
    Foo("New thread");

    return 0;
}

void Foo(const char *msg)
{
    /* static */ int i;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", msg, i);
        Sleep(1000);
    }
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
```

```

        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
    fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
    LocalFree(lpszErr);
}

exit(EXIT_FAILURE);
}

```

Thread'lerin Birbirleriyle Haberleşmesi

Thread'ler aynı prosesin parçaları olduğuna göre bunların birbirleriyle haberleşmesi global nesnelere ya da heap yoluyla yapılabilir. Global nesnelere Prosesin "data" veya "bss" denilen bölgelerinde yaratılmaktadır. Buraya da tüm thread'ler erişebilir. Heap de thread'ler arasında ortak kullanılan tekil bir alandır. Yani iki thread heap'te tahsis edilmiş nesnelere erişerek de haberleşme yapabilirler. Ancak yukarıda da belirtildiği gibi thread'lerin stack'leri birbirlerinden ayrılmıştır.

Thread'lerin haberleşmesi için özel bir yöntem (boru, paylaşılan bellek alanı gibi) gerek yoktur. Zaten "data", "bss" ve "heap" alanları ortak alanlardır. Yani bu alanlar zaten adeta paylaşılan bellek alanı gibidirler. C'de ilkdeğer verilmiş global değişkenler, static yerel değişkenler ve string ifadeleri "data" alanında, ilkdeğer verilmemiş global değişkenler ve static yerel değişkenler "bss" alanında, parametre değişkenleri ve yerel değişkenler de stack alanında yaratılmaktadır.

Thread'lerin Sonlanmasının Beklenmesi

Bazen bir thread bitene kadar diğer bir thread'in bloke edilerek bekletilmesi gerekebilir. İşte bunun için Windows sistemlerinde WaitForSingleObject (ya da WaitForMultipleObjects) isimli API fonksiyonu, UNIX/Linux sistemlerinde pthread_join isimli POSIX fonksiyonu kullanılmaktadır.

WaitForSingleObject aslında bekleme işlemini yapan genel bir fonksiyondur ve zaten ilerinde açıklanacaktır. Fonksiyonun prototipi şöyledir:

```

DWORD WINAPI WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMilliseconds
);

```

Fonksiyonun birinci parametresi thread'in handle değerini alır. İkinci parametre milisaniye cinsinden zaman aşımını belirtmektedir. Eğer ikinci parametrede bir zaman aşımı belirtilmişse en kötü olasılıkla eğer bitmesi beklenen thread bitmemişse bu zaman aşımı değeri dolduğunda bloke çözülür. Bu zaman aşımı değeri özel olarak INFINITE biçiminde de girilebilir. Bu durumda herhangi bir zaman aşımına bakılmaz. Yani fonksiyon ilgili thread sonlanana kadar kendisini çağıran thread'i blokede bekletir. Fonksiyonun geri dönüş değeri ileride daha detaylı olarak değerlendirilecektir. Ancak fonksiyon başarısızlık durumunda WAIT_FAILED özel değerine geri döner. Örneğin:

```

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc(void *param);

int main(void)
{
    DWORD dwThreadId;
    HANDLE hThread;

    if ((hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &dwThreadId)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    WaitForSingleObject(hThread, INFINITE);

    printf("Ok\n");
}

```

```

    return 0;
}

DWORD __stdcall ThreadProc(void *param)
{
    for (int i = 0; i < 10; ++i) {
        printf("New Thread: %d\n", i);
        Sleep(1000);
    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Windows sistemlerinde thread'in exit kodu GetExitCodeThread isimli API fonksiyonuyla elde edilebilmektedir. Fonksiyonun prototipi şöyledir:

```

BOOL GetExitCodeThread(
    HANDLE hThread,
    LPDWORD lpExitCode
);

```

Fonksiyonun birinci parametresi exit kodu alınacak thread'in handle değerini ikinci parametresi ise exit kodunun yerleştirileceği DWORD türünden nesnenin adresini alır. Fonksiyon başarısızlık durumunda 0 değerine başarı durumunda 0 dışı bir değere geri dönmektedir. Örneğin:

```

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc1(LPVOID lpvParam);
DWORD __stdcall ThreadProc2(LPVOID lpvParam);
void Foo(LPCSTR lpszName);

int main(void)
{
    HANDLE hThread1, hThread2;
    DWORD dwThreadID1, dwThreadID2;
    DWORD dwExitCode;

    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadID1)) == NULL)
        ExitSys("CreateThread");

    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadID2)) == NULL)
        ExitSys("CreateThread");

    WaitForSingleObject(hThread1, INFINITE);

    if (!GetExitCodeThread(hThread1, &dwExitCode))
        ExitSys("CreateThread");
}

```



```

printf("Thread-1 has been finished with exit code %lu\n", (unsigned long)dwExitCode);

WaitForSingleObject(hThread2, INFINITE);

if (!GetExitCodeThread(hThread2, &dwExitCode))
    ExitSys("CreateThread");

printf("Thread-2 has been finished with exit code %lu\n", (unsigned long)dwExitCode);

return 0;
}

DWORD __stdcall ThreadProc1(LPVOID lpvParam)
{
    Foo("Thread-1");

    return 100;
}

DWORD __stdcall ThreadProc2(LPVOID lpvParam)
{
    Foo("Thread-2");

    return 200;
}

void Foo(LPCSTR lpszName)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", lpszName, i);
        Sleep(1000);
    }
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastErr = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastErr,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

UNIX/Linux sistemlerindeki pthread_join POSIX fonksiyonun prototipi de şöyledir:

```

#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

```

Fonksiyonun birinci parametresi thread'in id değerini, ikinci parametresi thread'in exit kodunun yerleştirileceği void * türünden nesnenin adresini alır. Bu parametre NULL geçilebilir. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda hata kodunun kendisine geri döner. pthread_join fonksiyonunda bir zaman aşımı parametresi yoktur.

```

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

```

```

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

void *thread_proc1(void *param);
void *thread_proc2(void *param);
void foo(const char *name);
void exit_errno(const char *msg, int result);

int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    int result;
    void *tret;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
        exit_errno("pthread_create", result);

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
        exit_errno("pthread_create", result);

    if ((result = pthread_join(tid1, &tret)) != 0)
        exit_errno("pthread_join", result);

    printf("Thread-1 hasbeen finished with exit code %ld\n", (long)tret);

    if ((result = pthread_join(tid2, &tret)) != 0)
        exit_errno("pthread_join", result);

    printf("Thread-2 has been finished with exit code %ld\n", (long)tret);

    return 0;
}

void *thread_proc1(void *param)
{
    foo("Thread-1");

    return (void *)100;
}

void *thread_proc2(void *param)
{
    foo("Thread-2");

    return (void *)200;
}

void foo(const char *name)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", name, i);
        sleep(1);
    }
}

void exit_errno(const char *msg, int result)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(result));

    exit(EXIT_FAILURE);
}

```

Windows sistemlerindeki WaitForSingleObject, UNIX/Linux sistemlerindeki pthread_join fonksiyonun bekleme işlemini ilgili thread'i bloke ederek CPU zamanı harcamadna yaptığına dikkat ediniz. Bu işlem global bir değişken yoluyla meşgul bir döngüyle de yapılabilirdi. Ancak o zaman meşgul döngüdeki thread CPU zamanı harcardı. Örneğin:

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc(void *param);

int g_flag = 0;

int main(void)
{
    HANDLE hThread;
    DWORD threadId;
    int i;

    if ((hThread = CreateThread(NULL, 0, ThreadProc, NULL, 0, &threadId)) == NULL)
        ExitSys("CreateThread");

    while (g_flag == 0)
        ;

    printf("Ok\n");

    return 0;
}

DWORD __stdcall ThreadProc(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf("Other thread: %d\n", i);
        Sleep(1000);
    }

    g_flag = 1;

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}
```

Burada bekleme işleminin CPU zamanı harcanarak meşgul bir döngüde yapıldığını görüyorsunuz. Bu kötü bir tekniktir.

Thread Fonksiyonlarının Parametreleri

Thread akışının başlatılacağı fonksiyonun hem Windows sistemlerinde hem de UNIX/Linux sistemlerinde void * türünden bir parametreye sahip olduğunu görmüştük. Bu thread fonksiyonuna geçirilecek parametre thread'i yaratan

fonksiyonlarda belirtilmektedir. Windows'ta CreateThread API fonksiyonu, UNIX/Linux sistemlerinde pthread_create fonksiyonu bizden bu parametre için değeri alıp bu fonksiyona aktarmaktadır. Peki bu parametrenin amacı nedir? Birden fazla thread aynı fonksiyon kullanılarak yaratılıyor olabilir. Bu durumda bu thread'ler aynı işi yapacaklardır. Ancak yaptıkları bu iş onlara geçirilen parametre yoluyla farklı etkiler oluşturabilmektedir. Örneğin bir server program bir client ile bağlantı kurduğunda o client ile konuşmak için bir thread yaratabilir. Bu thread fonksiyonuna client'ın socket'ini aktarabilir. Böylece her client bağlantısında aslında aynı fonksiyon çalışacak fakat bu fonksiyon farklı client'larla konuşmuş olacaktır.

Thread fonksiyonlarına yerel nesnelerin adreslerini geçirmek hem Windows hem de UNIX/Linux sistemlerinde programın çökmesine yol açabilmektedir. Çünkü bu sistemlerde bir thread'in başka bir thread'in stack'ine erişmesi problemlidir. Bu durumda thread fonksiyonlarına biz global ya da heap'teki nesnelerin adreslerini geçirebiliriz. Tabii global nesnelere zaten tüm thread'ler erişebildiğine göre onların thread'lere parametre yoluyla aktarılmasının da bir anlamı yoktur. O halde tipik olarak thread'lere biz heap'te tahsis etmiş olduğumuz nesnelerin adreslerini geçirmeliyiz. Aşağıdaki örnekte 10 thread yaratılmıştır. Bu 10 thread'in de başlangıç fonksiyonu aynı yapılmıştır. Ancak bu thread fonksiyonlarına farklı değerler parametre olarak geçirilmiştir:

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc(void *param);

int g_flag = 0;

int main(void)
{
    HANDLE hThread[10];
    DWORD threadId[10];
    char *str;
    int i;

    for (i = 0; i < 10; ++i) {
        str = (char *)malloc(32);
        if (str == NULL) {
            fprintf(stderr, "cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
        sprintf(str, "Thread-%d", i);

        if ((hThread[i] = CreateThread(NULL, 0, ThreadProc, str, 0, &threadId[i])) == NULL)
            ExitSys("CreateThread");
    }

    for (i = 0; i < 10; ++i) /* WaitForMultipleObjects */
        WaitForSingleObject(hThread[i], INFINITE);

    printf("Ok\n");

    return 0;
}

DWORD __stdcall ThreadProc(void *param)
{
    int i;
    char *str = (char *)param;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", str, i);
        Sleep(1000);
    }

    free(str);
}
```

```

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Aynı örnek UNIX/Linux sistemlerinde de şöyle yazılabilir:

```

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

void exit_sys(const char *msg);
void *thread_proc(void *param);

int main(void)
{
    pthread_t tid[10];
    int i;
    char *str;
    int result;

    for (i = 0; i < 10; ++i) {
        str = (char *)malloc(32);
        if (str == NULL) {
            fprintf(stderr, "cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
        sprintf(str, "Thread-%d", i);

        if ((result = pthread_create(&tid[i], NULL, thread_proc, str)) != 0) {
            printf("pthread_create: %s\n", strerror(result));
            exit(EXIT_FAILURE);
        }
    }

    for (i = 0; i < 10; ++i)
        pthread_join(tid[i], NULL);

    printf("Ok\n");

    return 0;
}

void *thread_proc(void *param)
{
    int i;
    char *str = (char *)param;

    for (i = 0; i < 10; ++i) {
        printf("%s: %d\n", str, i);
    }
}

```

```

        sleep(1);
    }

    free(str);

    return NULL;
}

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

    exit(EXIT_FAILURE);
}

```

Thread'lerin Senkronizasyonu

Thread senkronizasyonu thread'ler konusunun en önemli ve en ayrıntılı bölümünü oluşturmaktadır. Thread'ler nadir olarak bağımsız iş yaparlar. Genellikle işbirliği içerisinde bir işi ortaklaşa yapacak biçimde organize edilirler. Thread'lerin ortak bir amacı gerçekleştirmek için çalışırken de kimi zaman birbirlerini beklemesi gerekebilmektedir. Genel olarak thread'lerin birbirlerini beklemesi ile ilgili konuya "thread senkronizasyonu" denilmektedir.

Windows, Linux, Mac OS X gibi preemptive sistemlerde bir thread'in quanta süresi dolduğunda (ya da başka birtakım gerekçelerle) o thread'in çalışmasına ansızın herhangi bir makine komutunda ara verilebilmektedir. İşte bu nedenle bir thread kritik bir işi henüz bitirmeden kesilebilir. Diğer bir thread yarım kalan iş ile ilgili birşeyler yaptığında tüm program birbirine girebilir, dolayısıyla çökebilir.

Kritik Kod (Critical Section) Kavramı

Başından sonuna kadar tek bir akış tarafından çalıştırılması gereken kod parçalarına kritik kod (critical section) denilmektedir. Pek çok durumda thread'ler ortak bir kaynak üzerinde bir arada çalışma yapıyor olabilir. Bu ortak kaynak bir veri yapısı olabileceği gibi dış dünyadaki donanımsal bir aygıt da olabilir. İşte bir thread ortak bir kaynak üzerinde ilerlerken o sırada thread'ler arası geçiş olursa o kaynak kararsız bir durumda kalır. Diğer bir thread kaynağı kullanmak istediğinde sorun çıkar. Örneğin bir thread'in ortak kullanılan bir bağlı listeye ekleme yaptığını diğer thread'in de burada arama yaptığını düşünelim. Ekleme yapan thread tam eklemenin ortasında kesilirse ve diğer thread arama yapmaya çalışırsa program çökebilir. Bu tür ortak kaynak kullanan thread'lerin işin tamamı bitene kadar birbirlerini beklemesi gerekmektedir. Yani örneğimizde ekleme yapan thread arada kesilse bile diğer thread ekleme işlemi bitene kadar onun bu işi bitirmesini beklemelidir. Tabii beklemenin meşgul bir döngü yoluyla değil bloke yoluyla yani CPU zamanı harcamadan yapılması arzu edilir.

Örneği aşağıdaki kodda iki thread aynı global değişkeni birer milyon kere artırmıştır. Acaba global değişken iki milyon değerine ulaşır mı?

```

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc1(void *param);
DWORD __stdcall ThreadProc2(void *param);

int g_i;

int main(void)
{
    DWORD dwThreadId1, dwThreadId2;
    HANDLE hThread1, hThread2;

    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);
}

```

```

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)
    ExitSys("CreateThread", EXIT_FAILURE);

WaitForSingleObject(hThread1, INFINITE);
WaitForSingleObject(hThread2, INFINITE);

printf("%d\n", g_i);

return 0;
}

DWORD __stdcall ThreadProc1(void *param)
{
    for (int i = 0; i < 1000000; ++i)
        ++g_i;

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    for (int i = 0; i < 1000000; ++i)
        ++g_i;

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Burada bizim işlemi tek bir ++ operatörüyle yapmış olmamız bunun atomik olacağı anlamına gelmez. Makine komutları atomiktir. Yani bir makine komutu çalıştırılırken zaten thread'ler arası geçiş oluşamaz. Ancak thread'ler iki makine komutu arasında thread'ler arası geçiş olabilir. İşte derleyiciler ++g_i ifadesini tek bir makine komutuyla yapmak zorunda değildiler. Örneğin bu işlemi aşağıdaki gibi üç makine komutuyla yapabilirler:

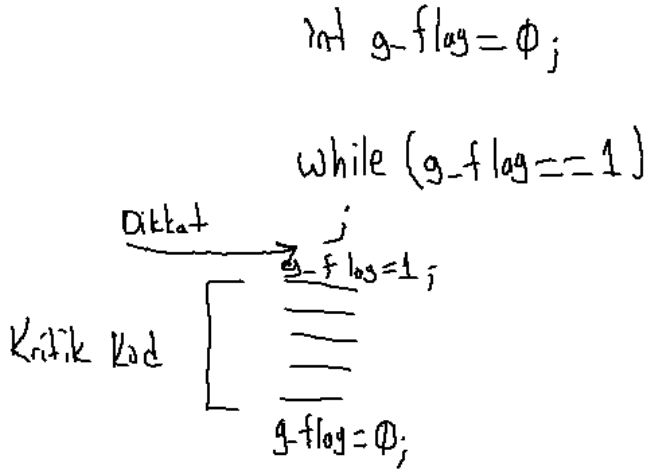
```

MOV    reg, g_i
INC    reg
MOV    g_i, reg

```

Gerçekten de yukarıdaki program çalışırsa muhtemelen iki milyon değeri görülmeyecektir. İşte bu örnekteki artırım işlemi kritik bir kodu temsil eder. Bu artırım başından sonuna kadar tek bir thread akışı tarafından yapılmalıdır.

Kritik kodlar manuel olarak oluşturulamazlar. Örneğin aşağıdaki gibi bir kodla kritik kod oluşturamayız:



Bu kodun iki sorunu vardır: Birincisi bekleyen thread meşgul bir döngüde (busy loop) bekleme yapar. İkincisi burada ok ile gösterilen noktada thread'ler arası geçiş olursa birden fazla thread kritik koda girebilir.

Peki bu işlem güvenli bir biçimde nasıl yapılabilir? Aslında yine bir bayrak tutulabilir. Ancak kontrol ve set işlemi yapılırken thread'ler arası geçiş kapatılabilir. Böylece bayrak güvenli olarak set edilebilir. Fakat thread'ler arası geçişin kapatılabilmesi için gereken makine komutlarını ancak kernel mod programlar kullanabilmektedir. Örneğin işletim sisteminin kernel modda çalışan sistem fonksiyonları bunu yapabilir. Eğer bayrak set edilmişse thread'i uykuya geçirebilir. Bu yöntem kullanılarak senkronizasyon yapan kodlara "kernel mod senkronizasyon nesnelere" denilmektedir. Son 20 yıldır makine dillerine karşılaştırma ve jump işlemini birlikte yapan özel komutlar eklenmiştir. Bu komutlar bayrakları güvenli set etmeyi sağlamaktadır. Böylece hiç olmazsa kontrol yapılırken kernel moda geçiş engellenebilmektedir. Fakat ne olursa olsun bu işi yapan fonksiyonların özel bir biçimde yazılması gerekir. İşte işletim sistemleri ya da temel kütüphaneler bunları yapan hazır fonksiyonlar bulundurmaktadır.

Windows'ta Kritik Kodların Oluşturulması

Windows'ta kritik kod oluşturmak için CriticalSection nesnelere, mutex ya da semaphore nesnelere faydalanılmaktadır. Burada öncelikle nesnenin kullanımını göreceğiz. CriticalSection nesnesi işlemlerini user modda yapmaktadır. CriticalSection nesnesinin kullanımı şöyledir:

- 1) CRITICAL_SECTION isimli yapı türünden global bir nesne yaratılır.
- 2) Bu nesne InitializeCriticalSection isimli API fonksiyonuyla ilkdeğerlenir.

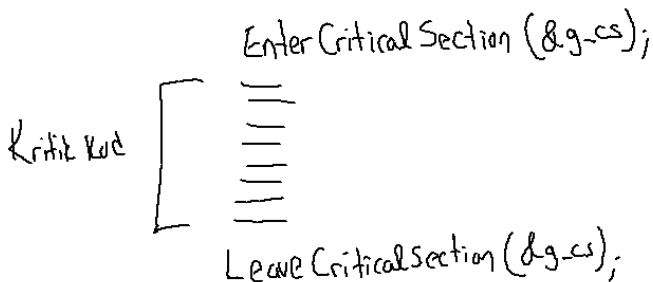
```

void WINAPI InitializeCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);

```

Fonksiyon parametre olarak CRITICAL_SECTION türünden nesnenin adresini almaktadır.

- 3) Kritik kod aşağıdaki gibi oluşturulur:



Bir thread EnterCriticalSection fonksiyonundan girdiğinde artık başka bir thread oradan girmeye çalışırsa bloke olur ve bekler. Taki diğeri LeaveCriticalSection ile kritik kod kilidini bırakana kadar. Böylece EnterCriticalSection ve LeaveCriticalSection arasındaki kodu tek bir thread akışı başından sonuna kadar çalıştırmış olur. Fonksiyonların prototipleri şöyledir:

```
void WINAPI EnterCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
);
```

```
void WINAPI LeaveCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
);
```

Görüldüğü gibi fonksiyonlar CRITICAL_SECTION nesnesinin adresini parametre olarak almaktadır.

Birden fazla thread EnterCriticalSection noktasında blokede beklerse kilit açıldığında hangisi kritik koda girecektir? Şüphesiz adil bir sistemin uygulanması istenir. Ancak Windows bunun bir garantisini bize vermemektedir.

4) İşimiz bitince DeleteCriticalSection fonksiyonuyla yapılan işlemler geri alınmalıdır:

```
void WINAPI DeleteCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
);
```

Bu fonksiyon da CRITICAL_SECTION nesnesinin adresini parametre olarak almaktadır.

Örneğin:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <Windows.h>  
  
void ExitSys(LPCSTR lpszMsg);  
DWORD __stdcall ThreadProc1(void *param);  
DWORD __stdcall ThreadProc2(void *param);  
  
int g_i;  
CRITICAL_SECTION g_cs;  
  
int main(void)  
{  
    DWORD dwThreadId1, dwThreadId2;  
    HANDLE hThread1, hThread2;  
  
    InitializeCriticalSection(&g_cs);  
  
    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)  
        ExitSys("CreateThread", EXIT_FAILURE);  
  
    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)  
        ExitSys("CreateThread", EXIT_FAILURE);  
  
    WaitForSingleObject(hThread1, INFINITE);  
    WaitForSingleObject(hThread2, INFINITE);  
  
    DeleteCriticalSection(&g_cs);  
  
    printf("%d\n", g_i);  
  
    return 0;  
}
```

```

DWORD __stdcall ThreadProc1(void *param)
{
    for (int i = 0; i < 1000000; ++i) {
        EnterCriticalSection(&g_cs);
        ++g_i;
        LeaveCriticalSection(&g_cs);
    }

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    for (int i = 0; i < 1000000; ++i) {
        EnterCriticalSection(&g_cs);
        ++g_i;
        LeaveCriticalSection(&g_cs);
    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

EnterCriticalSection fonksiyonuna verilen parametre kilidi temsil eder. Örneğin programın iki farklı yerinde biz EnterCriticalSection fonksiyonuna aynı argümanı geçerse bunlar aynı kilide ilişkin olurlar. Başka bir örnek şöyle olabilir:

```

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc1(void *param);
DWORD __stdcall ThreadProc2(void *param);

CRITICAL_SECTION g_cs;

int main(void)
{
    DWORD dwThreadId1, dwThreadId2;
    HANDLE hThread1, hThread2;

    InitializeCriticalSection(&g_cs);

    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);
}

```

```

DeleteCriticalSection(&g_cs);

return 0;
}

void CommonProc(const char *str)
{
EnterCriticalSection(&g_cs);

printf("%s: 1. Step\n", str);
Sleep(rand() % 100);
printf("%s: 2. Step\n", str);
Sleep(rand() % 100);
printf("%s: 3. Step\n", str);
Sleep(rand() % 100);
printf("%s: 4. Step\n", str);
Sleep(rand() % 100);
printf("%s: 5. Step\n", str);
Sleep(rand() % 100);
printf("-----\n");

LeaveCriticalSection(&g_cs);
}

DWORD __stdcall ThreadProc1(void *param)
{
int i;

for (i = 0; i < 10; ++i)
CommonProc("ThreadProc1");

return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
int i;

for (i = 0; i < 10; ++i)
CommonProc("ThreadProc2");

return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
DWORD dwLastError = GetLastError();
LPTSTR lpszErr;

if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
LocalFree(lpszErr);
}

exit(EXIT_FAILURE);
}

```

Ayrıca yukarıdaki API fonksiyonlarının dışında bir de TryEnterCriticalSection isimli bir API fonksiyonu vardır. Bu fonksiyon eğer CRITICAL_SECTION nesnesi açıksa (yani hiçbir thread onun kilidini almamışsa) nesnenin kilidini alır. Ancak CRITICAL_SECTION nesnesi kapalıysa (yani başka bir thread onun kilidini almışsa) bu durumda blokeye yol açmaz. Fonksiyonun prototipi şöyledir:

```

BOOL TryEnterCriticalSection(

```

```
LPCriticalSection lpCriticalSection
);
```

Fonksiyon parametre olarak CRITICAL_SECTION nesnesinin adresini alır. Eğer kilit başarılı olarak alınırsa sıfır dışı bir değere, eğer kilit zaten başka bir thread tarafından alınmışsa sıfır değerine geri döner. Pekiyi bu fonksiyonun kullanım amacı ne olabilir? İşte bazen ilgili thread kritik koda girmek istediğinde bloke olucaksa o noktada beklemek yerine başka bir şey yapmak isteyebilir. Bazen kilitlenme durumları (deadlock) bu tür yöntemlerle çözülmektedir.

Aynı thread aynı CRITICAL_SECTION nesnesinin kilidini EnterCriticalSection ya da TryEnterCriticalSection fonksiyonlarıyla ikinci kez bloke olmadan alabilmektedir. Ancak bu tür durumlarda bu thread'in kilidi kaç kez almışsa o kadar LeaveCriticalSection fonksiyonunu uygulaması gerekmektedir. CRITICAL_SECTION nesnelerinin tıpkı ileride göreceğimiz mutex nesnelerinde olduğu gibi thread temelinde sahipliği vardır. Yani bir thread kilitlenmediği CRITICAL_SECTION nesnesinin kilidini LeaveCriticalSection fonksiyonuyla bırakamaz. Bu durumda LeaveCriticalSection fonksiyonu başarısız olmaktadır. Son olarak bir ayrıntı üzerinde de durmak istiyoruz. Bir thread CRITICAL_SECTION nesnesinin kilidini almışken sonlanırsa bu durumda kilidin buruma "tanımsız davranış" oluşturmaktadır. Yani kilidin durumunun ne olacağı hakkında bir garanti verilmemektedir.

Örneğin C++'ta iki thread dinamik büyüyen bir diziye (vector) ekleme yapacak olsun. Ekleme işleminin yarıda kesilmemesi gerekir. Bunun için bu işlem kritik kod kontrolüyle yapılmalıdır. Örneğin eğer aşağıdaki koddan senkronizasyon kısmı kaldırılırsa program çökebilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <vector>

using namespace std;

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc1(void *param);
DWORD __stdcall ThreadProc2(void *param);

CRITICAL_SECTION g_cs;

vector<int> g_v;

int main(void)
{
    DWORD dwThreadId1, dwThreadId2;
    HANDLE hThread1, hThread2;

    InitializeCriticalSection(&g_cs);

    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    DeleteCriticalSection(&g_cs);

    return 0;
}

DWORD __stdcall ThreadProc1(void *param)
{
    for (int i = 0; i < 1000000; ++i) {
        EnterCriticalSection(&g_cs);
        g_v.push_back(i);
    }
}
```

```

        LeaveCriticalSection(&g_cs);
    }

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    for (int i = 0; i < 1000000; ++i) {
        EnterCriticalSection(&g_cs);
        g_v.push_back(i);
        LeaveCriticalSection(&g_cs);
    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Windows sistemlerinde CRITICAL_SECTION nesnesi aynı prosesin thread'leri arasında kritik kod oluşturmak için alternatif diğer yöntemlere göre daha hızlı bir düzenek sunmaktadır.

Windows'ta WaitForSingleObject ve WaitForMultipleObjects API Fonksiyonları

Windows'ta WaitForSingleObject ve WaitForMultipleObjects isimli API fonksiyonları senkronizasyon için kullanılan genel fonksiyonlardır. Bir kernel nesnesinin açık (signaled) ve kapalı (nonsignaled) biçiminde iki durumu vardır. WaitForSingleObject eğer nesne kapalıysa açılana kadar ilgili thread'i bloke ederek bekler. Eğer nesne açık durumdaysa WaitForSingleObject fonksiyonu hiç bekleme yapmadan hemen geri döner. Her nesnenin hangi durumda kapalı hangi durumda açık olduğu belirlenmiştir. Örneğin aslında proseslerin ve thread'lerin kendileri de birer kernel senkronizasyon nesnesi olarak kullanılabilir. Bir proses ya da thread sonlanmıyorsa bu nesnelere senkronizasyon bağlamında kapalı, sonlanmışsa açık durumda olur. Yani biz WaitForSingleObject fonksiyonu ile prosesleri ve thread'leri de bekleyebiliriz. WaitForSingleObject fonksiyonunun prototipi şöyledir:

```

DWORD WINAPI WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMilliseconds
);

```

Fonksiyonun birinci parametresi senkronizasyon nesnesinin (duruma göre thread, proses, mutex, semaphore vs.) handle değerini alır. İkinci parametre zaman aşımını belirtir. Eğer nesne açılmazsa en kötü olasılıkla fonksiyon ikinci parametrede belirtilen milisaniye cinsinden zaman dolduğunda beklemeyi sonlandırır. Ancak INFINITE özel değeri zaman aşımını kaldırmaktadır. Yani ikinci parametre INFINITE olarak geçilirse yalnızca nesnenin açılması durumunda fonksiyon sonlanır. Fonksiyon zaman aşımından dolayı sonlanmışsa WAIT_TIMEOUT özel değerine geri döner. (Ancak "mutex" nesnelerinde mutex'e sahip thread sonlanmışsa fonksiyon WAIT_ABANDONED özel değeriyle geri dönmektedir.) Eğer nesne açık duruma geldiğinden dolayı fonksiyon geri dönmüşse WAIT_OBJECT_0 değeri elde edilir. Fonksiyon başarısız olabilir. Başarısızlık durumunda WAIT_FAILED değerine geri döner.

WaitForMultipleObjects fonksiyonu birden fazla nesneyi beklemek için kullanılır. Bunların hepsi açık duruma geçene kadar ya da en az biri açık duruma geçene kadar bekleme yapılabilir. Fonksiyonun prototipi şöyledir:

```
DWORD WINAPI WaitForMultipleObjects(  
    DWORD nCount,  
    const HANDLE* LpHandles,  
    BOOL bWaitAll,  
    DWORD dwMilliseconds  
);
```

Fonksiyonun birinci parametresi dizideki nesnelerin sayısını belirtir. İkinci parametre HANDLE değerlerinin bulunduğu dizinin adresidir. Üçüncü parametre eğer sıfır dışı ise tüm nesneler açık duruma geçene kadar bekleme yapılır, 0 ise yalnızca bunlardan herhangi biri açık duruma geçene kadar bekleme yapılır. Son parametre zaman aşımını belirtmektedir. Fonksiyon eğer zaman aşımından dolayı sonlanmışsa yine WAIT_TIMEOUT özel değerine geri döner. Eğer üçüncü parametre FALSE ise geri dönüş değeri WAIT_OBJECT_0 + n biçimindedir. Bu n değeri açık duruma geçmiş nesnenin dizideki indeksini belirtir. (Benzer biçimde eğer fonksiyon mutex'e sahip bir thread'in sonlanmasıyla sonlanmışsa bu durumda geri dönüş değeri WAIT_ABANDONED_0 + n biçimindedir.) Buradaki n yine dizide indeks belirtmektedir. Fonksiyon başarısız olursa yine yine WAIT_FAILED değerine geri döner.

Şimdiye kadar biz yalnızca iki tane kernel senkronizasyon nesnesi gördük. Onlar proses ve thread nesnelerydi. Ancak mutex, semaphore, waitabletimer, event gibi daha pek çok senkronizasyon nesnesi vardır. Yukarıda kullandığımız CriticalSection nesnesi bir kernel senkronizasyon nesnesi değildir. Bu nedenle daha hızlı çalışma eğilimindedir. Biz CriticalSection nesnelerini WaitFor fonksiyonlarıyla kullanamayız. Windows'ta en önemli üç kernel senkronizasyon nesnesi mutex, event ve semaphore nesnelere aittir.

Windows Sistemlerinde Kritik Kodların Mutex Nesneleriyle Oluşturulması

Windows'ta mutex nesnelere kritik kod bloklarını oluşturmak için kullanılan kernel senkronizasyon nesnelere aittir. Burada "kernel senkronizasyon nesnesi" terimi "kernel moda geçiş yaparak çalışma" anlamına gelmektedir. Aslında Windows'ta mutex nesnelere CriticalSection nesnelere aynı amaçla kullanılırlar. Ancak CriticalSection nesnelere kernel moda geçmeden tamamen user modda çalışmaktadır. Bu nedenle CriticalSection nesnesi mutex nesnesinden daha hızlı olma eğilimindedir. Pekiyi Windows'ta CriticalSection nesnesi varken biz neden mutex nesnelere tercih edebiliriz? İşte Windows'ta kernel senkronizasyon nesnelere isimli nesnelere aittir. Biz onları onları istersek farklı proseslerin threadleri arasında da senkronizasyon amacıyla kullanabiliriz. Halbuki CriticalSection nesnelere yalnızca aynı prosesin threadleri arasında kullanılabilir. Ayrıca mutex nesnelere ileride de ele alındığı gibi sahiplik temelinde çalışmaktadır. Bu anlamda bazı tür uygulamalarda daha güvenli olduğu söylenebilir. Tabii Windows'ta eğer proseslerarası bir kullanım söz konusu değilse programcının kritik kodları CriticalSection nesnelere ile oluşturması tavsiye edilir.

Mutex ismi İngilizce "Mutual Exclusion" sözcüklerinden türetilmiştir. Mutex yalnızca Windows sistemlerinde değil pek çok işletim sisteminde kullanılan temel bir senkronizasyon nesnesidir. Mutex nesnesinin sahipliği (ownership) denilen bir kavram vardır. Mutex nesnelere thread temelinde bir sahipliği vardır. Bir mutex nesnesinin sahipliği bir thread tarafından alınır. Sahiplik ancak onu alan thread tarafından bırakılabilir. Mutex nesnelere thread tarafından alınmışsa nesne kapalı durumdadır. Eğer bunların sahipliği bir thread tarafından alınmamışsa nesne açık durumdadır.

Mutex nesnelere şöyle kullanılmaktadır:

1) Mutex nesnesi CreateMutex API fonksiyonuyla yaratılır. Proses içi kullanımda mutex'in handle değeri global bir değışkene atanmalıdır:

```
HANDLE WINAPI CreateMutex(  
    LPSECURITY_ATTRIBUTES LpMutexAttributes,  
    BOOL bInitialOwner,  
    LPCTSTR LpName  
);
```

Fonksiyonun birinci parametresi kernel nesnesinin güvenlik bilgilerini belirtir. Bu parametre NULL geçilebilir. İkinci parametre mutex'in başlangıçtaki sahipliğini belirlemede kullanılır. Eğer bu parametre sıfır dışı (TRUE) geçilirse

CreateMutex'i çağıran thread nesnenin sahipliğini de işin başında almış olur. Sıfır (FALSE) geçilirse nesne sahipliği alınmadan yaratılır. Son parametre proseslerarası kullanımdaki ismi belirtir. Aynı proses içindeki kullanımda bu parametre NULL geçilebilir. Fonksiyon başarı durumunda mutex'in handle değerine, başarısızlık durumunda NULL değerine geri döner.

2) Kritik kod şöyle oluşturulur:

```
WaitForSingleObject(g_hMutex, INFINITE);  
[  
Kritik  
Kod  
]  
ReleaseMutex(g_hMutex);
```

Mutex nesnesinin sahipliği bir thread tarafından alınmışsa nesne kapalı (nonsignaled) durumdadır. Nesne kapalıyken WaitForSingleObject fonksiyonu blokede bekler. Nesnenin sahipliği onu almış olan thread tarafından bırakıldığında nesne açık duruma geçer ve bloke sonlandırılır. Böylece akış WaitForSingleObject fonksiyonundan çıkacaktır. WaitForSingleObject nesne açık duruma geçtiğinde aynı zamanda nesnenin sahipliğini de almaktadır. ReleaseMutex fonksiyonu ise nesnenin sahipliğini bırakmak için kullanılmaktadır:

```
BOOL WINAPI ReleaseMutex(  
HANDLE hMutex  
);
```

3) İşlem bittiğinde mutex nesnesi CloseHandle fonksiyonuyla yok edilir. Tabii nesne kapatılmamış olsa bile proses bittiğinde işletim sistemi tarafından tüm kernel nesnelere otomatik olarak kapatılacaktır.

Peki mutex'in sahipliğini almış bir thread sahipliği bırakmadan sonlanırsa ne olur? Çünkü onu bekleyen başka thread'ler olabilir. İşte bu durumda mutex'lere terk edilmiş mutex'ler (abandoned mutex) denilmektedir. Eğer mutex'in sahipliğini almış olan bir thread onu bırakmadan sonlanırsa onu WaitForXXX fonksiyonlarıyla bekleyen thread'ler başarısızlıkla geri dönerler. Böylece onu bekleyen thread'ler bir kilitlenme (deadlock) durumuyla karşılaşmazlar.

Peki bir thread'in bizzat kendisi mutex'in sahipliğini yeniden alabilir mi? Örneğin foo fonksiyonu içerisinde mutex'le oluşturulmuş bir kritik kod olsun bu kritik kod içerisinde bar fonksiyonunu çağırmış olalım. bar fonksiyonu başka yerden de çağrılabilmesi için onun içinde de aynı mutex'e ilişkin kritik kod bulunuyor olsun. Bu durumda thread kendi kendisini kilitler mi? Yanıt Windows sistemlerinde hayır. Yani Windows sistemlerinde bir thread mutex nesnesinin sahipliğini almışsa artık kendisi WaitForXXX fonksiyonlarında bu mutex nesnesi dolayısıyla bloke olmaz. Ancak Windows sistemlerinde aynı thread ilgili mutex nesnesi için ne kadar WaitForSingleObject fonksiyonundan geçmişse o kadar ReleaseMutex uygulamalıdır.

Örneğin:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <windows.h>  
  
void ExitSys(LPCSTR lpszMsg);  
DWORD __stdcall ThreadProc1(void *param);  
DWORD __stdcall ThreadProc2(void *param);  
  
HANDLE g_hMutex;  
  
int main(void)  
{  
    DWORD dwThreadId1, dwThreadId2;  
    HANDLE hThread1, hThread2;
```

```

if ((g_hMutex = CreateMutex(NULL, FALSE, NULL)) == NULL)
    ExitSys("CreateMutex");

if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)
    ExitSys("CreateThread");

if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)
    ExitSys("CreateThread");

WaitForSingleObject(hThread1, INFINITE);
WaitForSingleObject(hThread2, INFINITE);

CloseHandle(g_hMutex);

return 0;
}

void CommonProc(const char *str)
{
    WaitForSingleObject(g_hMutex, INFINITE);

    printf("%s: 1. Step\n", str);
    Sleep(rand() % 100);
    printf("%s: 2. Step\n", str);
    Sleep(rand() % 100);
    printf("%s: 3. Step\n", str);
    Sleep(rand() % 100);
    printf("%s: 4. Step\n", str);
    Sleep(rand() % 100);
    printf("%s: 5. Step\n", str);
    Sleep(rand() % 100);
    printf("-----\n");

    ReleaseMutex(g_hMutex);
}

DWORD __stdcall ThreadProc1(void *param)
{
    int i;

    for (i = 0; i < 10; ++i)
        CommonProc("ThreadProc1");

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    int i;

    for (i = 0; i < 10; ++i)
        CommonProc("ThreadProc2");

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }
}

```



```
    exit(EXIT_FAILURE);  
}
```

Bir mutex nesnesinin açılmasını birden fazla thread WaitForXXX fonksiyonlarıyla bekliyor olsun. Mutex nesnesi açıldığında onu bekleyen hangi thread'in mutex nesnesinin sahipliğini alacağı konusunda bir garanti verilmemiştir.

UNIX/Linux Sistemlerinde Mutex Nesneleriyle Kritik Kodların Oluşturulması

UNIX/Linux sistemlerinde Windows sistemlerindeki gibi bir CriticalSection nesnesi yoktur. Bu sistemlerde kritik kodlar mutex nesneleriyle oluşturulmaktadır. Fakat UNIX/Linux sistemlerindeki mutex nesneleri proseslerarası kullanılmayacaksa zaten kernel moda geçmez. Yani UNIX/Linux sistemlerindeki mutex nesneleri performans bakımından Windows sistemlerindeki CriticalSection nesneleri gibidir.

UNIX/Linux sistemlerinde mutex nesneleri şöyle kullanılır:

1) Mutex'i temsil eden pthread_mutex_t türünden global bir nesne tanımlanır. Örneğin:

```
pthread_mutex_t g_mutex;
```

2) Bu mutex nesnesi pthread_mutex_init fonksiyonula ilkdeğerlenir.

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Fonksiyonun birinci parametresi global nesnenin adresini, ikinci parametresi mutex özelliklerini belirten yapı nesnesinin adresini alır. İkinci parametre NULL geçilebilir. Bu durumda mutex default özelliklerle yaratılır. Fonksiyon başarı durumunda sıfır, başarısızlık durumunda hata kodunun kendisine geri döner.

Aslında pthread_mutex_t türü bir yapı belirtmektedir. Mutex nesnesi PTHREAD_MUTEX_INITIALIZER makrosuyla da ilkdeğerlenebilir. Örneğin:

```
pthread_mutex_t g_mutex = PTHREAD_MUTEX_INITIALIZER;
```

3) Kritik kod aşağıdaki gibi oluşturulur:

```
pthread_mutex_lock(&g_mutex);  
[  
  _____  
  _____  
  _____  
  _____  
  _____  
pthread_mutex_unlock(&g_mutex);
```

Kritik Kod

pthread_mutex_lock eğer mutex nesnesinin sahipliği başka bir thread tarafından alınmışsa sahiplik pthread_mutex_unlock fonksiyonuyla bırakılana kadar thread'i blokede bekletir. Eğer mutex'in sahipliği başka bir thread tarafından alınmamışsa mutex'in sahipliğini alarak geri döner. Fonksiyonların prototipleri şöyledir:

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Fonksiyonlar mutex nesnesinin adresini alır, başarı durumunda sıfır, başarısızlık durumunda hata kodunun kendisine geri dönerler.

4. İşlem bittiğinde mutex nesnesi pthread_mutex_destroy fonksiyonuyla yok edilir:

```
#include <pthread.h>
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Fonksiyon parametre olarak mutex nesnesinin adresini almaktadır. Başarı durumunda 0 değerine, başarısızlık durumunda hata kodunun kendisine geri döner.

Örneğin:

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

```
void *thread_proc1(void *param);
void *thread_proc2(void *param);
void common_proc(const char *str);
```

```
pthread_mutex_t g_mutex;
```

```
int main(void)
{
    pthread_t tid1, tid2;
    int result;

    if ((result = pthread_mutex_init(&g_mutex, NULL)) != 0) {
        printf("pthread_mutex_init: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0) {
        printf("pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0) {
        printf("pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_mutex_destroy(&g_mutex);

    return 0;
}
```

```
void common_proc(const char *str)
{
    pthread_mutex_lock(&g_mutex);

    printf("%s: 1. Step\n", str);
    usleep(rand() % 300000);
    printf("%s: 2. Step\n", str);
    usleep(rand() % 300000);
    printf("%s: 3. Step\n", str);
    usleep(rand() % 300000);
    printf("%s: 4. Step\n", str);
    usleep(rand() % 300000);
}
```

```

printf("%s: 5. Step\n", str);
usleep(rand() % 300000);
printf("-----\n");

pthread_mutex_unlock(&g_mutex);
}

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < 10; ++i)
        common_proc("ThreadProc1");

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 10; ++i)
        common_proc("ThreadProc2");

    return NULL;
}

```

UNIX/Linux sistemlerinde default durumda mutex nesnelere özyinelemeli değildir. Yani bir thread mutex nesnesinin sahipliğini aldıktan sonra yeniden almaya çalışırsa kendi kendini kilitler. (Windows'ta mutex nesnelere özyinelemeli olduğunu anımsayınız.) Ancak biz istersek UNIX/Linux sistemlerinde mutex nesnesini özyinelemeli hale getirebiliriz. Bunun için pthread_mutex_init fonksiyonunun ikinci parametresi olan mutex özneliklerinin uygun biçimde set edilmesi gerekir. Bu set işlemi için önce pthread_mutexattr_t türünden bir nesne alınıp bu nesne pthread_mutexattr_init fonksiyonuyla ilkdeğerlenir. Sonra da özyinelemeli mutex bu öznelik nesnesinin pthread_mutexattr_settype fonksiyonu ile set edilmesi gerekmektedir. Bu işlem şöyle yapılabilir:

```

pthread_mutexattr_t mutexattr;
...
pthread_mutexattr_init(&mutexattr);
pthread_mutexattr_settype(&mutexattr, PTHREAD_MUTEX_RECURSIVE);

if ((result = pthread_mutex_init(&g_mutex, &mutexattr)) != 0)
    exit_sys_result("pthread_mutex_init", result);

```

Tabii yine UNIX/Linux sistemlerinde de özyinelemeli mutex nesnelere için ne kadar sayıda pthread_mutex_lock fonksiyonu uygulanmışsa nesne açmak için o sayıda pthread_mutex_unlock fonksiyonu uygulanmalıdır.

Semaphore Nesnelere

Trenlerdeki dur geç lambalarına semaphore denilmektedir. Semaphore'lar sayaçlı senkronizasyon nesnelereidir. Bir kritik koda en fazla n tane akışın girebilmesi için kullanılırlar. Semaforun bir sayacı vardır. Bu sayaç sıfırdan büyükse nesne açık durumdadır. Yani semafor kilitli değildir. Her geçiş yapıldığında sayaç bir eksiltilir. Sayaç sıfıra geldiğinde semafor nesnesi kapalı duruma geçer. Yani kilitlenir. Artık geçiş yapılmak istendiğinde ilgili thread bloke edilir. Kritik koddan çıkıldığında sayaç 1 artırılır. Böylece kritik koda en fazla n tane thread'in girmesi sağlanır.

Peki kritik koda neden n tane akış girsin ki? İşte eğer n tane kaynak varsa ama bunu elde etmek için rekabet eden çok thread bulunuyorsa bazı thread'lerin beklemesi gerekir. Böylece kritik koda giren her thread'e bir kaynak tahsis edilir. Semaphore sayacı sıfıra geldiğinde artık tahsis edilecek kaynak kalmamıştır. Bir thread işini bitirdiğinde kaynağı bırakır. Kritik koddan çıkar. Sayaç böylece bir artırılır. Yeni bir thread artık kritik koda girebilir. Yani semaphore'lar özellikle belli sayıda kaynağı senkronize biçimde paylaşdırmak amacıyla kullanılmaktadır.

Semafor sayacı başlangıçta 1 ise böyle semaphore'lara ikili (binary) semaphore denilmektedir. İkili semaphore'lar mutex nesnelere çok benzerdirler. Ancak yine de mutex'le ikili semaphore arasında önemli bir fark da vardır. Bir mutex nesnesinin sahipliği ancak onu almış thread tarafından bırakılabilir. Ancak bir semaphore'un sayacı başka thread'ler tarafından artırılabilir.

Semaphore nesnelere özellikle üretici-tüketici (producer-consumer) probleminin çözümü için kullanılmaktadır.

Windows Sistemlerinde Semaphore Nesnelere Kullanılması

Windows sistemlerinde semaphore nesnelere şöyle kullanılır:

1) CreateSemaphore fonksiyonuyla semaphore yaratılır ve handle değeri global bir değişkenin içerisine yerleştirilir:

```
HANDLE WINAPI CreateSemaphore(  
    LPSECURITY_ATTRIBUTES LpSemaphoreAttributes,  
    LONG lInitialCount,  
    LONG lMaximumCount,  
    LPCTSTR LpName  
);
```

Fonksiyonun birinci parametresi semaphore nesnesinin güvenlik bilgilerini belirtir. Bu parametre NULL geçilebilir. İkinci parametre başlangıçtaki semaphore sayacının değeri. Programcı bu değeri kritik koda paylaşılacak kaynak sayısını referans olarak girer. Üçüncü parametre semaphore sayacının çıkacağı maksimum değeri belirtir. Genellikle ikinci ve üçüncü parametreler aynı değeri girilerek oluşturulmaktadır. Son parametre proseslerarası kullanım için gereken ismi belirtir. Bu parametre proses içi kullanımda NULL geçilebilir. Fonksiyon başarı durumunda semaphore nesnesinin handle değerine, başarısızlık durumunda NULL adrese geri döner.

2) Kritik kod şöyle oluşturulur:

```
WaitForSingleObject(g_hSem, INFINITE);  
[  
    Kritik Kod  
]  
ReleaseSemaphore(g_hSem, 1, NULL);
```

Bir semaphore nesnesi sayacı sıfırdan büyükse açık durumda, sıfır ise kapalı durumdadır. Böylece WaitForSingleObject fonksiyonu eğer semaphore sayacı 0'dan büyükse blokeye yol açmadan thread'in kritik koda girmesine izin verecektir. Ancak WaitForSingleObject fonksiyonuyla bir semaphore bekleniyorsa ve nesne açık durumdaysa bu fonksiyon aynı zamanda otomatik olarak sayacı 1 eksiltmektedir. Başka bir deyişle akış WaitForSingleObject fonksiyonunu her geçtiğinde semaphore sayacı 1 eksiltmiş olur. ReleaseSemaphore fonksiyonu ise semaphore sayacını artırmaktadır. ReleaseSemaphore fonksiyonun prototipi şöyledir:

```
BOOL WINAPI ReleaseSemaphore(  
    HANDLE hSemaphore,  
    LONG lReleaseCount,  
    LPLONG lpPreviousCount  
);
```

Fonksiyonun birinci parametresi semaphore'un handle değerini, ikinci parametresi sayacın artırılabilecek değeri belirtir. Üçüncü parametre sayacın önceki değerinin yerleştirileceği long türden nesnenin adresini alır. Bu parametre NULL geçilebilir.

3) İşlem bitince semaphore nesnesi CloseHandle fonksiyonuyla yok edilir.

Aşağıdaki örnekte semaphore sayacının başlangıçtaki değeri 3 tutulmuştur. Sonra 10 tane thread yaratılmış ve 10 thread'in de 100 kere bir döngü içerisinde kritik koda girmesi sağlanmıştır. Semaphore sayesinde kritik koda aynı anda

yalnızca 3 thread girebilecektir. Örneğimizde semaphore sayacının değeri alınarak ekrana yazdırılmaktadır. Bu değerin 0 olması kritik kodda 3 thread'in 1 olması 2 thread'in 2 olması ise 1 thread'in bulunduğu anlamına gelmektedir.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>

#define SEMCOUNT      3
#define NTHREADS       10

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc(void *param);
void doSomething(void);

HANDLE g_hSemaphore;
int g_count;

int main(void)
{
    HANDLE hThreads[NTHREADS];
    DWORD threadIds[NTHREADS];
    int i;

    srand(time(NULL));

    if ((g_hSemaphore = CreateSemaphore(NULL, SEMCOUNT, SEMCOUNT, NULL)) == NULL)
        ExitSys("CreateSemaphore");

    for (i = 0; i < NTHREADS; ++i)
        if ((hThreads[i] = CreateThread(NULL, 0, ThreadProc, NULL, 0, &threadIds[i])) == NULL)
            ExitSys("CreateThread");

    WaitForMultipleObjects(NTHREADS, hThreads, TRUE, INFINITE);
    CloseHandle(g_hSemaphore);

    return 0;
}

DWORD __stdcall ThreadProc(void *param)
{
    int i;

    for (i = 0; i < 100; ++i) {
        doSomething();
        Sleep(rand() % 300);
    }

    return 0;
}

void doSomething()
{
    long semCount;

    WaitForSingleObject(g_hSemaphore, INFINITE);

    Sleep(rand() % 100);

    /* Critical Sections */

    ReleaseSemaphore(g_hSemaphore, 1, &semCount);
    printf("%d ", semCount);
}

void ExitSys(LPCSTR lpszMsg)
```


bayrakla birlikte O_EXCL bayrağı da kullanılabilir. O_CREAT ve O_EXCL birlikte kullanılırsa semaphore nesnesi zaten daha önce yaratılmışsa fonksiyon başarısız olmaktadır. Tabii açış modunda O_CREAT girilmişse artık fonksiyona iki argüman daha girmek gerekecektir. Bu durumda üçüncü argüman nesnenin erişim haklarını dördüncü argüman ise semaphore sayacının başlangıç değerini belirtir. sem_open fonksiyonu başarı durumunda sem_t türünden semaphore nesnesinin adresine, başarısızlık durumunda ise SEM_FAILED değerine geri döner. Bu fonksiyon errno değerini set etmektedir.

2) Kritik kod şöyle oluşturulur:

```
sem_wait(&g_sem);  
[  
Kritik Kod  
]  
sem_post(&g_sem);
```

sem_wait fonksiyonu semaphore sayacı sıfırdan büyükse bloke olmadan kritik koda geçiş yapar. Ancak geçiş yaparken semaphore sayacını 1 eksiltir. sem_post ise semaphore sayacını 1 artırmaktadır.

```
#include <semaphore.h>
```

```
int sem_wait(sem_t *sem);  
int sem_post(sem_t *sem);
```

sem_wait ve sem_post fonksiyonları başarı durumunda 0, başarısızlık durumunda -1 değerine geri dönmektedir. Bunlar errno değerini set ederler.

3) İşlemler bitince semaphore nesnesi sem_destroy fonksiyonuyla boşaltılabilir.

```
#include <semaphore.h>
```

```
int sem_destroy(sem_t *sem);
```

Aşağıda ikili semaphore nesnesi ile kritik kod oluşturmaya bir örnek verilmektedir:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <time.h>  
#include <unistd.h>  
#include <pthread.h>  
#include <semaphore.h>
```

```
void exit_sys(const char *msg);  
void exit_sys_result(const char *msg, int result);
```

```
void *thread_proc1(void *param);  
void *thread_proc2(void *param);
```

```
void do_something(const char *msg);
```

```
sem_t g_sem;
```

```
int main(void)  
{  
    int result;  
    pthread_t tid1, tid2;  
  
    srand(time(NULL));
```

```

if (sem_init(&g_sem, 0, 1) == -1)
    exit_sys("sem_init");

if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0)
    exit_sys_result("pthread_create", result);

if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0)
    exit_sys_result("pthread_create", result);

pthread_join(tid1, NULL);
pthread_join(tid2, NULL);

sem_destroy(&g_sem);

return 0;
}

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        do_something("thread-1");
    }

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        do_something("thread-2");
    }

    return NULL;
}

void do_something(const char *str)
{
    sem_wait(&g_sem);

    printf("-----\n");
    printf("%s: Step-1\n", str);
    usleep(rand() % 300000);
    printf("%s: Step-2\n", str);
    usleep(rand() % 30000);
    printf("%s: Step-3\n", str);
    usleep(rand() % 300000);
    printf("%s: Step-4\n", str);
    usleep(rand() % 300000);
    printf("%s: Step-5\n", str);
    usleep(rand() % 300000);
    sem_post(&g_sem);
}

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

    exit(EXIT_FAILURE);
}

void exit_sys_result(const char *msg, int result)

```



```
{
    fprintf(stderr, "%s: %s\n", msg, strerror(result));
    exit(EXIT_FAILURE);
}
```

UNIX/Linux sistemlerine daha sonraları zaman aşımli sem_timedwait isimli bir semaphore bekleme fonksiyonu da eklenmiştir:

```
#include <semaphore.h>

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

Bu fonksiyon eğer semaphore nesnesi açılmadıysa en kötü olasılıkla ikinci parametresinde belirtilen zaman dolduğunda blokeyi çözmektedir.

Yine UNIX/Linux sistemlerinde sem_trywait isimli POSIX fonksiyonu blokesiz semaphore kontrolü yapmak için kullanılmaktadır. Bu fonksiyon hiçbir zaman blokeye yol açmaz. Eğer semaphore sayacı 0 ise fonksiyon -1 ile geri döner. Eğer semaphore sayacı 0'dan büyükse fonksiyon sıfır değerine geri dönmektedir.

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

Nihayet bu sistemlerde semaphore sayacının değeri sem_getvalue POSIX fonksiyonuyla elde edilebilir:

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

Üretici-Tüketici Problemi (Producer-Consumer Problem)

Üretici-Tüketici problemi uygulamalarda en fazla karşılaşılan senkronizasyon problemlerinden bir tanesidir. Bu problemde işlemleri hızlandırmak için programcı iki thread kullanmaktadır. Birinci thread bir bilgiyi elde eder fakat işlemez. Bunu işlemesi için diğer thread'e verir. Bu verme işlemini o bilgiyi ortak paylaşılan alana yazarak yapar. Diğer thread de bilgiyi oradan alarak işler. İki thread de bunları bir döngü içerisinde çok defalar yapmaktadır. Böyle bir sistem hızlandırma sağlar. Çünkü eğer tek bir thread hem bilgiyi elde edip hem de işlerse toplamda birim zamanda daha az bilgi işlenmiş olur. Halbuki thread'lerden biri bilgiyi elde ederken diğeri işlerse birim zamanda işlenen bilgi miktarı artar. Bu problemde bilgiyi elde eden thread'e üretici thread (producer thread), onu işleyen thread'e de tüketici thread (consumer thread) denilmektedir.

Üretici-Tüketici problemindeki senkronizasyon sorunu şudur: Üretici thread henüz tüketici eski bilgiyi almadan yeni bilgiyi paylaşılan alana yerleştirmemelidir. Yerleştirirse eski bilgi ezilir. Benzer biçimde tüketici thread de henüz üretici thread yeni bilgiyi yerleştirmeden eski bilgiyi paylaşılan alandan ikinci kez almamalıdır. Aşağıda bir üretici-tüketici problemi senkronizasyon olmadan simüle edilmeye çalışılmıştır. Bu programı çalıştırarak buradaki senkronizasyon sorununu hemen anlayabilirsiniz.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ProducerProc(LPVOID param);
DWORD __stdcall ConsumerProc(LPVOID param);

int g_shared;

int main(void)
{
```

```

HANDLE hThreadProducer, hThreadConsumer;
DWORD dwProducerID, dwConsumerID;

srand(time(NULL));

if ((hThreadProducer = CreateThread(NULL, 0, ProducerProc, NULL, 0, &dwProducerID)) == NULL)
    ExitSys("CreateThread");

if ((hThreadConsumer = CreateThread(NULL, 0, ConsumerProc, NULL, 0, &dwConsumerID)) == NULL)
    ExitSys("CreateThread");

WaitForSingleObject(hThreadProducer, INFINITE);
WaitForSingleObject(hThreadConsumer, INFINITE);

return 0;
}

DWORD __stdcall ProducerProc(LPVOID param)
{
    int val = 0;

    for (;;) {

        g_shared = val;
        Sleep(rand() % 300);
        if (val == 99)
            break;
        ++val;
    }
}

DWORD __stdcall ConsumerProc(LPVOID param)
{
    int val;

    for (;;) {
        val = g_shared;
        printf("%d ", val);
        Sleep(rand() % 300);
        if (val == 99)
            break;
    }
    printf("\n");
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Burada üretici thread 0'dan 100'e kadar (100 dahil değil) sayıları g_shared ile temsil edilen paylaşılan alana yerleştirmektedir:

```

DWORD __stdcall ProducerProc(LPVOID param)
{
    int val = 0;

```

```

for (;;) {
    g_shared = val;
    Sleep(rand() % 300);
    if (val == 99)
        break;
    ++val;
}
}

```

Değerleri elde ederken 0 ile 300 milisaniye arasında rastgele bir bekleme yapıldığına dikkat ediniz. Çünkü üretici thread'in değeri elde etmesi farklı zaman aralıklarında yapılabilecektir. Tüketici thread ise bir döngüde bu g_shared paylaşılan alan içerisindeki değeri alarak kullanmaktadır:

```

DWORD __stdcall ConsumerProc(LPVOID param)
{
    int val;

    for (;;) {
        val = g_shared;
        printf("%d ", val);
        Sleep(rand() % 300);
        if (val == 99)
            break;
    }
    printf("\n");
}

```

Tüketici thread'in paylaşılan alandan aldığı bilgiyi işlerken rastgele bekleme yaptığına dikkat ediniz. Çünkü işleme işlemi değişik zaman aralıklarıyla yapılabilmektedir. Bu programın örnek bir çıktısı şöyledir:

```

0 0 1 3 3 5 6 7 8 9 10 11 12 12 13 15 16 17 17 19 20 20 21 22 24 25 26 27 27 28 29 30 31 32 33 34
35 37 37 39 39 40 42 42 43 44 46 47 47 49 50 50 52 53 53 54 55 56 58 58 59 60 61 62 64 64 65 66
68 69 69 70 71 73 74 74 75 76 78 78 79 81 81 82 83 84 86 86 87 89 89 90 91 92 93 95 95 97 97 99

```

Burada tüketicinin bazı değerleri alamadığına bazılarını ise birden fazla kez aldığına dikkat ediniz.

İşte üretici-tüketici problemleri tipik olarak semaphore nesneleri kullanılarak gerçekleştirilir.

Üretici Tüketici Probleminin Semaphore Nesneleriyle Çözümü

Semaphore'lar tipik olarak üretici-tüketici problemlerini çözmek için kullanılabilir. Bunun için Mutex nesnesi kullanılamaz. Çünkü mutex'in sahipliğini ancak onu alan thread bırakabilmektedir. Üretici tüketici probleminin semaphore nesneleriyle tipik çözümü şöyledir:

```

HANDLE g_hSemProducer;
HANDLE g_hSemConsumer;
DATA_TYPE g_shared;
g_hSemProducer = CreateSemaphore(NULL, 1, 1, NULL); // Başlangıç sayacı = 1
g_hSemConsumer = CreateSemaphore(NULL, 0, 1, NULL); // Başlangıç sayacı = 0

```

ÜRETİCİ

```

for(;;) {
    <bilgiyi ekle>
    WaitForSingleObject(g_hSemProducer, INFINITE);
    <bilgiyi paylaşılana yerleştir>
    ReleaseSemaphore(g_hSemConsumer, 1, NULL);
}

```

TÜKETİCİ

```

for(;;) {
    WaitForSingleObject(g_hSemConsumer, INFINITE);
    <bilgiyi paylaşılana al>
    ReleaseSemaphore(g_hSemProducer, 1, NULL);
    <bilgiyi kullan>
}

```

Burada başlangıçta üretici semaphore'unun sayacı 1'dir. Bu nedenle işin başında üretici semaphore hiç bloke olmadan kritik koddan geçer ve sayaç sıfıra düşer. Bu sırada tüketici beklemektedir. Çünkü tüketici semaphore'un sayacı 0'dır. Üretici bilgiyi paylaşılan alana yerleştirdikten sonra tüketicinin semaphore sayacını bir artırır. Böylece tüketici blokedan kurtulur ve bilgiyi alır. Bu sırada üretici yine başa dönüp beklemektedir. Çünkü artık onun semaphore sayacı sıfıra düşmüştür. Tüketici bilgiyi alınca bu sefer üreticinin semaphore sayacını 1 artırır. Görüldüğü gibi algortimada üretici tüketiciyi, tüketici ise üreticiyi blokedan kurtarmaktadır.

Üretici Tüketici problemi Windows'ta aşağıdaki gibi kodlanabilir:

```
/* ProducerConsumer.c */

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProcProducer(void *param);
DWORD __stdcall ThreadProcConsumer(void *param);
HANDLE g_hSemProducer;
HANDLE g_hSemConsumer;
int g_shared;

int main(void)
{
    DWORD dwThreadIdProducer, dwThreadIdConsumer;
    HANDLE hThreadProducer, hThreadConsumer;

    srand(time(NULL));

    if ((g_hSemProducer = CreateSemaphore(NULL, 1, 1, NULL)) == NULL)
        ExitSys("CreateSemaphore");

    if ((g_hSemConsumer = CreateSemaphore(NULL, 0, 1, NULL)) == NULL)
        ExitSys("CreateSemaphore");

    if ((hThreadProducer = CreateThread(NULL, 0, ThreadProcProducer, NULL, 0, &dwThreadIdProducer)) ==
    NULL)
        ExitSys("CreateThread");

    if ((hThreadConsumer = CreateThread(NULL, 0, ThreadProcConsumer, NULL, 0, &dwThreadIdConsumer)) ==
    NULL)
        ExitSys("CreateThread");

    WaitForSingleObject(hThreadProducer, INFINITE);
    WaitForSingleObject(hThreadConsumer, INFINITE);

    CloseHandle(g_hSemProducer);
    CloseHandle(g_hSemConsumer);

    return 0;
}

DWORD __stdcall ThreadProcProducer(void *param)
{
    int i;

    i = 0;
    for (;;) {
        Sleep(rand() % 500);

        WaitForSingleObject(g_hSemProducer, INFINITE);
        g_shared = i;
        ReleaseSemaphore(g_hSemConsumer, 1, NULL);
    }
}
```

```

        if (i == 99)
            break;
        ++i;
    }

    return 0;
}

DWORD __stdcall ThreadProcConsumer(void *param)
{
    int val;

    for (;;) {
        WaitForSingleObject(g_hSemConsumer, INFINITE);
        val = g_shared;
        ReleaseSemaphore(g_hSemProducer, 1, NULL);

        Sleep(rand() % 300);
        printf("%d ", val);
        fflush(stdout);
        if (val == 99)
            break;
    }
    printf("\n");

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Yukarıdaki program UNIX/Linux Sistemlerinde şöyle yazılabilir:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_proc_producer(void *param);
void *thread_proc_consumer(void *param);

sem_t g_sem_producer;
sem_t g_sem_consumer;
int g_shared;

int main(void)
{
    pthread_t tid_producer, tid_consumer;
    int result;

    if ((result = sem_init(&g_sem_producer, 0, 1)) != 0) {

```

```

    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

if ((result = sem_init(&g_sem_consumer, 0, 0)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer, NULL)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer, NULL)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

pthread_join(tid_producer, NULL);
pthread_join(tid_consumer, NULL);

sem_destroy(&g_sem_producer);
sem_destroy(&g_sem_consumer);

return 0;
}

void *thread_proc_producer(void *param)
{
    int i;

    i = 0;
    for (;;) {
        usleep(rand() % 300000);
        sem_wait(&g_sem_producer);
        g_shared = i;
        sem_post(&g_sem_consumer);
        if (i == 99)
            break;
        ++i;
    }

    return NULL;
}

void *thread_proc_consumer(void *param)
{
    int val;

    for (;;) {
        sem_wait(&g_sem_consumer);
        val = g_shared;
        sem_post(&g_sem_producer);

        usleep(rand() % 300000);
        printf("%d ", val);
        fflush(stdout);
        if (val == 99)
            break;
    }
    printf("\n");

    return NULL;
}

```

Üretici Tüketici Probleminin Kuyruklu Versiyonu

Üretici Tüketici probleminde ortadaki paylaşılan alanın tek bir elemandan oluşması yerine onun birden fazla elemandan oluşan bir kuyruk sistemi olması paylaşımı hızlandırır. Çünkü bu durumda bekleme olasılığı azalacaktır. Şöyle ki: Bu durumda üretici kuyruk tamamen doluyken, tüketici de kuruk tamamen boşken bekleme yapar. Bu çözümde semaphore sayaçları kuyruk uzunluğu kadar olmalıdır. Kuyruklar sonraki bölümde ele alınacaktır. Ancak burada sanki ele alınmış gibi kodlama yapacağız.

Tamponlu örnek çözüm şöyle olabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <Windows.h>

#define QSIZE      10

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProcProducer(void *param);
DWORD __stdcall ThreadProcConsumer(void *param);
HANDLE g_hSemProducer;
HANDLE g_hSemConsumer;
int g_queue[QSIZE];
int g_head, g_tail;

int main(void)
{
    DWORD dwThreadIdProducer, dwThreadIdConsumer;
    HANDLE hThreadProducer, hThreadConsumer;

    srand(time(NULL));

    if ((g_hSemProducer = CreateSemaphore(NULL, QSIZE, QSIZE, NULL)) == NULL)
        ExitSys("CreateSemaphore");

    if ((g_hSemConsumer = CreateSemaphore(NULL, 0, QSIZE, NULL)) == NULL)
        ExitSys("CreateSemaphore");

    if ((hThreadProducer = CreateThread(NULL, 0, ThreadProcProducer, NULL, 0, &dwThreadIdProducer)) ==
        NULL)
        ExitSys("CreateThread");

    if ((hThreadConsumer = CreateThread(NULL, 0, ThreadProcConsumer, NULL, 0, &dwThreadIdConsumer)) ==
        NULL)
        ExitSys("CreateThread");

    WaitForSingleObject(hThreadProducer, INFINITE);
    WaitForSingleObject(hThreadConsumer, INFINITE);

    CloseHandle(g_hSemProducer);
    CloseHandle(g_hSemConsumer);

    return 0;
}

DWORD __stdcall ThreadProcProducer(void *param)
{
    int i;

    i = 0;
    for (;;) {
        Sleep(rand() % 300);

        WaitForSingleObject(g_hSemProducer, INFINITE);
```

```

    g_queue[g_tail] = i;
    ReleaseSemaphore(g_hSemConsumer, 1, NULL);

    ++g_tail;
    g_tail %= QSIZE;

    if (i == 99)
        break;
    ++i;
}

return 0;
}

DWORD __stdcall ThreadProcConsumer(void *param)
{
    int val;

    for (;;) {
        WaitForSingleObject(g_hSemConsumer, INFINITE);
        val = g_queue[g_head];
        ReleaseSemaphore(g_hSemProducer, 1, NULL);

        ++g_head;
        g_head %= QSIZE;

        Sleep(rand() % 300);
        printf("%d ", val);
        fflush(stdout);
        if (val == 99)
            break;
    }
    printf("\n");

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Yukarıdaki program UNIX/Linux Sistemlerinde şöyle yazılabilir:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define QUEUE_SIZE    100

void *thread_proc_producer(void *param);
void *thread_proc_consumer(void *param);

```



```

sem_t g_sem_producer;
sem_t g_sem_consumer;
int g_queue[QUEUE_SIZE];
int g_tail, g_head;

int main(void)
{
    pthread_t tid_producer, tid_consumer;
    int result;

    if ((result = sem_init(&g_sem_producer, 0, QUEUE_SIZE)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = sem_init(&g_sem_consumer, 0, 0)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid_producer, NULL, thread_proc_producer, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid_consumer, NULL, thread_proc_consumer, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    pthread_join(tid_producer, NULL);
    pthread_join(tid_consumer, NULL);

    sem_destroy(&g_sem_producer);
    sem_destroy(&g_sem_consumer);

    return 0;
}

void *thread_proc_producer(void *param)
{
    int i;

    i = 0;
    for (;;) {
        usleep(rand() % 300000);
        sem_wait(&g_sem_producer);
        g_queue[g_tail] = i;
        sem_post(&g_sem_consumer);
        g_tail++;
        g_tail %= QUEUE_SIZE;
        if (i == 99)
            break;
        ++i;
    }

    return NULL;
}

void *thread_proc_consumer(void *param)
{
    int val;

    for (;;) {
        sem_wait(&g_sem_consumer);

```

```

val = g_queue[g_head];
sem_post(&g_sem_producer);

g_head++;
g_head %= QUEUE_SIZE;

usleep(rand() % 300000);
printf("%d ", val);
fflush(stdout);
if (val == 99)
    break;
}
printf("\n");

return NULL;
}

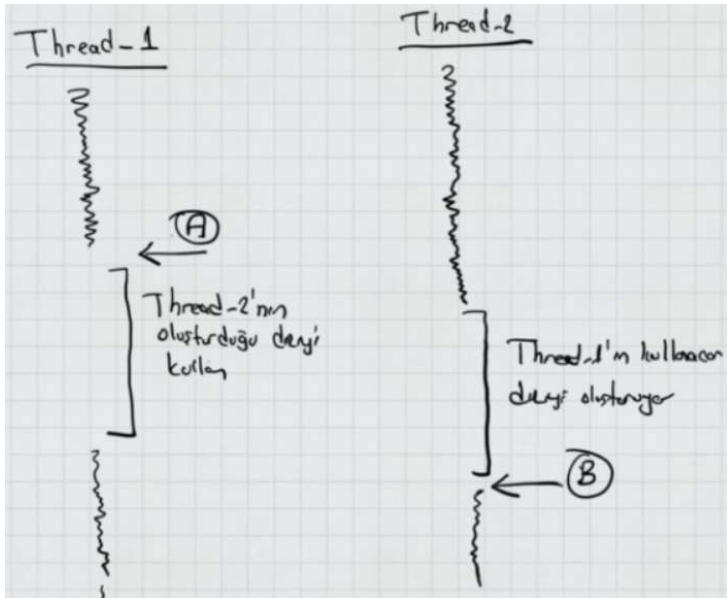
```

Üretici Tüketici Probleminin Diğer Biçimleri

Üretici tüketici probleminde birden fazla üretici ve/veya birden fazla tüketici olabilir. Birden fazla üreticinin olduğu durumda aslında fazlaca bir sorun yoktur. Üreticiler aynı semaphore'u kullanırlar. Birden fazla üretici ve birden fazla tüketicinin bulunması durumunda problemin kuyruklu versiyonunda kuyruk sistemine birden fazla thread'in yazma ve okuma amaçlı erişimi söz konusu olacağından kuyruk sisteminin de ek bir kritik kod içerisinde ele alınması gerekir. Ayrıca çok üretici ve çok tüketicili üretici tüketici problemlerinde iletim sonlandığında bunların bazıları blokede bekler durumda kalabilir. Bunların o durumdan kurtarılması gerekir. Bu değişik biçimlerde yapılabilir. Birden fazla tüketici de yine aynı semaphore'u kullanabilir. Aynı blokede kalma durumu yine tüketiciler için de söz konusu olabilmektedir.

Windows'ta Event Senkronizasyon Nesneleri

Windows'ta event isminde bir senkronizasyon nesnesi vardır. Bu nesne belli bir koşul sağlanana kadar blokeye yol açar. Örneğin bir thread belli bir noktadan sonra diğer bir thread'in oluşturduğu bir bilgiyi kullanmak isteyebilir. Fakat diğer thread bu bilgiyi henüz oluşturmamış olabilir. Bu nedenle söz konusu thread belli bir noktada artık diğerinin bu işi bitirmesini bekleyebilir. Örneğin:



Burada 1 Numaralı Thread A noktasına geldiğinde 2 Numaralı Thread B noktasından henüz geçmemişse o noktada beklemesi gerekir. Ta ki 2 Numaralı thread B noktasından geçene kadar. İşte event nesneleri bu tür amaçlarla kullanılmaktadır. Event nesnelere benzeri UNIX/Linux sistemlerinde "koşul değişkenleri (conditional variables)" ismiyle bulunmaktadır.

Event nesnelere kullanımını şöyledir:

1) Nesne CreateEvent fonksiyonuyla yaratılır ve handle değeri global bir değişkende saklanır.

```
HANDLE WINAPI CreateEvent(  
    LPSECURITY_ATTRIBUTES lpEventAttributes,  
    BOOL bManualReset,  
    BOOL bInitialState,  
    LPCTSTR lpName  
);
```

Fonksiyonun birinci parametresi nesnenin güvenlik bilgilerini belirtir. Bu parametre NULL geçilebilir. İkinci parametre event nesnesinin "otomatik" mi yoksa "manuel" mi olacağını belirtmektedir. Bu parametre FALSE girilirse event nesnesi otomatik, TRUE girilirse manuel moda olur. Nesne açık duruma geçtiğinde WaitForXXX fonksiyonlarından geçiş yapıldığında nesne otomatik olarak yeniden kapalı duruma geçiyorsa bu moda otomatik mod denilmektedir. Manuel moda nesneyi yeniden kapalı duruma geçirmek için ResetEvent fonksiyonu çağrılmalıdır. Üçüncü parametre event nesnesinin başlangıçta "açık" mı "yoksa "kapalı" mı olacağını belirtmektedir. Son parametre proseslerarası kullanım için gereken isimdir. Aynı prostedeki kullanımda bu parametre de NULL geçilebilir.

2) Olay gerçekleşene kadar bekleme işlemi WaitForXXX fonksiyonlarıyla yapılır.

3) Nesneyi açık duruma geçirmek için SetEvent fonksiyonu kullanılır:

```
BOOL WINAPI SetEvent(  
    HANDLE hEvent  
);
```

Event nesnesi otomatikse WaitForXXX fonksiyonlarından geçildiğinde nesne otomatik olarak yeniden kapalı duruma geçer. Eğer event nesnesi manuele kapalı duruma geçirmek için ResetEvent fonksiyonu çağrılmalıdır:

```
BOOL WINAPI ResetEvent(  
    HANDLE hEvent  
);
```

4) Kullanım bittikten sonra nesne CloseHandle fonksiyonuyla yok edilir:

Örnek bir event kullanım şöyle olabilir:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <Windows.h>  
  
void ExitSys(LPCSTR lpszMsg);  
DWORD __stdcall ThreadProcProducer(void *param);  
DWORD __stdcall ThreadProcConsumer(void *param);  
HANDLE g_hEvent;  
  
int main(void)  
{  
    DWORD dwThreadIdProducer, dwThreadIdConsumer;  
    HANDLE hThreadProducer, hThreadConsumer;  
  
    if ((g_hEvent = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)  
        ExitSys("CreateEvent");  
  
    if ((hThreadProducer = CreateThread(NULL, 0, ThreadProcProducer, NULL, 0, &dwThreadIdProducer)) ==  
        NULL)  
        ExitSys("CreateThread");  
  
    if ((hThreadConsumer = CreateThread(NULL, 0, ThreadProcConsumer, NULL, 0, &dwThreadIdConsumer)) ==  
        NULL)  
        ExitSys("CreateThread");
```

```

    WaitForSingleObject(hThreadProducer, INFINITE);
    WaitForSingleObject(hThreadConsumer, INFINITE);

    CloseHandle(g_hEvent);

    return 0;
}

DWORD __stdcall ThreadProcProducer(void *param)
{
    int i;

    for (i = 0; i < 5; ++i) {
        printf("!");
        fflush(stdout);
        Sleep(500);
    }

    WaitForSingleObject(g_hEvent, INFINITE);

    printf("Ok!\n");

    return 0;
}

DWORD __stdcall ThreadProcConsumer(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        printf(".");
        fflush(stdout);
        Sleep(500);
    }
    SetEvent(g_hEvent);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Üretici tüketici probleminin tekli versiyonu event nesneleriyle de çözülebilir. (Fakat tamponlu versiyonu çözülemez):

```

/* ProducerConsumer.c */

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProcProducer(void *param);
DWORD __stdcall ThreadProcConsumer(void *param);

```

```

HANDLE g_hEventProducer;
HANDLE g_hEventConsumer;
int g_share;

int main(void)
{
    DWORD dwThreadIdProducer, dwThreadIdConsumer;
    HANDLE hThreadProducer, hThreadConsumer;

    srand(time(NULL));

    if ((g_hEventProducer = CreateEvent(NULL, FALSE, TRUE, NULL)) == NULL)
        ExitSys("CreateSemaphore", EXIT_FAILURE);

    if ((g_hEventConsumer = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
        ExitSys("CreateSemaphore", EXIT_FAILURE);

    if ((hThreadProducer = CreateThread(NULL, 0, ThreadProcProducer, NULL, 0, &dwThreadIdProducer)) ==
    NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    if ((hThreadConsumer = CreateThread(NULL, 0, ThreadProcConsumer, NULL, 0, &dwThreadIdConsumer)) ==
    NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    WaitForSingleObject(hThreadProducer, INFINITE);
    WaitForSingleObject(hThreadConsumer, INFINITE);

    CloseHandle(g_hEventProducer);
    CloseHandle(g_hEventConsumer);

    return 0;
}

DWORD __stdcall ThreadProcProducer(void *param)
{
    int i = 0;

    for (;;) {
        Sleep(rand() % 300);

        WaitForSingleObject(g_hEventProducer, INFINITE);
        g_share = i;
        SetEvent(g_hEventConsumer);

        if (i == 99)
            break;
        ++i;
    }

    return 0;
}

DWORD __stdcall ThreadProcConsumer(void *param)
{
    int val;

    for (;;) {
        WaitForSingleObject(g_hEventConsumer, INFINITE);
        val = g_share;
        SetEvent(g_hEventProducer);

        printf("%d ", val);
        Sleep(rand() % 300);
        if (val == 99)

```

```

        break;
    }
    printf("\n");

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Okuma Yazma Kilitleri (Read/Write Locks)

Paylaşılan bir veri yapısına birden fazla akışın okuma ve yazma yapma amacıyla eriştiğini düşünelim. Burada şüphesiz kritik kod oluşturmak gerekir. Ancak veri yapısından birden fazla thread'in okuma yapmasında sakınca yoktur. Ancak bir thread okuma yaparken diğer thread yazma yapmamalıdır. Bir thread yazma yapmak için veri yapısına erişmişse diğer thread'ler ne okuma ne de yazma amaçlı aynı veri yapısına erişmelidir. Bu durumu aşağıdaki tabloyla özetleyebiliriz:

Thread Erişimi	Diğer Thread Erişimi
Okuma Amaçlı	Okuma amaçlıysa Beklememeli, yazma amaçlıysa okuma işlemi bitene kadar beklemeli
Yazma Amaçlı	Hem okuma amaçlı hem de yazma amaçlıysa yazma işlemi bitene kadar beklemeli

İşte bunu sağlayan senkronizasyon nesnesine okuma yazma kilitleri denilmektedir. Örneğin bir bağlı listeye birkaç thread'in eleman eklediğini birkaç thread'in de bu bağlı listede arama işlemi yaptığını düşünelim. Eğer bir thread bağlı listeye eleman ekliyorsa diğer threadler bu bağlı listede arama yapmamalı ve eleman eklemeye çalışmamalıdır. Ta ki eleman ekleyen thread işini bitirince kadar. Benzer biçimde bir thread bağlı listede arama yaparken diğer thread'ler bağlı listeye eleman eklemeye çalışmamalıdır. Taki arama işlemi yapan thread bu işlemi bitirince kadar. Ancak birden fazla thread'in arama işlemi yapmasında bir sakınca yoktur. Bu örnekte eleman ekleme mantıksal olarak "yazma" işlemine arama yapma da mantıksal olarak "okuma" işlemine karşılık gelmektedir. Bu problemi normal senkronizasyon nesneleriyle çözmeye çalışalım:

Okuma İşlemi yapan thread

```

<kilidi al>
<okumayı yap>
<kilidi aç>

```

Yazma İşlemi Yapan Thread

```

<kilidi al>
<yazmayı yap>
<kilidi aç>

```

Burada birden fazla thread okuma yapmak istediğinde bunu aynı anda yapamayacaklardır. İşye okuma yazma kilitleri bu sorunu çözmektedir. Okuma yazma kilitleri hem Windows'ta hem de UNIX/Linux sistemlerinde benzer biçimde bulunmaktadır.

Windows'ta Okuma Yazma Kilitleri

Windows'ta okuma yazma kilitleri şöyle kullanılmaktadır:

1) Global düzeyde SRWLOCK isimli yapı türünden bir nesne tanımlanır. Ve bu nesne InitializeSRWLock fonksiyonuyla ilklendir.

```
VOID WINAPI InitializeSRWLock(  
    PSRWLOCK SRWLock  
);
```

Fonksiyon SRWLOCK türünden yapı nesnesinin adresini alır.

2) Kaynağa okuma amaçlı erişiliyorsa kritik kod şöyle oluşturulmalıdır:

```
    AcquireSRWLockShared(&g_rwLock);  
Kritik Kod {  
    _____  
    _____  
    _____  
    _____  
    ReleaseSRWLockShared(&g_rwLock);
```

3) Kaynağa yazma amaçlı erişiliyorsa kritik kod şöyle oluşturulmalıdır:

```
    AcquireSRWLockExclusive(&g_rwLock);  
Kritik Kod {  
    _____  
    _____  
    _____  
    _____  
    ReleaseSRWLockExclusive(&g_rwLock);
```

Söz konusu fonksiyonların prototipleri şöyledir:

```
VOID WINAPI AcquireSRWLockShared(  
    PSRWLOCK SRWLock  
);
```

```
VOID WINAPI AcquireSRWLockExclusive(  
    PSRWLOCK SRWLock  
);
```

```
VOID WINAPI ReleaseSRWLockShared(  
    PSRWLOCK SRWLock  
);
```

```
VOID WINAPI ReleaseSRWLockExclusive(  
    PSRWLOCK SRWLock  
);
```

Burada görüldüğü gibi bir thread ortak kaynağa okuma amaçlı erişecekse kilidi AcquireSRWLockShared fonksiyonuyla elde etmeye çalışmaktadır. Bu durumda başka bir thread AcquireSRWLockShared işleminde bloke olmaz. Fakat AcquireSRWLockExclusive işleminde bloke olur. Eğer thread kaynağa yazma amacıyla erişecekse bu durumda kilidi AcquireSRWLockExclusive fonksiyonuyla elde etmeye çalışmalıdır. Bu durumda diğer bir thread eğer kilidi AcquireSRWLockShared fonksiyonuyla ya da AcquireSRWLockExclusive fonksiyonuyla elde etmek istediğinde blokede bekler.

Okuma-Yazma kilitlerinin testi için aşağıdaki gibi bir örnek verilebilir:

```

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

void ExitSys(LPCSTR lpszMsg);
DWORD __stdcall ThreadProc1(void *param);
DWORD __stdcall ThreadProc2(void *param);
DWORD __stdcall ThreadProc3(void *param);
DWORD __stdcall ThreadProc4(void *param);

SRWLOCK g_srwLock;

int main(void)
{
    DWORD dwThreadId1, dwThreadId2, dwThreadId3, dwThreadId4;
    HANDLE hThread1, hThread2, hThread3, hThread4;

    srand(time(NULL));

    InitializeSRWLock(&g_srwLock);

    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)
        ExitSys("CreateThread");

    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)
        ExitSys("CreateThread");

    if ((hThread3 = CreateThread(NULL, 0, ThreadProc3, NULL, 0, &dwThreadId3)) == NULL)
        ExitSys("CreateThread");

    if ((hThread4 = CreateThread(NULL, 0, ThreadProc4, NULL, 0, &dwThreadId4)) == NULL)
        ExitSys("CreateThread");

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);
    WaitForSingleObject(hThread3, INFINITE);
    WaitForSingleObject(hThread4, INFINITE);

    return 0;
}

DWORD __stdcall ThreadProc1(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        Sleep(rand() % 100);
        AcquireSRWLockShared(&g_srwLock);
        printf("Thread-1 enters for READING...\n");
        Sleep(rand() % 100);
        printf("Thread-1 exits from READING...\n");
        ReleaseSRWLockShared(&g_srwLock);
    }

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        Sleep(rand() % 100);
        AcquireSRWLockExclusive(&g_srwLock);
    }
}

```



```

        printf("Thread-2 enters for WRITING...\n");
        Sleep(rand() % 100);
        printf("Thread-2 exits from WRITING...\n");
        ReleaseSRWLockExclusive(&g_srwLock);
    }

    return 0;
}

DWORD __stdcall ThreadProc3(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        Sleep(rand() % 100);
        AcquireSRWLockShared(&g_srwLock);
        printf("Thread-3 enters for READING...\n");
        Sleep(rand() % 100);
        printf("Thread-3 exits from READING...\n");
        ReleaseSRWLockShared(&g_srwLock);
    }

    return 0;
}

DWORD __stdcall ThreadProc4(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        Sleep(rand() % 100);
        AcquireSRWLockExclusive(&g_srwLock);
        printf("Thread-4 enters for WRITING...\n");
        Sleep(rand() % 100);
        printf("Thread-4 exits from WRITING...\n");
        ReleaseSRWLockExclusive(&g_srwLock);
    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

Bu örnekte birden fazla okumanın birlikte yapılabildiği ancak okuma ile yazmanın ve yazma ile yazmanın birlikte yapılamadığı görülebilir. Yukarıdaki programın ekran çıktısının bir bölümü aşağıdaki gibi elde edilmiştir:

```

...
Thread-1 enters for READING...
Thread-3 enters for READING...
Thread-3 exits from READING...
Thread-1 exits from READING...
Thread-2 enters for WRITING...
Thread-2 exits from WRITING...
Thread-4 enters for WRITING...

```

```

Thread-4 exits from WRITING...
Thread-3 enters for READING...
Thread-1 enters for READING...
Thread-1 exits from READING...
Thread-3 exits from READING...
Thread-2 enters for WRITING...
Thread-2 exits from WRITING...
Thread-1 enters for READING...
Thread-1 exits from READING...
Thread-4 enters for WRITING...
Thread-4 exits from WRITING...
Thread-3 enters for READING...
Thread-3 exits from READING...
Thread-2 enters for WRITING...
Thread-2 exits from WRITING...
Thread-4 enters for WRITING...
...

```

UNIX/Linux Sistemlerinde Okuma Yazma Kilitleri

UNIX/Linux sistemlerinde okuma yazma kilitleri Windows sistemlerindeki gibi benzerdir. Bu sistemlerde de okuma yazma kilitleri şöyle kullanılmaktadır:

1) `pthread_rwlock_t` türünden global bir nesne tanımlanır ve bu nesneye `pthread_rwlock_init` fonksiyonuyla ilkdeğer verilir.

```
#include <pthread.h>
```

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
```

Fonksiyonun ikinci parametre nesnenin diğer özelliklerini belirtir. Bu parametre NULL geçilirse nesne default özelliklerle yaratılır.

2) Kaynağa okuma amaçlı erişiliyorsa kritik kod şöyle oluşturulmalıdır:

```

pthread_rwlock_rdlock(&g_rwlock);
Kritik Kod [ ≡≡≡
pthread_rwlock_unlock(&g_rwlock);

```

Kaynağa yazma amaçlı erişiliyorsa kritik kod şöyle oluşturulmalıdır:

```

pthread_rwlock_wrlock(&g_rwlock);
Kritik Kod [ ≡≡≡
pthread_rwlock_unlock(&g_rwlock);

```

Fonksiyonların prototipleri şöyledir:

```
#include <pthread.h>
```

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Fonksiyonlar başarı durumunda 0, başarısızlık durumunda hata koduna geri dönmektedir.

3) İşlemler bitince rwlock nesnesi pthread_rwlock_destroy fonksiyonuyla yok edilir:

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Görüldüğü gibi UNIX/Linux sistemlerindeki okuma-yazma kilitlerinin kullanımı Windows sistemlerine çok benzemektedir. Ancak UNIX/Linux sistemlerinde kilidi serbest bırakmak için ayrı fonksiyonlar yoktur. Kilit ister okuma amaçlı alınmış olsun, ister yazma amaçlı alınmış olsun her iki durumda da pthread_rwlock_unlock fonksiyonuyla serbest bırakılır.

Okuma-Yazma kilitlerinin Unix/Linux sistemlerindeki testi için aşağıdaki gibi bir örnek verilebilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_proc1(void *param);
void *thread_proc2(void *param);
void *thread_proc3(void *param);
void *thread_proc4(void *param);

pthread_rwlock_t g_rwlock;

int main(void)
{
    pthread_t tid1, tid2, tid3, tid4;
    int result;

    srand((unsigned int)time(NULL));

    if ((result = pthread_rwlock_init(&g_rwlock, NULL)) != 0) {
        fprintf(stderr, "pthread_rwlock_init: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid3, NULL, thread_proc3, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid4, NULL, thread_proc4, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);
    pthread_join(tid4, NULL);

    pthread_rwlock_destroy(&g_rwlock);

    return 0;
}
```

```

}

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        usleep(rand() % 100);
        pthread_rwlock_rdlock(&g_rwlock);
        printf("Thread1 enters for reading\n");
        usleep(rand() % 100);
        printf("Thread1 exits reading\n");
        pthread_rwlock_unlock(&g_rwlock);
    }

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        usleep(rand() % 100);
        pthread_rwlock_wrlock(&g_rwlock);
        printf("Thread2 enters for writing\n");
        usleep(rand() % 100);
        printf("Thread2 exits writing\n");
        pthread_rwlock_unlock(&g_rwlock);
    }

    return NULL;
}

void *thread_proc3(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        usleep(rand() % 100);
        pthread_rwlock_rdlock(&g_rwlock);
        printf("Thread3 enters for reading\n");
        usleep(rand() % 100);
        printf("Thread3 exits reading\n");
        pthread_rwlock_unlock(&g_rwlock);
    }

    return NULL;
}

void *thread_proc4(void *param)
{
    int i;

    for (i = 0; i < 10; ++i) {
        usleep(rand() % 100);
        pthread_rwlock_wrlock(&g_rwlock);
        printf("Thread4 enters for writing\n");
        usleep(rand() % 100);
        printf("Thread4 exits writing\n");
        pthread_rwlock_unlock(&g_rwlock);
    }
}

```

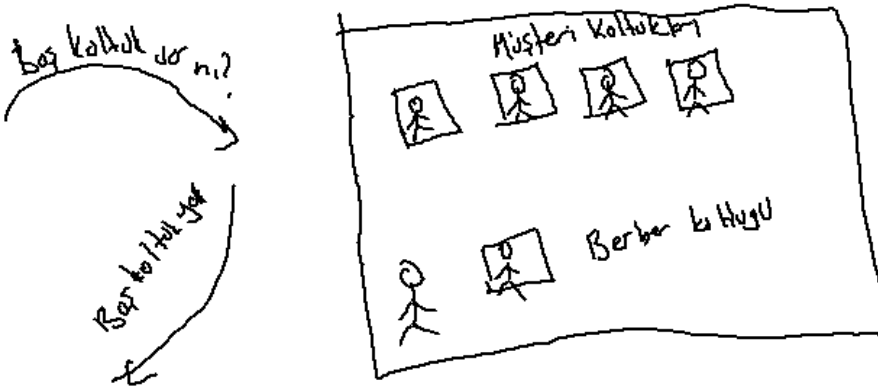
```
    return NULL;
}
```

Literatürde Çok Geçen Fakat Uygulamada Çok Önemli Olmayan Birkaç Senkronizasyon Problemi

Üretici-Tüketici problemi uygulamada en fazla karşılaşılan senkronizasyon problemidir. Ancak literatürde geçen birkaç senkronizasyon problemi daha vardır. Bunların önemli bir uygulama alanı olduğu söylenemez. Burada bunlardan kısaca bahsedeceğiz.

Uyuyan Berber Problemi (Sleeping Barber Problem)

Bu problemde bir berber dükkanı vardır. Dükkanda belirli sayıda bekleme koltuğu ve tabii bir de berber koltuğu bulunmaktadır. Berber hiç müşteri yoksa uyur. (Yani bloke olur. CPU zamanı harcamaz.) Müşteri içeri girmek istediğinde koltuklara bakar. Koltuklar doluysa beklemez. Bekleme koltukları boşsa orada bekler. Beklerken de uyur (bloke olur). Berber koltuğundaki kişinin tıraşı bittiğinde bekleyen kişilerden sıradaki uyandırılarak koltuğa oturtulur.



Problemin çözümünde bekleme koltukları için bir semafor kullanılır. Bu semaforun başlangıçtaki sayacı sıfırdır. Berber bu semafora bakarak işlem yapar. Demek ki başlangıçta berber hiç müşteri olmadığı için uyuyacaktır. Bir müşteri geldiğinde semafor sayacı 1 artar. Berber böylece uyanır. Onu koltuğa alır. Semafor sayacını bir eksiltir. Dışarıdan gelen müşterinin koltukların dolu olup olmadığını anlaması gerekmektedir. Bunun için bir sayaç tutulur. Tabii sayaç da kritik kod bloğu ile azaltılıp yükseltilir. Pekiyi müşterinin koltukta uyuması ve berberin de müşteriyi uyandırması nasıl olacaktır? İşte bunun için de ayrı bir senkronizasyon nesnesine ihtiyaç vardır. Binary bir semafor işi görebilir. Mutex olmaz. Çünkü başka bir thread mutex'in kilidini açamaz. (Bu amaçla Windows'taki Event nesnelere de kullanılabilir. Benzer biçimde UNIX/Linux sistemlerindeki "koşullu değişkenler (conditional variables)" de bu amaçla kullanılabilirler.)

Uyuyan Berber Probleminin Çözümü Windows sistemlerinde şöyle yapılabilir:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <Windows.h>

#define NUMBER_OF_SEATS          3
#define NUMBER_OF_CUSTOMERS     20

void ExitSys(LPCSTR lpszMsg, int status);
DWORD __stdcall ThreadProcBarber(void *param);
DWORD __stdcall ThreadProcCustomer(void *param);
HANDLE CreateBarber(const char *name);
HANDLE CreateCustomer(const char *name);

HANDLE g_hSemaphoreBarber;
HANDLE g_hMutexSeat;
HANDLE g_hEventCustomer;
HANDLE g_hEventShaving;
int g_emptySeatCount;
```

```

int main(void)
{
    HANDLE allHandles[10];
    char *name;
    int i;

    g_emptySeatCount = NUMBER_OF_SEATS;
    if ((g_hSemaphoreBarber = CreateSemaphore(NULL, 0, NUMBER_OF_SEATS, NULL)) == NULL)
        ExitSys("CreateSemaphore", EXIT_FAILURE);

    if ((g_hMutexSeat = CreateMutex(NULL, FALSE, NULL)) == NULL)
        ExitSys("CreateMutex", EXIT_FAILURE);

    if ((g_hEventCustomer = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
        ExitSys("CreateSemaphore", EXIT_FAILURE);

    if ((g_hEventShaving = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
        ExitSys("CreateSemaphore", EXIT_FAILURE);

    allHandles[0] = CreateBarber("Barber");
    Sleep(500);
    for (i = 1; i <= NUMBER_OF_CUSTOMERS; ++i) {
        if ((name = (char *)malloc(10)) == NULL) {
            fprintf(stderr, "cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
        sprintf(name, "Customer-%d", i);
        allHandles[i] = CreateCustomer(name);
        Sleep(rand() % 300);
    }

    WaitForMultipleObjects(NUMBER_OF_CUSTOMERS, allHandles, TRUE, INFINITE);

    return 0;
}

HANDLE CreateBarber(const char *name)
{
    HANDLE hThreadBarber;
    DWORD dwThreadIdBarber;

    if ((hThreadBarber = CreateThread(NULL, 0, ThreadProcBarber, (void *)name, 0, &dwThreadIdBarber))
    == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    return hThreadBarber;
}

HANDLE CreateCustomer(const char *name)
{
    HANDLE hThreadCustomer;
    DWORD dwThreadIdCustomer;

    if ((hThreadCustomer = CreateThread(NULL, 0, ThreadProcCustomer, (void *)name, 0,
    &dwThreadIdCustomer)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    return hThreadCustomer;
}

DWORD __stdcall ThreadProcBarber(void *param)
{
    const char *name = (const char *)param;

    for (;;) {

```

```

Sleep(200);
printf("(s) Berber uyuyor\n", name);
WaitForSingleObject(g_hSemaphoreBarber, INFINITE);

printf("(s) Berber uyandı müşteriyi koltuga davet etti\n", name);
SetEvent(g_hEventCustomer);

printf("Berber müşteriyi tiras ediyor\n");
Sleep(rand() % 300);

printf("Berber tirsasi bitirdi\n");
SetEvent(g_hEventShaving);

WaitForSingleObject(g_hMutexSeat, INFINITE);
++g_emptySeatCount;
ReleaseMutex(g_hMutexSeat);
}

return 0;
}

DWORD __stdcall ThreadProcCustomer(void *param)
{
    const char *name = (const char *)param;

    printf("(s) müşteri geldi ve dukkana bakıyor\n", name);
    Sleep(rand() % 300);

    WaitForSingleObject(g_hMutexSeat, INFINITE);

    if (g_emptySeatCount > 0) {
        printf("(s) bos koltuga oturdu ve uyudu\n", name);
        --g_emptySeatCount;
        ReleaseMutex(g_hMutexSeat);

        ReleaseSemaphore(g_hSemaphoreBarber, 1, NULL);

        WaitForSingleObject(g_hEventCustomer, INFINITE);
        printf("(s) müşteri uyandı berber koltuguna oturdu\n", name);
        WaitForSingleObject(g_hEventShaving, INFINITE);
        printf("(s) müşteri tirsasini oldu, dukkandan cikiyor\n", name);
    }
    else {
        printf("(s) müşteri dukkana giremedn ayriliyor\n", name);
        ReleaseMutex(g_hMutexSeat);
    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

```

Burada birden fazla müşteri koltukta bekliyorsa onlardan bir tanesinin uyandırılması Event nesneleriyle yapılmıştır. Bu tür nesnelerin hepsi (hem Windows'ta hem de UNIX/Linux sistemlerinde) bekleyen thread'lerden hangisinin

uyandırılacağı konusunda bir garanti vermemektedir. Bu garantini verilmesi için bir kuyruk sistemi kullanılabilir. Kuyruğun içerisinde koltukta bekleyen müşterilere ilişkin event nesnelere olur.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <Windows.h>

#define NUMBER_OF_SEATS          3
#define NUMBER_OF_CUSTOMERS     20

void ExitSys(LPCSTR lpszMsg, int status);
DWORD __stdcall ThreadProcBarber(void *param);
DWORD __stdcall ThreadProcCustomer(void *param);
HANDLE CreateBarber(const char *name);
HANDLE CreateCustomer(const char *name);

HANDLE g_hSemaphoreBarber;
HANDLE g_hMutexSeat;
HANDLE g_hEventCustomer;
HANDLE g_hEventShaving;
int g_emptySeatCount;

int main(void)
{
    HANDLE allHandles[10];
    char *name;
    int i;

    g_emptySeatCount = NUMBER_OF_SEATS;
    if ((g_hSemaphoreBarber = CreateSemaphore(NULL, 0, NUMBER_OF_SEATS, NULL)) == NULL)
        ExitSys("CreateSemaphore", EXIT_FAILURE);

    if ((g_hMutexSeat = CreateMutex(NULL, FALSE, NULL)) == NULL)
        ExitSys("CreateMutex", EXIT_FAILURE);

    if ((g_hEventCustomer = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
        ExitSys("CreateSemaphore", EXIT_FAILURE);

    if ((g_hEventShaving = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
        ExitSys("CreateSemaphore", EXIT_FAILURE);

    allHandles[0] = CreateBarber("Berber");
    Sleep(500);
    for (i = 1; i <= NUMBER_OF_CUSTOMERS; ++i) {
        if ((name = (char *)malloc(10)) == NULL) {
            fprintf(stderr, "cannot allocate memory!..\n");
            exit(EXIT_FAILURE);
        }
        sprintf(name, "Musteri-%d", i);
        allHandles[i] = CreateCustomer(name);
        Sleep(rand() % 300);
    }

    WaitForMultipleObjects(NUMBER_OF_CUSTOMERS, allHandles, TRUE, INFINITE);

    return 0;
}

HANDLE CreateBarber(const char *name)
{
    HANDLE hThreadBarber;
    DWORD dwThreadIdBarber;

    if ((hThreadBarber = CreateThread(NULL, 0, ThreadProcBarber, (void *)name, 0, &dwThreadIdBarber))
    == NULL)

```



```

        ExitSys("CreateThread", EXIT_FAILURE);

    return hThreadBarber;
}

HANDLE CreateCustomer(const char *name)
{
    HANDLE hThreadCustomer;
    DWORD dwThreadIdCustomer;

    if ((hThreadCustomer = CreateThread(NULL, 0, ThreadProcCustomer, (void *)name, 0,
&dwThreadIdCustomer)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    return hThreadCustomer;
}

DWORD __stdcall ThreadProcBarber(void *param)
{
    const char *name = (const char *)param;

    for (;;) {
        Sleep(200);
        printf("(%s) Berber uyuyor\n", name);
        WaitForSingleObject(g_hSemaphoreBarber, INFINITE);

        printf("(%s) Berber uyandı musteriye koltuga davet etti\n", name);
        SetEvent(g_hEventCustomer);

        Sleep(rand() % 300);

        printf("Berber tırasi bitirdi\n");
        SetEvent(g_hEventShaving);

        WaitForSingleObject(g_hMutexSeat, INFINITE);
        ++g_emptySeatCount;
        ReleaseMutex(g_hMutexSeat);
    }

    return 0;
}

DWORD __stdcall ThreadProcCustomer(void *param)
{
    const char *name = (const char *)param;

    printf("(%s) müşteri geldi ve dukkana bakıyor\n", name);
    Sleep(rand() % 300);

    WaitForSingleObject(g_hMutexSeat, INFINITE);

    if (g_emptySeatCount > 0) {
        printf("(%s) bos koltuga oturdu ve uyudu\n", name);
        --g_emptySeatCount;
        ReleaseMutex(g_hMutexSeat);

        ReleaseSemaphore(g_hSemaphoreBarber, 1, NULL);

        WaitForSingleObject(g_hEventCustomer, INFINITE);
        printf("(%s) müşteri uyandı berber koltuguna oturdu ve tıraş oluyor...\n", name);
        WaitForSingleObject(g_hEventShaving, INFINITE);
        printf("(%s) müşteri tırasini oldu, dukkandan cikiyor\n", name);
    }
    else {
        printf("(%s) müşteri dukkana giremedn ayriliyor\n", name);
        ReleaseMutex(g_hMutexSeat);
    }
}

```

```

    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

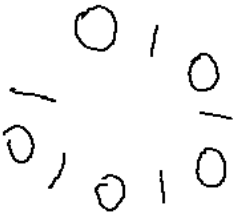
    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

```

Yemek Yiyen Filozoflar Problemi (Dining Philosophers)

Bu problemde bir grup kişi yuvarlak yemek masasında oturmaktadır. Onların sol ve sağında birer çatal vardır. Fakat yemeği yiyebilmek için iki çatala da ihtiyaç duyarlar. Yemek yeme süreci bir döngü içerisinde iki çatalı alıp biraz yiyip bırakmak biçiminde olmaktadır.



Problem 1965 yılında Edsger Dijkstra tarafından tanımlanmıştır. Burada amaç kişilerin aç kalmasını (starvation) ve kilitlenmeyi (deadlock) engellemektir. Kilitlenme bir thread diğerini beklerken diğerinin de başka bir şeyi beklemesi ve böylece sonsuz bir beklemenin oluşması durumudur. Örneğin her kişi önce solundaki ya da sağındaki çatalı alırsa bir kilitlenme oluşur. Her kişi iki çatalı birden almaya çalışırsa da alamayabilir. Yine kilitlenme oluşma riski vardır. Ayrıca bazı kişilere iki çatalı alma sırası hiç gelemeyebilir. Yani bazılarının aç kalma riski de vardır.

Problemin çözümü burada yapılmayacaktır. Ancak klasik çözümde kişiler iki senkronizasyon nesnesiyle (örneğin semaphore) önce soldaki, sonra sağdaki çataları almaya çalışırlar. (Çözüm için Tanenbaum'un "Operating System Design And Implementation" kitabına bakılabilir.)

Sigara İçen Kişiler Problemi (Cigarette Smokers Problem)

Bu problemde sigara içmek isteyen üç kişi vardır. Bunların sigara içmesi için tütün, kağıt ve kibriti elde etmesi gerekir. Problemde bir de yürütücü (agent) vardır. Yürütücü rastgele iki malzemeyi masaya koyar. İhtiyacı olanlar alıp sigara içmeye çalışır. Ancak sigara içmek isteyen kişilerde bu malzemelerden yalnızca biri zaten sınırsız ölçüde vardır. Yani birinde tütün, birinde kağıt, birinde de kibrit sınırsız ölçüde zaten bulunmaktadır. O halde bu kişiler diğer iki malzemeyi tedarik etmeye çalışırlar. Örneğin, yürütücü masaya kibrit ve kağıt koysun. Bunun ikisini de tütüne sahip kişi alırsa sigarayı içer. Fakat farklı kişiler alırsa yürütücünün diğer iki malzemeyi bırakmasını bekler. Problemde hedef en az bekleme oluşturacak biçimde tiryakilerin maksimum ölçüde sigara içmesini sağlamaktır.

Çözüm için yine her malzemeyi elde bulunduranlar diğer ikisi için iki ayrı semaphore ile beklerler. Ancak çözüm burada verilmeyecektir.

Atomik İşlemler

Atomiklik (atomicity) bir işlemin (bir grup makine kodunun) hiç kesilme olmadan (yani thread'ler arası geçiş oluşmadan) yürütülmesi anlamına gelmektedir. Yani atomik bir işlem sırasında thread'ler arası geçiş olmaz, işlem kesiksiz bir biçimde yürütülür. Dolayısıyla atomik işlemler aynı zamanda "thread güvenli (thread safe)" işlemlerdir. Atomik işlemlerin ayrıca senkronize edilmeleri gerekmez.

Normal olarak makine komutları atomiktir. Yani bir makine komutu çalışırken onun ortasında kesilme (thread'ler arası geçiş olmaz.) Thread'ler arası geçiş (context switch) ancak bir makine komutunun çalışması bittiğinde oluşabilmektedir. Ancak daha önceden de belirtildiği gibi thread'ler arası geçiş (context switch) herhangi iki makine komutu arasında gerçekleşebilmektedir.

Biz C'de bir işlemi tek bir ifadeyle yapıyor olmamız derleyicinin onu tek bir makine komutuyla yapacağı dolayısıyla bunun atomik olacağı anlamına gelmez. Örneğin:

```
++g_val;
```

Bizim bunu tek bir operatörle yapmış olmamız derleyicinin bunu tek bir makine komutuyla yapacağı anlamına gelmez. Örneğin Intel'de bu işlem tek bir komutuyla yapılabilir:

```
INC g_val
```

Ya da bu işlem birkaç makine komutuyla da yapılabilir:

```
MOV EAX, g_val
INC EAX
MOV g_val, EAX
```

Örneğin aynı global değişkeni iki thread de belli miktar artıracak olsun. Acaba sonuçta global değişkenin değışkenin değeri olması gereken değerde olacak mıdır?

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

#define N 1000000

void ExitSys(LPCSTR lpszMsg, int status);
DWORD __stdcall ThreadProc1(void *param);
DWORD __stdcall ThreadProc2(void *param);

int g_val;

int main(void)
{
    DWORD dwThreadId1, dwThreadId2;
    HANDLE hThread1, hThread2;

    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    printf("%d\n", g_val);

    return 0;
}
```

```

DWORD __stdcall ThreadProc1(void *param)
{
    int i;

    for (i = 0; i < N; ++i)
        ++g_val;

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    int i;

    for (i = 0; i < N; ++i)
        ++g_val;

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

```

Yukarıdaki programda sayacın değerinin 2000000 olması gerekir. Halbuki muhtemelen böyle çıkmayacaktır. Bu tür basit işlemlerin bile kritik kod içerisinde yapılması gerekir. Örneğin:

```

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

#define N        1000000

void ExitSys(LPCSTR lpszMsg, int status);
DWORD __stdcall ThreadProc1(void *param);
DWORD __stdcall ThreadProc2(void *param);

int g_val;

CRITICAL_SECTION g_cs;

int main(void)
{
    DWORD dwThreadId1, dwThreadId2;
    HANDLE hThread1, hThread2;

    InitializeCriticalSection(&g_cs);

    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)
        ExitSys("CreateThread", EXIT_FAILURE);

    WaitForSingleObject(hThread1, INFINITE);
}

```

```

    WaitForSingleObject(hThread2, INFINITE);

    printf("%d\n", g_val);

    DeleteCriticalSection(&g_cs);

    return 0;
}

DWORD __stdcall ThreadProc1(void *param)
{
    int i;

    for (i = 0; i < N; ++i) {
        EnterCriticalSection(&g_cs);
        ++g_val;
        LeaveCriticalSection(&g_cs);
    }

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    int i;

    for (i = 0; i < N; ++i) {
        EnterCriticalSection(&g_cs);
        ++g_val;
        LeaveCriticalSection(&g_cs);
    }

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

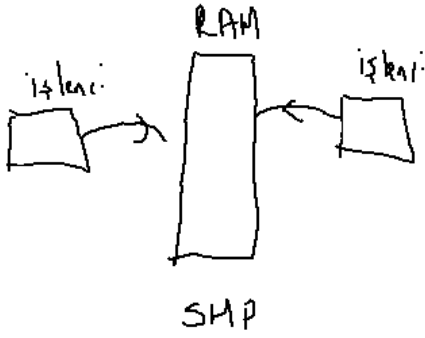
```

Tabii bu tür basit işlemler için bile kritik kod oluşturulması biraz sıkıcı olabilmektedir.

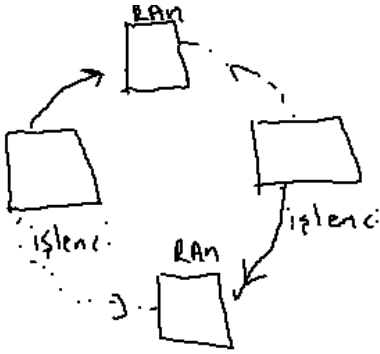
Ayrıca çok işlemcili ya da çok çekirdekli sistemlerde tek bir makine komutunun atomikliği de tartışılır hale gelebilmektedir. Örneğin iki çekirdek de aşağıdaki makine komutunu tesadüfen aynı anda çalıştırmış olsun:

```
INC    g_val
```

Burada bazı sistemlerde yine bozulma olabilir. Bugün çok işlemcili ya da çok çekirdekli sistemlerde iki mimari kullanılmaktadır: SMP ve NUMA. SMP (Symmetric MultiProcessor) mimarisinde işlemciler ya da çekirdekler aynı RAM'e bağlıdır. Dolayısıyla bus çatışması olmaması için biri RAM'e erişirken diğeri durdurur. (Yani bu süreç yüzünden iki çekirdekli bir sistem iki kat hızlı değildir. Olsa olsa %20-25 daha hızlıdır. Dört çekirdekli sistem de dört kat hızlı değildir.)

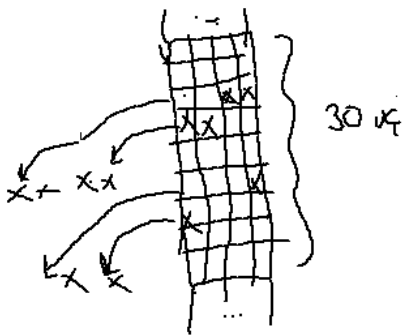


NUMA'da (Non Unified Memory Access) ise her işlemcinin ya da çekirdeğin RAM bloğu (bank) ayrıdır. Fakat bunlar birbirlerinin RAM'lerine (bank'lerine) de erişebilmektedir. Tabi bu durumda erişim daha yavaş olur (Zaten Non Unified Memory Access ismi buradan gelmektedir.) Böylece bu mimaride işlemcilerin ya da çekirdeklerin RAM'e erişirken birbirlerini durdurmasıyla daha az karşılaşılmaktadır.



Bizim kullandığımız ana kartların çok büyük çoğunluğu SMP mimarisine sahiptir. (Ancak örneğin AMD'nin Opteron işlemcilerinin kullanıldığı ana kartlar NUMA mimarisine sahiptir.) Her iki mimarinin de bazı avantajları ve dezavantajları vardır. NUMA mimarisi daha çok server sistemlerinde tercih edilmektedir.

Bazı işlemciler belleğe erişen makine komutlarında bus'ı başından sonuna kadar tutmazlar. Biraz tutup bırakıp, sonra yeniden tutup bırakabilirler. İşte durumda diğer işlemci ya da çekirdek aynı yere erişirse bozuk bir değer oluşabilmektedir. Örneğin Intel işlemcilerinde bazı makine komutlarında bu durum vardır. Ayrıca hizalanmamış bilgilere erişen makine komutlarının çoğunda da bu problem söz konusudur.



Yani Intel işlemcileri için şunları söyleyebiliriz:

- 1) Bazı makine komutları farklı işlemciler ya da çekirdekler tarafından aynı anda aynı bellek bölgesine eriştiğinde okuma yazma sırasında bozulmalara yol açabilmektedir.
- 2) Pek çok makine komutu aynı adrese eriştiğinde eğer adres hizalanmamışsa okuma yazma sırasında bozulmalara yol açabilmektedir.

Fakat Intel işlemcilerinde (diğer işlemcilerde de benzer mekanizma vardır) bir komutun başına LOCK prefix'i getirilirse o komut çok işlemcili sistemlerde de hiç bus'ı bırakmadan sorunsuz çalışmaktadır. Yani başına LOCK getirilmiş makine komutları başka işlemcileri ya da çekirdekleri durdurarak belleğe yalnızca ilgili işlemci ya da çekirdeğin erişmesini sağlamaktadır. Böylece sistem programcısı aynı bellek bölgesine erişirken orada bir bozulmanın oluşmaması için bu tür makine komutlarının başına LOCK prefix'i getirir. Örneğin:

```
LOCK ADD [EBX], EAX
```

Başka işlemcilerde de bu durum göz önüne alınmalıdır. Özetle tek bir makine komutu bile çok işlemcili ya da çok çekirdekli sistemlerde aynı bellek bölgesine erişilirken soruna yol açabilmektedir. Bu konunun o sistem için araştırılması gerekir.

Tek işlemcili ya da tek çekirdekli sistemlerde her zaman makine komutları atomiktir. Çünkü zaten başka bir akış aynı yere erişemez. Dolayısıyla bu sistemlerde bu amaçla LOCK prefix'inin kullanılması gerekmez.

İşte tek makina komutuyla yapılabilecek artırma, eksiltme gibi basit işlem için kritik kod oluşturmaya gerek kalmasın diye Microsoft bir grup InterlockedXXX isimli API fonksiyonu buldurmuştur. Bu fonksiyonlar işlemlerini tek bir makine komutuyla yaparlar üstelik de komutun başına LOCK prefix'i getirirler. Böylece kritik kod oluşturmanın maliyetinden kaçınılmış olur. IntelockedIncrement lock'lu tek bir makina komutuyla bir nesneyi artırmaktadır:

```
LONG __cdecl InterlockedIncrement(  
    LONG volatile *Addend  
);
```

InterlockedAdd ise atomik toplama yapmaktadır:

```
LONG __cdecl InterlockedAdd(  
    LONG volatile *Addend,  
    LONG Value  
);
```

InterlockedExchange ise atomik atama yapmaktadır:

```
LONG InterlockedExchange(  
    LONG volatile *Target,  
    LONG Value  
);
```

InterlockedIncrement API fonksiyonu long bir nesnenin adresini alıp onu nesneyi atomik bir biçimde (yani tek bir makine komutuyla lock yaparak) bir artırır. InterlockedAdd ise nesneyi belli bir değer kadar artırmaktadır. InterlockedExchange ise nesneye belli bir değeri atamak için kullanılır. Bu fonksiyonlar nesnede bulunan önceki değerle geri dönerler.

Windows'ta InterlockedXXX fonksiyonlarının sayısı bir hayli fazladır. Bunların tam listesini MSDN dokümanlarından inceleyebilirsiniz.

Şimdi hiç kritik kod oluşturmadan yukarıdaki global değişkenin artırılması örneğini InterlockedIncrement ile yapalım:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <Windows.h>  
  
#define N      1000000  
  
void ExitSys(LPCSTR lpszMsg);  
DWORD __stdcall ThreadProc1(void *param);  
DWORD __stdcall ThreadProc2(void *param);  
  
int g_val;
```

```

int main(void)
{
    DWORD dwThreadId1, dwThreadId2;
    HANDLE hThread1, hThread2;

    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)
        ExitSys("CreateThread");

    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)
        ExitSys("CreateThread");

    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    printf("%d\n", g_val);

    return 0;
}

DWORD __stdcall ThreadProc1(void *param)
{
    int i;

    for (i = 0; i < N; ++i)
        InterlockedIncrement(&g_val);

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    int i;

    for (i = 0; i < N; ++i)
        InterlockedIncrement(&g_val);

    return 0;
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

İçsel fonksiyonlar (intrinsic functions) derleyicinin kendisinin derleme sırasında hiç kütüphaneye başvurmadan tanıdığı ve ele aldığı fonksiyonlardır. Bunlar birer makro gibi koda dahil edilebildikleri gibi normal birer fonksiyon gibi de koda dahil edilebilmektedir. Derleyicilerin içsel fonksiyonları için herhangi bir başlık dosyasının include edilmesine de gerek yoktur. İşte Microsoft derleyicilerinde yukarıda açıkladığımız InterLockedXXX API fonksiyonlarının _InterlockedXXX biçiminde içsel (intrinsic) biçimleri de vardır. Örneğin aşağıdaki fonksiyonlar Microsoft C ve C++ derleyicileri tarafından derleme aşamasında tanınıp işleme sokulabilmektedir:

```

long _InterlockedIncrement(
    long volatile * lpAddend
);

```



```

long _InterlockedExchange(
    long volatile * Target,
    long Value
);

```

Genel olarak (ancak her zaman değil) aynı işleve sahip olan kütüphane fonksiyonu ile içsel fonksiyonlar söz konusu olduğunda içsel fonksiyonların daha hızlı çalıştığı söylenebilir.

UNIX/Linux sistemlerinde Microsoft'un InterlockedXXX fonsiyonlarının benzerleri gcc derleyicilerinde içsel biçimde bulundurulmuştur. gcc'nin içsel atomik fonksiyonlarından önemli olanları şunlardır:

```

type __sync_fetch_and_add (type *ptr, type value, ...)
type __sync_fetch_and_sub (type *ptr, type value, ...)
type __sync_fetch_and_or (type *ptr, type value, ...)
type __sync_fetch_and_and (type *ptr, type value, ...)
type __sync_fetch_and_xor (type *ptr, type value, ...)
type __sync_fetch_and_nand (type *ptr, type value, ...)

```

Bu fonksiyonlar ilgili işlemi yaparlar. Ancak nesnenin işlem görmeden önceki değerini geri dönüş değeri olarak verirler. Aşağıdaki fonksiyonlar ise işlemi yapmakla birlikte bize nesnenin yeni değerini vermektedir:

```

type __sync_add_and_fetch (type *ptr, type value, ...)
type __sync_sub_and_fetch (type *ptr, type value, ...)
type __sync_or_and_fetch (type *ptr, type value, ...)
type __sync_and_and_fetch (type *ptr, type value, ...)
type __sync_xor_and_fetch (type *ptr, type value, ...)
type __sync_nand_and_fetch (type *ptr, type value, ...)

```

Bu fonksiyonlar gcc'nin Windows versiyonu olan MinGW'de de kullanılabilir. Diğer atomik içsel gcc fonksiyonları için gcc'nin referans kitaplarına başvurulabilir ("Built-in functions for memory access" başlığı).

Hiç kritik kod oluşturmadan içsel fonksiyon kullanılarak global bir değişkenin adresi Unix/Linux sistemlerinde aşağıdaki gibi artırılabilir:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

void *thread_proc(void *param);

int g_val;

int main(void)
{
    pthread_t tid1, tid2;
    int result;

    if ((result = pthread_create(&tid1, NULL, thread_proc, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid2, NULL, thread_proc, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    pthread_join(tid1, NULL);

```

```

pthread_join(tid2, NULL);

printf("g_val=%d\n", g_val);

return 0;
}

void *thread_proc(void *param)
{
    int i;

    for (i = 0; i < 10000000; ++i) {
        __sync_fetch_and_add(&g_val, 1);
    }

    return NULL;
}

```

Basit atomik işlemlerin kolay yapılabilmesi için zamanla dillere eklentiler de yapılmıştır. Örneğin C'nin 2011 yılındaki yeni revizyonunda (ISO/IEC 9899:2011 ya da C11) dile `_Atomic` isminde isteğe bağlı (optional) bir tür belirleyicisi (type specifier) ve tür niteleyicisi (type qualifier) eklenmiştir. C11 derleyicilerinin atomik işlemleri destekleyip desteklemediği built-in `__STDC_NO_ATOMICS__` makrosunun define edilmiş olup olmadığına bağlıdır. Yani programcı bu makronun define edilip edilmediğine bakarak derleyicisinin atomik işlemleri destekleyip desteklemediğini anlayabilir. Ayrıca `<stdatomic.h>` dosyası içerisinde de `_Atomic <tür>` biçimindeki türler `atomic_<tür>` biçiminde typedef de edilmiştir. C11'deki `_Atomic` anahtar sözcüğü yukarıda da belirtildiği gibi hem bir tür belirleyicisi (type specifier) hem de bir tür niteleyicisi (type qualifier) görevindedir. Örneğin aşağıdaki bildirimler geçerlidir:

```

_Atomic int x;
_Atomic double y;
int a, _Atomic b;

```

Benzer biçimde eğer `<stdatomic.h>` başlık dosyası include edilmişse

```
atomic_int a;
```

bildirimi ile:

```
_Atomic int a;
```

bildirimi eşdeğerdir. Microsoft derleyicileri bu yeni `_Atomic` türünü desteklememektedir. Ancak gcc ve clang derleyicileri `-std=c11` ya da `-std=c18` seçenekleriyle bu türü desteklemektedir. Örneğin:

```

/* sample.c */

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

void *thread_proc1(void *param);
void *thread_proc2(void *param);

_Atomic int g_count;

int main(void)
{
    pthread_t tid1, tid2;
    int result;

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
    }
}

```

```

    exit(EXIT_FAILURE);
}

if ((result = pthread_create(&tid2, NULL, thread_proc1, NULL)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

pthread_join(tid1, NULL);
pthread_join(tid2, NULL);

printf("%d\n", g_count);

return 0;
}

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < 1000000; ++i)
        ++g_count;

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 1000000; ++i)
        ++g_count;

    return NULL;
}

```

Derleme işlemini şöyle yapabilirsiniz:

```
gcc -std=c11 -o sample sample.c -lpthread
```

C++'a da C++11 ile (ISO/IEC 14882: 2011) atomic isminde bir template sınıf eklenmiştir. Bu sınıfın ++, -- +=, *= gibi operatör metotları atomic işlemler yapmaktadır. Bu sınıf hem Microsoft C derleyicileri tarafından hem de gcc ve clang derleyicileri tarafından desteklenmektedir. Örneğin:

```

/* sample.cpp */

#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <pthread.h>
#include <atomic>

using namespace std;

void *thread_proc1(void *param);
void *thread_proc2(void *param);

atomic<int> g_count;

int main(void)
{
    pthread_t tid1, tid2;
    int result;

```

```

if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

if ((result = pthread_create(&tid2, NULL, thread_proc1, NULL)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

pthread_join(tid1, NULL);
pthread_join(tid2, NULL);

printf("%d\n", static_cast<int>(g_count));

return 0;
}

void *thread_proc1(void *param)
{
    int i;

    for (i = 0; i < 1000000; ++i)
        ++g_count;

    return NULL;
}

void *thread_proc2(void *param)
{
    int i;

    for (i = 0; i < 1000000; ++i)
        ++g_count;

    return NULL;
}

```

Fonksiyonların Thread Güvenliliği (Thread Safety)

Bir fonksiyonun thread güvenli (thread safe) olması demek o fonksiyonun birden fazla thread tarafından çağrıldığında soruna yol açmaması demektir. Yalnızca parametre değişkenlerini ve yerel değişkenleri kullanan fonksiyonlar thread güvenlidir. Çünkü yerel değişkenler ve parametre değişkenleri stack'te yaratılırlar ve thread'in ayrı bir stack'i vardır. Fakat global ya da static yerel nesnelere, ortak kaynaklara kilitsiz erişen fonksiyonlar thread güvenli değildir. Örneğin:

```

const char *myitoa(int val)
{
    static char buf[100];

    sprintf(buf, "%d", val);

    return buf;
}

```

Burada int bir değeri static yerel bir diziye yazı olarak yerleştiren ve bu static yerel diziyle geri dönen bir fonksiyon görüyorsunuz. Bu fonksiyon farklı thread'lerden aynı anda çağrılırsa soruna yol açabilir. Çünkü örneğin thread'lerden biri bu diziye yerleştirme yaparken kesilebilir. Bu durumda başka bir thread de aynı diziye yerleştirme yapacağından bu yerel dizide bozuk bilgiler oluşabilir. Halbuki bu fonksiyon aşağıdaki gibi yazılıysaydı thread güvenli olurdu:

```

char *myitoa(int val, char *str)
{
    sprintf(str, "%d", val);
}

```

```
    return str;
}
```

Burada artık fonksiyon programcıdan aldığı adrese yerleştirme yapmaktadır.

Thread güvenli olmayan fonksiyonların çok thread'li uygulamalarda kritik kod içerisinde çağırılması gerekir. Başka bir deyişle thread güvenli olmayan fonksiyonları çağırın kişi senkronizasyon nesnelileriyle onun aynı anda tek bir thread tarafından çağırılmasını garanti altına almalıdır. İşte sonuç olarak programcının çok thread'li uygulamalarda fonksiyonların thread güvenli olup olmadığına dikkat etmesi gerekir. Genellikle kütüphanelerin dokümantasyonlarında fonksiyonların thread güvenli olup olmadığı belirtilmektedir. Örneğin Windows'un API fonksiyonlarının hepsi thread güvenlidir.

Peki standart C fonksiyonları thread güvenli midir? İşte bazı fonksiyonlar static data kullandığı için thread güvenli olmama eğilimindedir. Örneğin strtok, localtime, asctime, rand, tmpnam gibi fonksiyonlar statik data kullanmaktadır.

C standartlarında C11'e kadar hiç thread lafı edilmemiştir. C11'de C'nin standart kütüphanesine isteğe bağlı (optional) thread fonksiyonları eklenmiştir. Ancak maalesef henüz Microsoft derleyicileri de gcc ve clang derleyicileri de standart kütüphanedeki thread fonksiyonlarını desteklememtedir. C11'e kadar standartlarda thread lafının edilmemesi derleyici yazarların bu fonksiyonları thread güvenli ya da thread güvensiz yazabileceği anlamına gelmektedir. Peki örneğin biz localtime fonksiyonunu iki thread'ten de kullanıyorsak senkronize etmeli miyiz? Yanıt: Derleyiciyi tanımamız localtime fonksiyonunun o derleyici de thread güvenli olup olmadığını bilmemiz gerekir.

İşte Microsoft Visual Studio 2005'e kadar standart C kütüphanesinin "single threaded" ve "multi threaded" olmak üzere iki versiyonunu da bulunduruyordu. Programcı da hangi kütüpheneyle programını link edileceğini proje ayarlarından seçiyordu. Yani Microsoft'ta eskiden standart C kütüphanesinin iki ayrı versiyonu vardı. Fakat Visual Studio 2005'ten itibaren artık Microsoft yalnızca "multi threaded" C kütüphanesi bulundurmaktadır. (Tek thread'li uygulamalarda kütüphanenin "multi threaded" versiyonunu kullanmak "single threaded" versiyonuna göre biraz zaman kaybı oluştursa da artık Microsoft bu kaybın önemli olmadığını düşünmektedir.)

gcc derleyicilerinde (genel olarak tüm POSIX fonksiyonlarında) sorunlu C fonksiyonlarının ayrı isimlerle iki versiyonundan da bulundurulmuştur. Sonu _r ile biten fonksiyonlar (r harfi "reentrant" sözcüğünden geliyor) thread güvenlidir. Normal isimdekiler thread güvenli değildir. Yani örneğin gcc derleyicilerinde localtime fonksiyonu thread güvenli değildir. Halbuki localtime_r fonksiyonu thread güvenlidir. Örneğin:

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

typedef struct tagPARAM {
    char name[100];
    int min, max, count;
} PARAM;

void *thread_proc(void *param);

unsigned int g_seed;

int main(void)
{
    pthread_t tid1, tid2;
    int result;
    PARAM p1 = { "t1", 10, 20, 10000 };
    PARAM p2 = { "t2", 10, 20, 10000 };
    void *ret_val1, *ret_val2;

    g_seed = (unsigned int)time(NULL);
```

```

if ((result = pthread_create(&tid1, NULL, thread_proc, (void *)&p1)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

if ((result = pthread_create(&tid2, NULL, thread_proc, (void *)&p2)) != 0) {
    fprintf(stderr, "pthread_create: %s\n", strerror(result));
    exit(EXIT_FAILURE);
}

pthread_join(tid1, &ret_val1);
pthread_join(tid2, &ret_val2);

printf("Result1:%d\n", (int)(intptr_t)ret_val1);
printf("Result2:%d\n", (int)(intptr_t)ret_val2);

return 0;
}

void *thread_proc(void *param)
{
    PARAM *p = (PARAM *)param;
    int min, max, count;
    char name[100];
    int i;
    int sum;

    min = p->min;
    max = p->max;
    count = p->count;
    strcpy(name, p->name);

    sum = 0;
    while (count--) {
        int val = rand_r(&g_seed) % (max - min) + min;
        sum += val;
    }

    return (void *)(intptr_t)sum;
}

```

Thread'e Özgü Statik Alanlar

Bazen global değişken gibi davranan thread'e özgü nesnelere gereksinim duyulmaktadır. Bilindiği gibi normal olarak bir global değişken tüm thread'lerde ortak biçimde kullanılmaktadır. Thread'e özgü global değişkenler olsaydı, nasıl bir etki oluşurdu? Örneğin aşağıdaki gibi foo ve bar fonksiyonları olsun:

```

int g_x;

void foo(void)
{
    /* .... */

    ++g_x;

    /* .... */

    ++g_x;

    /* .... */
}

void bar(void)
{
    /* .... */
}

```

```

printf("%d\n", g_x);

/* .... */
}

```

Burada foo fonksiyonu thread güvenli değildir. Çünkü farklı thread'lerden aynı anda çağrıldığında verilerde bozulma oluşur. Pekiyi bu örnekte her thread'in ayrı bir g_x global değişkeni olsaydı bir sorun oluşur muydu? Yanıt hayır, oluşmazdı. İşte thread'e özgü static alanlar aslında buna benzer bir işlemi gerçekleştirmektedir.

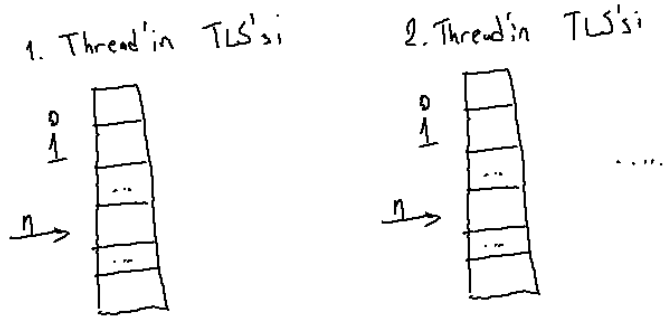
Thread'e özgü statik alanlara Windows dünyasında "Thread Local Storage (TLS)", UNIX/Linux dünyasında "Thread Specific Data (TSD)" denilmektedir. Thread'e özgü statik alanlar adeta thread'e özgü global değişken etkisi yaratmaktadır.

Thread'e özgü statik alanlar işletim sistemi tarafından her thread için ayrı ayrı tahsis edilmektedir. Yani nasıl her thread'in ayrı bir stack'i varsa aynı zamanda ayrı bir de statik alanı vardır.

Windows'ta TLS (Thread Local Storage) Kullanımı

Windows'ta TLS kullanımı şöyledir:

1) Önce (tipik olarak thread'ler yaratılmadan önce) TlsAlloc API fonksiyonu ile bir slot oluşturulur ve o slotun indeksi alınır. İşletim sistemi her thread için ayrı bir TLS alanı oluşturmaktadır. Thread'lerin TLS alanları slotlardan oluşan bir dizi gibidir. Her thread'in TLS'sinin ilgili indeksi aslında farklı bir global değişken gibi davranmaktadır.



TlsAlloc fonksiyonunun prototipi şöyledir:

```
DWORD WINAPI TlsAlloc(void);
```

Fonksiyon başarı durumunda TLS slotunun numarasına, başarısızlık durumunda TLS_OUT_OF_INDEXES değerine geri döner. Slot numarasının global bir değişkende saklanması uygun olur. TlsAlloc ile bir slot tahsis edildiğinde aslında o anda yaratılmış olan ve daha sonra yaratılacak olan tüm thread'lerde bir slot tahsisatı yapılmış olur. Bu slot numarası hangi thread'de kullanılırsa o thread'in TLS'sinin içerisindeki dizinin ilgili elemanına erişilir. Örneğin TlsAlloc bize 5 değerini vermiş olsun. Biz şimdi hangi thread'te 5 numaralı slota erişirsek o thread'in TLS'sindeki 5 numaralı slota erişmiş oluruz. Böylece aynı kod farklı thread'lerde farklı nesnelere temsil ediyor gibi olur.

2) TLS slotuna değer yerleştirmek için TlsSetValue, değer almak için TlsGetValue API fonksiyonları kullanılır.

```

BOOL WINAPI TlsSetValue(
    DWORD dwTlsIndex,
    LPVOID lpTlsValue
);

```

```

LPVOID WINAPI TlsGetValue(
    DWORD dwTlsIndex
);

```

Slotlarda saklanan bilgi bir göstericidir. Aslında bir gösterici her şeyi tutabilmeye adaydır. Eğer biz birden fazla bilgi tutacaksak bunu bir yapı olarak oluştururuz, sonra malloc fonksiyonu ile heap'te tahsisat yaparız. Onun adresini slota yerleştiririz.

3) Kullanım bitince TlsAlloc ile elde edilen slot TlsFree API fonksiyonuyla serbest bırakılır:

```
BOOL WINAPI TlsFree(  
    DWORD dwTlsIndex  
);
```

Windows sistemleri için örnek bir TLS kullanımı şöyle olabilir:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <Windows.h>  
  
typedef struct tagPERSON {  
    char name[32];  
    int no;  
} PERSON;  
  
void ExitSys(LPCSTR lpszMsg);  
DWORD __stdcall ThreadProc1(void *param);  
DWORD __stdcall ThreadProc2(void *param);  
void Foo(void);  
  
DWORD g_tlsVarIndex;  
  
int main(void)  
{  
    DWORD dwThreadId1, dwThreadId2;  
    HANDLE hThread1, hThread2;  
  
    if ((g_tlsVarIndex = TlsAlloc()) == TLS_OUT_OF_INDEXES)  
        ExitSys("AlsAlloc", EXIT_FAILURE);  
  
    if ((hThread1 = CreateThread(NULL, 0, ThreadProc1, NULL, 0, &dwThreadId1)) == NULL)  
        ExitSys("CreateThread", EXIT_FAILURE);  
  
    if ((hThread2 = CreateThread(NULL, 0, ThreadProc2, NULL, 0, &dwThreadId2)) == NULL)  
        ExitSys("CreateThread", EXIT_FAILURE);  
  
    WaitForSingleObject(hThread1, INFINITE);  
    WaitForSingleObject(hThread2, INFINITE);  
  
    TlsFree(g_tlsVarIndex);  
  
    return 0;  
}  
  
DWORD __stdcall ThreadProc1(void *param)  
{  
    PERSON *per;  
  
    if ((per = (PERSON *)malloc(sizeof(PERSON))) == NULL) {  
        fprintf(stderr, "cannot allocate memory!..\n");  
        exit(EXIT_FAILURE);  
    }  
    strcpy(per->name, "Kaan Aslan");  
    per->no = 123;  
  
    TlsSetValue(g_tlsVarIndex, per);
```



```

    Foo();

    free(per);

    return 0;
}

DWORD __stdcall ThreadProc2(void *param)
{
    PERSON *per;

    if ((per = (PERSON *)malloc(sizeof(PERSON))) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }

    strcpy(per->name, "Ali Serce");
    per->no = 674;

    TlsSetValue(g_tlsVarIndex, per);

    Foo();

    free(per);

    return 0;
}

void Foo(void)
{
    PERSON *per;

    per = (PERSON *)TlsGetValue(g_tlsVarIndex);
    printf("%s, %d\n", per->name, per->no);
}

void ExitSys(LPCSTR lpszMsg)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(EXIT_FAILURE);
}

```

UNIX/Linux Sistemlerinde TSD (Thread Specific Data) Kullanımı

Thread'e özgü alanların UNIX/Linux sistemlerindeki kullanımı Windows sistemlerindeki kullanımına oldukça benzemektedir. İşlemler sırasıyla şu adımlardan geçilerek gerçekleştirilir:

1) Önce pthread_key_create fonksiyonu ile bir slot yaratılır. (Bu fonksiyon Windows'taki TlsAlloc fonksiyonuna benzemektedir.)

```
#include <pthread.h>
```

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

Fonksiyonun birinci parametresi slotun numarasının yerleştirileceği pthread_key_t türünden nesnenin adresini alır. İkinci parametre slot yok edilirken çağrılacak fonksiyonu belirtilir. Bu parametre NULL geçilebilir. Fonksiyon başarı durumunda 0 değerine başarısızlık durumunda hata değerine geri dönmektedir.

2) Slota değer pthread_setspecific fonksiyonuyla yerleştirilip, pthread_getspecific fonksiyonuyla geri alınabilir:

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

Burada da slota bir gösterici yerleştirilip geri alınmaktadır.

3) İşlem bitince slot pthread_key_delete fonksiyonuyla yok edilir:

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

Örnek bir kullanım şöyle olabilir:

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

typedef struct tagPERSON {
    char name[32];
    int no;
} PERSON;

void *thread_proc1(void *param);
void *thread_proc2(void *param);
void foo(void);

pthread_key_t g_slotKey;

int main(void)
{
    pthread_t tid1, tid2;
    int result;

    if (pthread_key_create(&g_slotKey, NULL) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid1, NULL, thread_proc1, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    if ((result = pthread_create(&tid2, NULL, thread_proc2, NULL)) != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_key_delete(g_slotKey);
```

```

    return 0;
}

void *thread_proc1(void *param)
{
    PERSON *per;

    if ((per = (PERSON *)malloc(sizeof(PERSON))) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    strcpy(per->name, "Kaan Aslan");
    per->no = 123;

    pthread_setspecific(g_slotKey, per);

    foo();

    free(per);

    return NULL;
}

void *thread_proc2(void *param)
{
    PERSON *per;

    if ((per = (PERSON *)malloc(sizeof(PERSON))) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }

    strcpy(per->name, "Ali Serce");
    per->no = 674;

    pthread_setspecific(g_slotKey, per);

    foo();

    free(per);

    return NULL;
}

void foo(void)
{
    PERSON *per;

    per = (PERSON *)pthread_getspecific(g_slotKey);
    printf("%s, %d\n", per->name, per->no);
}

```

Thread Havuzları (Thrad Pools)

IO yoğun uygulamalarda (örneğin tipik olarak server socket uygulamalarında) çok sayıda kısa ömürlü thread'in yaratılması gerekebilmektedir. Örneğin server programa bir client'tan istek gelir. Server bir thread açarak o thread'in bunu yapmasını sağlar. Çünkü o sırada server başka client'ların isteklerini de bekletmek istemez. Thread'lerin yaratılma ve yok edilme zamanları görel olarak maliyetlidir. İşte bu tür uygulamalarda thread'lerin yaratılması ve yok edilmesini hızlandırmak için thread havuzları düşünülmüştür.

Thread havuzları aslında thread mekanizmasını kullanan bir organizasyondur. Thread havuzlarında belli miktarda thread yaratılmış ama çalışmıyor biçimde (suspend biçimde) bekletilir. Programcı havuzdan bir thread istediği zaman havuzda zaten yaratılmış olan bir thread ona verilir. Programcının thread ile işi bittiğinde thread yok edilmez, yeniden havuzda

bekletilir. Tabii havuz yetersiz kalırsa ihtiyaca göre büyütülebilmektedir. Havuzda çok fazla sayıda thread atıl biçimde bekliorsa yine havuz dinamik olarak küçültülebilmektedir.

Thread havuzları Windows sistemlerinde API fonksiyonları tarafından desteklenmektedir. Yani thread havuzundan thread alan, thread'i bırakan API fonksiyonları vardır. Tabii bu API fonksiyonları aslında taban fonksiyonlar değildir. Bunlar thread API fonksiyonlarını kendi içerisinde çağırılmaktadır. UNIX/Linux sistemlerinde thread havuzu işlemlerini yapan standart POSIX fonksiyonları yoktur. Dolayısıyla thread havuz organizasyonunu programcıların yapması beklenmektedir. Qt gibi, Java ve .NET, Mono gibi framework'lerde thread havuzu işlemlerini yapan sınıflar bulunmaktadır. C'de UNIX/Linux sistemlerinde thread havuzu işlemlerini yapan başkaları tarafından yazılmış kütüphaneler de vardır (Google'dan "thread pool implementation in C" aramasını yapınız.) C++'a C++11 ile isteğe bağlı thread kütüphanesi eklenmiş olsa da bir thread havuzu mekanizması eklenmemiştir. Bunun için boost kütüphanesindeki thread_pool sınıfı kullanılabilir.

Thread havuzları yukarıda da belirtildiği gibi kısa ömürlü çok sayıda thread'in gereksinim duyulduğu durumlarda kullanılmalıdır. Az sayıda uzun süre çalışan thread'ler için havuz organizasyonunun hiçbir faydası yoktur.

Thread Öncelikleri ve Thread'lerin Çizelgelenmesi (Thread Scheduling)

Thread çizelgelemesinin ayrıntıları işletim sisteminden işletim sistemine hatta aynı işletim sisteminde versiyondan versiyona değişebilmektedir. Bu nedenle bu konu ayrıntılı olarak "Windows Sistem Programlama" ve "UNIX/Linux Sistem Programlama" kurslarında ele alınmaktadır. Burada bir fikir oluşsun diye Windows ve Linux thread çizelgelemesinin dayandığı fikir açıklanacaktır.

Windows'ta uygulanan thread çizelgeleme algoritmasına "öncelik sınıfları ile döngüsel çizelgeleme (priority class based round robin scheduling) denilmektedir.

Windows'ta her thread'in [0-31] arasında bir öncelik derecesi vardır. Sistemde aynı öncelik derecesine sahip thread'ler bir sınıf oluşturur. İşletim sistemi sanki diğer thread'ler yokmuş gibi en yüksek öncelikli gruptaki thread'leri döngüsel çizelgeleme yöntemiyle çalıştırır. Ancak bu gruptaki thread'lerin hepsi bloke olduğunda ya da bittiğinde sonraki öncelik sınıfına geçilir. Örneğin sistemde şu önceliklere sahip thread'ler bulunuyor olsun:

T1: 8
T2: 8
T3: 8
T4: 12
T5: 12
T6: 18

Burada önce yalnızca T6 thread'i tek başına çalıştırılır. O thread bloke olduğunda ya da çalışması bittiğinde T4 ve T5 döngüsel çizelgelenir. Bunlar da bloke olduğunda ya da bunların da çalışması bittiğinde bu sefer T1, T2 ve T3 thread'leri döngüsel olarak çalıştırılacaktır. Düşük öncelikli bir sınıf çalıştırılırken o anda yüksek öncelikli bir thread'in blokesi kalktığında çalışma hemen o öncelik sınıfına döner. Ancak Windows birden fazla CPU ya da çekirdek söz konusu olduğunda her CPU ya da çekirdek için ayrı kuyruk oluşturmaktadır. Dolayısıyla bir CPU ya da çekirdek 18 öncelikli thread'i çizelgelerken diğeri 12 öncelikli thread'leri çizelgeliyor olabilir. Bu konunun Windows özelinde ayrıntıları vardır. Yani Windows'un çok CPU ya da çekirdekli sistemlerdeki çizelgelemesi bazı ayrıntıları içermektedir. Bu konu "Windows Sistem Programlama" kursunda ele alınmaktadır.

Windows'un uyguladığı bu çizelgeleme algoritmasında en dikkat çekici nokta yüksek öncelikli thread'lerin CPU'yu tekeline alabilme durumudur. Windows sistemlerinde en normal olan durum IO yoğun thread'lerin önceliğini yükseltmektir. IO yoğun thread'ler çok büyük zamanlarını uykuda geçirirler. Bir IO olayı gerçekleştiğinde uyanarak ona yanıt verirler. Bazı uygulamalarda hızlı yanıt için bu tür thread'lerin önceliklerinin yükseltilmesi tercih edilebilmektedir. Yukarıda da belirtildiği gibi yüksek bir thread uykudayken IO olayı gerçekleşirse hemen düşük öncelikli thread'in çalışmasına ara verilir ve o thread CPU'ya atanır. Tabii CPU yoğun bir thread'in önceliğini yükseltirsek gerçekten bu thread CPU'yu tekeline alabilir.

Peki Windows'ta bir thread'in önceliği nasıl değiştirilmektedir? Thread'ler için 8 normal bir önceliktir. Bizim çalıştırdığımız programlardaki thread'lerin önceliği varsayılan durumda 8'dir. Thread'lerin öncelikleri iki sayının toplamı biçiminde oluşturulmaktadır:

- 1) Prosesin öncelik sınıfı
- 2) Thread'in görelî önceliği

Prosesin öncelik sınıfı bir taban değeri oluşturmaktadır. Bu taban değere thread'in görelî önceliği (+ ya da - olarak) toplanır. Prosesin öncelik sınıfı üst processten alt processe aktarılmaktadır. Prosesin öncelik sınıfı GetPriorityClass fonksiyonuyla alınıp SetPriorityClass fonksiyonuyla set edilebilmektedir. Thread'in görelî önceliği ise GetThreadPriority fonksiyonuyla alınıp SetThreadPriority fonksiyonuyla set edilebilir. MSDN yardım sisteminde [0,31] arası thread öncelikleri için prosesin öncelik sınıfının ve thread'in görelî önceliğinin nasıl ayarlanacağı belirtilmiştir. Aşağıda bu tablonun bir bölümünü görüyorsunuz:

	Process Priority Class	Thread Priority Level
1	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
2	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
3	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
4	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
4	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
5	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
5	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
5	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
6	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
6	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
6	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
7	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
7	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
7	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
8	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
8	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
8	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
8	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
9	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
9	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
9	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
10	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
10	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL

Linux sistemlerindeki çizelgeleme algoritmaları genel olarak POSIX standartlarına uygunluk göstermektedir. POSIX standartları thread çizelgelemesi için genel bir çerçeve oluşturmaktadır. POSIX uyumlu sistemler de o çerçeveye çeşitli farklılıklarla uymaktadır. Biz burada Linux sistemlerindeki çizelgelemenin temellerini ele alacağız. Linux çekirdeklerinde çizelgeleme zamanla üç defa değiştirilmiştir. Eski klasik çizelgeleme algoritması 2.4 çekirdeğine kadar uygulanmıştı. 2.4 çekirdeğiyle birlikte "O(1) çizelgelemesi" denilen yeni bir algoritmaya geçilmiştir. Ancak 2.6 ile bundan vaz geçilmiş ve eski sistemin bir benzerine dönmüştür. Linux'un 2.6 ve sonraki versiyonlarındaki çizelgeleme algoritmasına "CFS (Completely Fair Scheduling)" denilmektedir.

Linux işletim sisteminde "Proses" ve "Thread" çizelgeleme bağlamında aynı anlamda kullanılmaktadır. Yani yine Linux sistemlerinde çizelgelenen öğeler aslında thread'lerdir. Ancak Linux işletim sistemi her thread için prosesler ile aynı veri yapısını (task_struct) kullandığı için bu bağlamda proses ile thread aynı kavramı ifade etmektedir.

POSIX standartlarına göre her thread'in bir "çizelgeleme politikası (scheduling policy)" vardır. Bu politika şunlardan biri olabilmektedir:

SCHED_FIFO
SCHED_RR

SCHED_OTHER

Normal thread'ler SCHED_OTHER çizelgeleme politikasına sahiptir. POSIX standartlarında SCHED_OTHER politikası büyük ölçüde işletim sisteminin kendi isteğine bırakılmıştır. SCHED_FIFO ve SCHED_RR politikalarına "gerçek zamanlı (real time)" çizelgeleme politikaları denmektedir. Sistemde SCHED_FIFO ve SCHED_RR politikasına sahip thread'ler varsa her zaman bu thread'ler kendi aralarında CPU'yu kullanır. Yani bu thread'ler varken SCHED_OTHER thread'ler çalışmamaktadır. Ancak SCHED_FIFO ve SCHED_RR politikasına sahip thread'ler bloke olduğunda SCHED_OTHER politikasına sahip thread'ler çalıştırılır. Tabii çok işlemcili ya da çok çekirdekli sistemlerde eğer boшта işlemci ya da çekirdek varsa bu SCHED_OTHER thread'leri o işlemcilerde ya da çekirdeklerde çalışmaya devam edebilirler.

Linux'ta SCHED_FIFO ve SCHED_RR politikasına sahip thread'lerin birer statik öncelik derecesi vardır. Bu öncelik derecesi Windows'taki thread önceliklerine benzetilebilir. SCHED_FIFO ve SCHED_RR politikasına sahip thread'lerin bu statik öncelik dereceleri 1 ile 99 arasındadır. Linux'ta SCHED_OTHER çizelgeleme politikasına sahip thread'lerin ise birer dinamik öncelik dereceleri vardır. Linux'un çizelgeleyicisi her zaman kuyruktaki en yüksek statik önceliğe sahip kuyruğun daha önündeki thread'i CPU'ya atar. Bu bağlamda Linux'taki çizelgeleme şöyle yapılmaktadır:

- SCHED_FIFO politikası en gerçek zamanlı politikadır. Sistemde n'inci statik öncelikteki bir SCHED_FIFO thread'i CPU'ya atanırsa bu politikaya sahip thread quan'talı yani zaman paylaşımlı çalışmaz. Bloke olana kadar ya da kendisinden yüksek öncelikli bir SCHED_FIFO ya da SCHED_RR thread'i uyanana kadar CPU'yu tekeline alarak kullanır. Yani yüksek statik öncelikli bir SCHED_FIFO thread'i bloke olmadıktan sonra CPU'yu tekeline almaktadır. Şimdi biz statik önceliği 40 olan bir SCHED_FIFO thread'inin CPU'ya atandığını varsayalım. Arık bu thread'in CPU'dan alınması için ya bloke olması ya da daha yüksek statik öncelikli bir thread'in çalışmaya hazır duruma gelmesi gerekir. Eğer SCHED_FIFO bir thread kendisinden yüksek statik öncelikli bir SCHED_FIFO ya da SCHED_OTHER thread tarafından kesilirse kuyruğun başına alınır. Yani CPU almış olan thread bıraktığı anda yeniden bu thread CPU'ya atanacaktır. Eğer SCHED_FIFO bir thread bloke olursa bloke çözüldüğünde kuyruğun sonuna aınır.

- SCHED_RR politikasına sahip thread'ler eğer aynı statik önceliğe sahipse döngüsel (round robin) çizelgenirler. Örneğin sistemde iki tane statik önceliği 40 olan SCHED_RR thread'i üç tane de statik önceliği 30 olan SCHED_RR thread'i bulunuyor olsun. Öncelikle statik önceliği 40 olan SCHED_RR thread'leri kendi aralarında zaman paylaşımlı olarak döngüsel biçimde çalıştırılırlar. Bunlar bloke olurlarsa bu sefer 30 önceliğe sahip üç SCHED_RR thread'i kendi aralarında döngüsel çalıştırılırlar.

- SCHED_OTHER thread'ler Linux'ta normal thread'lerdir. Yani bizim pthread_create fonksiyonuyla yarattığımız thread'ler default olarak SCHED_OTHER politikasına sahiptir. SCHED_OTHER politikasına sahip olan thread'ler zaman paylaşımlı olarak ama eğer kuyrukta SCHED_FIFO ve SCHED_RR önceliğinde hiçbir thread yoksa çalıştırılır. SCHED_OTHER thread'lerinin statik öncelikleri yoktur (ya da 0'dır). Dolayısıyla bunlar hiçbir zaman SCHED_FIFO ve SCHED_RR thread'leriyle rekabet edemezler. Pekiyi tüm SCHED_OTHER thread'ler aynı miktar CPU mu kullanmatadır? Yanıt hayır. Bunların kullandığı CPU miktarları bunların dinamik önceliklerine göre değişebilmektedir. SCHED_OTHER thread'lerinin dinamik önceliklerine "nice" değerleri de denilmektedir. Dinamik öncelikler [0, 40] arasında değişebilmektedir. Normal dinamik öncelik 20'dir. İşte Linux sistemlerinde SCHED_OTHER thread'ler kendi aralarında döngüsel (round robin) zaman paylaşımlı çalışmakla birlikte CPU kullanma zamanları arasında dinamik önceliğe bağlı olarak farklar oluşabilmektedir. Sistem özetle şöyle çalışmaktadır:

- SCHED_OTHER ve 20 dinamik önceliğe sahip normal thread'ler 200 ms. quanta süresine sahiptir. Her dinamik öncelik +10 ms ya da -10 ms quantayı azaltır. Örneğin 30 dinamik önceliğin quanta süresi 300 ms., 10 dinamik önceliği 100 ms. dir.

- Linux SCHED_OTHER thread'lerinin hepsinin quantası 0'a düştüğünde onlara yeni quanta'ları atamaktadır. Yani örneğin kuyrukta tüm quanta sürelerini kullanmadığından dolayı 80, 20, 160, 190 ms. quantaları kalan 4 SCHED_OTHER thread bulunuyor olsun. Bunların da taban quanta süreleri sırasıyla 300 ms, 200 ms, 200 ms, 400 ms. olsun. İşte önce bu thread'lerin hepsi quantalarını sıfırlar. Ondan sonra bunlara yeni taze taban quanta değerleri hep birlikte doldurulur. Örneğin 200 ms. quantaya sahip SCHED_OTHER thread CPU'ya atanmış olsun. Bu thread'in hiç bloke olmadan 180 ms. çalıştığını düşünelim. 180 ms. sonra bloke olsun. Şimdi quantası 20 ms. kalmıştır. Bu thread CPU atandığında bu 20 ms'i kullanıp diğer tüm thread'lerin quantasını bitirmesini bekleyecektir. Ondan sonra hepsine birlikte dinamik önceliklerine bağlı olarak quantaları doldurulacaktır.

UNIX/Linux sistemlerinde prosesin çizelgeleme politikası sched_getscheduler fonksiyonuyla alınıp sched_setscheduler fonksiyonuyla set edilebilmektedir. Tabii bu set işlemini yapılabilmesi için prosesin etkin kullanıcı id'sinin 0 olması gerekir.

```
#include <sched.h>
```

```
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
int sched_getscheduler(pid_t pid);
```

sched_setscheduler fonksiyonun birinci parametresi politikası değiştirilecek prosesin id'sini, ikinci parametresi yeni çizelgeleme politikasını üçüncü parametresi de statik ya da dinamik öncelik derecesini belirtir. Üçüncü parametre bir yapı olmasına karşın bu yapının şimdilik anlamlı tek elemanı vardır:

```
struct sched_param {
    ...
    int sched_priority;
    ...
};
```

sched_getscheduler fonksiyonu da çizelgeleme politikası alınacak olan prosesin id değerini parametre olarak alır ve çizelgeleme politikasına geri döner. POSIX standartlarına göre biz bir prosesin çizelgeleme politikasını değiştirdiğimiz zaman onun tüm thread'lerinin de çizelgeleme politikasının değiştiriliyor olması gerekmektedir. Ancak Linux sistemleri bu anlamda POSIX standartlarına tam uyum göstermemektedir. Yani Linux sistemlerinde biz prosesin çizelgeleme politikasını değiştirdiğimizde yalnızca o prosesin ana thread'inin çizelgeleme politikasını değiştirmiş oluruz. Pekiye POSIX'te ya da Linux'ta belli bir thread'in çizelgeleme politikası nasıl değiştirilmektedir? İşte POSIX standartlarına göre belli bir thread'in çizelgeleme politikasını elde etmek ve değiştirmek için pthread_getschedparam, pthread_setschedparam fonksiyonları kullanılmaktadır:

```
#include <pthread.h>
```

```
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);
```

Linux çekirdeği aslında her thread için proses kontrol bloğu yapısının aynısını (task_struct) oluşturmaktadır. Dolayısıyla aslında Linux'ta thread'ler bellek alanı ortak olan prosesler gibidir. Biz Linux'ta istersek bir thread'in prosesmiş gibi proses id'sini alabiliriz. Bunun için gettid fonksiyonu kullanılmaktadır:

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

```
pid_t gettid(void);
```

Yani örneğin biz bu fonksiyon ile o anda çalışmakta olan thread'e ilişkin sistem genelinde tek olan id değerini alıp bunu proses id gibi diğer fonksiyonlarda kullanabiliriz. Tabii bu fonksiyon tamamen Linux'a özgüdür.

Anımsanacağı gibi Linux normal thread'ler default olarak SCHED_OTHER çizelgeleme politikasına sahiptir. SCHED_OTHER thread'ler de quanta süreleri dikkate alınarak döngüsel çizelgeleniyordu. SCHED_OTHER thread'lerin quanta sürelerinin onların dinamik önceliği (dynamic priority) ile ilişkili olduğunu anımsayınız. SCHED_OTHER thread'lerinin dinamik öncelikleri Linux'ta [1, 40] arasındadır. Pekiye biz bir SCHED_OTHER thread'inin dinamik önceliğini nasıl değiştirebiliriz?

POSIX standartlarına göre (Linux'ta buna uymaktadır) bir SCHED_OTHER thread'inin dinamik önceliği nice isimli fonksiyonla değiştirilir. nice fonksiyonunun prototipi şöyledir:

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

```
int nice(int inc);
```


nice fonksiyonu Linux'ta -20 ile +10 arasında bir değeri argüman olarak alır. O anda çalışmakta olan SCHED_OTHER prosesinin dinamik önceliğine argümanla aldığı değerin negatifini ekler. Örneğin o anda çalışmakta olan SCHED_OTHER prosesinin dinamik önceliği 20 olsun (Linux'ta default durum). Biz şimdi nice (-10) çağrısı yaparsak prosesin dinamik önceliği 30 olacaktır. O halde negatif değerler dinamik önceliği artırırken pozitif değerler azaltmaktadır. Bu dinamik önceliğin kullanılacak quanta süresiyle ilgili olduğunu anımsayınız. Yetkisiz prosesler dinamik önceliklerini artıramazlar (yani nice fonksiyonunu negatif değerle çağırılmazlar). Yetkisiz prosesler yalnızca dinamik önceliklerini düşürebilirler (yani nice fonksiyonunu pozitif değerle çağırabilirler.) POSIX standartlarına göre nice fonksiyonu prosesin tüm thread'leri üzerinde etkili olmaktadır. Ancak Linux bu anlamda standartlara uymamaktadır. nice fonksiyonu Linux'ta yalnızca prosesin ana thread'inin dinamik önceliğini değiştirmektedir.

nice fonksiyonun kendi prosesinin dinamik önceliğini değiştirmeye çalıştığına dikkat ediniz. Eğer başka bir prosesin dinamik önceliği değiştirilmek isteniyorsa bu durumda nice fonksiyonu yerine setpriority POSIX fonksiyonu kullanılmalıdır. getpriority fonksiyonu ile de başka bir SCHED_OTHER prosesin nice değeri elde edilebilir. setpriority ve getpriority fonksiyonlarının prototipleri şöyledir:

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

```
int getpriority(int which, id_t who);  
int setpriority(int which, id_t who, int value);
```

setpriority fonksiyonunun birinci parametresi neyin nice değerinin değiştirileceğini belirtir. Bu parametre PRIO_PROCESS girilirse ilgili prosesin, PRIO_PGRP girilirse ilgili proses grubundaki tüm proseslerin ve PRIO_USER girilirse etkin kullanıcı id'si belli bir değerde olan proseslerin nice değerleri değiştirilir. Fonksiyonun ikinci parametresi birinci parametrenin durumuna göre proses id, proses grup id ya da kullanıcı id'si olabilmektedir. Üçüncü parametre ise yeni nice değerini belirtir. setpriority fonksiyonu başarı durumunda yeni nice değerine başarısızlık durumunda -1 değerine geri dönmektedir. getpriority fonksiyonu da o andaki mevcut nice değerini elde etmektedir.

Yukarıda da belirttiğimiz gibi POSIX standartlarıncı belli bir thread'in nice değerini değiştirebilen bir fonksiyon bulunmamaktadır. Ancak Linux sistemlerinde thread'ler için ayrılan veri yapısı prosesler için ayrılan proses kontrol blok ile aynı olduğu için adeta bu sistemlerde thread'ler de birer proses id'sine sahip gibidirler. Yukarıda da belirttiğimiz gibi bir thread için onun proses id değeri gettid fonksiyonuyla elde edilebilmektedir. Bu durumda biz belli bir SCHED_OTHER thread'in Linux sistemlerinde dinamik önceliğini gettid ve setpriority fonksiyonlarını kullanarak değiştirebiliriz.

Veri Yapıları ve Algoritmalar

Bu bölümde veri yapıları ve algoritmalar konusu temel düzeyde ele alınacaktır. Veri yapıları ile algoritmalar konusu birbirleri ile yakın biçimde ilişkilidir. Bu nedenle bu iki konu genellikle birlikte ele alınmaktadır.

Algoritma Nedir?

Bir problemi çözüme götüren adımlar topluluğuna algoritma (algorithm) denilmektedir. Sözcük etimolojik bakımdan cebirin kurucusu El-Harizmi'nin isminden türetilmiştir. Algoritma problemi kesin çözüme götürür. Ancak bazı problemlerin kesin çözümü çok uzun zaman alabilmektedir. Bunlar için kısa sürede iyi bir çözüm ile yetinilebilir. İşte problemin kesin çözümünü bulmayan ama iyi bir çözüm vaat eden adımlar topluluğuna "sezgisel yöntemler (heuristic)" denilmektedir.

Algoritmalar nasıl ifade edilirler? Bunun için çok formel bir yöntem yoktur. Bazen akış diyagramları, sahte kodlar (pseudo codes) kullanılabilir. Hemen her zaman sözel bir anlatım buna eşlik eder. Algoritma anlatmak için özel diller tasarlanmış olsa da bunlar yaygın bir kullanım bulamamıştır. Algoritmayı anlatmak için en çok kullanılan yöntem popüler bir dille algoritmayı gerçekleştirmektir. Eskiden bu amaçla C yaygın kullanılırdı. Şimdi C'nin yanı sıra C++, Java, C#, Python gibi diller de bu amaçla kullanılmaktadır.

Peki algoritma dilden bağımsız mıdır? Algoritmaların çoğu dilden bağımsız olarak ifade edilebilir. Ancak ince noktalara gelindiğinde algoritmalar da dile bağımlı hale gelebilmektedir. Örneğin bir dilde belli işi daha kolay yapan deyimler

olabilir. Bu durumda algoritma o deyimlerle gerçekleştirilebilir. Ya da örneğin fonksiyonel dillerde algoritmalar özyleneleme içerecek biçimde tasarlanmaktadır.

Bazı problemlerin çözümü için tek bir yol bulunuyor olabilir. Fakat bazıları için birden fazla algoritma söz konusu olabilir. Bu durumda bu algoritmaları kıyaslamak isteyebiliriz. Kıyaslamak için ise ölçütler gerekir. Algoritmaları kıyaslamak için iki önemli ölçüt vardır: Hız ve kaynak kullanımı. Ancak baskın ölçüt hızdır. Bu nedenle algoritmaları karşılaştırmak denildiğinde default olarak hız ölçütüne göre karşılaştırma anlaşılır. Algoritmaların masaya yatırılıp karşılaştırılması sürecine "algoritma analizi (analysis of algorithms)" denilmektedir.

Algoritma Analizi

İki algoritmanın hızı nasıl ölçülebilir? İlk akla gelen yöntem simülasyon yöntemidir. Bu yöntemde algoritmalar çalıştırılıp aldığı zamanlar ölçülür. Tabii algoritmaların çoğu birtakım girdiler üzerinde işlem yapmaktadır (örneğin bir dizi üzerinde). Bu durumda o girdilerin dağılımı da önemlidir. Eğer biz simülasyonu bir kez yaparsak yanlış sonuçlar elde edebiliriz. Rastgele girdiler üzerinde çok sayıda denememinin bir ortalaması ancak bir fikir verebilir.

Simülasyon yöntemi matematiksel bir yöntem değildir. Bu nedenle bize önemli bir bilgi vermez. Algoritmaları cebirsel olarak ölçmek daha sağlam bir zemin oluşturabilmektedir. Algoritmaları analiz ederken kullanılacak sağlam yöntemlerden biri algoritmadaki çözüm için toplamda kaç işlemin gerektiğine bakmaktır. (Tabii tüm işlemlerin makine zamanları aynı değildir fakat bu durum ihmal edilebilir.) Örneğin iki sort algoritmasını karşılaştıracak olalım. İkisinde de sort işlemi için kaç işlem gerektiğini hesaplamaya çalışabiliriz. Tabii algoritmaların girdi parametreleri vardır. Sort işleminde dizinin uzunluğu olan N bir girdi parametresidir. Algoritma için gereken işlemlerin sayısı bu N ile ilgili olacaktır.

Algoritmadaki işlemlerin sayısı bulunabilir mi? Ya algoritmada karışık if deyimleri varsa ne olacaktır? Biz akışın nasıl olacağını giridiyi bilmeden anlayamayız. Pekiyi bu durumda işlemlerin sayısı nasıl hesaplanacaktır? Örneğin aşağıdaki en büyük sayıyı bulma algoritmasına bakalım:

```
int a[N] = {...};
int max, i;

max = a[0];
for (i = 1; i < N; +i)
    if (max < a[i])
        max = a[i];
```

Burada toplam işlemin sayısı $\max = a[i]$ işlemine bağlıdır. Pekiyi bu işlemten kaç tane olacağını diziyi görmeden bilebilir miyiz? Yanıt tabii ki hayır. İşte algoritma analizinde işlemlerin sayısını hesaplarken üç durum dikkate alınabilmektedir:

- 1) Ortalama durum (average case condition)
- 2) En kötü durum (worst case condition)
- 3) En iyi durum (best case condition)

En kötü durum olabileceklerin en kötüsüdür. Yukarıdaki örnekte en kötü durumda $\max = a[i]$ işlemi $N - 1$ kez yapılır. Ortalama durum tüm olasılıkların ortalamasını temsil eder. Yukarıdaki örnekte ortalama olarak $\max = a[i]$ işlemi $(N - 1) / 2$ kez yapılmaktadır. En iyi durum da olabileceklerin en iyisini temsil eder. Yukarı örnekte en iyi durumda $\max = a[i]$ işlemi hiç yapılmaz.

Tabii analizde en iyi durumun çok yararı yoktur. Çünkü aşırı iyimserlik programlamada çok değerli değildir. En kötü durum senaryosu önemlidir. Çünkü bazı durumlarda en kötü duruma bile hazırlıklı olmak gerekebilir. Şüphesiz en önemli durum ortalama durumdur. Ortalama durum algoritmanın karakterini en iyi yansıtan durumdur. O halde bu üç durumun önemi şöyle sıralanabilir: Ortalama durum, en kötü durum, en iyi durum. Genellikle algoritmalar ele alınırken hem ortalama durum hem de en kötü durum analizi yapılır.

Algoritmaların işlem sayısına ilişkin analizler de sanıldığı kadar kolay değildir. Çünkü özellikle ortalama durumu hesaplamak bazen çok karmaşık olabilmektedir. Algoritmaları pratik bakımdan kıyaslamak için asimtotik notasyonlardan faydalanılmaktadır. Bunların en yaygın kullanılanı Big O notasyonudur.

Big O Notasyonu

Big O notasyonu algoritmaları karmaşıklıklarına göre kategorilere (sınıflara) ayırır. Kategoriler arasında da iyilik kötülük ilişkisi vardır. Yani Big O notasyonunda karmaşıklıkta algoritmaları aynı kategoriye (sınıfa) içerisine yerleştirilir. Bu kategoriler de birbirlerine göre iyi ya da kötü olabilmektedir. Aynı kategoride olan algoritmalar aralarında farklılıklar olsa bile aynı düzeyde iyi kabul edilmektedir.

Big O notasyonu kaba bir kıyaslama biçimidir. Eğer algoritmalar ayrıntılı bir biçimde kıyaslanacaksa yukarıda belirtilen işlem sayıları hesaplanmalıdır. Big O notasyonunda belli karakterdeki algoritmalar aynı kategoridedir. Adeta bunlar arasında fark yokmuş gibi davranılır. Eğer algoritma birden fazla kategoriye giriyorsa en kötü kategori onun gerçek kategorisini belirtir. İyiden kötüye doğru Big O kategorileri şöyledir:

O(1): Bu kategoriye "sabit karmaşıklık" da denilmektedir. Eğer algoritmada hiç döngü yoksa her şey tekil işlemlerle yapılıyorsa algoritma $O(1)$ karmaşıklıktadır. (Örneğin üçgenin alanının bulunması gibi). Tekil işlemlerin sayısı önemli değildir.

O(log N): Bu karmaşıklıkta N uzunluğunda bir girdi kümesi (örneğin bir dizi) vardır. Algoritmada açık ya da özinelemeli olarak bir döngü bulunur. Ancak bu döngü N kadar değil log N kadar dönmektedir. (log default olarak 2 tabanına ilişkin düşünülmelidir.). Örneğin "ikili arama (binary search)" algoritması $O(\log N)$ karmaşıklıktadır. (Tabi yukarıda da belirtildiği gibi başka tekil işlemler de algoritmada bulunabilir.) Bu karmaşıklık kategorisine "logaritmik karmaşıklık" da denilmektedir.

O(N): Bu kategoriye "doğrusal karmaşıklık" da denilmektedir. Bu tür algoritmalarda iç içe olmayan tekil birden fazla döngü bulunabilir. Girdi uzunluğu N ise bu döngüler N ile bağlantılı olarak (tipik olarak N defa ya da $N/2$ defa ya da $N-5$ defa vs.) dönerler. Fakat dönüş logaritmik değildir. N ile oransal olabilir (Örneğin $2N$, $5N$, $N/2$ gibi). Örneğin en büyük sayıyı bulma algoritması, doğrusal arama işlemi, bir dizinin arasına bir elemanın eklenmesi işlemi doğrusal karmaşıklıktadır.

O(N log N): Bu tür algoritmalarda iç içe iki döngü vardır. Ancak döngülerden biri N ile orantılı biçimde dönerken diğeri log N kadar döner. Örneğin "quick sort" algoritması $O(N \log N)$ karmaşıklıktadır.

O(N²): Bu kategoriye "karesel karmaşıklık" da denilmektedir. Bu tür algoritmalarda iç içe iki döngü vardır (tabii başka tekil döngüler de olabilir.) Döngülerin ikisi de N ile orantılı biçimde dönerler. Örneğin "kabarcık sıralaması (bubble sort)", "seçerek sıralama (selection sort)" algoritmaları karesel karmaşıklıktadır.

O(N³): Bu kategoriye "küpsel karmaşıklık" da denilmektedir. Burada iç içe N ile orantılı 3 döngü vardır. Örneğin matris çarpımı böyle bir döngüsel yapı gerektirmektedir.

O(N^k): Bu kategoriye K'sal karmaşıklık denir. İç içe k tane N ile orantılı döngü bulunmaktadır.

Yukarıdaki tüm kategorilere "polinomsal karmaşıklık" da denilmektedir. Polinomsal karmaşıklıkta problemlerin bugünkü bilgisayarlarla makul zamanda çözülmesi mümkündür.

O(K^N): Bu kategoriye üstel karmaşıklık (exponential complexity) da denilmektedir. Bu kategorideki algoritmalarındaki işlem sayısı N'e göre çok hızlı artmaktadır. Bugünkü bilgisayarlarla bile üstel karmaşıklığa sahip algoritmaların kesin çözümleri çok fazla zaman alabilir. Bu nedenle bunların bilgisayar çözümleri mümkün olmayabilir. Bu tür algoritmalarda sezgisel yöntemler (heuristic) önem kazanmaktadır. Örneğin N elemanlı bir kümenin alt kümelerinin sayısı 2^N tanedir. Bir kümenin tüm alt kümelerine bakarak işlem yapan algoritmalar (böyle çok algoritma vardır) üstel karmaşıklıktadır.

O(N!): Bu karmaşıklığa faktöryel karmaşıklığı denilmektedir. Bu en kötü karmaşıklık grubudur. Örneğin gezgin satıcı probleminde satıcı bir merkezden (düğümünden) çıkarak tüm şehirleri (düğümleri) dolaşır tekrar başlangıç yerine geri gelir. Amaç en kısa turu atacak çözümün bulunmasıdır. Bu problemde N tane düğüm varsa $(N-1)!$ tane rotayı hesaplamak gerekir. O halde gezgin satıcı problemi faktöryel karmaşıklıkta bir algoritmadır. İş sıralama çizelgeleme (sequencing and scheduling) problemleri de genel olarak bu karmaşıklıktadır.

Üstel ve faktöryel karmaşıklıklara "polinomsal olmayan (Non-Polynomial ya da kısaca NP)" karmaşıklıkta problem denilmektedir. İşte bu tür problemler hala üzerinde en çok çalışılan ve özel yöntem bulunmaya çalışılan problemlerdir.

İşte algoritmaları karşılaştırırken onların Big O kategorileri belirlenir. Sonra hangi kategori değerinden daha iyiye o algoritmanın kategorik olarak diğerinden daha iyi olduğu söylenir. Tabii bu yöntemde aynı kategoriye sahip algoritmalar sanki aynı etkinlikteymiş gibi değerlendirilmektedir. Big O gibi asimtotik notasyonlar hızlı bir biçimde genel bir fikir edinmek için kullanılmaktadır. Algoritmanın ayrıntılı analizinde "ortalama" ve "en kötü" işlem zamanları belirtilmelidir. Örneğin bir algoritma iç içe iki döngü içeriyorsa ve otuz kadar tekil döngü içeriyorsa bu algoritma karesel karmaşıklıktadır. Kategorik olarak bu algoritmanın iç içe iki döngü içeren ve on tekil döngü içeren algoritmadan bir farkı yoktur. Fakat ayrıntılı analizde bunun diğerinden daha iyi olduğu sonucu çıkartılabilir.

Aslında algoritmalar dünyasında çoğu kez bir algoritmanın mutlak iyi olduğunu söyleyemeyiz. Algoritmanın karmaşıklık kategorisi diğerinden yüksek olsa bile ya da ortalama ve en kötü olasılıktaki işlem sayısı diğerinden kötü olsa bile bazı durumlarda diğerinden hızlı çalışabilmektedir. Örneğin "quick sort" algoritması kategorik olarak $O(N \log N)$ karmaşıklıktadır. Bubble sort algoritması ise $O(N^2)$ karmaşıklıktadır. Fakat elemanların çoğu sıralı az sayıda elemanın sırasının bozuk olduğu dizilimlerde "bubble sort" algoritması "quick sort" algoritmasından hızlı çalışabilmektedir. O halde biz sırasının çok az bozuk olduğunu bildiğimiz olguları sıraya dizmek istiyorsak burada "bubble sort" algoritmasını tercih edebiliriz. O halde aslında karmaşıklığı kötü olan algoritmaların bazı özel durumlarda daha etkin olabilmektedir. Biz de sistemi değerlendirip ona göre hangi algoritmanın kullanılacağına karar veririz. Yukarıdaki örnekte şüphesiz dizi dağılımı hakkında hiçbir bilgimiz yoksa "quick sort" algoritmasını tercih ederiz. Ancak dizinin çoğu elemanlarının zaten sıralı olduğunu biliyorsak "bubble sort" algoritmasını tercih ederiz.

Algoritmaların Sınıflandırılması

Algoritmalar çeşitli ölçütlere göre sınıflandırılabilir. Tipik kullanılan sınıflandırma ölçütleri şunlardır:

Gerçekleştirim (Implementation) Biçimlerine Göre Sınıflandırma: Bu sınıflandırmada algoritmalar onların kodlarının nasıl yazıldığına göre sınıflandırılmaktadır. Tipik alt sınıflar şunlardır:

- Özyinelemeli Algoritmalar
- Mantıksal Algoritmalar
- Seri Algoritmalar
- Paralel ya da Dağıtık (Distributed) Algoritmalar
- Deterministik ya da Deterministik Olmayan (Probabilistic, Stochastic) Algoritmalar
- Quantum Bilgisayarlarına Yönelik Algoritmalar

Tasarım Biçimine Göre Sınıflandırma: Bu sınıflandırma algoritmaları genel tasarımına göre sınıflandırmaktadır. Tipik alt sınıflar şunlardır:

- Böl ve Ele Geçir (Divide and Conquer) Algoritmaları
- Rastgele İşlemlerin Uygulandığı Algoritmalar
- Karmaşıklığın Azaltılması ile Çözüm Bulunmaya Çalışılan Algoritmalar
- Sürekli İyileştirme Yapılarak Geri Dönüşlü (Backtracking) Algoritmalar

Optimizasyon Tekniğine Göre Sınıflandırma: Bu sınıflandırma genellikle algoritmaların bir en iyi değer bulma çabasına göre yapılmaktadır. Tipik alt sınıfları şunlardır:

- Doğrusal Programlama
- Dinamik Programlama
- Greedy Algoritmaları
- Sezgisel (Heuristic) Yöntemler

Karmaşıklığa Göre Sınıflandırma: Bu sınıflandırmada algoritmanın karmaşıklığına bakılır. Zaten bu sınıflandırma yukarıda ele alınmıştır.

- $O(1)$ Karmaşıklıkta Algoritmalar
- $O(\log N)$ Karmaşıklıkta Algoritmalar
- $O(N)$ Karmaşıklıkta Algoritmalar
- $O(N \log N)$ Karmaşıklıkta Algoritmalar
- $O(N^2)$ Karmaşıklıkta Algoritmalar
- $O(N^3)$ Karmaşıklıkta Algoritmalar
- $O(N^k)$ Karmaşıklıkta Algoritmalar
- Üstel Karmaşıklıkta Algoritmalar ($O(a^N)$)
- Faktöryel Karmaşıklıkta Algoritmalar ($O(N!)$)

Donald Knuth "The Art Of Computing Programming" kitap serisinde algoritmaları ayrı kitaplar biçiminde şöyle sınıflandırmıştır:

- Temel Algoritmalar (Fundamental Algorithms): Bunlar günlük programlamada karşılaşılan temel algoritmalarıdır.
- Yarı Nümerik Algoritmalar (Seminumerical Algorithms): Bunlar nümerik tarafı da olan ama programlama dünyasında karşılaşılabilen algoritmalarıdır.
- Arama ve Sıraya Dizme Algoritmaları (Sorting and Searching Algorithms): Bunlar arama ve sıraya dizme işlemlerine ilişkin algoritmalarıdır.
- Graf Algoritmaları (Graph Algorithms): Graflar üzerinde dolaşım ve optimizasyon içeren algoritmalarıdır.
- Optimizasyon Algoritmaları (Optimization Algorithms): Matematiksel en iyi değerleri bulmaya çalışan algoritmalarıdır.
- Nümerik Algoritmalar (Numerical Algorithms): Sayısal olarak kök bulma, denklemleri çözme, türev, integral gibi nümerik işlemleri konu edinen algoritmalarıdır.

Veri Yapıları (Data Structure)

Aralarında mantıksal ya da fiziksel ilişki bulunan nesnelerin oluşturduğu topluluğa "veri yapısı (data structure)" denilmektedir. Bazı veri yapıları dilin sentaksı tarafından zaten doğrudan desteklenmektedir. Bazılarını ise biz kullandığımız dilin olanaklarıyla fonksiyonlar ya da sınıflar biçiminde oluştururuz. C'de derleyici tarafından desteklenen (built-in) veri yapıları şunlardır:

- Diziler (arrays)
- Yapılar (structures)
- Birlikler (unions)

Elemanları aynı türden olan ve bellekte ardışıl bir biçimde bulunan veri yapılarına dizi denilmektedir. Elemanlar aynı türden olduğu için dizi bildirimleri oldukça basittir. Örneğin:

```
int a[10];
```

Dizi elemanların ardışıl olmasının iki önemli faydası vardır. Ardışılıktan dolayı dizinin herhangi bir elemanına $O(1)$ karmaşıklıkta yani çok hızlı bir biçimde erişilebilmektedir. Gerçekten de işlemcilerin belli bir adresten n ileriye ve n geriye erişim yapabilen makine komutları vardır. Yani işlemciler A adresindeki ya da $A + N$ adresindeki bilgiye aşağı yukarı aynı hızda erişirler. Bu tür erişimlere "rastgele erişimler (random access)" de denilmektedir. Buradaki "rastgele" sözcüğü gelişigüzel anlamında değil "her yere aynı hızda erişim" anlamındadır. Örneğin disk erişimleri de bu anlamda rastgeledir. Dizi elemanlarının ardışıl olmasının diğer bir faydası da onların başlangıç adresleri yoluyla fonksiyonlara aktarılabilmesidir. Biz bir diziyi fonksiyona parametre yoluyla aktaracak olsak dizinin her elemanını ayrı ayrı aktarmayız. Dizinin başlangıç adresini fonksiyona aktarırız. Fonksiyon da dizinin her elemanına erişebilir.

Yapılar elemanları farklı türlerden olabilen ve bellekte ardışıl bir biçimde bulunan veri yapılarıdır. Bu anlamda yapılar aslında dizilerin daha genel bir biçimidir. Tabii yapıların elemanları farklı türlerden olabildiği için hangi elemanın hangi

türden olduğunu programcının derleyiciye söylemesi gerekir. Bu nedenle yapılarla çalışmadan önce bir yapı bildirimini gerekmektedir. Örneğin:

```
struct SAMPLE {  
    int a;  
    long b;  
    double c;  
};
```

```
struct SAMPLE s;
```

Yapı elemanlarına erişim $O(1)$ karmaşıklıktadır. Erişim C'de nokta ya da ok operatörü ile yapılmaktadır. Pekiyi yapı elemanlarına dizilerde olduğu gibi köşeli parantezlerle erişilemez miydi? C statik tür sistemine sahip bir dildir. eğer yapı elemanlarına köşeli parantez içerisinde erişim mümkün olsaydı köşeli parantez içerisindeki ifadenin sabit ifadesi olması gerekirdi. Elemanlara birer vermek ve onlara bu isimler yoluyla erişmek daha okunabilir bir kullanım sunmaktadır. Yapı elemanları da bellekte ardışıl bir biçimde bulunduğu için yapı nesnelere fonksiyonlara başlangıç adresi yoluyla aktarılabilir.

Yapı elemanlarının aralarında "hizalama (alignment)" amaçlı boşluklar (padding) bırakıldığını biliyorsunuz. Bu boşluklar kurallı bir biçimde bırakıldığı için bu durum yapı elemanlarının ardışıklığını bozmamaktadır. Burada C açısından önemli bir durum vardır. Biz bir yapıyı kullanan fonksiyonları derleyip kütüphaneye yerleştirmiş olalım. Onu kullanırken yapı bildirimindeki elemanların yerlerini değiştiremeyiz. Eğer elemanların yerlerini değiştirsek derlenmiş kodla onu kullanan kod için elemanların yerleri farklı olacaktır. Benzer biçimde derleyerek kütüphaneye yerleştirdiğimiz yapı kullanan bir kodun hizalama gereksinimi ile o kütüphaneyi kullanan kodun hizalama gereksinimlerinin aynı olması gerekir.

Birlikler elemanları aynı adresten itibaren çakışık yerleştirilen veri yapılarıdır. Bu durumda bir birliğin belli bir elemanı değiştirildiğinde diğer elemanların değerleri de değişecektir. Bir birlik nesnesi birliğin en uzun elemanı kadar yer kaplamaktadır. Gerçekten de C'de bir birlik nesnesine küme parantezleri içerisinde ilkdeğer verilirken yalnızca onun ilk elemanı için ilkdeğer verilmektedir.

Bazı dillerin standart kütüphanelerinde pek çok veri yapısı bulunabilmektedir. Özellikle nesne yönelimli programlama dillerinin kütüphanelerinde pek çok veri yapısı sınıfsal biçimde gerçekleştirilmiştir. Örneğin C++'ın, Java'nın, .NET'in standart kütüphanelerinde pek çok veri yapısı hazır biçimde zaten bulunmaktadır. O ortamlarda çalışan programcıların bunları yeniden yazması gerekmez. Bazı framework'ler de yine çeşitli veri yapılarını barındırmaktadır. (Örneğin Qt'de MFC'de Cocoa'da temel veri yapıları yine sınıflar biçiminde bulunmaktadır.) Ancak C'nin standart kütüphanesinde genel olarak veri yapılarına ilişkin fonksiyonlar bulunmamaktadır.

Veri yapıları konusunda "Soyut Veri Türü (Abstract Data Type (ADT))" isimli bir kavram da çok sık kullanılmaktadır. Soyut veri türü denildiğinde bir veri yapısı üzerinde işlem yapan fonksiyonlar anlaşılmaktadır. Soyut bir veri yapısında veri yapısını idare için gereken birtakım veriler ve onları yöneten fonksiyonlar bulunur. Aslında soyut veri yapısı adeta veri yapısını oluşturan bir sınıf gibi düşünülebilir. Örneğin "Stack ADT" başlığını gördüğümüzde stack veri yapısını organize eden ve bunun üzerinde işlemler yapan fonksiyonlardan oluşan kodlar aklımıza gelir. Soyut veri yapısı özellikle nesne yönelimli teknikte birlikte kullanılmaya başlanmış bir terimdir.

Temel Veri Yapıları

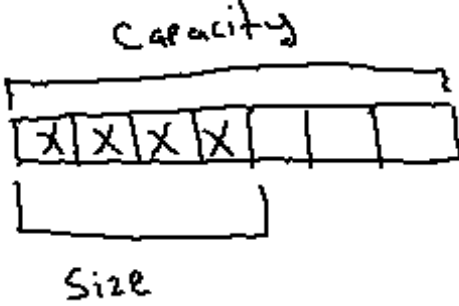
Dinamik diziler, kuyruk sistemleri, stack sistemleri, bağlı listeler ve hash tabloları en çok kullanılan temel veri yapılarıdır. Bu bölümde bunların nasıl oluşturulacağı ve algoritmik özellikleri üzerinde duracağız.

Dinamik Diziler (Dynamic Arrays)

Pek çok uygulamada gereksinim duyulan dizinin uzunluğu programın çalışma zamanı sırasında değişebilmektedir. Örneğin bir çizim programında çizilen şekildeki noktaların bir dizide tutulduğunu düşünelim. Başlangıçta bu dizinin hangi uzunlukta olacağını bilemeyiz. Bu dizi duruma göre büyüyebilir. Ya da bir dizin içerisindeki dosyaları bir diziyeye yerleştirmek isteyelim. Dizindeki dosyaları bulurken önceden bunların kaç tane olduğunu bilemeyiz. İşte bu tür

durumlarda bir dizi oluşturarak duruma göre bunu büyütme gerekir. Bu işlemi yapan veri yapısı ve fonksiyonlara "Dinamik Dizi" dizi denilmektedir.

Dinamik dizi gerçekleştirilmesinde dizi için önce küçük bir alan tahsis edilir. Bu alan dolduğunda yeniden tahsisat yapılarak daha büyük bir alana geçilir. Tabii eski alandaki bilgiler yeni alana kopyalanacak ve eski alan da serbest bırakılacaktır. (C'de bu işlem zaten realloc fonksiyonuyla yapılmaktadır.) Geleneksel olarak dizi için tahsis edilmiş olan toplam alana "capacity", dolu olan eleman sayısına ise "size" ya da "count" denilmektedir.



Dinamik diziyeye eleman eklenirken eleman "size" ile belirtilen indekse eklenir, "size" bir artırılır. "size" değeri "capacity" değerine geldiğinde yeniden tahsisat yapılarak (reallocation) "capacity" değeri artırılır. Pekiyi capacity değeri ne kadar artırılmalıdır? Capacity değeri eskisinin katı olarak artırılırsa (yani geometrik olarak artırılırsa) sona eleman ekleme işlemi sabit karmaşıklığa yaklaşır. Eğer Capacity değeri eskisinden N fazla olacak biçimde artırılırsa bu durumda sona eleman ekleme işlemi doğrusal karmaşıklığa yaklaşır. Geometrik artırımdaki sona eleman ekleme karmaşıklığına "ek maliyetli sabit zamanlı karmaşıklık (amortized constant time complexity)" denilmektedir. "Ek maliyetli sabit zamanlı karmaşıklık" demek, işlemin çoğu kez sabit karmaşıklıkta ancak üstel aralıklarla doğrusal karmaşıklıkta yapılması demektir.

Yukarıda da belirtildiği gibi tipik olarak dinamik dizilerde "capacity" artırımını eskisinin iki katı olacak biçimde yapılmaktadır. Buradaki karmaşıklığı şöyle açıklayabiliriz: Örneğin artırımın hep beşer beşer yapıldığını düşünelim. Bu durumda dinamik dizinin sonuna eleman ekleme sırasında her 5 eklemede bir yeniden tahsisat yapılacaktır. Bu yeniden tahsisat işlemi sırasında doğrusal karmaşıklıkta bir kopyalama işlemi devreye girer (ayrıca tahsisat işleminin kendisinin de doğrusal karmaşıklıkta yapıldığını varsayabiliriz). Bu durumda sona eleman ekleme işlemi aslında doğrusal karmaşıklıktadır. Yani biz her eklemeyi 1/5 kadar dönen bir döngüyle yaptığımızı düşünebiliriz. Halbuki büyütme eskisinin iki katı kadar yapılırsa sona ekleme işlemi her beşte bir değil logaritmik bir karmaşıklığa çekilecektir. Yani arada sırada nadiren yeniden tahsisat yapılacaktır. İşte bu sisteme "ek maliyetli sabit zamanlı (amortized constant time)" ekleme denilmektedir.

Dinamik dizilerde elemana erişim normal dizilerde olduğu gibi sabit karmaşıklıkta (yani $O(1)$ karmaşıklıkta) yapılmaktadır. Araya eleman ekleme ve aradan eleman silme işlemi ise doğrusal yani $O(N)$ karmaşıklıkta yapılmaktadır. Çünkü araya eleman eklerken o noktadan dizinin kaydırılması gerekir ("expand" işlemi). Aradan eleman silerken de o noktadan dizinin sıkıştırılması gerekir ("shrink" işlemi).

Dinamik dizileri handle sistemi kullanarak aşağıdaki gibi oluşturabiliriz:

```
/* DynamicArray.h */

#ifndef DYNAMICARRAY_H_
#define DYNAMICARRAY_H_

/* Symbolic Constants */

#define DEF_CAPACITY 4
#define DARRAY_FAILED ((size_t)-1)

#include <stddef.h>

/* Type Declarations */
```

```

typedef int DATATYPE;

typedef struct tagDARRAY {
    DATATYPE *pArray;
    size_t size;
    size_t capacity;
} DARRAY, *HDARRAY;

/* Function Prototypes */

HDARRAY CreatedArray(void);
HDARRAY CreatedArrayWithCapacity(size_t capacity);
size_t AddItem(HDARRAY hDArray, DATATYPE val);
void DeleteItemIndex(HDARRAY hDArray, size_t index);
size_t DeleteItem(HDARRAY hDArray, DATATYPE val);
size_t InsertItem(HDARRAY hDArray, size_t index, DATATYPE val);
DATATYPE *FindItem(HDARRAY hDArray, DATATYPE val);
DATATYPE *FindItemLast(HDARRAY hDArray, DATATYPE val);
size_t SetCapacity(HDARRAY hDArray, size_t newCapacity);
void TrimToSize(HDARRAY hDArray);
void CloseDArray(HDARRAY hDArray);

/* Macros */

#define GetCapacity(hDArray)      ((hDArray)->capacity)
#define GetSize(hDArray)         ((hDArray)->size)
#define GetItem(hDArray, index)  ((hDArray)->pArray[index])
#define Clear(hDArray)           ((hDArray)->size = 0)

#endif

/* DynamicArray.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "DynamicArray.h"

/* Function Definitions */

HDARRAY CreatedArray(void)
{
    return CreatedArrayWithCapacity(DEF_CAPACITY);
}

HDARRAY CreatedArrayWithCapacity(size_t capacity)
{
    HDARRAY hDArray;

    if ((hDArray = (HDARRAY)malloc(sizeof(DARRAY))) == NULL)
        return NULL;

    if ((hDArray->pArray = (DATATYPE *)malloc(sizeof(DATATYPE) * capacity)) == NULL) {
        free(hDArray);
        return NULL;
    }
    hDArray->capacity = capacity;
    hDArray->size = 0;

    return hDArray;
}

size_t AddItem(HDARRAY hDArray, DATATYPE val)
{
    if (hDArray->size == hDArray->capacity && SetCapacity(hDArray, hDArray->size * 2) == DARRAY_FAILED)
        return DARRAY_FAILED;
}

```



```

    hDArray->pArray[hDArray->size++] = val;

    return hDArray->size - 1;
}

size_t InsertItem(HDARRAY hDArray, size_t index, DATATYPE val)
{
    if (hDArray->size == hDArray->capacity && SetCapacity(hDArray, hDArray->size * 2) == DARRAY_FAILED)
        return DARRAY_FAILED;

    memmove(&hDArray->pArray[index + 1], &hDArray->pArray[index], (hDArray->size - index) *
sizeof(DATATYPE));
    hDArray->pArray[index] = val;
    ++hDArray->size;

    return index;
}

void DeleteItemIndex(HDARRAY hDArray, size_t index)
{
    memmove(&hDArray->pArray[index], &hDArray->pArray[index + 1], (hDArray->size - index - 1) *
sizeof(DATATYPE));
    --hDArray->size;
}

size_t DeleteItem(HDARRAY hDArray, DATATYPE val)
{
    size_t i;

    for (i = 0; i < hDArray->size; ++i)
        if (hDArray->pArray[i] == val) {
            memmove(&hDArray->pArray[i], &hDArray->pArray[i + 1], (hDArray->size - i - 1) *
sizeof(DATATYPE));
            --hDArray->size;
            return i;
        }

    return DARRAY_FAILED;
}

DATATYPE *FindItem(HDARRAY hDArray, DATATYPE val)
{
    size_t i;

    for (i = 0; i < hDArray->size; ++i)
        if (hDArray->pArray[i] == val)
            return &hDArray->pArray[i];

    return NULL;
}

DATATYPE *FindItemLast(HDARRAY hDArray, DATATYPE val)
{
    size_t i;

    for (i = hDArray->size - 1; ; ) {
        if (hDArray->pArray[i] == val)
            return &hDArray->pArray[i];
        if (i == 0)
            break;
    }

    return NULL;
}

```



```

size_t SetCapacity(HDARRAY hDArray, size_t newCapacity)
{
    DATATYPE *pArray;

    if (newCapacity < hDArray->size)
        return DARRAY_FAILED;

    if ((pArray = (DATATYPE *)realloc(hDArray->pArray, sizeof(DATATYPE) * newCapacity)) == NULL)
        return DARRAY_FAILED;
    hDArray->pArray = pArray;
    hDArray->capacity = newCapacity;

    return newCapacity;
}

void TrimToSize(HDARRAY hDArray)
{
    hDArray->pArray = realloc(hDArray->pArray, hDArray->size);

    hDArray->capacity = hDArray->size;
}

void CloseDArray(HDARRAY hDArray)
{
    free(hDArray->pArray);
    free(hDArray);
}

/* Test.c */

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

int main(void)
{
    HDARRAY hDArray;
    DATATYPE val;
    int i;

    if ((hDArray = CreateDArrayWithCapacity(30)) == NULL) {
        fprintf(stderr, "Cannot create dynamic array!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        AddItem(hDArray, i);

    for (size_t i = 0; i < GetSize(hDArray); ++i)
        printf("%d ", GetItem(hDArray, i));
    printf("\n");

    DeleteItem(hDArray, 20);

    for (size_t i = 0; i < GetSize(hDArray); ++i)
        printf("%d ", GetItem(hDArray, i));
    printf("\n");

    printf("%d\n", *FindItem(hDArray, 7));

    printf("Capacity: %lu, Size: %lu\n", (unsigned long)GetCapacity(hDArray),
        (unsigned long)GetSize(hDArray));
    TrimToSize(hDArray);
    printf("Capacity: %lu, Size: %lu\n", (unsigned long)GetCapacity(hDArray),
        (unsigned long)GetSize(hDArray));
    Clear(hDArray);
}

```

```

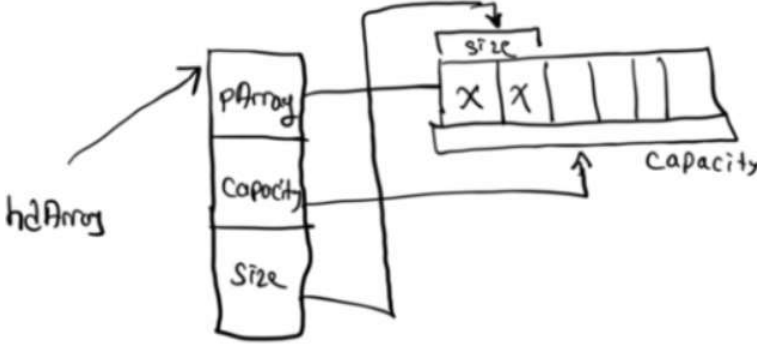
printf("Capacity: %lu, Size: %lu\n", (unsigned long)GetCapacity(hDArray),
      (unsigned long)GetSize(hDArray));

CloseDArray(hDArray);

return 0;
}

```

Yukarıdaki veri yapısını ve fonksiyonları biraz açıklayalım. Dinamik dizi DARRAY isimli bir yapıyla temsil edilmiştir. Yapının pArray elemanı dinamik tahsis edilen dizinin başlangıç adresini, capacity elemanı onun için yapılmış olan tahsisat miktarını, size elemanı ise dizinin dolu olan elemanlarının sayısını belirtir.



Eleman ekleme işlemi sırasında size değerinin capacity değerine erişip erişmediğine bakılmıştır. Eğer size değeri capacity değerine erişmişse dinamik dizi iki kat büyütülmüştür. InsertItem belli bir noktadan itibaren diziyi açar. DeleteItem ise büber. Bazı fonksiyonların makro biçiminde yazılmış olduğuna dikkat ediniz. Clear fonksiyonu dizideki tüm elemanları silmektedir. Eleman silinirken dizi kapasitesine azaltmak iyi bir teknik değildir. Çünkü genellikle bir kez büyümüş olan dizinin yeniden büyüme olasılığı vardır. Tabii programcı isterse fazla kapasiteyi TrimToSize fonksiyonuyla ortadan kaldırabilir.

Giriş kısmında da belirttiğimiz gibi nesne yönelimli dillerde bu çeşit veri yapıları tipik olarak sınıflarla temsil edilmiştir ve o dillerin kütüphanelerinde hazır olarak bulunmaktadır. Dinamik diziler C++'ın standart kütüphanesinde vector isimli sınıfla, C# ve Java'nın kütüphanelerinde ArrayList isimli sınıfla temsil edilmişlerdir. Yine diğer nesne yönelimli dillerde ve framework'lerde dinamik dizi işlemlerini yapan sınıflar vardır.

Aşağıda C++'ta vector sınıfının kullanımına ilişkin bir örnek görüyorsunuz.

```

#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v;

    for (int i = 0; i < 100; ++i)
        v.push_back(i);

    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}

```

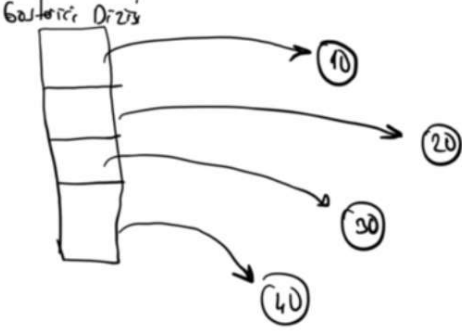
Bağlı Listeler

Veri yapıları dünyasında aralarında öncelik sonralık ilişkisinin olduğu veri yapılarına kategorik olarak liste (list) denilmektedir. Bu tanıma göre diziler de birer listedir. Bir listede elemanlar bellekte ardışıl bir biçimde bulunmazsa

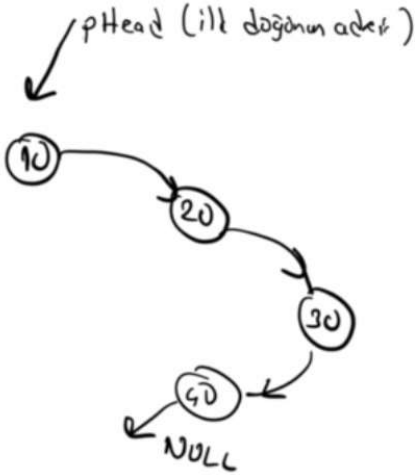
listenin elemanlarına nasıl erişilebilir? Bunun tipik olarak iki yöntemi olabilir.

- 1) Elemanların adreslerini bir gösterici dizisinde tutmak
- 2) Her elemanın bir sonraki elemanın yerini göstermesini sağlamak

Elemanları bellekte ardışıl olmayan listelerin eleman adreslerinin ayrı bir dizide tutulması çoğu kez uygun değildir. Çünkü zaten eleman ardışılığının ortadan kaldırılmasının istendiği bir durumda yeniden bir ardışıl oluşturmak uygun olmaz. Örneğin:

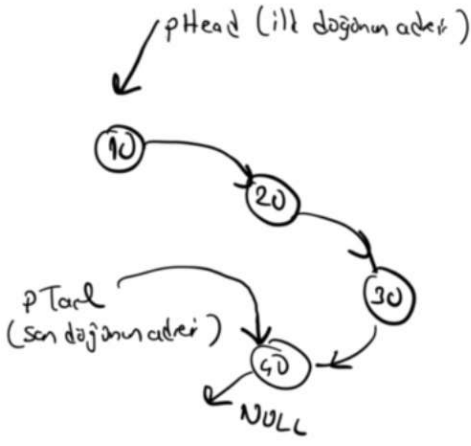


Önceki elemanın sonraki elemanı göstermesi biçiminde bir organizasyon çok daha uygundur. Örneğin:



İşte veri yapıları dünyasında önceki elemanın sonraki elemanı gösterdiği listelere bağlı listeler (linked lists) denilmektedir. Bağlı listeler elemanların ardışıl olma zorunluluğunun kaldırıldığı diziler gibidir. Başka bir deyişle "elemanları aynı türden olan fakat bellekte ardışıl biçimde bulunmak zorunda olmayan dizilere" bağlı liste diyebiliriz.

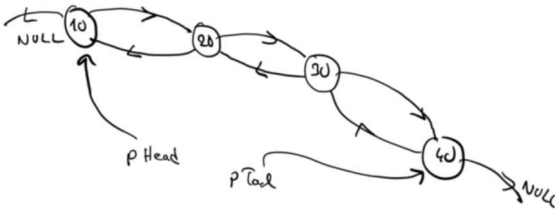
Bağlı listelerdeki her elemana düğüm (node) denilmektedir. Bağlı listenin ilk düğümünün yeri bir göstericide tutulur. tutulur. (Genellikle bu göstericiye İngilizce "head pointer" denilmektedir.) Her ne kadar zorunlu olmasa da bağlı listelerde genellikle son düğümünün yeri de tutulmaktadır. (Genellikle son düğümün yerini tutan göstericiye de İngilizce "tail pointer" denilmektedir.)



Bir bağlı listedeki düğümlerde hem o elemanda tutulacak bilgi hem de sonraki düğümün adresi bulunmalıdır. Bu durumda biz C'de bir bağlı liste düğümünü ancak bir yapı ile temsil edebiliriz. Örneğin:

```
struct NODE {
    DATATYPE val;
    struct NODE *pNext;
};
```

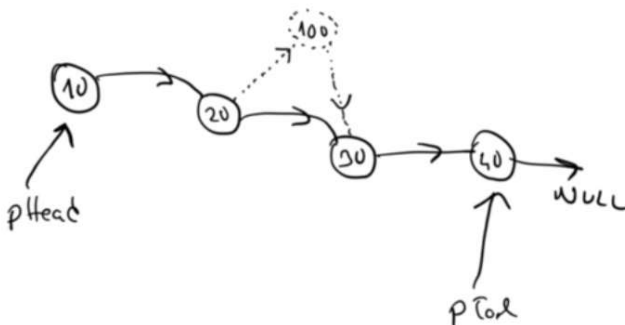
Eğer bir bağlı listede yukarıdaki gibi her düğüm yalnızca kendisinden bir sonraki düğümün adresini tutuyorsa böyle bağlı listelere "tek bağlı listeler (single linked list)" denilmektedir. Fakat eğer bir bağlı listede her düğüm hem kendinden sonraki düğümün hem de kendin önceki düğümün adresini tutuyorsa böyle bağlı listelere "çift bağlı listeler (doubly linked list)" denilmektedir.



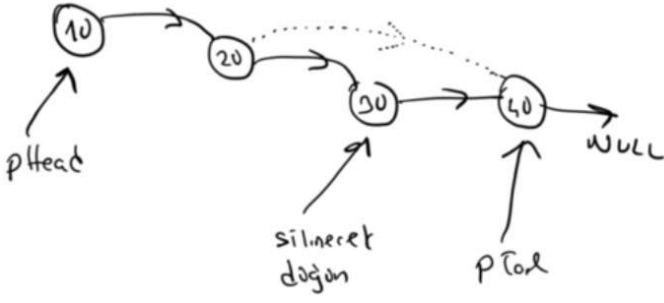
Tek bağlı listelerde bir düğümden yalnızca ileriye gidilebileceğine çift bağlı listelerde ise hem ileriye hem de geriye gidilebileceğine dikkat ediniz. Çift bağlı listelerin düğümleri C'de aşağıdaki gibi bir yapıyla temsil edilebilir:

```
struct NODE {
    DATATYPE val;
    struct NODE *pPrev;
    struct NODE *pNext;
};
```

Bağlı listelerde belirli indeksteki elemana erişmek doğrusal yani $O(N)$ karmaşıklıktadır. Çünkü o elemana erişmek için bir döngü gerekir. Fakat bağlı listelerde bir düğümün yerini biliyorsak oraya eleman insert etmek sabit karmaşıklıktadır.



Benzer biçimde eğer uygun düğümün yerini biliyorsak eleman silme işlemi de sabit karmaşıklıkta yapılabilir:



Bağlı listelerin başına ve sonuna eleman ekleme de yine sabit karmaşıklıkta bir işlemdir.

Peki tek bağlı listelerle çift bağlı listelerin arasındaki farklılıklar nelerdir? Şüphesiz çift bağlı listelerde yalnızca ileriye değil geriye de gidilebilmektedir. Örneğin biz tek bağlı listeyi etkin biçimde tersten dolaşamayız. Ancak çift bağlı listeyi dolaşabiliriz. Fakat aslında çift bağlı listelerin en önemli kullanıma gerekçeleri eleman silme işlemleri içindir. Biz çift bağlı listelerde bir elemanın adresini biliyorsak (ki genellikle biliriz) o elemanı sabit zamanlı olarak $O(1)$ karmaşıklıkta silebiliriz. Halbuki tek bağlı listelerde adresini bildiğimiz elemanı sabit zamanlı olarak $O(1)$ karmaşıklıkta silemeyiz. Ancak onun önündeki elemanı silebiliriz. Benzer biçimde çift bağlı listelerde adresini bildiğimiz bir elemanın önüne ya da gerisine eleman ekleyebiliriz. Uygulamada çift bağlı listeler tek bağlı listelere göre daha fazla kullanılmaktadır. Tabii çift bağlı listeler toplamda bellekte daha fazla yer kaplamaktadır.

Bağlı Listelere Neden Gereksinim Duyulmaktadır?

1) Bağlı listelerde elemanların bellekte ardışıl bulunması gerekmez. Böylece ardışıl bellek sorununun olduğu durumlarda bağlı listeler tercih edilirler. Ardışılık bölünmeye (fragmentation) yol açar. Bölünme de bellek verimini düşürme eğilimindedir. Özellikle çok sayıda aynı bölgeyi (örneğin hepa'i) kullanan dinamik dizilerin söz konusu olduğu durumlarda bağlı listeler belleğin daha verimli kullanılmasını sağlamaktadır. Sistem programlamada genellikle miktarı belli olmayan diziler -eğer elemanlara çok sık erişilmiyorsa- dinamik dizi yerine bağlı liste biçiminde oluşturulmaktadır.

2) Çok sayıda insert delete işleminin yapıldığı durumlarda bağlı listeler tercih edilebilir. Örneğin bir işletim sisteminde prosesler için proses kontrol blokları oluşturulmaktadır. Proses kontrol blokları çekirdeğin heap alanında dinamik olarak tahsis edilirler. Bunlar bağlı liste halinde birbirlerine bağlanırlar. Proses bittiğinde bunlar yok edileceklerdir. İşte bunların bağlı listelerden silinmesi dinamik dizilere göre çok daha kolaydır. İşletim sistemi gerçekleştiriminde benzer pek çok veri yapısı bağlı listeler biçiminde organize edilmektedir.

Bağlı Listelerle Dizilerin Karşılaştırılması

1) Belli bir indeksteki elemana erişim dizilerde sabit zamanlıdır fakat bağlı listelerde doğrusal karmaşıklıktadır. Bu nedenle elemana erişimin çok fazla yapıldığı sistemlerde normal diziler ya da dinamik diziler tercih edilmelidir.

2) Düğümü bilinen bir elemanın önüne (ya da gerisine) insert işlemi bağlı listelerde sabit zamanlıdır ancak dizilerde doğrusal karmaşıklıktadır. O halde insert işleminin yoğun yapıldığı durumlarda bağlı listeler tercih edilebilir.

3) Düğüm adresi bilinen bir elemanın silinmesi bağlı listelerde sabit karmaşıklıktadır ancak dizilerde doğrusal karmaşıklıktadır. (Tabii tek bağlı listelerde silinecek düğümün değil ondan önceki düğümün adresi bilinmelidir.) O halde silme işleminin de yoğun yapıldığı sistemlerde bağlı listeler dizilere tercih edilebilir.

4) Başa eleman ekleme dizilerde doğrusal karmaşıklıktadır ancak bağlı listelerde sabit karmaşıklıktadır. Sona eleman ekleme her iki veri yapısında da sabit karmaşıklıktadır.

5) Dizilerin ardışıl alana gereksinim duymaları bir dezavantajdır. Bağlı listeler ardışıl alana gereksinim duymazlar.

6) Bağlı listelerin bellekte toplam kapladığı alan dizilerden fazladır. (Ancak bölünme bundan çok daha büyük bir belleğin kullanım dışı kalmasına yol açan bir etkidir.)

Bağlı Listelerin Gerçekleştirilmesi

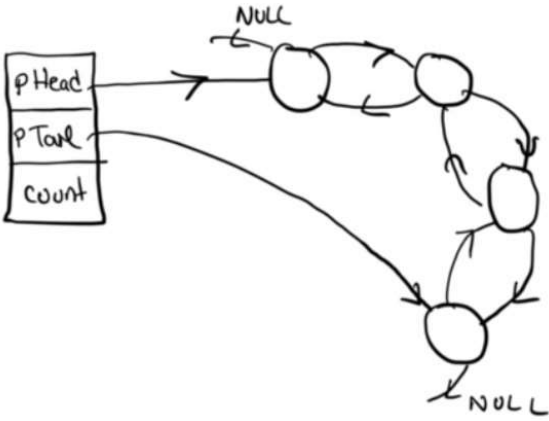
Uygulamada çift bağlı listeler tek bağlı listelerden çok daha fazla kullanılmaktadır. Çünkü yukarıda da belirtildiği gibi adresi bilinen bir düğümün silinmesi uygulamalarda çok gereksinim duyulan durumlardan biridir. Bu nedenle biz burada çift bağlı liste gerçekleştirimi üzerinde duracağız.

Çift bağlı listelerde her düğüm aşağıdaki gibi bir yapıyla temsil edilebilir:

```
typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pPrev;
    struct tagNODE *pNext;
} NODE;
```

Bağlı listeyi kontrol eden handle alanı şöyle olabilir:

```
typedef struct tagLLIST {
    NODE *pHead;
    NODE *pTail;
    size_t count;
} LLIST, *HLLIST;
```



Çift bağlı listenin örnek gerçekleştirimi şöyle olabilir:

```
/* LinkedList.h */

#ifndef LINKEDLIST_H_
#define LINKEDLIST_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE 0
#define TRUE 1

/* Type Definitions */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
    struct tagNODE *pPrev;
}
```

```

} NODE;

typedef struct tagLLIST {
    NODE *pHead;
    NODE *pTail;
    size_t count;
} LLIST, *HLLIST;

/* Function Prototypes */

HLLIST CreateLList(void);
NODE *AddItemTail(HLLIST hLList, DATATYPE val);
NODE *AddItemHead(HLLIST hLList, DATATYPE val);
NODE *InsertItemPrev(HLLIST hLList, NODE *pNode, DATATYPE val);
NODE *InsertItemNext(HLLIST hLList, NODE *pNode, DATATYPE val);
NODE *InsertItemIndex(HLLIST hLList, size_t index, DATATYPE val);
void DeleteItem(HLLIST hLList, NODE *pNode);
BOOL DeleteItemIndex(HLLIST hLList, size_t index);
DATATYPE *FindItem(HLLIST hLList, BOOL(*Compare)(const DATATYPE *));
NODE *FindItemNode(HLLIST hLList, BOOL(*Compare)(const DATATYPE *));
BOOL WalkList(HLLIST hLList, BOOL(*Proc)(DATATYPE *));
BOOL WalkListRev(HLLIST hLList, BOOL(*Proc)(DATATYPE *));
void Clear(HLLIST hLList);
void CloseList(HLLIST hLList);

/* Macros */

#define GetCount(hLList) ((hLList)->count)
#define IsEmpty(hLList) ((hLList)->count == 0)

#endif

/* LinkedList.c */

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

HLLIST CreateLList(void)
{
    HLLIST hLList;

    if ((hLList = (HLLIST)malloc(sizeof(LLIST))) == NULL)
        return NULL;

    hLList->pHead = hLList->pTail = NULL;
    hLList->count = 0;

    return hLList;
}

NODE *AddItemTail(HLLIST hLList, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (hLList->pHead != NULL)
        hLList->pTail->pNext = pNewNode;
    else
        hLList->pHead = pNewNode;

    pNewNode->pNext = NULL;
    pNewNode->pPrev = hLList->pTail;
}

```

```

    hLList->pTail = pNewNode;

    ++hLList->count;

    return pNewNode;
}

NODE *AddItemHead(HLLIST hLList, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (hLList->pHead != NULL)
        hLList->pHead->pPrev = pNewNode;
    else
        hLList->pTail = pNewNode;

    pNewNode->pNext = hLList->pHead;
    pNewNode->pPrev = NULL;
    hLList->pHead = pNewNode;

    ++hLList->count;

    return pNewNode;
}

NODE *InsertItemPrev(HLLIST hLList, NODE *pNode, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (pNode == hLList->pHead)
        hLList->pHead = pNewNode;
    else
        pNode->pPrev->pNext = pNewNode;

    pNewNode->pPrev = pNode->pPrev;
    pNewNode->pNext = pNode;
    pNode->pPrev = pNewNode;

    ++hLList->count;

    return pNewNode;
}

NODE *InsertItemNext(HLLIST hLList, NODE *pNode, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (hLList->pTail == pNode)
        hLList->pTail = pNewNode;
    else
        pNode->pNext->pPrev = pNewNode;

    pNewNode->pPrev = pNode;
    pNewNode->pNext = pNode->pNext;
}

```



```

    pNode->pNext = pNewNode;

    ++hLList->count;

    return pNewNode;
}

NODE *InsertItemIndex(HLLIST hLList, size_t index, DATATYPE val)
{
    size_t i;
    NODE *pNode;

    if (index > hLList->count)
        return NULL;

    if (index == hLList->count)
        return AddItemTail(hLList, val);

    pNode = hLList->pHead;
    for (i = 0; i < index; ++i)
        pNode = pNode->pNext;

    return InsertItemPrev(hLList, pNode, val);
}

void DeleteItem(HLLIST hLList, NODE *pNode)
{
    if (pNode == hLList->pHead)
        hLList->pHead = pNode->pNext;
    else
        pNode->pPrev->pNext = pNode->pNext;

    if (pNode == hLList->pTail)
        hLList->pTail = pNode->pPrev;
    else
        pNode->pNext->pPrev = pNode->pPrev;

    --hLList->count;

    free(pNode);
}

BOOL DeleteItemIndex(HLLIST hLList, size_t index)
{
    size_t i;
    NODE *pNode;

    if (index >= hLList->count)
        return FALSE;

    pNode = hLList->pHead;
    for (i = 0; i < index; ++i)
        pNode = pNode->pNext;

    DeleteItem(hLList, pNode);

    return TRUE;
}

DATATYPE *FindItem(HLLIST hLList, BOOL(*Compare)(const DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLList->pHead; pNode != NULL; pNode = pNode->pNext)
        if (Compare(&pNode->val))
            return &pNode->val;
}

```

```

    return NULL;
}

NODE *FindItemNode(HLLIST hLList, BOOL(*Compare)(const DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLList->pHead; pNode != NULL; pNode = pNode->pNext)
        if (Compare(&pNode->val))
            return pNode;

    return NULL;
}

BOOL WalkList(HLLIST hLList, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLList->pHead; pNode != NULL; pNode = pNode->pNext)
        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

BOOL WalkListRev(HLLIST hLList, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLList->pTail; pNode != NULL; pNode = pNode->pPrev)
        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

void Clear(HLLIST hLList)
{
    NODE *pNode, *pTempNode;

    pNode = hLList->pHead;
    while (pNode != NULL) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    hLList->pHead = hLList->pTail = NULL;
    hLList->count = 0;
}

void CloseList(HLLIST hLList)
{
    NODE *pNode, *pTempNode;

    pNode = hLList->pHead;
    while (pNode != NULL) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    free(hLList);
}

```

```

/* Test.c */

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

BOOL DispItem(DATATYPE *pVal)
{
    printf("%d ", *pVal);

    return TRUE;
}

BOOL MyComparer(const DATATYPE *pVal)
{
    if (*pVal == 5)
        return TRUE;

    return FALSE;
}

int main(void)
{
    HLLIST hLList;
    int i;
    NODE *pNode;
    DATATYPE *pVal;

    if ((hLList = CreateLList()) == NULL) {
        fprintf(stderr, "cannot create linked list\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i) {
        if ((pNode = AddItemTail(hLList, i)) == NULL) {
            fprintf(stderr, "cannot add node!..\n");
            exit(EXIT_FAILURE);
        }
    }

    WalkList(hLList, DispItem);
    printf("\n");

    DeleteItemIndex(hLList, 7);

    WalkList(hLList, DispItem);
    printf("\n");

    if ((pVal = FindItem(hLList, MyComparer)) == NULL) {
        fprintf(stderr, "cannot find item!");
        exit(EXIT_FAILURE);
    }

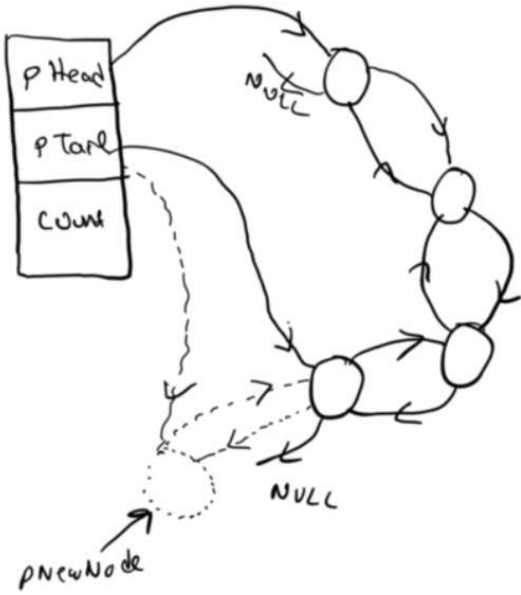
    printf("Found: %d\n", *pVal);
    printf("Total Item: %u\n", GetCount(hLList));

    CloseList(hLList);

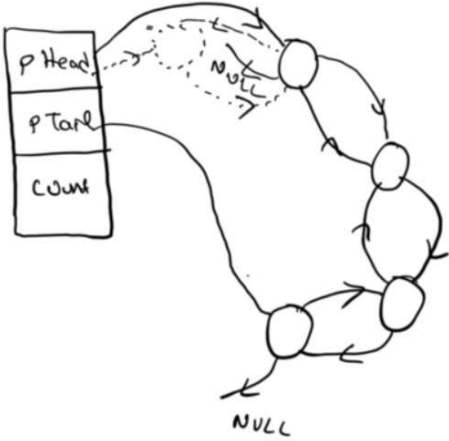
    return 0;
}

```

Sona eleman ekleme işleminde handle alanındaki pTail göstericisi eklenen elemanı gösterecek biçime getirilmektedir. Listenin tamamen boş olması durumu da ayrıca kontrol edilmiştir:



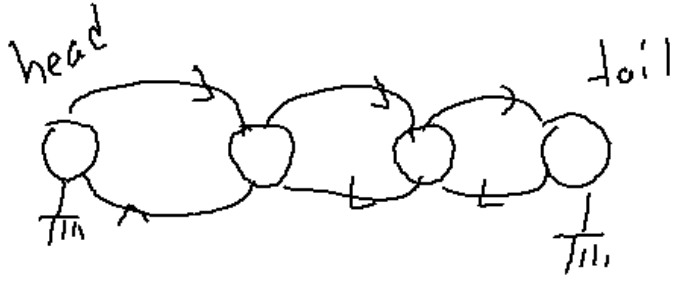
Listenin başına eleman eklenmesi de benzer bir çaba gerektirmektedir:



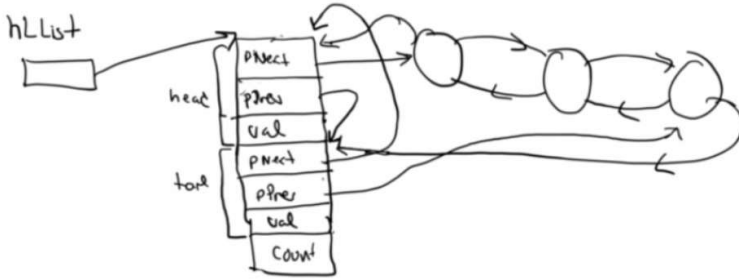
Çift bağlı liste gerçekleştiriminde handle alanında ilk ve son düğümlerin adresleri değil de düğümün kendisi tutulursa bazı özel durumlar özel olmaktan çıkartılabilir. Dolayısıyla tasarım daha sadeleşebilir. Tabii handle alanında tutulan ilk ve son düğümler sözde düğümler olur. Bu tasarım düğümdeki veri miktarının küçük olduğu durumda tercih edilebilir. Örneğin bu durumda handle alanı şöyle olacaktır:

```
typedef struct tagLLIST {
    NODE head;
    NODE tail;
    size_t count;
} LLIST, *HLLIST;
```

Burada son düğümün pNext göstericisi NULL'değerini değil handle alanındaki tail düğümünü gösterecektir. Benzer biçimde ilk düğümün pPrev göstericisi de aslında handle alanındaki head düğümünü gösterir. Yani bu tasarımda head ve tail düğümleri sanki baştan listeye ekliymiş gibi ele alınmaktadır:



Bu şekli biraz daha ayrıntılı biçimde şöyle de çizabiliriz:



Bu tasarımda handle alanındaki head ve tail düğümleri sözde (pseudo) head ve tail düğümleridir. Başka bir deyişle handle alanındaki head düğümü aslında ilk düğümün öncesinde bulunan sözde bir düğümdür. Benzer biçimde tail düğümü de aslında son düğümün sonradan bulunan sözde bir düğümdür. Fakat bu sözde düğümlerin yerleştirilmesi ekleme ve silme işlemlerinde bazı özel durumları ortadan kaldırarak daha sade bir kodun oluşmasına yol açarlar. Bu tasarımın örnek bir gerçekleştirimi şöyle olabilir:

```

/* LinkedList.h */

#ifndef LINKEDLIST_H_
#define LINKEDLIST_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE    0
#define TRUE     1

/* Type Definitions */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
    struct tagNODE *pPrev;
} NODE;

typedef struct tagLLIST {
    NODE head;
    NODE tail;
    size_t count;
} LLIST, *HLLIST;

/* Function Prototypes */

HLLIST CreateLList(void);
NODE *AddItemTail(HLLIST hList, DATATYPE val);

```

```

NODE *AddItemHead(HLLIST hLList, DATATYPE val);
NODE *InsertItemPrev(HLLIST hLList, NODE *pNode, DATATYPE val);
NODE *InsertItemNext(HLLIST hLList, NODE *pNode, DATATYPE val);
NODE *InsertItemIndex(HLLIST hLList, size_t index, DATATYPE val);
void DeleteItem(HLLIST hLList, NODE *pNode);
BOOL DeleteItemIndex(HLLIST hLList, size_t index);
DATATYPE *FindItem(HLLIST hLList, BOOL(*Compare)(const DATATYPE *));
NODE *FindItemNode(HLLIST hLList, BOOL(*Compare)(const DATATYPE *));
BOOL WalkList(HLLIST hLList, BOOL(*Proc)(DATATYPE *));
BOOL WalkListRev(HLLIST hLList, BOOL(*Proc)(DATATYPE *));
void Clear(HLLIST hLList);
void CloseList(HLLIST hLList);

/* Macros */

#define GetCount(hLList) ((hLList)->count)
#define IsEmpty(hLList) ((hLList)->count == 0)

#endif

/* LinkedList.c */

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

HLLIST CreateLList(void)
{
    HLLIST hLList;

    if ((hLList = (HLLIST)malloc(sizeof(LLIST))) == NULL)
        return NULL;

    hLList->head.pNext = &hLList->tail;
    hLList->head.pPrev = &hLList->tail;
    hLList->tail.pNext = &hLList->head;
    hLList->tail.pPrev = &hLList->head;

    hLList->count = 0;

    return hLList;
}

NODE *AddItemTail(HLLIST hLList, DATATYPE val)
{
    return InsertItemPrev(hLList, &hLList->tail, val);
}

NODE *AddItemHead(HLLIST hLList, DATATYPE val)
{
    return InsertItemNext(hLList, &hLList->head, val);
}

NODE *InsertItemPrev(HLLIST hLList, NODE *pNode, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    pNewNode->pNext = pNode;
    pNewNode->pPrev = pNode->pPrev;
    pNode->pPrev->pNext = pNewNode;
    pNode->pPrev = pNewNode;
}

```

```

    ++hLList->count;

    return pNewNode;
}

NODE *InsertItemNext(HLLIST hLList, NODE *pNode, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    pNewNode->pPrev = pNode;
    pNewNode->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNewNode;
    pNode->pNext = pNewNode;

    ++hLList->count;

    return pNewNode;
}

NODE *InsertItemIndex(HLLIST hLList, size_t index, DATATYPE val)
{
    size_t i;
    NODE *pNode;

    if (index > hLList->count)
        return NULL;

    if (index == hLList->count)
        return AddItemTail(hLList, val);

    pNode = hLList->head.pNext;
    for (i = 0; i < index; ++i)
        pNode = pNode->pNext;

    return InsertItemPrev(hLList, pNode, val);
}

void DeleteItem(HLLIST hLList, NODE *pNode)
{
    pNode->pPrev->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNode->pPrev;

    --hLList->count;

    free(pNode);
}

BOOL DeleteItemIndex(HLLIST hLList, size_t index)
{
    size_t i;
    NODE *pNode;

    if (index >= hLList->count)
        return FALSE;

    pNode = hLList->head.pNext;
    for (i = 0; i < index; ++i)
        pNode = pNode->pNext;

    DeleteItem(hLList, pNode);

    return TRUE;
}

```

```

}

DATATYPE *FindItem(HLLIST hLList, BOOL(*Compare)(const DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLList->head.pNext; pNode != &hLList->tail; pNode = pNode->pNext)
        if (Compare(&pNode->val))
            return &pNode->val;

    return NULL;
}

NODE *FindItemNode(HLLIST hLList, BOOL(*Compare)(const DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLList->head.pNext; pNode != &hLList->tail; pNode = pNode->pNext)
        if (Compare(&pNode->val))
            return pNode;

    return NULL;
}

BOOL WalkList(HLLIST hLList, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLList->head.pNext; pNode != &hLList->tail; pNode = pNode->pNext)
        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

BOOL WalkListRev(HLLIST hLList, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLList->tail.pPrev; pNode != &hLList->head; pNode = pNode->pPrev)
        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

void Clear(HLLIST hLList)
{
    NODE *pNode, *pTempNode;

    pNode = hLList->head.pNext;
    while (pNode != &hLList->tail) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    hLList->head.pNext = &hLList->tail;
    hLList->tail.pPrev = &hLList->head;

    hLList->count = 0;
}

void CloseList(HLLIST hLList)
{
    NODE *pNode, *pTempNode;

```



```

    pNode = hLList->head.pNext;
    while (pNode != &hLList->tail) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    free(hLList);
}

/* Test.c */

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

BOOL DispItem(DATATYPE *pVal)
{
    printf("%d ", *pVal);

    return TRUE;
}

BOOL MyComparer(const DATATYPE *pVal)
{
    if (*pVal == 5)
        return TRUE;

    return FALSE;
}

int main(void)
{
    HLLIST hLList;
    int i;
    NODE *pNode;
    NODE *pInsNode;
    DATATYPE *pVal;

    if ((hLList = CreateLList()) == NULL) {
        fprintf(stderr, "cannot create linked list\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i) {
        if ((pNode = AddItemTail(hLList, i)) == NULL) {
            fprintf(stderr, "cannot add node!..\n");
            exit(EXIT_FAILURE);
        }
        if (i == 6)
            pInsNode = pNode;
    }

    WalkList(hLList, DispItem);
    printf("\n");

    InsertItemPrev(hLList, pInsNode, 1000);

    WalkList(hLList, DispItem);
    printf("\n");

    Clear(hLList);

    WalkList(hLList, DispItem);
    printf("\n");
}

```

```

CloseList(hLList);

return 0;
}

```

Bağlı listelere ilişkin verdiğimiz iki örnekte de bağlı listelerin içerisinde tıpkı dinamik dizi örneğinde olduğu gibi DATATYPE türünden nesnelere olduğu varsaydık. Buradaki DATATYPE örneklerimizde int olarak typedef edilmiştir. Ancak DATATYPE bir yapı biçiminde typedef edilirse fonksiyonların parametrelerinin DATATYPE * türünden olması daha anlamlı olur. Örneğin yukarıdaki kodlarda AddItemTail fonksiyonunun prototipi şöyledir:

```

NODE *AddItemTail(HLLIST hLList, DATATYPE val);

```

Eğer DATATYPE bir yapıysa bu fonksiyonun aşağıdaki gibi bir prototipe sahip olması daha uygun olacaktır:

```

NODE *AddItemTail(HLLIST hLList, const DATATYPE *val);

```

Çünkü yapıların fonksiyonlara adres yoluyla aktarılması (call by reference) değer yoluyla aktarılmasından (call by value) daha etkindir. Pekiyi bu biçimde oluşturduğumuz bağlı listelerde aynı projede farklı türlerden (yani farklı DATATYPE'lardan) birden fazla bağlı liste oluşturabilir miyiz? Yanıt maalesef hayır. Çünkü DATATYPE typedef edildiğine göre programda bir daha değiştirilemez. Böylece biz aynı projede aynı türden birden fazla bağlı liste oluşturabiliriz ancak farklı türlerden oluşturamayız. Aynı durum burada ele aldığımız diğer veri yapıları için de söz konusudur. İşte aynı projede farklı türlerden veri yapılarının oluşturulabilmesi için onların genelleştirilmesi gerekmektedir. Genelleştirme işlemi izleyen bölümlerde ele alınacaktır.

Kuyruk (Queue) Veri Yapısı

Kuyruk FIFO prensibiyle çalışan bir veri yapısıdır. Kuyrukla ilgili iki temel işlem söz konusudur: "Kuyruğa eleman ekleme" ve "kuyruktan eleman alma". Kuyruklarda araya eleman ekleme ve herhangi bir elemanı silme işlemi anlamlı değildir. Kuyruğa eleman eklendiğinde eleman sona eklenir. Kuyruktan eleman alındığında baştaki eleman alınır. Kuyruktan alınan eleman aynı zamanda silinmektedir. Kuyruğa eleman yerleştirme işlemi için geleneksel olarak İngilizce "put" ya da "enqueue" sözcükleri eleman almak için de "get" ya da "dequeue" sözcükleri tercih edilmektedir.

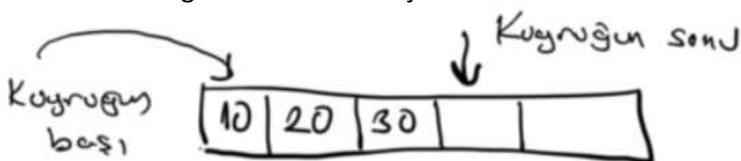
Kuyruk Veri Yapısına Neden Gereksinim Duyulmaktadır?

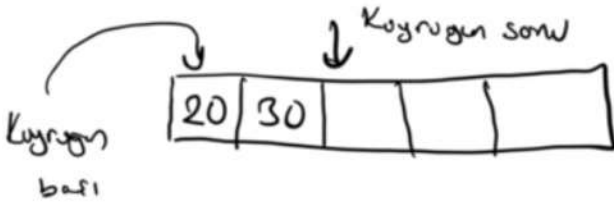
Kuyruklar genellikle bilgilerin geçici olarak sırası bozulmadan bekletilmesi için kullanılırlar. (Yani tipik olarak kuyruklar tampon alanların gerçekleştirilmesinde kullanılmaktadır.) Kuyruk veri yapısıyla pek çok yerde karşılaşılabilir. Örneğin arabalı vapura binme ve inme kuyruğu, yemek kuyrukları gerçek yaşamda karşılaştığımız kuyruklardır. Klavyeden basılan tuşlar işletim sistemi tarafından bir klavye tamponuna yerleştirilir. Bu da bir kuyruk sistemidir. Yazıcıya birden fazla iş gönderilirse yazıcı bunları bir kuyruk sisteminde saklar ve yazdırma işlemini bu sıraya göre gerçekleştirir. Ya da örneğin işletim sisteminin blokedeki prosesleri ya da thread'leri beklettiği veri yapıları da birer kuyruk sistemi gibidir.

Kuyruk Veri Yapısının Gerçekleştirimi

Kuyruklar üç biçimde gerçekleştirilebilirler:

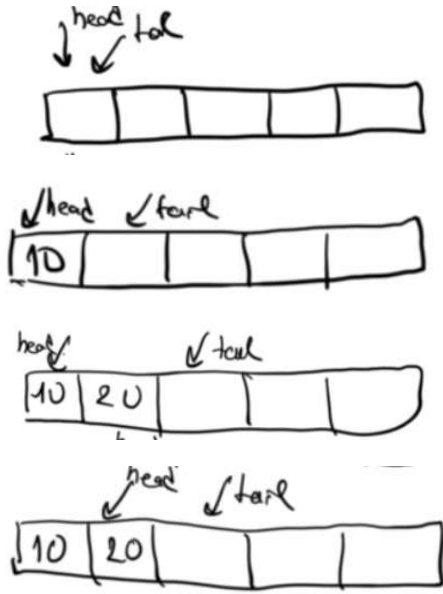
En basit yöntem dizi yöntemidir. Bu yöntemde kuyruk için bir dizi yaratılır. Bir index kuyruğun sonunu tutar. Eleman kuyruğa yerleştirilmek istendiğinde bu indeksin gösterdiği yere yerleştirilir ve indeks bir artırılır. Kuyruktan eleman alınmak istendiğinde de dizinin başındaki eleman alınır. Dizi ve indeks bir kaydırılır.



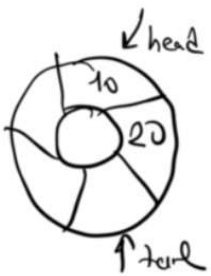


Bu yöntemde kuyruğa eleman yerleştirmek $O(1)$ karmaşıklıkta, kuyruktan eleman almak ise $O(N)$ karmaşıklıkta yapılmaktadır. Uygulamada diğer iki yöntem daha iyi olduğu için bu yöntem tercih edilmemektedir.

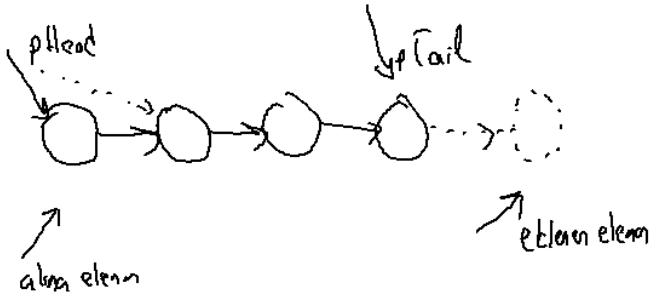
İkinci yöntemde (buna döngüsel kuyruk gerçekleştirimi de denilmektedir) yine kuyruk için bir dizi yaratılır. Kuyruğun başı ve sonu iki gösterici ya da indeksle tutulur. Geleneksel olarak bunlardan birine "head" göstericisi, diğerine "tail" göstericisi denilmektedir. Eleman "tail" göstericisinin gösterdiği yere yerleştirilir ve "tail" göstericisi bir artırılır. Eleman "head" göstericisinin gösterdiği yerden alınır ve "head" göstericisi bir artırılır. Tabii dizinin sonuna gelindiğinde yeniden başa dönülmelidir.



Yukarıda da belirtildiği gibi bu yöntemde "head" ve "tail" göstericileri dizinin sonuna geldiğinde yeniden başa çekilir. Böylece sanki dizi döngüsel yapıdaymış gibi bir etki oluşturulur:



Üçüncü yöntem bağlı listeye kuyruk oluşturma yöntemidir. Buna "linked list queue" da denilmektedir. Bu yöntemde eleman bağlı listenin sonuna eklenir, başından alınır. Örneğin:



Bu yöntemde kuyruk dinamik olarak büyütülüp küçültülebilmektedir. Halbuki döngüsel kuyruk sisteminde kuyruk için gereken alan işin başında tahsis edilmek zorundadır. Fakat dinamik tahsisatların da belli bir zamansal maliyeti vardır. Dinamik tahsisat sistemleri genel olarak $O(N)$ karmaşıklıkta tahsisat yapmaktadır.

Döngüsel yöntemle kuyruk aşağıdaki gibi oluşturulabilir:

```

/* Queue.h */

#ifndef QUEUE_H_
#define QUEUE_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE          0
#define TRUE           1

/* Type Declarations */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagQUEUE {
    DATATYPE *pQueue;
    size_t size;
    size_t head;
    size_t tail;
    size_t count;
} QUEUE, *HQUEUE;

/* Function Prototypes */

HQUEUE CreateQueue(size_t size);
BOOL PutItem(HQUEUE hQueue, DATATYPE val);
BOOL GetItem(HQUEUE hQueue, DATATYPE *pVal);
BOOL Walk(HQUEUE hQueue, BOOL(*Proc)(DATATYPE *));
void Clear(HQUEUE hQueue);
void CloseQueue(HQUEUE hQueue);

/* Macros */

#define IsEmpty(hQueue)      ((hQueue)->count == 0)
#define GetCount(hQueue)    ((hQueue)->count)
#define GetSize(hQueue)     ((hQueue)->size)

#endif

/* Queue.c */

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

```

```

HQUEUE CreateQueue(size_t size)
{
    HQUEUE hQueue;

    if ((hQueue = (HQUEUE)malloc(sizeof(Queue))) == NULL)
        return NULL;

    if ((hQueue->pQueue = (DATATYPE *)malloc(sizeof(DATATYPE) * size)) == NULL) {
        free(hQueue);
        return NULL;
    }

    hQueue->size = size;
    hQueue->head = hQueue->tail = 0;
    hQueue->count = 0;

    return hQueue;
}

BOOL PutItem(HQUEUE hQueue, DATATYPE val)
{
    if (hQueue->count == hQueue->size)
        return FALSE;

    hQueue->pQueue[hQueue->tail++] = val;
    hQueue->tail %= hQueue->size;

    ++hQueue->count;

    return TRUE;
}

BOOL GetItem(HQUEUE hQueue, DATATYPE *pVal)
{
    if (hQueue->count == 0)
        return FALSE;

    *pVal = hQueue->pQueue[hQueue->head++];
    hQueue->head %= hQueue->size;

    --hQueue->count;

    return TRUE;
}

BOOL Walk(HQUEUE hQueue, BOOL (*Proc)(DATATYPE *))
{
    size_t count;
    size_t i;

    count = hQueue->count;
    i = hQueue->head;
    while (count-- > 0) {
        if (!Proc(&hQueue->pQueue[i++]))
            return FALSE;
        i %= hQueue->size;
    }

    return TRUE;
}

void Clear(HQUEUE hQueue)
{
    hQueue->head = hQueue->tail = 0;
    hQueue->count = 0;
}

```

```

void CloseQueue(HQUEUE hQueue)
{
    free(hQueue->pQueue);
    free(hQueue);
}

/* Test.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Queue.h"

#define MAX_CMD_LINE    1024
#define MAX_PARAMS      10

void parse_cmdline(void);
void proc_get(void);
void proc_put(void);
void proc_clear(void);
void proc_count(void);
void proc_disp(void);

typedef struct tagCMD {
    const char *cmdText;
    void(*Proc)(void);
} CMD;

CMD g_cmds[] = {
    {"get", proc_get}, {"put", proc_put}, {"clear", proc_clear},
    {"count", proc_count}, {"disp", proc_disp}, {NULL, NULL}
};

HQUEUE g_hQueue;
char g_cmdLine[MAX_CMD_LINE];
char *g_params[MAX_PARAMS];
int g_nparams;

int main(void)
{
    int i;

    if ((g_hQueue = CreateQueue(10)) == NULL) {
        fprintf(stderr, "cannot create queue!..\n");
        exit(EXIT_FAILURE);
    }

    for (;;) {
        printf("CSD>");
        fgets(g_cmdLine, MAX_CMD_LINE, stdin);
        parse_cmdline();
        if (g_nparams == 0)
            continue;
        if (!strcmp(g_params[0], "quit"))
            break;
        for (i = 0; g_cmds[i].cmdText != NULL; ++i)
            if (!strcmp(g_params[0], g_cmds[i].cmdText)) {
                g_cmds[i].Proc();
                break;
            }
        if (g_cmds[i].cmdText == NULL)
            printf("command not found: %s\n", g_params[0]);
    }

    CloseQueue(g_hQueue);
}

```

```

    return 0;
}

void parse_cmdline(void)
{
    char *str;

    g_nparams = 0;
    for (str = strtok(g_cmdLine, " \t\n"); str != NULL; str = strtok(NULL, " \t\n"))
        g_params[g_nparams++] = str;
}

void proc_get(void)
{
    DATATYPE val;

    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    if (!GetItem(g_hQueue, &val)) {
        printf("empty queue!\n");
        return;
    }
    printf("%d\n", val);
}

void proc_put(void)
{
    int val;

    if (g_nparams == 1) {
        printf("too few parameters!\n");
        return;
    }

    if (g_nparams > 2) {
        printf("too many parameters!\n");
        return;
    }

    val = atoi(g_params[1]);
    if (!PutItem(g_hQueue, val)) {
        printf("queue is full!\n");
        return;
    }
}

void proc_clear(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    Clear(g_hQueue);
}

void proc_count(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }

    printf("%u\n", GetCount(g_hQueue));
}

```

```

}

static BOOL disp(DATATYPE *val)
{
    printf("%d ", *val);

    return TRUE;
}

void proc_disp(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    Walk(g_hQueue, disp);
    if (!GetCount(g_hQueue))
        printf("Empty queue");
    printf("\n");
}

```

Kuyruk veri yapısının bağlı listelerle gerçekleştiriminde bağlı listenin çift bağlı (double linked) olmasına gerek yoktur. Örnek bir gerçekleştirim şöyle olabilir:

```

/* Queue.h */

#ifndef QUEUE_H_
#define QUEUE_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE      0
#define TRUE       1

/* Type Declarations */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
} NODE;

typedef struct tagQUEUE {
    NODE *pHead;
    NODE *pTail;
    size_t count;
} QUEUE, *HQUEUE;

/* Function Prototypes */

HQUEUE CreateQueue(void);
BOOL PutItem(HQUEUE hQueue, DATATYPE val);
BOOL GetItem(HQUEUE hQueue, DATATYPE *pVal);
BOOL Walk(HQUEUE hQueue, BOOL(*Proc)(DATATYPE *));
void Clear(HQUEUE hQueue);
void CloseQueue(HQUEUE hQueue);

/* Macros */

#define IsEmpty(hQueue)      ((hQueue)->count == 0)
#define GetCount(hQueue)    ((hQueue)->count)

```



```

#endif

/* Queue.c */

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

HQUEUE CreateQueue(void)
{
    HQUEUE hQueue;

    if ((hQueue = (HQUEUE)malloc(sizeof(Queue))) == NULL)
        return NULL;

    hQueue->pHead = hQueue->pTail = NULL;
    hQueue->count = 0;

    return hQueue;
}

BOOL PutItem(HQUEUE hQueue, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return FALSE;
    pNewNode->val = val;
    pNewNode->pNext = NULL;

    if (hQueue->pTail == NULL)
        hQueue->pHead = pNewNode;
    else
        hQueue->pTail->pNext = pNewNode;
    hQueue->pTail = pNewNode;

    ++hQueue->count;

    return TRUE;
}

BOOL GetItem(HQUEUE hQueue, DATATYPE *pVal)
{
    NODE *pNode;

    if (hQueue->pHead == NULL)
        return FALSE;

    pNode = hQueue->pHead;
    hQueue->pHead = pNode->pNext;
    if (pNode->pNext == NULL)
        hQueue->pTail = NULL;
    *pVal = pNode->val;

    free(pNode);

    --hQueue->count;

    return TRUE;
}

BOOL Walk(HQUEUE hQueue, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;

    for (pNode = hQueue->pHead; pNode != NULL; pNode = pNode->pNext)

```

```

        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

void Clear(HQUEUE hQueue)
{
    NODE *pTempNode, *pNode;

    pNode = hQueue->pHead;

    while (pNode != NULL) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    hQueue->pHead = hQueue->pTail = NULL;
    hQueue->count = 0;
}

void CloseQueue(HQUEUE hQueue)
{
    NODE *pTempNode, *pNode;

    pNode = hQueue->pHead;

    while (pNode != NULL) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    free(hQueue);
}

/* Test.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Queue.h"

#define MAX_CMD_LINE    1024
#define MAX_PARAMS      10

void parse_cmdline(void);
void proc_get(void);
void proc_put(void);
void proc_clear(void);
void proc_count(void);
void proc_disp(void);

typedef struct tagCMD {
    const char *cmdText;
    void(*Proc)(void);
} CMD;

CMD g_cmds[] = {
    {"get", proc_get}, {"put", proc_put}, {"clear", proc_clear},
    {"count", proc_count}, {"disp", proc_disp}, {NULL, NULL}
};

HQUEUE g_hQueue;
char g_cmdLine[MAX_CMD_LINE];

```

```

char *g_params[MAX_PARAMS];
int g_nparams;

int main(void)
{
    int i;

    if ((g_hQueue = CreateQueue()) == NULL) {
        fprintf(stderr, "cannot create queue!..\n");
        exit(EXIT_FAILURE);
    }

    for (;;) {
        printf("CSD>");
        fgets(g_cmdLine, MAX_CMD_LINE, stdin);
        parse_cmdline();
        if (g_nparams == 0)
            continue;
        if (!strcmp(g_params[0], "quit"))
            break;
        for (i = 0; g_cmds[i].cmdText != NULL; ++i)
            if (!strcmp(g_params[0], g_cmds[i].cmdText)) {
                g_cmds[i].Proc();
                break;
            }
        if (g_cmds[i].cmdText == NULL)
            printf("command not found: %s\n", g_params[0]);
    }

    CloseQueue(g_hQueue);

    return 0;
}

void parse_cmdline(void)
{
    char *str;

    g_nparams = 0;
    for (str = strtok(g_cmdLine, " \t\n"); str != NULL; str = strtok(NULL, " \t\n"))
        g_params[g_nparams++] = str;
}

void proc_get(void)
{
    DATATYPE val;

    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    if (!GetItem(g_hQueue, &val)) {
        printf("empty queue!\n");
        return;
    }
    printf("%d\n", val);
}

void proc_put(void)
{
    int val;

    if (g_nparams == 1) {
        printf("too few parameters!\n");
        return;
    }
}

```

```

if (g_nparams > 2) {
    printf("too many parameters!\n");
    return;
}

val = atoi(g_params[1]);
if (!PutItem(g_hQueue, val)) {
    printf("queue is full!\n");
    return;
}
}

void proc_clear(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    Clear(g_hQueue);
}

void proc_count(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }

    printf("%u\n", GetCount(g_hQueue));
}

static BOOL disp(DATATYPE *val)
{
    printf("%d ", *val);

    return TRUE;
}

void proc_disp(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    Walk(g_hQueue, disp);
    if (!GetCount(g_hQueue))
        printf("Empty queue");
    printf("\n");
}
}

```

Pekiye döngüsel kuyruk sistemi ile bağlı listeli kuyruk sistemi arasındaki farklar nelerdir? Döngüsel kuyruk sisteminde tüm tahsisat için başında bir hamlede yapılmaktadır. Dolayısıyla her eleman eklendiğinde yeniden tahsisat yapılmaz. Bu nedenle döngüsel kuyruk sistemi bağlı listeli kuyruk sistemine daha hızlı olma eğilimindedir. Tabii eğer kuyrukta saklanacak bilgiler (yani sizeof(DATATYPE)) çok büyükse döngüsel kuyruk sisteminde işin başında tüm kuyruk için yapılan tahsisat da yüksek olur. Kısıtlı belleklerin söz konusu olduğu sistemlerde bu durum göz önüne alınmalıdır. Öte yandan bağlı listeli kuyruk sisteminde kuyruğun uzunluğu baştan sınırlı değildir. Heap alanı yettikçe kuyruğa yeni eleman eklenebilir. O halde eleman sayısı belli olan ve bu sayının çok büyük olmadığı sistemlerde döngüsel kuyruk sistemi, diğer sistemlerde bağlı listeli kuyruk sistemi tercih edilebilir. Bellek tahsisat işleminin pek çok durumda $O(N)$ karmaşıklıkta olduğu düşünülürse kuyruğa eleman ekleme işlemi döngüsel kuyruk sisteminde $O(1)$ karmaşıklıkta, bağlı listeli kuyruk sisteminde ise $O(N)$ karmaşıklıkta. Kuyruktan eleman alma işlemi her iki sistemde de $O(1)$ karmaşıklıkta yapılabilmektedir.

Stack Veri Yapısı

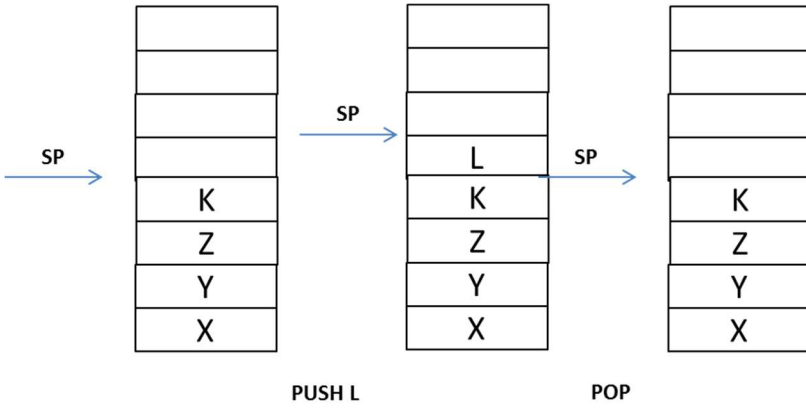
Stack LIFO prensibiyle çalışan bir kuyruk sistemidir. Yani stack'e son yerleştirilen bilgi ilk olarak alınır. Geleneksel olarak stack'e eleman ekleyen fonksiyonlar İngilizce "push", stack'ten eleman alan fonksiyonlar "pop" biçiminde isimlendirilmektedir. Push ve pop işlemleri $O(1)$ karmaşıklıkta gerçekleştirilmektedir.

Stack Veri Yapısına Neden Gereksinim Duyulmaktadır?

Stack sistemleriyle gerçek hayatta da karşılaşılmaktadır. Örneğin asansöre son binenler (genellikle) ilk inerler. Tabakları üst üste koyduğumuzda en üsttekini önce alırız. Alışveriş arabalarındaki sistem benzerdir. Programlama da da stack sistemleriyle karşılaşılmaktadır. Örneğin "undo" mekanizması bir stack sistemini kullanır. (Son yapılan değişikliği ilk geri alırız). Ya da örneğin bir pencere aktifken onu kapattığımızda en üstteki pencere aktif yapılmaktadır. Bu da stack sistemini çağırır. Bazı algoritmaların gerçekleştiriminde de stack sistemleri kullanılmaktadır. Örneğin aslında özyineleme içeren algoritmalar özyineleme olmadan yapay bir stack'le de gerçekleştirilebilirler. Parsing algoritmaları stack kullanan algoritmalar için tipik örneklerdendir. Stack sistemleri mikroişlemcilerin çalışabilmesi için mutlak bulunması gereken bir mekanizmadır.

Stack Veri Yapısının Gerçekleştirilmesi

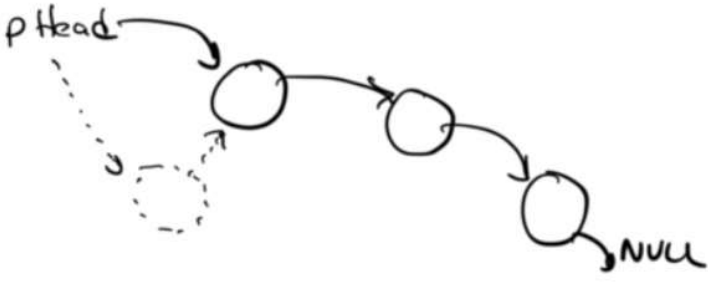
Stack sistemleri de dizilerle ve bağlı listelerle gerçekleştirilebilir. Dizilerle gerçekleştirimde stack bir dizi olarak oluşturulur. Stack'in aktif noktası bir göstericiyle gösterilir. Bu göstericiye geleneksel olarak "stack gösterici (stack pointer)" denilmektedir. Push işlemi sırasında stack göstericisi bir azaltılır ve eleman stack göstericisinin gösterdiği yere yerleştirilir. Pop işleminde de stack göstericisinin gösterdiği yerdeki eleman alınır ve stack göstericisi bir artırılır. İşin başında stack göstericisi dizinin sonundadır.



Şüphesiz push ve pop işlemleri aslında tam ters olarak da yapılabilir. Yani örneğin işin başında stack göstericisi dizinin başını gösterebilir. Push işlemi sırasında eleman stack göstericisinin gösterdiği yere atanıp stack göstericisi bir artırılabilir. Pop işlemi sırasında da önce stack göstericisi bir azaltılıp oradaki eleman alınabilir. Fakat geleneksel durum stack göstericisinin başlangıçta dizinin sonunu göstermesi durumudur.

Eğer stack'e çok fazla eleman yerleştirilirse (push edilirse) bu durumda stack yukarıdan taşar (stack overflow). Stack'ten çok fazla eleman alınmaya çalışılırsa (yani push edilmemiş elemanlar pop edilmeye çalışılırsa) bu kez stack aşağıdan taşacaktır (stack underflow).

Bağlı liste tekniğinde (list stack) push işlemi sırasında bağlı listenin önüne eleman eklenir. Pop işleminde de bağlı listenin önündeki eleman alınır.



Tıpkı kuyruk sistemlerinde olduğu gibi stack sistemlerinde de dizi tekniği ile bağlı liste tekniği aynı avantaj ve dezavantajlara sahiptir. Yani dizi tekniğinde push ve pop işlemleri $O(1)$ karmaşıklıktadır. Bağlı liste tekniğinde bellek tahsisatı söz konusu olacağından push işlemi çoğu durumda $O(N)$ karmaşıklıkta yapılır. Yani bellek tahsisat işleminin ek bir zamansal maliyeti de vardır. Tabii bağlı liste gerçekleştiriminde stack'in uzunluğu baştan belirlenmek zorunda değildir.

Dizi kullanılarak stack gerçekleştirimi şöyle yapılabilir:

```

/* Stack.h */

#ifndef STACK_H_
#define STACK_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE      0
#define TRUE       1

/* Type Declarations */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagSTACK {
    DATATYPE *pStack;
    size_t size;
    DATATYPE *sp;
    size_t count;
} STACK, *HSTACK;

/* Function Prototypes */

HSTACK CreateStack(size_t size);
BOOL Push(HSTACK hStack, DATATYPE val);
DATATYPE Pop(HSTACK hStack);
BOOL PopSecure(HSTACK hStack, DATATYPE *pVal);
BOOL Walk(HSTACK hStack, BOOL(*Proc)(DATATYPE *));
void Clear(HSTACK hStack);
void CloseStack(HSTACK hStack);

/* Macros */

#define IsEmpty(hStack)      ((hStack)->count == 0)
#define GetCount(hStack)    ((hStack)->count)
#define GetSize(hStack)     ((hStack)->size)

#endif

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

```

```

HSTACK CreateStack(size_t size)
{
    HSTACK hStack;

    if ((hStack = malloc(sizeof(STACK))) == NULL)
        return NULL;

    if ((hStack->pStack = malloc(sizeof(DATATYPE) * size)) == NULL) {
        free(hStack);
        return NULL;
    }

    hStack->size = size;
    hStack->count = 0;
    hStack->sp = hStack->pStack + size;

    return hStack;
}

BOOL Push(HSTACK hStack, DATATYPE val)
{
    if (hStack->count == hStack->size)    /* hStack->sp == hStack->pStack */
        return FALSE;

    *--hStack->sp = val;
    ++hStack->count;

    return TRUE;
}

DATATYPE Pop(HSTACK hStack)
{
    --hStack->count;

    return *hStack->sp++;
}

BOOL PopSecure(HSTACK hStack, DATATYPE *pVal)
{
    if (!hStack->count)    /* hStack->sp == hStack->pStack + hStack->size */
        return FALSE;

    *pVal = *hStack->sp++;
    --hStack->count;

    return TRUE;
}

BOOL Walk(HSTACK hStack, BOOL(*Proc)(DATATYPE *))
{
    size_t i;

    for (i = 0; i < hStack->count; ++i)
        if (!Proc(&hStack->sp[i]))
            return FALSE;

    return TRUE;
}

void Clear(HSTACK hStack)
{
    hStack->sp = hStack->pStack + hStack->size;
    hStack->count = 0;
}

```

```

void CloseStack(HSTACK hStack)
{
    free(hStack->pStack);
    free(hStack);
}

/* Test.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Stack.h"

#define MAX_CMD_LINE    1024
#define MAX_PARAMS      10

void parse_cmdline(void);
void proc_pop(void);
void proc_push(void);
void proc_clear(void);
void proc_count(void);
void proc_disp(void);

typedef struct tagCMD {
    const char *cmdText;
    void(*Proc)(void);
} CMD;

CMD g_cmds[] = {
    {"pop", proc_pop}, {"push", proc_push}, {"clear", proc_clear},
    {"count", proc_count}, {"disp", proc_disp}, {NULL, NULL}
};

HSTACK g_hStack;
char g_cmdLine[MAX_CMD_LINE];
char *g_params[MAX_PARAMS];
int g_nparams;

int main(void)
{
    int i;

    if ((g_hStack = CreateStack(10)) == NULL) {
        fprintf(stderr, "cannot create queue!..\n");
        exit(EXIT_FAILURE);
    }

    for (;;) {
        printf("CSD>");
        fgets(g_cmdLine, MAX_CMD_LINE, stdin);
        parse_cmdline();
        if (g_nparams == 0)
            continue;
        if (!strcmp(g_params[0], "quit"))
            break;
        for (i = 0; g_cmds[i].cmdText != NULL; ++i)
            if (!strcmp(g_params[0], g_cmds[i].cmdText)) {
                g_cmds[i].Proc();
                break;
            }
        if (g_cmds[i].cmdText == NULL)
            printf("command not found: %s\n", g_params[0]);
    }

    CloseStack(g_hStack);
}

```



```

    return 0;
}

void parse_cmdline(void)
{
    char *str;

    g_nparams = 0;
    for (str = strtok(g_cmdLine, " \t\n"); str != NULL; str = strtok(NULL, " \t\n"))
        g_params[g_nparams++] = str;
}

void proc_pop(void)
{
    DATATYPE val;

    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    if (!PopSecure(g_hStack, &val)) {
        printf("empty stack!\n");
        return;
    }
    printf("%d\n", val);
}

void proc_push(void)
{
    int val;

    if (g_nparams == 1) {
        printf("too few parameters!\n");
        return;
    }

    if (g_nparams > 2) {
        printf("too many parameters!\n");
        return;
    }

    val = atoi(g_params[1]);
    if (!Push(g_hStack, val)) {
        printf("stack is full!\n");
        return;
    }
}

void proc_clear(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    Clear(g_hStack);
}

void proc_count(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }

    printf("%u\n", GetCount(g_hStack));
}

```

```

static BOOL disp(DATATYPE *val)
{
    printf("%d ", *val);

    return TRUE;
}

void proc_disp(void)
{
    if (g_nparams > 1) {
        printf("too many parameters!\n");
        return;
    }
    Walk(g_hStack, disp);
    if (!GetCount(g_hStack))
        printf("Empty stack");
    printf("\n");
}

```

Bağlı liste kullanılarak stack gerçekleştirimi de şöyle yapılabilir:

```

/* StackList.h */

#ifndef STACKLIST_H_
#define STACKLIST_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE 0
#define TRUE 1

/* Type Definitions */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
} NODE;

typedef struct tagSTACK {
    NODE *pHead;
    size_t count;
} STACK, *HSTACK;

/* Function Prototypes */

HSTACK CreateStack(void);
BOOL Push(HSTACK hStack, DATATYPE val);
DATATYPE Pop(HSTACK hStack);
BOOL PopSecure(HSTACK hStack, DATATYPE *val);
void ClearStack(HSTACK hStack);
void CloseStack(HSTACK hStack);

#define GetItemCount(hStack)((hStack)->count)
#define IsEmptyStack(hStack)((hStack)->count == 0)

#endif

/* StackList.c */

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

```

```

#include "StackList.h"

/* Function Definitions */

HSTACK CreateStack(void)
{
    HSTACK hStack;

    if ((hStack = (HSTACK)malloc(sizeof(STACK))) == NULL)
        return NULL;

    hStack->pHead = NULL;
    hStack->count = 0;

    return hStack;
}

BOOL Push(HSTACK hStack, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return FALSE;

    pNewNode->val = val;
    pNewNode->pNext = hStack->pHead;
    hStack->pHead = pNewNode;

    ++hStack->count;

    return TRUE;
}

DATATYPE Pop(HSTACK hStack)
{
    NODE *pNode;
    DATATYPE val;

    pNode = hStack->pHead;
    hStack->pHead = pNode->pNext;
    val = pNode->val;
    --hStack->count;

    free(pNode);

    return val;
}

BOOL PopSecure(HSTACK hStack, DATATYPE *val)
{
    NODE *pNode;

    if (hStack->pHead == NULL)
        return FALSE;

    pNode = hStack->pHead;
    hStack->pHead = pNode->pNext;
    *val = pNode->val;
    --hStack->count;

    free(pNode);

    return TRUE;
}

void ClearStack(HSTACK hStack)
{
    NODE *pNode, *pTemp;

    pNode = hStack->pHead;

```

```

while (pNode != NULL) {
    pTemp = pNode;
    pNode = pNode->pNext;
    free(pTemp);
}

hStack->pHead = NULL;
hStack->count = 0;
}

void CloseStack(HSTACK hStack)
{
    ClearStack(hStack);
    free(hStack);
}

/* App.c */

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

int main(void)
{
    HSTACK hStack;
    int i;
    DATATYPE val;

    if ((hStack = CreateStack(10)) == NULL) {
        fprintf(stderr, "cannot create stack!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        Push(hStack, i);

    while (!IsEmptyStack(hStack)) {
        PopSecure(hStack, &val);
        printf("%d ", val);
    }
    printf("\n");

    ClearStack(hStack);

    for (i = 0; "ankara"[i] != '\0'; ++i)
        Push(hStack, "ankara"[i]);

    while (!IsEmptyStack(hStack))
        putchar(Pop(hStack));

    printf("\n");

    CloseStack(hStack);

    return 0;
}

```

Çift Yönlü Dinamik Diziler (Double Ended Queue (Deque))

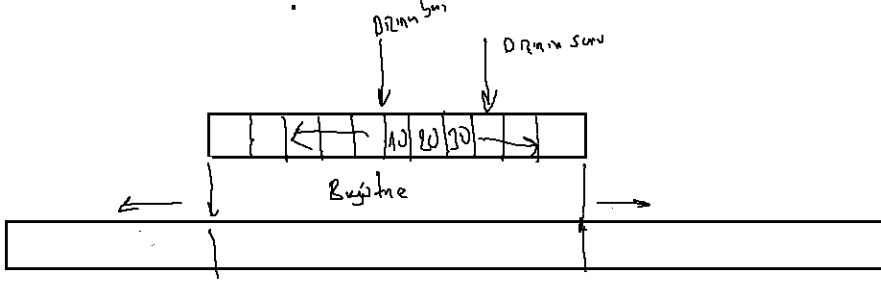
Çift yönlü dinamik dizilerin normal dinamik dizilerden tek farkı başa ve sona eleman eklemenin ek maliyetli sabit zamanlı olmasıdır. Anımsanacağı gibi normal dinamik dizilerde sona eleman ekleme ek maliyetli sabit karmaşıklıkta başa (ya da araya) eleman eklemek doğrusal karmaşıklıktadır. Oysa çift yönlü dinamik dizilerde yalnızca sona değil başa da eleman eklemek ek maliyetli sabit karmaşıklıkta bir işlemdir. Çift yönlü dinamik dizilerde elemana erişmek dinamik dizilerde olduğu gibi yine sabit karmaşıklığa sahip bir işlemdir.

Çift Yönlü Dinamik Dizilere Neden Gereksinim Duyulmaktadır?

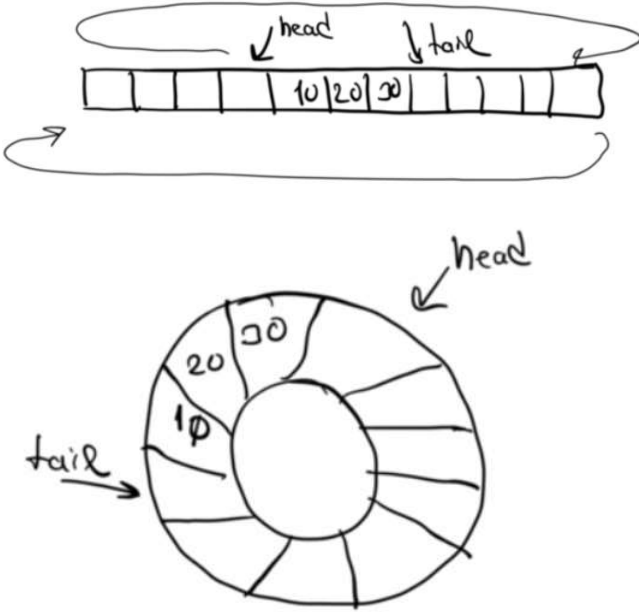
Bazı sistemlerde hem başa hem de sona eleman eklemek mümkün olabilmektedir. Örneğin gelen tek sayıları dizinin başına çift sayıları dizinin sonuna yerleştirmek istediğimizi düşünelim. Bunu normal dinamik diziyile yapmak istersek eleman ekleme işlemi için doğrusal karmaşıklık gerekir. Fakat çift yönlü dinamik dizilerde hem başa hem de sona eleman ekleme sabit karmaşıklıkta yapılabilir. Çift yönlü dinamik diziler genel bir veri yapısı olarak da kullanılabilir. Yani örneğin bunlar hem stack hem kuyruk de gerçekleştiriminde kullanılabilir.

Çift Yönlü Dinamik Dizilerin Gerçekleştirilmesi

Çift yönlü dinamik dizilerin gerçekleştirimi tipik olarak iki biçimde yapılabilir. Birinci biçimde bir dizi alınır. Fakat dizinin indeks olarak başı ve sonu ayrı bir göstericiyle tutturulur. Böylece başa ve sona ekleme sabit zamanlı yapılabilir. Örneğin:

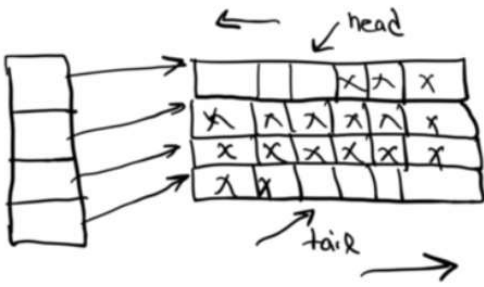


Tabii istenirse bu dizi sanki döngüsel kuyruk sisteminde olduğu gibi de ele alınabilir. Yani dizinin başını ve sonunu gösteren göstericiler iki uca geldiklerinde diğer uçtan çıkabilirler.



Bu biçimdeki gerçekleştirimde genellikle dizi de dinamik olarak büyütülür. Büyütme yine eskisinin iki katı olacak biçimde yapılır.

İkinci yöntemde bloklar bir listeye birbirine bağlanır. Bu bağlama işlemi sabit uzunluktaki blokların adreslerinin bir gösterici dizisinde tutulmasıyla yapılabilir. Böylece elemana erişim yine $O(1)$ karmaşıklıkta gerçekleştirilir. Büyütme işlemi küçük blokların tahsis edilmesi yoluyla yapıldığından hem daha hızlı olur hem de heap alanının bölünmesi (fragmentation) daha makul düzeye çekilmiş olur.



Aşağıda başa ve sona eleman ekleme fonksiyonları olan bir deque veri yapısı örneği verilmiştir:

```

/* Deque.h */

#ifndef DEQUE_H_
#define DEQUE_H_

#include <stddef.h>

#define DEF_CAPACITY 8
#define DEQUE_FAILED ((size_t)-1)

/* Type Declarations */

typedef struct tagPERSON {
    char name[32];
    int no;
} PERSON;

typedef PERSON DATATYPE;

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
} NODE;

typedef struct tagDEQUE {
    DATATYPE *pDeque;
    size_t head;
    size_t tail;
    size_t capacity;
    size_t size;
} DEQUE, *HDEQUE;

/* Function Prototypes */

HDEQUE CreateDeque(void);
HDEQUE CreateDequeWithCapacity(size_t capacity);
size_t AddItem(HDEQUE hDeque, const DATATYPE *pVal);
size_t AddItemFront(HDEQUE hDeque, const DATATYPE *pVal);
size_t SetCapacity(HDEQUE hDeque, size_t newCapacity);
void GetItem(HDEQUE hDeque, size_t index, DATATYPE *pVal);

/* Macros */

#define GetItem(hDeque, index, pVal) (*(pVal) = (hDeque)->pDeque[((hDeque)->head + (index)) % (hDeque)->capacity])

#endif

/* Deque.c */

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

```

```

#define MIN(a, b) ((a) < (b) ? (a) : (b))

HDEQUE CreateDeque(void)
{
    return CreateDequeWithCapacity(DEF_CAPACITY);
}

HDEQUE CreateDequeWithCapacity(size_t capacity)
{
    HDEQUE hDeque;

    if ((hDeque = (HDEQUE)malloc(sizeof(DEQUE))) == NULL)
        return NULL;

    if ((hDeque->pDeque = (DATATYPE *)malloc(sizeof(DATATYPE) * capacity)) == NULL) {
        free(hDeque);
        return NULL;
    }
    hDeque->head = hDeque->tail = hDeque->size = 0;
    hDeque->capacity = capacity;

    return hDeque;
}

size_t AddItem(HDEQUE hDeque, const DATATYPE *pVal)
{
    if (hDeque->size == hDeque->capacity)
        if (SetCapacity(hDeque, hDeque->capacity * 2) == DEQUE_FAILED)
            return DEQUE_FAILED;
    hDeque->pDeque[hDeque->tail++] = *pVal;
    hDeque->tail = hDeque->tail % hDeque->capacity;
    ++hDeque->size;

    return hDeque->size - 1;
}

size_t AddItemFront(HDEQUE hDeque, const DATATYPE *pVal)
{
    if (hDeque->size == hDeque->capacity)
        if (SetCapacity(hDeque, hDeque->capacity * 2) == DEQUE_FAILED)
            return DEQUE_FAILED;

    if (hDeque->head == 0)
        hDeque->head = hDeque->capacity - 1;
    else
        --hDeque->head;

    hDeque->pDeque[hDeque->head] = *pVal;
    ++hDeque->size;

    return 0;
}

size_t SetCapacity(HDEQUE hDeque, size_t newCapacity)
{
    DATATYPE *pDeque;
    size_t size1, size2;

    if (newCapacity < hDeque->size)
        return DEQUE_FAILED;

    if ((pDeque = (DATATYPE *)malloc(sizeof(DATATYPE) * newCapacity)) == NULL)
        return DEQUE_FAILED;

    size1 = MIN(hDeque->capacity - hDeque->head, hDeque->size);

```

```

size2 = hDeque->size - size1;

memcpy(pDeque, &hDeque->pDeque[hDeque->head], size1 * sizeof(DATATYPE));
memcpy(pDeque + size1, &hDeque->pDeque, size2 * sizeof(DATATYPE));

free(hDeque->pDeque);
hDeque->pDeque = pDeque;

hDeque->head = 0;
hDeque->tail = hDeque->size;
hDeque->capacity = newCapacity;

return newCapacity;
}

/* Test.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Deque.h"

int main(void)
{
    HDEQUE hDeque;
    PERSON per;
    int i;

    if ((hDeque = CreateDequeWithCapacity(2)) == NULL) {
        fprintf(stderr, "Cannot create deque!..\n");
        exit(EXIT_FAILURE);
    }

    strcpy(per.name, "Kaan Aslan");
    per.no = 123;

    AddItem(hDeque, &per);

    strcpy(per.name, "Ali Serce");
    per.no = 456;

    AddItem(hDeque, &per);

    strcpy(per.name, "Sacit Bulut");
    per.no = 456;

    AddItem(hDeque, &per);

    strcpy(per.name, "Burhan Kuzu");
    per.no = 965;

    AddItemFront(hDeque, &per);

    for (i = 0; i < hDeque->size; ++i) {
        GetItem(hDeque, i, &per);
        printf("%s, %d\n", per.name, per.no);
    }

    return 0;
}

```

Arama İşlemleri

Bilgisayar bilimlerinde bir anahtar verildiğinde onun değerinin ya da yerinin bulunması sürecine arama (searching) denilmektedir. Arama işlemleri içsel (internal) ve dışsal (external) olmak üzere ikiye ayrılır. Eğer arama birincil bellekte

saklanan bilgilerin üzerinde yapılıyorsa buna içsel arama (internal search), diskteki dosyaların içerisinde yapılıyorsa buna da dışsal arama (external search) denilmektedir.. Ancak yalnızca arama denildiğinde içsel arama anlaşılmaktadır.

Arama işleminin mümkün olduğu kadar çabuk yapılması arzu edilir. Pekiyi arama işlemleri nasıl hızlandırılabilir? Örneğin bir dizi içerisinde bir değeri aramak isteyelim, hangi yöntemleri kullanabiliriz? Öncelikle ideal bir arama nasıl olabilir bunun üzerinde düşünelim. Şüphesiz ideal durumda aramanın hızlı bir biçimde adeta rastgele erişimli olarak yani $O(1)$ karmaşıklıkta yapılması arzu edilir. İşte bunun tipik bir uygulaması indeksli arama (index search) denilen yöntemdir. Bu yöntemde bir dizi açılır. Anahtar diziyeye indeks yapılarak eleman dizinin o indeksine yerleştirilir. Sonra arama yapılırken doğrudan anahtara ilişkin indekse bakılır. Örneğin 100 kişinin bilgilerini bir diziyeye yerleştirip aramak isteyelim. Burada kişilerin bilgilerini bir yapıyla temsil ederiz. Sonra bu yapı türünden 100 elemanlı bir dizi açarız. Kişileri de numaralarına göre bu yapı dizisinin ilgili indekslerine yerleştiririz. Arakende de o numaraların belirttiği indekslere bakarız. Böyle bir sistem $O(1)$ karmaşıklıkta ideal aramaya olanak sağlar. Ancak bu sistem pratikte pek çok durum için uygun değildir. Eğer anahtarın indeks genişliği çok büyükse yöntem kullanılabilir olmaktan çıkar. (Örneğin 100 kişiyi TC numaralarına göre bu yöntemle diziyeye yerleştirmek isteyelim. Bizim çok çok büyük bir dizi açmamız gerekir. Bu dizinin çok az kısmı dolu olacaktır.) Bu yöntemin diğer bir problemi de anahtarın bire bir bir indekse dönüşürülmesi gerekliliğidir. Örneğin anahtar TC kimlik numarası değil de ad soya biçiminde olsa biz belli bir kişiyi dizinin kaçınıcı indeksine yerleştireceğiz? İşte bizim yazıyı indeks numarasına dönüştüren bir hash fonksiyonuna ihtiyacımız olur. Bu hash fonksiyonun da bire bir olması gerekir. Fakat pratikte bire bir hash fonksiyonu mümkün değildir. (Yani örneğin iki farklı ad soyad aynı indeks değerini üretebilmektedir.) Fakat yine de indeksli arama özel durumlarda ideal biçimde kullanılabilir. Örneğin bir 1000 kişiye birer numara verip onları $O(1)$ karmaşıklıkta numaralarına göre arayabiliriz.

Pekiyi yukarıdaki nedenlerden dolayı indeksli arama mümkün değilse ne yapabiliriz? Örneğin 100 civarında kişiyi TC kimlik numaralarına göre bir diziyeye yerleştirip aramak isteyelim. Nasıl bir arama yöntemi kullanabiliriz? Eğer bir dizilimde elemanlar arasında hiçbir kural yoksa ilk akla gelen yöntem sıralı aramadır (sequential search). Sıralı aramada dizideki eleman sayısı N ise ortalama karmaşıklık $(N + 1) / 2$ 'dir.

$$\frac{1+2+3+\dots+n}{n} = \frac{n(n+1)}{2} = \frac{(n+1)}{2}$$

O halde Big O notasyonuna göre sıralı arama doğrusal karmaşıklıkta yani $O(N)$ karmaşıklıkta yapılmaktadır. (Örneğin 1000000 elemanlı dizi için ortalama olarak 500000 karşılaştırma yapmak gerekir.) Sıralı arama 20'den küçük sayıda diziler için belki de en iyi arama yöntemidir. Çünkü ileride sözünü edeceğimiz algoritmik arama yöntemlerinin kurulumu için belli bir zaman harcanmaktadır. Oysa sıralı aramada böyle bir hazırlığın yapılmasına gerek olmaz. Bu nedenle çok az sayıda eleman için özel bir algoritmik yöntemin düşünülmesine hiç gerek yoktur.

Eğer dizilimdeki elemanlar sıraya dizilmişse ikili arama yöntemi (binary search) en iyi yöntemdir. İkili aramada iki çubuk (yani indeks) tutulur. Ortadaki elemana bakılır. Sol ya da sağ çubuk oraya çekilir. Yani her defasında dizilim yarısı kadar küçültülmektedir. İkili aramanın en kötü durum karmaşıklığı $\log_2 N$ 'dir. Big O notasyonuna göre ikili aramanın karmaşıklığı ise $\log(N)$ biçimindedir. (Örneğin 1024 eleman için 10, 1000000 için yaklaşık 20 karşılaştırma). İkili arama algoritması tipik olarak şöyle gerçekleştirilmektedir.

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

typedef int DATATYPE;

DATATYPE *BinarySearch(DATATYPE *pArray, size_t size, DATATYPE val)
{
    size_t left, right, mid;

    left = 0;
    right = size - 1;

    while (left <= right) {
        mid = (left + right) / 2;
        if (pArray[mid] < val)
            left = mid + 1;
        else if (pArray[mid] > val) {
            if (right == 0)
                return -1;
            right = mid - 1;
        }
    }
    return pArray[mid];
}
```

```

        break;
        right = mid - 1;
    }
    else
        return &pArray[mid];
}

return NULL;
}

int main(void)
{
    DATATYPE array[10] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
    DATATYPE *pVal;

    if ((pVal = BinarySearch(array, 10, 0)) == NULL)
        printf("Bulunamadi!..\n");
    else
        printf("Bulundu: %d\n", *pVal);

    return 0;
}

```

Peki sıralı olmayan diziyi önce sıraya dizip sonra ikili arama yapsak nasıl olur? Diziyi sıraya dizmenin en iyi karmaşıklığı $N \log N$ 'dir. Bu durumda sıraya dizmenin maliyeti zaten sıralı aramanın maliyetinden yüksek olur. Şüphesiz diziyeye hiç eleman eklenmeyecekse fakat çok sayıda arama yapılacaksa bu yöntem uygun hale gelebilir. Hatta diziyeye seyrek olarak eleman eklendiği fakat çok sayıda aramanın yapıldığı durumlarda da bu yöntem uygun olabilmektedir. Fakat uygulamada çoğu kez durum böyle değildir. Şöyle bir durum düşünelim. Bir yarışa 100 kişi katılmış olsun. Her kişinin aldığı dereceyi bir diziyeye herhangi bir sırada eklemiş olalım. Sonra da bize telefon edilerek yarışa katılanlar kaçınıcı olduklarını sorsunlar. Biz önce bu N kişiyi sıraya dizip sonra N kişi için ikili arama yapmak isteyelim. Sıralamanın en iyi maliyeti $N \log(N)$ 'dir. Daha sonra N tane kişiyi de $\log(N)$ karmaşıklıkta arayacağımız düşünelim. İşlemin toplam karmaşıklığı $N \log(N) + N \log(N)$ olur. Bu da $2 N \log(N)$ anlamına gelir. Big O notasyonuna göre karmaşıklık $N \log(N)$ 'dir. Diziyi sıraya dizmeyip N kişi için sıralı arama yaparsak toplam karmaşıklık $N * N / 2$ olur ki bu da Big O notasyonuna göre $O(N^2)$ karmaşıklık anlamına gelir. Görüldüğü gibi açık biçimde bu durumda diziyi sıraya dizip ikili arama yapmak daha avantajlıdır. Şimdi yarışa katılan herkesin değil de yalnızca 10 kişinin durumunu soracağımızı düşünelim. Bu durumda doğrusal arama $O(N)$ karmaşıklıkta kalacaktır ve diğerinden daha iyi olacaktır.

Peki sıralı diziler için ikili aramadan daha iyi bir yöntem olabilir mi? Eğer biz dizilimin ortasına değil de daha uygun bir yerine bakabilirsek sol ya da sağ çubukları daha etkin konumlandırabiliriz. Başka bir deyişle her defasında diziyi ortadan değil de daha uygun bir oranda daraltırsak daha hızlı yakınsama sağlanabilir. (Örneğin her defasında çubukları yarıdan değil, $1/3$ 'ten hizalamak gibi). Tabi böyle bir çıkarımda bulunabilmek için dizilimin dağılımı hakkında bilgi sahibi olmak gerekir. Bu yöntem "enterpolasyon araması (interpolation search)" denilmektedir. Ancak dizilim hakkında bir bilgi yoksa böyle bir algoritma daha fazla zaman kaybına da yol açabilir. Dağılımı bilinmeyen sıralı dizilerde ikili arama en etkin yöntemdir.

İkili aramamanın diğer bir biçimine de "üstel arama (exponential search)" denilmektedir. Bu yöntemde önce sıralı tek taraftan ya da iki taraftan yarıya bölme yöntemiyle daraltılır. Sonra daraltılmış olan alanda ikili arama yöntemi uygulanır. Bu yöntem özellikle dizide uzunluk sınırının olmadığı (unbounded arrays) ve değerlerin düzgün dağılmadığı durumlarda hız kazancı sağlamaktadır. Üstel arama aşağıdaki gibi yapılabilir:

```

DATATYPE *ExponentialSearch(DATATYPE *pArray, size_t size, DATATYPE val)
{
    size_t i, left;

    if (val < pArray[0])
        return NULL;

    i = 1;
    while (pArray[i] < val && i < size)
        i *= 2;
    left = i / 2;
}

```

```

return BinarySearch(pArray + left, i - left + 1, val);
}

int main(void)
{
int a[10] = { 3, 9, 15, 26, 38, 45, 59, 65, 80, 94 };
int *pi;

if ((pi = ExponentialSearch(a, 10, 1)) == NULL) {
fprintf(stderr, "Bulunamadı!..\n");
exit(EXIT_FAILURE);
}

printf("Bulundu: %d\n", *pi);

return 0;
}

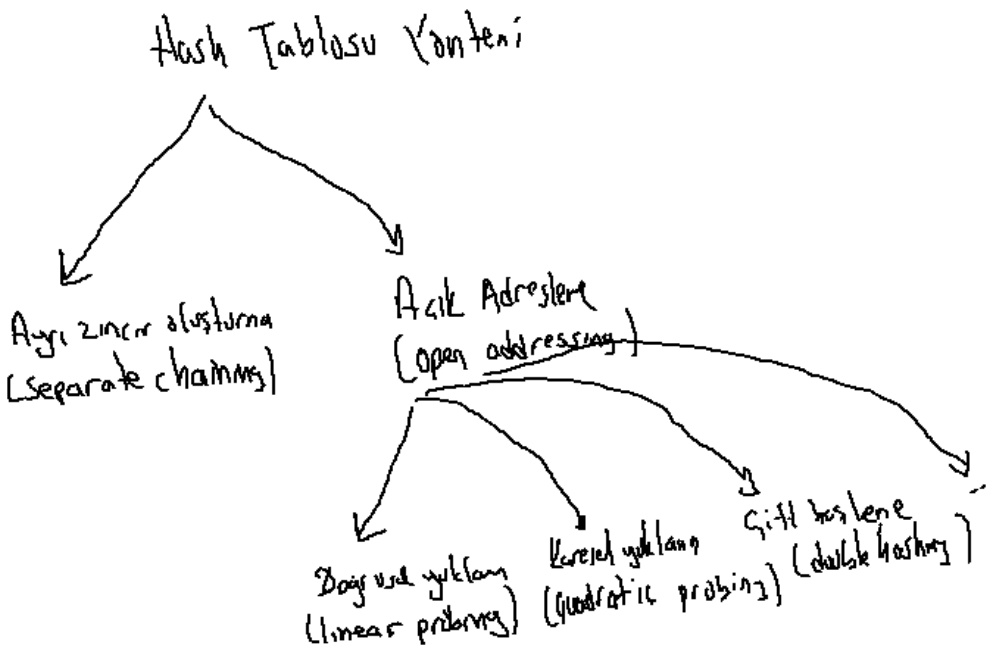
```

Peki dizinin sıralı olmadığını düşünelim. Daha etkin bir arama nasıl yapılabilir? İşte bunun için başka algoritmik arama yöntemleri kullanılmaktadır. Bu yöntemlerde aslında eleman daha veri yapısına yerleştirilirken onun aranabileceği fikriyle bazı notlar için başında alınmaktadır. Algoritmik arama yöntemleri tipik olarak "hash tabloları" ve "arama ağaçları" ve biçiminde iki gruba ayrılabilir. Bunların dışında başka algoritmik arama yöntemleri olsa da uygulamada bu iki yöntem grubu en sık kullanılanlardır.

Hash Tabloları İle Arama (Hashing)

Hash tabloları ile arama "indeksli arama" ile "sıralı arama"nın bir orta noktası gibidir. Bu yöntemde bir dizi açılır. Anahtar bir fonksiyona sokularak dizi indeksine dönüştürülür ve eleman o indekse yerleştirilmek istenir. Örneğin kişileri TC kimlik numaralarına göre bir hash tablosuna yerleştirmek isteyelim. Bunun için 100 elemanlı bir dizi açmış olalım. TC kimlik numarası 11 basamaklı büyük bir numaradır. Biz bu numaradan [0, 99] arasında bir indeks elde etmek isteyelim. Bu indeksi elde eden fonksiyona "hash fonksiyonu" denilmektedir. Örneğin 100'e bölümünden elde edilen kalan basit hash fonksiyonu olarak kullanılabilir. Bu durumda 41106234567 TC numaralı kişi bu dizinin 67'inci indeksine yerleştirilmek istenecektir. Eleman aranırken de aynı biçimde anahtardan aynı hash fonksiyonuyla dizi indeksi elde edilir ve o indekse bakılmak istenir.

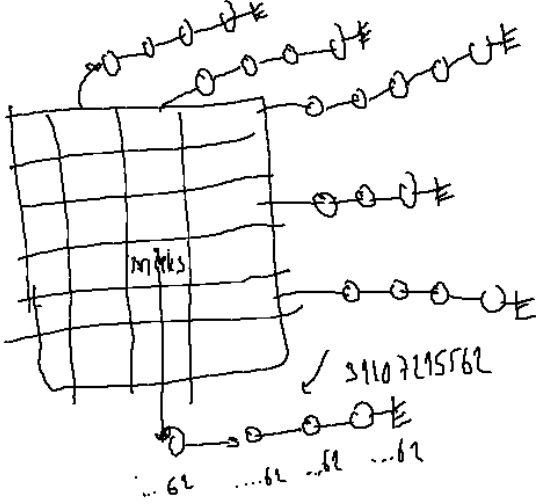
Şüphesiz yukarıda açıklanan yöntemde her zaman bir çakışma (collision) olasılığı vardır. Örneğin 23445623267 TC numaralı kişi için de aynı indeks ele edilecektir. İşte hash tablosu yöntemi çakışma durumundaki stratejiye göre iki temel alt gruba ayrılmaktadır:



Açık adresleme yöntemi de kendi aralarında pek çok alt yöntem ayrılmaktadır. En çok kullanılan çakışma çözüm yöntemi (collision resolution methods) "ayrı zincir oluşturma" yöntemidir.

Ayrı Zincir Oluşturma Yöntemi

Bu yöntemde N elemanlı bir dizi açılır. Fakat dizide elemanlar tutulmaz. Bu bir bağlı liste dizisidir. Yani dizinin her elemanı bir bağlı listenin head göstericisini tutar. Eleman yerleştirileceği zaman hash fonksiyonuna sokulur. Buradan bir dizi indeksi elde edilir. Sonra eleman bu indeksteki bağlı listeye eklenir. Arama yapılırken de benzer biçimde önce anahtar hash fonksiyonuna sokulur. Oradan elde edilen indeksteki bağlı listede doğrusal arama yapılır.



Bu yöntemde eleman eklemek sabit karmaşıklığa sahip bir işlemdir. Arama ise doğrusal karmaşıklıkta gözükmele birlikte az sayıda eleman sahip dizi içerisinde yapılacağı için çok etkindir. Gerçekten de Knuth bağlı listelerdeki eleman sayısı ortalama 10'u geçmedikten sonra bu yöntemi süper bir yöntem olarak nitelendirmektedir. Eğer bağlı listedeki elemanların sayısı çok fazla olursa (yani tablo küçük kalırsa) yöntem doğrusal aramaya yaklaşmaya başlar. O halde tablo uzunluğunu baştan iyi öngörmek gerekir. Örneğin ilgili sisteme ortalama 10000 eleman yerleştirilecekse bizim tabloyu 1000 civarında tutmamız uygun olur.

Hash tabloları özellikle işletim sistemlerinin çekirdek kodlamalarında çok sık karşımıza çıkmaktadır. Örneğin Linux'ta cache sistemlerinde arama yapılırken hep ayrı zincirli hash tabloları kullanılmıştır. Örneğin Linux'un buffer cache (disk cache) mekanizmasını düşünelim. Burada dosya fonksiyonları okunacak yerin disk blok adresini hesapladıktan sonra onun buffer cache içerisinde olup olmadığına bakmak ister. Blok adresleri birer tamsayıyla belirtilir. Örneğin read fonksiyonu okunacak dosya parçasınının diskin 181317'inci bloğunda olduğunu hesaplasın. Bu blok acaba cache'te midir? İşte Linux bu değeri anahtar yaparak bir hash tablosunda arama yapar. Eğer bulursa doğrudan hiç disk okuması yapmadan bilgiyi oradan alarak verir. Bnezer biçimde dizin girişleri (dentry cache), inode elemanları (inode cache) hep bu biçimde cache sistemleri içerisinde saklanmaktadır.

Ayrı zincir oluşturma yönteminin örnek bir gerçekleştirimi şöyle olabilir:

```
/* HashTable.h */

#ifndef HASHTABLE_H_
#define HASHTABLE_H_

/* Symbolic Constants */

#define TRUE 1
#define FALSE 0

/* Type Declarations */

typedef int BOOL;

typedef struct tagPERSON {
    char name[32];
```

```

    int no;
} PERSON;

typedef PERSON DATATYPE;

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
    struct tagNODE *pPrev;
} NODE;

typedef int KEY;

typedef struct tagHASHTABLE {
    NODE **ppHeads;
    size_t size;
    size_t count;
    size_t (*HashFunc)(size_t, KEY);
} HASHTABLE, *HHASHTABLE;

/* Function Prototypes */

HHASHTABLE CreateHashTable(size_t size);
NODE *InsertItem(HHASHTABLE hHashTable, DATATYPE *pVal);
PERSON *FindItem(HHASHTABLE hHashTable, KEY no);
void DeleteItemNode(HHASHTABLE hHashTable, NODE *pNode);
BOOL DeleteItem(HHASHTABLE hHashTable, KEY key);
void SetHashFunc(HHASHTABLE hHashTable, size_t(*HashFunc)(size_t, KEY));
void DisplayHashTable(HHASHTABLE hHashTable);
void ClearHashTable(HHASHTABLE hHashTable);
void CloseHashTable(HHASHTABLE hHashTable);

/* Macros */

#define GetCount(hHashTable) ((hHashTable)->count)
#define GetSize(hHashTable) ((hHashTable)->size)

#endif

/* HashTable.c */

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

static size_t defaultHashFunc(size_t size, KEY key);

HHASHTABLE CreateHashTable(size_t size)
{
    HHASHTABLE hHashTable;
    size_t i;

    if ((hHashTable = (HHASHTABLE)malloc(sizeof(HASHTABLE))) == NULL)
        return NULL;

    if ((hHashTable->ppHeads = (NODE **)malloc(sizeof(NODE *) * size)) == NULL) {
        free(hHashTable);
        return NULL;
    }

    for (i = 0; i < size; ++i)
        hHashTable->ppHeads[i] = NULL;

    hHashTable->size = size;
    hHashTable->count = 0;
    hHashTable->HashFunc = defaultHashFunc;
}

```

```

    return hHashTable;
}

NODE *InsertItem(HHASHTABLE hHashTable, DATATYPE *pVal)
{
    size_t index;
    NODE *pNewNode;

    index = hHashTable->HashFunc(hHashTable->size, pVal->no);
    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = *pVal;

    if (hHashTable->ppHeads[index] != NULL)
        hHashTable->ppHeads[index]->pPrev = pNewNode;
    pNewNode->pNext = hHashTable->ppHeads[index];
    pNewNode->pPrev = NULL;
    hHashTable->ppHeads[index] = pNewNode;

    ++hHashTable->count;

    return pNewNode;
}

PERSON *FindItem(HHASHTABLE hHashTable, KEY no)
{
    size_t index;
    NODE *pNode;

    index = hHashTable->HashFunc(hHashTable->size, no);

    pNode = hHashTable->ppHeads[index];
    while (pNode != NULL) {
        if (pNode->val.no == no)
            return &pNode->val;
        pNode = pNode->pNext;
    }

    return NULL;
}

void DeleteItemNode(HHASHTABLE hHashTable, NODE *pNode)
{
    size_t index;

    index = hHashTable->HashFunc(hHashTable->size, pNode->val.no);
    if (pNode->pPrev == NULL)
        hHashTable->ppHeads[index] = pNode->pNext;
    else
        pNode->pPrev->pNext = pNode->pNext;

    if (pNode->pNext != NULL)
        pNode->pNext->pPrev = pNode->pPrev;

    free(pNode);

    --hHashTable->count;
}

BOOL DeleteItem(HHASHTABLE hHashTable, KEY key)
{
    size_t index;
    NODE *pNode;
    index = hHashTable->HashFunc(hHashTable->size, key);

```

```

    pNode = hHashTable->ppHeads[index];
    while (pNode != NULL) {
        if (pNode->val.no == key) {
            DeleteItemNode(hHashTable, pNode);
            return TRUE;
        }
        pNode = pNode->pNext;
    }

    return FALSE;
}

void SetHashFunc(HHASHTABLE hHashTable, size_t(*HashFunc)(size_t, KEY))
{
    hHashTable->HashFunc = HashFunc;
}

void DisplayHashTable(HHASHTABLE hHashTable)
{
    size_t i;
    NODE *pNode;

    for (i = 0; i < hHashTable->size; ++i) {
        printf("%3d : \n", i);
        pNode = hHashTable->ppHeads[i];
        while (pNode != NULL) {
            printf("\t%5d %s\n", pNode->val.no, pNode->val.name);
            pNode = pNode->pNext;
        }
        printf("\n");
    }
}

void ClearHashTable(HHASHTABLE hHashTable)
{
    size_t i;
    NODE *pNode, *pTempNode;

    for (i = 0; i < hHashTable->size; ++i) {
        pNode = hHashTable->ppHeads[i];
        while (pNode != NULL) {
            pTempNode = pNode;
            pNode = pNode->pNext;
            free(pTempNode);
        }
    }

    for (i = 0; i < hHashTable->size; ++i)
        hHashTable->ppHeads[i] = NULL;

    hHashTable->count = 0;
}

void CloseHashTable(HHASHTABLE hHashTable)
{
    ClearHashTable(hHashTable);

    free(hHashTable->ppHeads);
    free(hHashTable);
}

static size_t defaultHashFunc(size_t size, KEY key)
{
    return key % size;
}

```

```

/* Test.c */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "HashTable.h"

void getRandomItem(PERSON *pper);

int main(void)
{
    HHASHTABLE hHashTable;
    int i;
    PERSON *pper;
    NODE *pNode;
    NODE *pDelNode;
    PERSON per;
    PERSON perSpec = { "Kaan Aslan", 7654 };

    //srand(time(NULL));

    if ((hHashTable = CreateHashTable(101)) == NULL) {
        fprintf(stderr, "cannot create hash table!\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 1000; ++i) {
        getRandomItem(&per);

        if ((pNode = InsertItem(hHashTable, &per)) == NULL) {
            fprintf(stderr, "cannot insert random item!");
            exit(EXIT_FAILURE);
        }
    }
    if ((pDelNode = InsertItem(hHashTable, &perSpec)) == NULL) {
        fprintf(stderr, "cannot insert item!");
        exit(EXIT_FAILURE);
    }

    /* DeleteItemNode(hHashTable, pDelNode); */

    if (!DeleteItem(hHashTable, 7654)) {
        fprintf(stderr, "cannot delete item!..\n");
        exit(EXIT_FAILURE);
    }
    /*
    if ((pper = FindItem(hHashTable, 7654)) == NULL) {
        fprintf(stderr, "cannot find item!..\n");
        exit(EXIT_FAILURE);
    }

    printf("Item found: %s, %d\n", pper->name, pper->no);
    */

    /* ClearHashTable(hHashTable); */

    DisplayHashTable(hHashTable);

    CloseHashTable(hHashTable);

    return 0;
}

void getRandomItem(PERSON *pper)
{

```



```

size_t k;

for (k = 0; k < 31; ++k)
    pper->name[k] = rand() % 26 + 'A';
pper->name[k] = '\0';
pper->no = rand() % 100000;
}

```

Hash Fonksiyonu Nasıl Olmalıdır?

Bilindiği gibi hash fonksiyonu anahtar dizi indeksine dönüştürmektedir. İyi bir hash fonksiyonunun şu özelliklere sahip olması beklenir:

- Tabloya yaydırmayı iyi yapmalıdır. Yani örneğin anahtar değerleri yanlı olsa bile hash fonksiyonun tabloya yansız bir biçimde eşit miktarda dağıtım yapabilecek yetenekte olması arzu edilir.

- Hash fonksiyonunun hızlı olması istenir. Çünkü eleman ekleme ve arama işlemlerinde devreye girmektedir.

Sayıdan sayı elde eden, yazıdan sayı elde eden kaliteli hash fonksiyonları için Internet'te araştırma yapılabilir. Örneğin yazıyı sayıya dönüştüren bir hash fonksiyonu şöyle olabilir:

```

size_t hash(const char *str, size_t size)
{
    unsigned long hash = 5381;

    while (*str != '\0')
        hash = ((hash << 5) + hash) + *str++;

    return hash % size;
}

```

Şimdi yukarıdaki örneği anahtar kişinin adı soyadı olacak biçimde değiştirelim:

```

/* HashTable.h */

#ifndef HASHTABLE_H_
#define HASHTABLE_H_

/* Symbolic Constants */

#define TRUE      1
#define FALSE    0

/* Type Declarations */

typedef int BOOL;

typedef struct tagPERSON {
    char name[32];
    int no;
} PERSON;

typedef PERSON DATATYPE;

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pNext;
    struct tagNODE *pPrev;
} NODE;

typedef char *KEY;

```

```

typedef struct tagHASHTABLE {
    NODE **ppHeads;
    size_t size;
    size_t count;
    size_t (*HashFunc)(size_t, KEY);
} HASHTABLE, *HHASHTABLE;

/* Function Prototypes */

HHASHTABLE CreateHashTable(size_t size);
NODE *InsertItem(HHASHTABLE hHashTable, DATATYPE *pVal);
PERSON *FindItem(HHASHTABLE hHashTable, KEY name);
void DeleteItemNode(HHASHTABLE hHashTable, NODE *pNode);
BOOL DeleteItem(HHASHTABLE hHashTable, KEY key);
void SetHashFunc(HHASHTABLE hHashTable, size_t(*HashFunc)(size_t, KEY));
void DisplayHashTable(HHASHTABLE hHashTable);
void ClearHashTable(HHASHTABLE hHashTable);
void CloseHashTable(HHASHTABLE hHashTable);

/* Macros */

#define GetCount(hHashTable) ((hHashTable)->count)
#define GetSize(hHashTable) ((hHashTable)->size)

#endif

/* HashTable.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "HashTable.h"

static size_t defaultHashFunc(size_t size, KEY key);

HHASHTABLE CreateHashTable(size_t size)
{
    HHASHTABLE hHashTable;
    size_t i;

    if ((hHashTable = (HHASHTABLE)malloc(sizeof(HASHTABLE))) == NULL)
        return NULL;

    if ((hHashTable->ppHeads = (NODE **)malloc(sizeof(NODE *) * size)) == NULL) {
        free(hHashTable);
        return NULL;
    }

    for (i = 0; i < size; ++i)
        hHashTable->ppHeads[i] = NULL;

    hHashTable->size = size;
    hHashTable->count = 0;
    hHashTable->HashFunc = defaultHashFunc;

    return hHashTable;
}

NODE *InsertItem(HHASHTABLE hHashTable, DATATYPE *pVal)
{
    size_t index;
    NODE *pNewNode;

    index = hHashTable->HashFunc(hHashTable->size, pVal->name);
    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;

```

```

pNewNode->val = *pVal;

if (hHashTable->ppHeads[index] != NULL)
    hHashTable->ppHeads[index]->pPrev = pNewNode;
pNewNode->pNext = hHashTable->ppHeads[index];
pNewNode->pPrev = NULL;
hHashTable->ppHeads[index] = pNewNode;

++hHashTable->count;

return pNewNode;
}

PERSON *FindItem(HHASHTABLE hHashTable, KEY name)
{
    size_t index;
    NODE *pNode;

    index = hHashTable->HashFunc(hHashTable->size, name);

    pNode = hHashTable->ppHeads[index];
    while (pNode != NULL) {
        if (!strcmp(pNode->val.name, name))
            return &pNode->val;
        pNode = pNode->pNext;
    }

    return NULL;
}

void DeleteItemNode(HHASHTABLE hHashTable, NODE *pNode)
{
    size_t index;

    index = hHashTable->HashFunc(hHashTable->size, pNode->val.name);
    if (pNode->pPrev == NULL)
        hHashTable->ppHeads[index] = pNode->pNext;
    else
        pNode->pPrev->pNext = pNode->pNext;

    if (pNode->pNext != NULL)
        pNode->pNext->pPrev = pNode->pPrev;

    free(pNode);

    --hHashTable->count;
}

BOOL DeleteItem(HHASHTABLE hHashTable, KEY key)
{
    size_t index;
    NODE *pNode;
    index = hHashTable->HashFunc(hHashTable->size, key);

    pNode = hHashTable->ppHeads[index];
    while (pNode != NULL) {
        if (!strcmp(pNode->val.name, key)) {
            DeleteItemNode(hHashTable, pNode);
            return TRUE;
        }
        pNode = pNode->pNext;
    }

    return FALSE;
}

```

```

void SetHashFunc(HHASHTABLE hHashTable, size_t(*HashFunc)(size_t, KEY))
{
    hHashTable->HashFunc = HashFunc;
}

void DisplayHashTable(HHASHTABLE hHashTable)
{
    size_t i;
    NODE *pNode;

    for (i = 0; i < hHashTable->size; ++i) {
        printf("%3d : \n", i);
        pNode = hHashTable->ppHeads[i];
        while (pNode != NULL) {
            printf("\t%5d %s\n", pNode->val.no, pNode->val.name);
            pNode = pNode->pNext;
        }
        printf("\n");
    }
}

void ClearHashTable(HHASHTABLE hHashTable)
{
    size_t i;
    NODE *pNode, *pTempNode;

    for (i = 0; i < hHashTable->size; ++i) {
        pNode = hHashTable->ppHeads[i];
        while (pNode != NULL) {
            pTempNode = pNode;
            pNode = pNode->pNext;
            free(pTempNode);
        }
    }

    for (i = 0; i < hHashTable->size; ++i)
        hHashTable->ppHeads[i] = NULL;

    hHashTable->count = 0;
}

void CloseHashTable(HHASHTABLE hHashTable)
{
    ClearHashTable(hHashTable);

    free(hHashTable->ppHeads);
    free(hHashTable);
}

static size_t defaultHashFunc(size_t size, KEY key)
{
    unsigned long hash = 5381;

    while (*key != '\0')
        hash = ((hash << 5) + hash) + *key++;

    return hash % size;
}

/* Test.c */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "HashTable.h"

```

```

void getRandomItem(PERSON *pper);

int main(void)
{
    HHASHTABLE hHashTable;
    int i;
    NODE *pNode;
    NODE *pDelNode;
    PERSON *pper;
    PERSON per;
    PERSON perSpec = { "Kaan Aslan", 7654 };

    //srand(time(NULL));

    if ((hHashTable = CreateHashTable(101)) == NULL) {
        fprintf(stderr, "cannot create hash table!\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 1000; ++i) {
        getRandomItem(&per);

        if ((pNode = InsertItem(hHashTable, &per)) == NULL) {
            fprintf(stderr, "cannot insert random item!");
            exit(EXIT_FAILURE);
        }
    }
    if ((pDelNode = InsertItem(hHashTable, &perSpec)) == NULL) {
        fprintf(stderr, "cannot insert item!");
        exit(EXIT_FAILURE);
    }

    /* DeleteItemNode(hHashTable, pDelNode);

    if (!DeleteItem(hHashTable, 7654)) {
        fprintf(stderr, "cannot delete item!..\n");
        exit(EXIT_FAILURE);
    }
    */
    if ((pper = FindItem(hHashTable, "Kaan Aslan")) == NULL) {
        fprintf(stderr, "cannot find item!..\n");
        exit(EXIT_FAILURE);
    }

    printf("Item found: %s, %d\n", pper->name, pper->no);

    /* ClearHashTable(hHashTable); */

    DisplayHashTable(hHashTable);

    CloseHashTable(hHashTable);

    return 0;
}

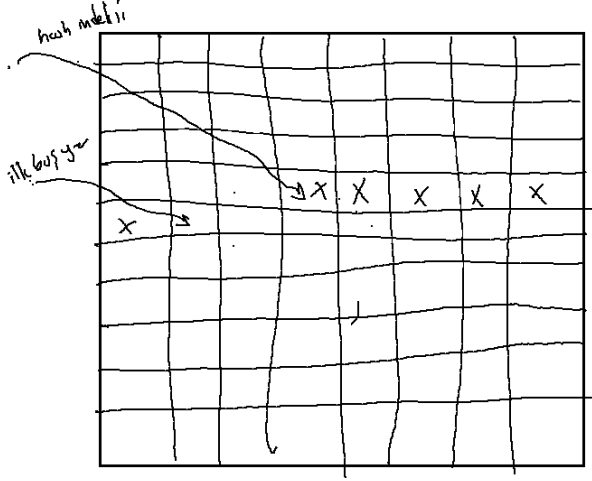
void getRandomItem(PERSON *pper)
{
    size_t k;

    for (k = 0; k < 31; ++k)
        pper->name[k] = rand() % 26 + 'A';
    pper->name[k] = '\0';
    pper->no = rand() % 100000;
}

```

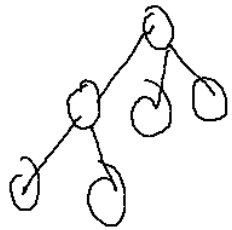
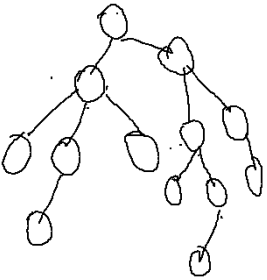
Hash Tablolarında Açık Adresleme (Open Addressing)

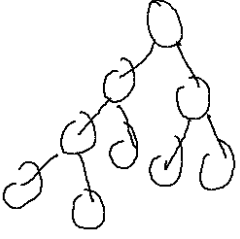
Yukarıda da belirtildiği gibi açık adresleme yöntemi pek çok alt yönetime ayrılmaktadır. En çok kullanılan açık adresleme yöntemi "doğrusal yoklama (linear probing)" denilen yöntemdir. Bu yöntemde anahtardan hash fonksiyonuyla bir tablo indeksi elde edilir. O indeksteki eleman (bucket) boşsa yerleştirme oraya yapılır. Doluysa sırasıyla elemanlar üzerinde ilerlenerek ilk boş yer bulunur. Eleman ilk boş yere eklenir. Arama yapılırken aynı biçimde yine dizi indeksi elde edilir ve elemanlar sırasıyla gözden geçirilir. Tabii ilk boş eleman görüldüğünde artık durulabilir. Bu yöntemde tablonun geniş açılması önemlidir. Eğer tablo küçük kalırsa bu durumda tabloda boş elemanlar azalır. Bu da hem eleman eklerken hem de ararken zaman kaybettirir.



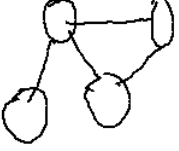
Ağaç Veri Yapıları (Tree Data Structure)

Ağaçlar düğümlerden (nodes) ve düğümleri birbirlerine bağlayan kenarlardan (edges) oluşmaktadır. Ağaçlar tek köke sahiptir ve ağaçlarda herhangi bir düğüme yalnızca tek bir yoldan ulaşılabilir. Ağacın köküne giden bir yol yoktur. Yani kök kendisine herhangi bir düğümden ulaşamayan başlangıç düğümüdür. Eğer düğümlerden ve kenarlardan oluşan veri yapılarında bir düğüme birden fazla yoldan gidilebiliyorsa artık onlara graf denilmektedir. Tabii ağaçlar da bir tür graf olarak değerlendirilebilir. Aşağıda birkaç örnek ağaç görülmektedir:





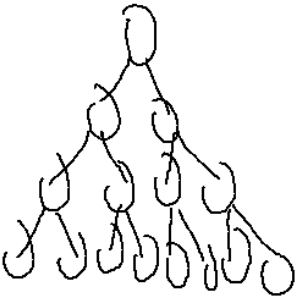
Ancak aşağıdaki veri yapısı bir ağaç değildir, graf belirtir:



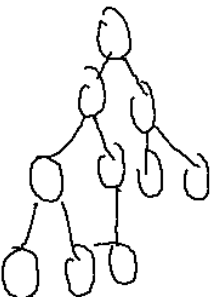
Ağaç Terminolojisi

Ağaçlarda kendisine hiçbir yerden gelinemeyen özel düğüme kök (root) denir. Bir düğümden gidilebilen ilk düğümlere o düğümün alt düğümleri (child nodes) denilmektedir. O düğüme gelinen ilk düğüme ise o düğümün üst düğümü (parent node) denilmektedir. Ağaçlarda kök dışındaki her düğümün bir ve yalnızca bir tane üst düğümü vardır. Fakat bir düğümün birden fazla alt düğümü olabilir. Üst düğümleri aynı olan düğümlere kardeş düğümler (sibling nodes) denilmektedir.

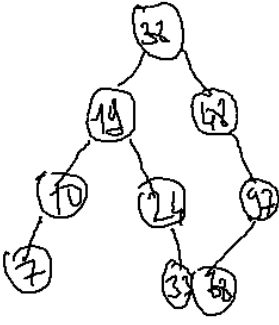
Eğer bir ağaçta her düğümün en fazla n tane alt düğümü olacağı yönünde bir koşul varsa böyle ağaçlara n 'li ağaçlar (n 'ary tree) denilmektedir. Pratikte en çok kullanılan n 'li ağaçlar ikili ağaçlardır. İkili ağaçlarda her düğümün sıfır, bir ya da en fazla iki alt düğümü olabilir. Ağaçlarda hiç alt düğümü olmayan düğümlere yaprak düğümler (leaf nodes) denilmektedir. Düğümleri birbirine bağlayan çizgiler kenar (edge) olarak isimlendirilir. Kökten bir düğüme gidebilmek için geçilen kenar sayısına o düğümün yüksekliği denilmektedir. Bir ağaçta kökten en uzaktaki yaprağa giden yüksekliğe ise ağacın yüksekliği denir. Bir n 'li ağaçta son kademe yapraklar dışındaki tüm düğümlerin tam olarak n tane alt düğümü varsa bu ağaca "tam dolu n 'li ağaç (full n 'ary tree)" denilmektedir. Örneğin aşağıda tam bir ikili ağaç görülmektedir:



Eğer bir ağaçta sondan bir önceki kademe dışındaki tüm düğümlerin tam olarak n tane alt düğümü varsa böyle ağaçlara tam n 'li ağaçlar (complete n 'ary tree) denilmektedir. Örneğin aşağıda tam bir ikili ağaç görülmektedir:



38, 19, 48, 10, 27, 24, 68, 7, 33



Ağaçların Veri Yapısı Olarak Temsil Edilmesi

Ağaçlar bir veri yapısı biçiminde oluşturulabilir. Bu durumda düğümler bir yapıyla temsil edilir. Bu yapının içerisinde anahtar ve o anahtara karşı gelen değer tutulur. Kenarlar aslında diğer düğümleri gösteren birer gösterici durumundadır. Örneğin bir ikili ağaç aşağıdaki gibi bir düğümlerle temsil edilebilir:

```
struct tagNODE {
    DATATYPE val;
    struct tagNODE *pLeft;
    struct tagNODE *pRight;
} NODE;
```

Tabii ağacın kökü ayrıca bir yerde saklanmalıdır. Yaprak düğümlerdeki kenar göstericilere NULL değeri yerleştirilir. Böylece bu düğümden başka bir yere gidilemeyeceği anlaşılır.

İkili Arama Ağaçları

İkili arama ağaçları (binary search tree) en çok kullanılan arama ağaçlarıdır. Burada her düğümün en fazla iki alt düğümü olabilir. Tipik olarak küçük değerler sol alt düğüme büyük değerler sağ alt düğüme yerleştirilir. Arama yapılırken kökten girilir. Aranacak değer o andaki düğümün değerinden küçükse sola, büyükse sağa sapılarak ilerlenir. Eleman eklenirken de sola, sağa gidilerek uygun yaprak pozisyonuna gelinir ekleme oraya yapılır. İkili arama ağacı için aşağıdaki bir handle alanı oluşturulabilir:

```
typedef int BOOL;

typedef struct tagPERSON {
    char name[32];
    /* */
} PERSON;

typedef int KEY;
typedef PERSON VALUE;

typedef struct tagNODE {
    struct tagNODE *pLeft;
    struct tagNODE *pRight;
    KEY key;
    VALUE value;
} NODE;

typedef struct tagQUEUE_NODE {
    NODE *pNode;
    struct tagQUEUE_NODE *pNext;
} QUEUE_NODE;
```

```

typedef struct tagBINARY_TREE {
    NODE *pRoot;
    size_t count;
    QUEUE_NODE *pHead;        /* Breadt First Traverve Queue Head Pointer */
    QUEUE_NODE *pTail;       /* Breadt First Traverve Queue Tail Pointer */
} BINARY_TREE, *HBINARY_TREE;

```

Burada handle alanı içerisindeki pRoot göstericisi ikili arama ağacının kök düğümünün yerini tutmaktadır. pHead ve pTail göstericileri daha ileride ele alacağımız "breadth first" dolaşım için gerekli olan kuyruk yapısı için bulundurulmuştur. Handle alanının yaratılması şöyle bir fonksiyonla yapılabilir:

```

HBINARY_TREE CreateBinaryTree(void)
{
    HBINARY_TREE hBTree;

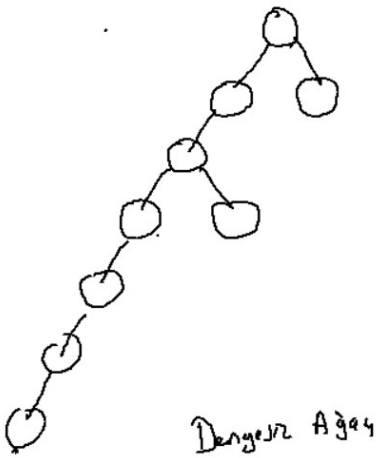
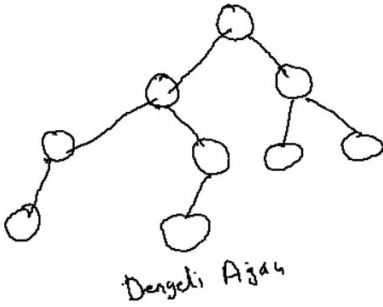
    if ((hBTree = (HBINARY_TREE)malloc(sizeof(BINARY_TREE))) == NULL)
        return NULL;

    hBTree->pRoot = NULL;
    hBTree->count = 0;
    hBTree->pTail = hBTree->pHead = NULL;

    return hBTree;
}

```

İkili ağaca eleman eklenmesi eğer ağaç dengeliyse logaritmik karmaşıklıkla yapılmaktadır. Ağaç dengeli değilse eleman ekleme işlemi en kötü olasılıkla (worst case) doğrusal karmaşıklıkla yapılmaktadır. Elemanın aranması süreci de eğer ağaç dengeliyse en kötü olasılıkla (worst case) logaritmik karmaşıklıkta dengeli değilse doğrusal karmaşıklıkta yapılabilmektedir. İkili ağacın dengeli olmaması yaprak yüksekliklerinin birbirlerinden çok farklı olabilmesi anlamına gelir. Hatta dengeli olmayan ikili ağaçlar en kötü durumda bir bağlı listeye benzerler. Örneğin:



Dengeli ikili arama ağaçlarında arama işlemlerinin "ikili arama (binary search)" biçimine geldiğine dikkat ediniz. O halde ikili arama ağaçları aslında sıralı olmayan düğümlerde ikili arama yönteminin uygulanmasını sağlamaktadır. Pekiyi dengelenmiş ikili ağaçlar yerine sıralı diziler kullanılamaz mı? İşte eğer biz diziyi sıralı tutmaya çalışırsak yeni bir elemanın

eklenmesinde kaydırma yapmak durumunda kalırız. Bukaydırma da doğrusal karmaşıklıkta yapılabilir. Halbuki dengelenmiş ikili ağaçlarda eleman eklenmesi de logaritmik karmaşıklıktadır.

İkili Ağaca Eleman Eklenmesi

İkil arama ağacına eleman eklerken önce ekleme yerinin bulunması gerekir. Bunun için yine kök düğümünden girilir, duruma göre sola ve sağa gidilerek eklenecek düğümün yeri tespit edilir ve ekleme yapılır. Örneğin:

```
BOOL InsertItem(HBINARY_TREE hBTree, KEY key, const VALUE *pValue)
{
    NODE *pNewNode, *pNode, *pParentNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return FALSE;
    pNewNode->pLeft = pNewNode->pRight = NULL;
    pNewNode->key = key;
    pNewNode->value = *pValue;

    if (hBTree->pRoot == NULL) {
        hBTree->pRoot = pNewNode;
        ++hBTree->count;
        return TRUE;
    }

    pNode = hBTree->pRoot;

    while (pNode != NULL) {
        pParentNode = pNode;
        if (key < pNode->key)
            pNode = pNode->pLeft;
        else if (key > pNode->key)
            pNode = pNode->pRight;
        else {
            pNode->value = *pValue;
            return TRUE;
        }
    }

    if (key > pParentNode->key)
        pParentNode->pRight = pNewNode;
    else
        pParentNode->pLeft = pNewNode;

    ++hBTree->count;

    return TRUE;
}
```

Ekleme işlemi özyinelemeli olarak da yapılabilir. Bunun için kök düğümünden girilir. Eklenecek sol ya da sağ yön tespit edilir. Eğer orada NULL gösterici varsa oraya ekleme yapılır. Yoksa o düğümle fonksiyon kendisini çağırır.

İkili Arama Ağacında Arama Yapılması

İkili arama ağaçlarında arama yapılırken kök düğümünden girilir. Duruma göre sola ve sağa sapılarak eleman bulunmaya çalışılır. Örnek bir arama işlemi şöyle yapılabilir:

```
VALUE *FindItem(HBINARY_TREE hBTree, KEY key)
{
    NODE *pNode;

    pNode = hBTree->pRoot;

    while (pNode != NULL) {
```

```

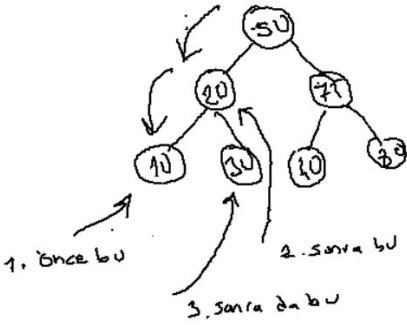
    if (key > pNode->key)
        pNode = pNode->pRight;
    else if (key < pNode->key)
        pNode = pNode->pLeft;
    else
        return &pNode->value;
}

return NULL;
}

```

İkili Ağacın Dolaşılması

İkili ağacın dolaşılması (traverse / walk) özyinelemeli bir algoritmayla üç farklı yolla yapılabilmektedir: Inorder, Preorder ve Postorder dolaşım. Bu isimler üst düğümün alt düğümlere göre hangi sırada ziyaret edileceğine göre verilmiştir. Preorder dolaşımında düğümler alt-üst-alt ziyaret edilir. Yani önce sol ya da sağ düğümü ziyaret etmek için özyineleme uygulanır. Özyinelemeden çıktığında üst düğüm ziyaret edilir. Sonra diğer alt düğüm ziyaret edilir.



Inorder dolaşım anahtarları sıralı bir biçimde elde etmeyi sağlar. Eğer önce sol alt düğüm ziyaret edilirse anahtarlar küçükten-büyüğe, eğer önce sağ alt düğüm ziyaret edilirse büyükten-küçüğe elde edilir.

Küçükten büyüğe inorder dolaşımın özyinelemeli temel algoritması şöyledir:

```

void WalkInOrderAscending(NODE *pNode)
{
    if (pNode->pLeft != NULL)
        WalkInOrder(pNode->pLeft);

    printf("%d\n", pNode->val.no);

    if (pNode->pRight != NULL)
        WalkInOrder(pNode->pRight);
}

```

Büyükten küçüğe inorder dolaşım da benzer biçimde yapılabilir:

```

void WalkInOrderDescending(NODE *pNode)
{
    if (pNode->pRight != NULL)
        WalkInOrder(pNode->pRight);

    printf("%d\n", pNode->val.no);

    if (pNode->pLeft != NULL)
        WalkInOrder(pNode->pLeft);
}

```

Postorder dolaşımında düğümler alt-alt-üst biçiminde ziyaret edilir. Düğümlerin silinmesi tipik olarak bu tarz bir dolaşım ile sağlanmaktadır. Postorder dolaşımın da özyinelemeli temel algoritması şöyledir:

```

void WalkPostOrder(NODE *pNode)
{
    if (pNode->pLeft != NULL)
        WalkPostOrder(pNode->pLeft);

    if (pNode->pRight != NULL)
        WalkPostOrder(pNode->pRight);

    printf("%d\n", pNode->val.no);
}

```

Preorder dolaşımında düğümler üst-alt-alt biçiminde ziyaret edilir. Preorder dolaşım tipik olarak ağacı görüntülemek amacıyla uygulanmaktadır. Predorder dolaşımın da özyinelemeli temel algoritması şöyledir:

```

void WalkPreOrder(NODE *pNode)
{
    printf("%d\n", pNode->val.no);

    if (pNode->pLeft != NULL)
        WalkPreOrder(pNode->pLeft);

    if (pNode->pRight != NULL)
        WalkPreOrder(pNode->pRight);
}

```

Yukarıdaki handle sistemiyle oluşturmuş olduğumuz ağacı dolaşırken ziyaret sırasında programcının ne yapacağını bilmediğimizden dolayı dolaşım işleminde bir "call back" fonksiyondan faydalanabiliriz. Bu callback fonksiyon FALSE ile geri dönerse dolaşım sonlandırılabilir. Budurumda örneğin Inorder dolaşım şöyle olacaktır:

```

static BOOL WalkInOrderAscendingRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (pNode->pLeft != NULL)
        if (!WalkInOrderAscendingRecur(pNode->pLeft, Proc))
            return FALSE;
    if (!Proc(&pNode->key, &pNode->value))
        return FALSE;
    if (pNode->pRight != NULL)
        if (!WalkInOrderAscendingRecur(pNode->pRight, Proc))
            return FALSE;

    return TRUE;
}

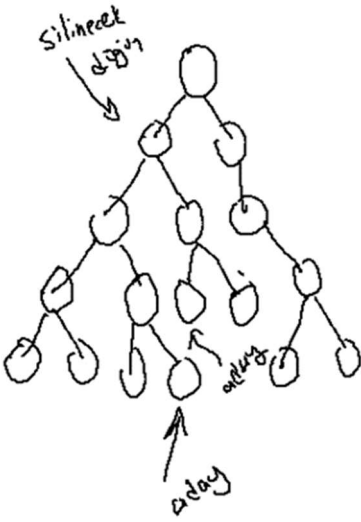
BOOL WalkInOrderAscending(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (hBTree->pRoot != NULL)
        return WalkInOrderAscendingRecur(hBTree->pRoot, Proc);

    return TRUE;
}

```

İkili Ağaçtan Düğüm Silinmesi

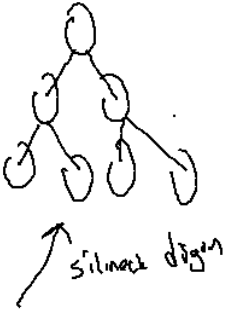
İkili ağaçtan eleman silmek biraz daha zahmetlidir. Çünkü silinecek elemanın yerine ağaçtaki başka elemanın taşınması gerekir. Basit bir fikir şöyle olabilir: Silinecek düğüm onun solundakilerden büyük sağındakilerden küçük olduğuna göre onun yerine geçecek düğüm onun solundakilerin en büyüğü ya da sağındakilerin en küçüğü olabilir. Örneğin:



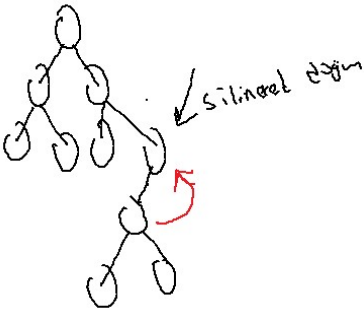
Silme işleminde 4 özel durum söz konusudur:

- 1) Silinecek düğümün hiç alt düğümü yoktur.
- 2) Silinecek düğümün yalnızca bir alt düğümü vardır.
- 3) Silinecek düğümün iki alt düğümü de vardır.
- 4) Silinecek düğüm köktür.

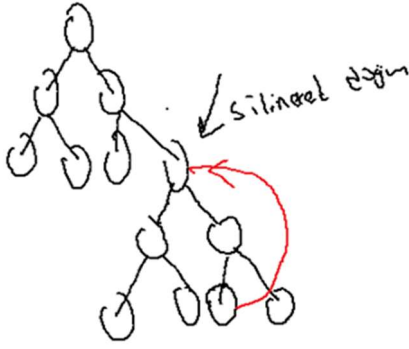
Silinecek düğümün hiç alt düğümü yoksa düğüm doğrudan silinir. Tabii onun üst düğümünün uygun linki NULL yapılır.



Silinecek düğümün yalnızca bir alt düğümü varsa bu alt düğüm silinecek düğümün yerine geçer:



Eğer silinecek düğümün iki alt düğümü de varsa bu durumda onun sağındakinin en küçüğü ya da solundakinin en büyüğü onun yerine geçer.



Eğer silinecek düğüm kökse handle alanında kökü güncellemek gerekir. Bu özel bir durum oluşturabilmektedir. Tabi handle alanında biz kök düğümün adresini değil onu bir düğüm olarak (dummy düğüm olarak) tutarsak bu özel durumdan kurtulabiliriz.

```
static void SubTreeShift(HBINARY_TREE hBTree, NODE *pUpParentNode, NODE *pUpNode, NODE *pDownNode)
{
    if (pUpParentNode == NULL)
        hBTree->pRoot = pDownNode;
    else if (pUpParentNode->pLeft == pUpNode)
        pUpParentNode->pLeft = pDownNode;
    else
        pUpParentNode->pRight = pDownNode;
}
```

```
BOOL DeleteItem(HBINARY_TREE hBTree, KEY key)
{
    NODE *pDeleteNode, *pDeleteParentNode, *pReplaceParentNode, *pReplaceNode;

    pDeleteNode = hBTree->pRoot;
    pDeleteParentNode = NULL;

    while (pDeleteNode != NULL) {
        if (pDeleteNode->key == key)
            break;
        pDeleteParentNode = pDeleteNode;
        if (pDeleteNode->key > key)
            pDeleteNode = pDeleteNode->pLeft;
        else if (pDeleteNode->key < key)
            pDeleteNode = pDeleteNode->pRight;
    }

    if (pDeleteNode == NULL)
        return FALSE;

    if (pDeleteNode->pLeft == NULL)
        SubTreeShift(hBTree, pDeleteParentNode, pDeleteNode, pDeleteNode->pRight);
    else if (pDeleteNode->pRight == NULL)
        SubTreeShift(hBTree, pDeleteParentNode, pDeleteNode, pDeleteNode->pLeft);
    else {
        pReplaceParentNode = pDeleteNode;
        pReplaceNode = pDeleteNode->pRight;
        while (pReplaceNode->pLeft != NULL) {
            pReplaceParentNode = pReplaceNode;
            pReplaceNode = pReplaceNode->pLeft;
        }
        if (pReplaceParentNode != pDeleteNode) {
            SubTreeShift(hBTree, pReplaceParentNode, pReplaceNode, pReplaceNode->pRight);
            pReplaceNode->pRight = pDeleteNode->pRight;
        }
        SubTreeShift(hBTree, pDeleteParentNode, pDeleteNode, pReplaceNode);
        pReplaceNode->pLeft = pDeleteNode->pLeft;
    }
}
```

```

}

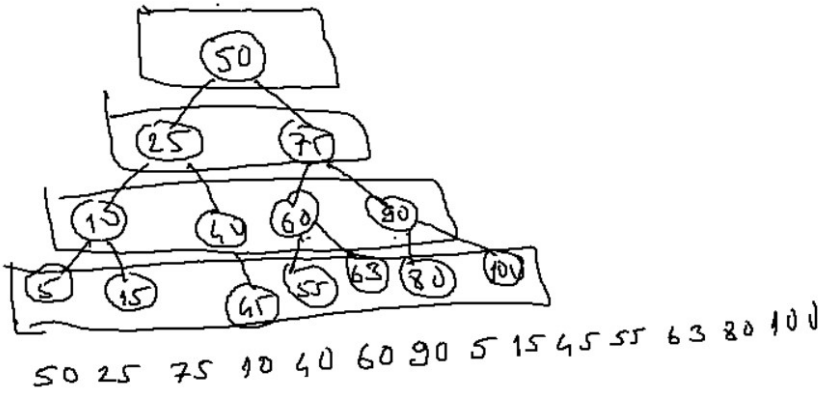
return TRUE;
}

```

Burada SubTreeShift fonksiyonu belli bir düğümü başka bir düğüme monte eder. Bir düğümü başka bir düğüme monte etmek için monte edilecek düğümün üst düğümünde değişiklik yapmak gerekir. Bu nedenle bu fonksiyon monte edilecek yer ile onun üst düğümünü de parametre olarak almıştır.

İkili Ağaçlarda Depth-First ve Breadth-First Dolaşım

Ağaçlarda arama yaparken eğer ağaç bir arama ağacı değilse her düğüme bakmak gerekir. Aslında bu durum genel olarak graflar için de söylenebilir. Böylece ağaçları dolaşırken iki strateji izlenebilmektedir. Depth-First dolaşımında her zaman dibine dalınarak özinelemeli bir gidiş vardır. Yukarıda incelediğimiz Inorder, postorder ve preorder dolaşım kategorisi olarak depth-first dolaşım türündendir. Halbuki breadth-first dolaşımında ağaç (ya da genel olarak graf) kademeli dolaşılır. Örneğin:



Breadth-First dolaşım için bir kuyruk sistemi oluşturulur. Kök düğüm kuyruğa atılarak işe başlanır. Sonra kuyruktan eleman alınır her alınan elemanın alt düğümleri kuyruğa atılır. Bu biçimde ilerlenerek gidilir. Bazı durumlarda breadth-first dolaşım istenebilmektedir. Örneğin bir arama ağacında değil fakat hiyerarşik bir organizasyonda üst yönetimden alt tarafa doğru kişileri dolaşmak isteyebiliriz.

İkili arama ağacı veri yapısının kodları aşağıda verilmiştir:

```

/* BinarySearchTree.h */

#ifndef BINARYTREE_H_
#define BINARYTREE_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE    0
#define TRUE     1

/* Type Declaraions */

typedef int BOOL;

typedef struct tagPERSON {
    char name[32];
    /* */
} PERSON;

typedef int KEY;
typedef PERSON VALUE;

```



```

typedef struct tagNODE {
    struct tagNODE *pLeft;
    struct tagNODE *pRight;
    KEY key;
    VALUE value;
} NODE;

typedef struct tagQUEUE_NODE {
    NODE *pNode;
    struct tagQUEUE_NODE *pNext;
} QUEUE_NODE;

typedef struct tagBINARY_TREE {
    NODE *pRoot;
    size_t count;
    QUEUE_NODE *pHead;          /* Breadt First Traverve Queue Head Pointer */
    QUEUE_NODE *pTail;         /* Breadt First Traverve Queue Tail Pointer */
} BINARY_TREE, *HBINARY_TREE;

/* Function prototypes */

HBINARY_TREE CreateBinaryTree(void);
BOOL InsertItem(HBINARY_TREE hBTree, KEY key, const VALUE *pValue);
BOOL InsertItemAlternativeWay(HBINARY_TREE hBTree, KEY key, const VALUE *pValue);
VALUE *FindItem(HBINARY_TREE hBTree, KEY key);
BOOL WalkInOrderAscending(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value));
BOOL WalkInOrderDescending(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value));
BOOL WalkPreOrderLeft(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value));
BOOL WalkPreOrderRight(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value));
BOOL WalkPostOrderLeft(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value));
BOOL WalkPostOrderRight(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value));
BOOL WalkBreadthFirst(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value));
BOOL DeleteItem(HBINARY_TREE hBTree, KEY key);
void ClearBinaryTree(HBINARY_TREE hBTree);

/* Macros */

#define GetCount(hBTree)      ((hBTree)->count)

#endif

/* BinarySearchTree.c */

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

static BOOL WalkInOrderAscendingRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value));
static BOOL WalkInOrderDescendingRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value));
static BOOL WalkPreOrderLeftRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value));
static BOOL WalkPreOrderRightRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value));
static BOOL WalkPostOrderLeftRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value));
static BOOL WalkPostOrderRightRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value));
static BOOL PutItem(HBINARY_TREE hBTree, const NODE *pNode);
static NODE *GetItem(HBINARY_TREE hBTree);
static void SubTreeShift(HBINARY_TREE hBTree, NODE *pUpParentNode, NODE *pUpNode, NODE *pDownNode);
static void ClearBinaryTreeRecur(NODE *pNode);
void ClearQueue(HBINARY_TREE hBTree);

HBINARY_TREE CreateBinaryTree(void)
{
    HBINARY_TREE hBTree;

    if ((hBTree = (HBINARY_TREE)malloc(sizeof(BINARY_TREE))) == NULL)
        return NULL;
}

```

```

hBTree->pRoot = NULL;
hBTree->count = 0;
hBTree->pTail = hBTree->pHead = NULL;

return hBTree;
}

BOOL InsertItem(HBINARY_TREE hBTree, KEY key, const VALUE *pValue)
{
    NODE *pNewNode, *pNode, *pParentNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return FALSE;
    pNewNode->pLeft = pNewNode->pRight = NULL;
    pNewNode->key = key;
    pNewNode->value = *pValue;

    if (hBTree->pRoot == NULL) {
        hBTree->pRoot = pNewNode;
        ++hBTree->count;
        return TRUE;
    }

    pNode = hBTree->pRoot;

    while (pNode != NULL) {
        pParentNode = pNode;
        if (key < pNode->key)
            pNode = pNode->pLeft;
        else if (key > pNode->key)
            pNode = pNode->pRight;
        else {
            pNode->value = *pValue;
            return TRUE;
        }
    }

    if (key > pParentNode->key)
        pParentNode->pRight = pNewNode;
    else
        pParentNode->pLeft = pNewNode;

    ++hBTree->count;

    return TRUE;
}

BOOL InsertItemAlternativeWay(HBINARY_TREE hBTree, KEY key, const VALUE *pValue)
{
    NODE *pNewNode, *pNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return FALSE;
    pNewNode->pLeft = pNewNode->pRight = NULL;
    pNewNode->key = key;
    pNewNode->value = *pValue;

    if (hBTree->pRoot == NULL) {
        hBTree->pRoot = pNewNode;
        ++hBTree->count;
        return TRUE;
    }

    pNode = hBTree->pRoot;

```

```

for (;;) {
    if (key < pNode->key)
        if (pNode->pLeft == NULL) {
            pNode->pLeft = pNewNode;
            break;
        }
        else
            pNode = pNode->pLeft;
    else if (key > pNode->key)
        if (pNode->pRight == NULL) {
            pNode->pRight = pNewNode;
            break;
        }
        else
            pNode = pNode->pRight;
    else {
        pNode->value = *pValue;
        return TRUE;
    }
}

++hBTree->count;

return TRUE;
}

VALUE *FindItem(HBINARY_TREE hBTree, KEY key)
{
    NODE *pNode;

    pNode = hBTree->pRoot;

    while (pNode != NULL) {
        if (key > pNode->key)
            pNode = pNode->pRight;
        else if (key < pNode->key)
            pNode = pNode->pLeft;
        else
            return &pNode->value;
    }

    return NULL;
}

static BOOL WalkInOrderAscendingRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (pNode->pLeft != NULL)
        if (!WalkInOrderAscendingRecur(pNode->pLeft, Proc))
            return FALSE;
    if (!Proc(&pNode->key, &pNode->value))
        return FALSE;
    if (pNode->pRight != NULL)
        if (!WalkInOrderAscendingRecur(pNode->pRight, Proc))
            return FALSE;

    return TRUE;
}

BOOL WalkInOrderAscending(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (hBTree->pRoot != NULL)
        return WalkInOrderAscendingRecur(hBTree->pRoot, Proc);

    return TRUE;
}

```

```

static BOOL WalkInOrderDescendingRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (pNode->pRight != NULL)
        if (!WalkInOrderDescendingRecur(pNode->pRight, Proc))
            return FALSE;
    if (!Proc(&pNode->key, &pNode->value))
        return FALSE;
    if (pNode->pLeft != NULL)
        if (!WalkInOrderDescendingRecur(pNode->pLeft, Proc))
            return FALSE;

    return TRUE;
}

BOOL WalkInOrderDescending(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (hBTree->pRoot != NULL)
        return WalkInOrderDescendingRecur(hBTree->pRoot, Proc);

    return TRUE;
}

static BOOL WalkPreOrderLeftRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (!Proc(&pNode->key, &pNode->value))
        return FALSE;

    if (pNode->pLeft != NULL)
        if (!WalkPreOrderLeftRecur(pNode->pLeft, Proc))
            return FALSE;

    if (pNode->pRight != NULL)
        if (!WalkPreOrderLeftRecur(pNode->pRight, Proc))
            return FALSE;

    return TRUE;
}

BOOL WalkPreOrderLeft(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (hBTree->pRoot != NULL)
        return WalkPreOrderLeftRecur(hBTree->pRoot, Proc);

    return TRUE;
}

static BOOL WalkPreOrderRightRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (!Proc(&pNode->key, &pNode->value))
        return FALSE;

    if (pNode->pRight != NULL)
        if (!WalkPreOrderRightRecur(pNode->pRight, Proc))
            return FALSE;

    if (pNode->pLeft != NULL)
        if (!WalkPreOrderRightRecur(pNode->pLeft, Proc))
            return FALSE;

    return TRUE;
}

BOOL WalkPreOrderRight(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (hBTree->pRoot != NULL)
        return WalkPreOrderRightRecur(hBTree->pRoot, Proc);
}

```

```

    return TRUE;
}

static BOOL WalkPostOrderLeftRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (pNode->pLeft != NULL)
        if (!WalkPostOrderLeftRecur(pNode->pLeft, Proc))
            return FALSE;

    if (pNode->pRight != NULL)
        if (!WalkPostOrderLeftRecur(pNode->pRight, Proc))
            return FALSE;

    if (!Proc(&pNode->key, &pNode->value))
        return FALSE;

    return TRUE;
}

BOOL WalkPostOrderLeft(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (hBTree->pRoot != NULL)
        return WalkPostOrderLeftRecur(hBTree->pRoot, Proc);

    return TRUE;
}

static BOOL WalkPostOrderRightRecur(NODE *pNode, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (pNode->pRight != NULL)
        if (!WalkPostOrderRightRecur(pNode->pRight, Proc))
            return FALSE;

    if (pNode->pLeft != NULL)
        if (!WalkPostOrderRightRecur(pNode->pLeft, Proc))
            return FALSE;

    if (!Proc(&pNode->key, &pNode->value))
        return FALSE;

    return TRUE;
}

BOOL WalkPostOrderRight(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    if (hBTree->pRoot != NULL)
        return WalkPostOrderRightRecur(hBTree->pRoot, Proc);

    return TRUE;
}

BOOL WalkBreadthFirst(HBINARY_TREE hBTree, BOOL(*Proc)(const KEY *key, VALUE *value))
{
    NODE *pNode;
    BOOL result;

    result = TRUE;

    if (hBTree->pRoot != NULL && !PutItem(hBTree, hBTree->pRoot))
        return FALSE;

    while ((pNode = GetItem(hBTree)) != NULL) {
        if (!Proc(&pNode->key, &pNode->value)) {
            result = FALSE;
            break;
        }
    }
}

```

```

    }
    if (pNode->pLeft != NULL && !PutItem(hBTree, pNode->pLeft)) {
        result = FALSE;
        break;
    }
    if (pNode->pRight != NULL && !PutItem(hBTree, pNode->pRight)) {
        result = FALSE;
        break;
    }
}

ClearQueue(hBTree);

return result;
}

static void SubTreeShift(HBINARY_TREE hBTree, NODE *pUpParentNode, NODE *pUpNode, NODE *pDownNode)
{
    if (pUpParentNode == NULL)
        hBTree->pRoot = pDownNode;
    else if (pUpParentNode->pLeft == pUpNode)
        pUpParentNode->pLeft = pDownNode;
    else
        pUpParentNode->pRight = pDownNode;
}

BOOL DeleteItem(HBINARY_TREE hBTree, KEY key)
{
    NODE *pDeleteNode, *pDeleteParentNode, *pReplaceParentNode, *pReplaceNode;

    pDeleteNode = hBTree->pRoot;
    pDeleteParentNode = NULL;

    while (pDeleteNode != NULL) {
        if (pDeleteNode->key == key)
            break;
        pDeleteParentNode = pDeleteNode;
        if (pDeleteNode->key > key)
            pDeleteNode = pDeleteNode->pLeft;
        else if (pDeleteNode->key < key)
            pDeleteNode = pDeleteNode->pRight;
    }

    if (pDeleteNode == NULL)
        return FALSE;

    if (pDeleteNode->pLeft == NULL)
        SubTreeShift(hBTree, pDeleteParentNode, pDeleteNode, pDeleteNode->pRight);
    else if (pDeleteNode->pRight == NULL)
        SubTreeShift(hBTree, pDeleteParentNode, pDeleteNode, pDeleteNode->pLeft);
    else {
        pReplaceParentNode = pDeleteNode;
        pReplaceNode = pDeleteNode->pRight;
        while (pReplaceNode->pLeft != NULL) {
            pReplaceParentNode = pReplaceNode;
            pReplaceNode = pReplaceNode->pLeft;
        }
        if (pReplaceParentNode != pDeleteNode) {
            SubTreeShift(hBTree, pReplaceParentNode, pReplaceNode, pReplaceNode->pRight);
            pReplaceNode->pRight = pDeleteNode->pRight;
        }
        SubTreeShift(hBTree, pDeleteParentNode, pDeleteNode, pReplaceNode);
        pReplaceNode->pLeft = pDeleteNode->pLeft;
    }

    return TRUE;
}

```

```

}

static void ClearBinaryTreeRecur(NODE *pNode)
{
    if (pNode->pLeft != NULL)
        ClearBinaryTreeRecur(pNode->pLeft);
    if (pNode->pRight != NULL)
        ClearBinaryTreeRecur(pNode->pRight);

    free(pNode);
}

void ClearBinaryTree(HBINARY_TREE hBTree)
{
    if (hBTree->pRoot != NULL) {
        ClearBinaryTreeRecur(hBTree->pRoot);

        hBTree->pRoot = NULL;
        hBTree->count = 0;
    }
}

static BOOL PutItem(HBINARY_TREE hBTree, const NODE *pNode)
{
    QUEUE_NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(QUEUE_NODE))) == NULL)
        return FALSE;
    pNewNode->pNode = pNode;
    pNewNode->pNext = NULL;

    if (hBTree->pTail != NULL)
        hBTree->pTail->pNext = pNewNode;
    else
        hBTree->pHead = pNewNode;
    hBTree->pTail = pNewNode;

    return TRUE;
}

static NODE *GetItem(HBINARY_TREE hBTree)
{
    QUEUE_NODE *pQueueNode;
    NODE *pNode = NULL;

    if (hBTree->pHead == NULL)
        return NULL;

    pQueueNode = hBTree->pHead;
    hBTree->pHead = pQueueNode->pNext;
    if (hBTree->pHead == NULL)
        hBTree->pTail = NULL;

    pNode = pQueueNode->pNode;
    free(pQueueNode);

    return pNode;
}

static void ClearQueue(HBINARY_TREE hBTree)
{
    QUEUE_NODE *pQueueNode, *pTempNode;

    pQueueNode = hBTree->pHead;
    while (pQueueNode != NULL) {
        pTempNode = pQueueNode->pNext;

```

```

        free(pQueueNode);
        pQueueNode = pTempNode;
    }

    hBTree->pHead = hBTree->pTail = NULL;
}

/* Test.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "BinaryTree.h"

void GetRandomKeyVal(KEY *pKey, VALUE *pVal)
{
    int i;

    for (i = 0; i < 31; ++i)
        pVal->name[i] = 'A' + rand() % 26;
    pVal->name[i] = '\0';

    *pKey = rand() % 1000;
}

BOOL Disp(const KEY *key, VALUE *value)
{
    printf("%d ---> %s\n", *key, value->name);

    return TRUE;
}

int main(void)
{
    HBINARY_TREE hBTree;
    int i;
    KEY key;
    VALUE value;
    KEY keys[] = { 100, 50, 120, 25, 75, 115, 150, 10, 30, 60, 80, 145, 200};
    VALUE values[] = { {"Sacit Erol"}, {"Ayşe Er"}, {"Demir Büyüközkaya"}, {"Harun Şanlı"},
        {"Fehmi Sakin"}, {"Olcay Torun"}, {"Kaan Aslan"}, {"Güray Sönmez"}, {"Bulut Hakkı"},
        {"Defne Yılmaz"}, {"Timur Selçuk"}, {"Kazım Koç"}, {"Gül Koca"} };

    srand(time(NULL));

    if ((hBTree = CreateBinaryTree()) == NULL) {
        fprintf(stderr, "Cannot create binary tree!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 20; ++i) {
        GetRandomKeyVal(&key, &value);
        if (!InsertItem(hBTree, key, &value)) {
            fprintf(stderr, "cannot insert item!..\n");
            exit(EXIT_FAILURE);
        }
        printf("%d --> %s\n", key, value.name);
    }

    for (i = 0; i < sizeof(keys) / sizeof(*keys); ++i) {
        if (!InsertItem(hBTree, keys[i], &values[i])) {
            fprintf(stderr, "cannot insert item!..\n");
            exit(EXIT_FAILURE);
        }
        printf("%d --> %s\n", keys[i], values[i].name);
    }
}

```



```

}

printf("-----\n");
printf("Inorder Ascending\n\n");
WalkInOrderAscending(hBTree, Disp);
printf("-----\n");
printf("Inorder Descending\n\n");
WalkInOrderDescending(hBTree, Disp);
printf("-----\n");
printf("Preorder Left\n\n");
WalkPreOrderLeft(hBTree, Disp);
printf("-----\n");
printf("Preorder Right\n\n");
WalkPreOrderRight(hBTree, Disp);
printf("-----\n");
printf("Postorder Left\n\n");
WalkPostOrderLeft(hBTree, Disp);
printf("-----\n");
printf("Postorder Right\n\n");
WalkPostOrderRight(hBTree, Disp);

ClearBinaryTree(hBTree);

for (i = 0; i < sizeof(keys) / sizeof(*keys); ++i) {
    if (!InsertItemAlternativeWay(hBTree, keys[i], &values[i])) {
        fprintf(stderr, "cannot insert item!..\n");
        exit(EXIT_FAILURE);
    }
}

printf("-----\n");
printf("Breadth First\n\n");

WalkBreadthFirst(hBTree, Disp);

if (!DeleteItem(hBTree, 100)) {
    fprintf(stderr, "Cannot delete item!..\n");
    exit(EXIT_FAILURE);
}

printf("-----\n");
printf("Breadth First\n\n");

WalkBreadthFirst(hBTree, Disp);

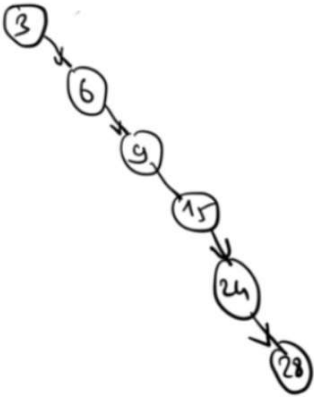
return 0;
}

```

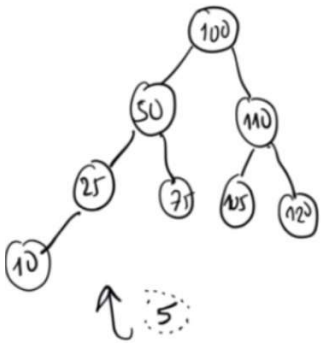
İkili Arama Ağaçlarının Dengelenmesi

İkili arama ağaçlarında yaprakların yükseklikleri birbirlerinden çok farklı olabilmektedir. Bu durum arama performansını düşürücü bir etki yaratır. Hatta öyle bir durum oluşabilir ki ikili arama ağacı bir bağlı liste biçimine bile dönüşebilir. Böylece arama işleminde en kötü durum karmaşıklığı $O(N)$ haline gelir. Örneğin aşağıdaki anahtarları ikili ağaca yerleştirmeye çalışalım:

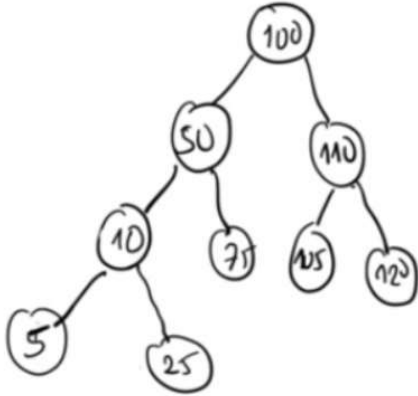
3 6 10 15 24 28



İşte ağacın her yaprağının birbirlerine yakın bir yükseklikte olmasını sağlamak için "dengelenmiş (balanced)" ağaçlar kullanılmaktadır. Dengeleme eleman eklenirken ya da silinirken $O(1)$ karmaşıklıkta ya da $O(\log N)$ karmaşıklıkta işlemlerle dengenin bozulmasını engelleme faaliyetine denilmektedir. Örneğin aşağıdaki ikili arama ağacına 5 anahtarını ekleyecek olalım:



Bu ekleme işlemi yapıldığında ağacın dengesi bozulacaktır. Biz bunu anlayıp bir düzeltme yapabiliriz:



Tabii eklenecek elemana göre dengeleme işleminde yapılması gereken şeyler farklılaşabilmektedir. Genel olarak ikili arama ağaçlarında en çok kullanılan dengeleme algoritmaları "AVL (Adelson-Velsky-Landis)" ve "Red Black Tree" isimli algoritmalarıdır. AVL dengelemesi pek çok yerde daha fazla tercih edilmektedir. İki dengeleme algoritmasının birbirlerine göre avantajları ve dezavantajları vardır. İkili arama ağaçlarının dengelenmesi "Sistem Programlama ve İleri C Uygulamaları-II" kursumuzda ele alınmaktadır.

İkili Arama Ağaçları İle Hash Tablolarının Karşılaştırılması

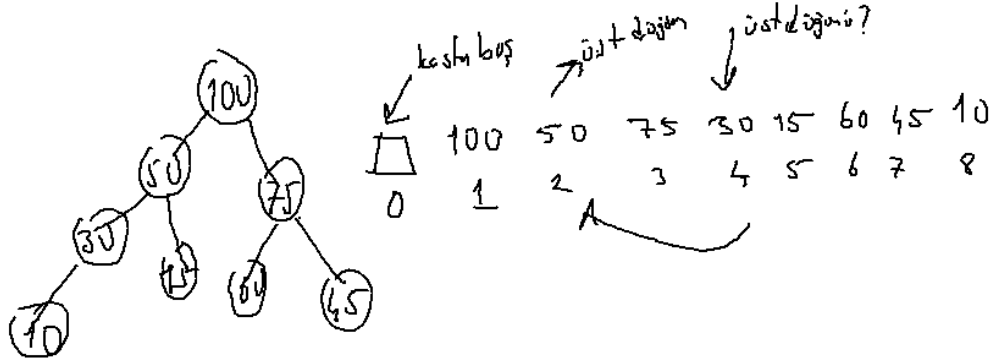
Hash tablolarında elemanlar hiç döngü olmadan $O(1)$ karmaşıklıkta eklenebilmektedir. Halbuki dengelenmiş ikili arama ağaçlarına elemanlar $O(\log N)$ karmaşıklıkta eklenir. Hash tablolarında arama tablonun uzunluğu makul tutulmuşsa yine $O(1)$ karmaşıklıkta gerçekleşebilmektedir. Halbuki dengelenmiş ikili arama ağaçlarında arama $O(\log N)$ karmaşıklıkta

yapılmaktadır. Örneğin 65000 civarında elemanın bulunduğu bir ikili arama ağacında en kötü olasılıkla arama için gereken adım sayısı 16'dır. Eleman silme işlemi eleman arama işlemine benzer karmaşıklıklarda yapılmaktadır. Hash tabloları için genel olarak daha fazla bir alana gereksinim duyulmaktadır. Halbuki dengelenmiş ikili ağaçlarda yalnızca ağaçtaki elemanlar için yerler ayrılmaktadır. Eklenecek eleman sayısı çok fazla olduğunda hash tablolarını büyütme dikkate değer bir zaman ve bellek kaybına yol açmaktadır. Halbuki dengelenmiş ikili ağaçlarda büyütme özel bir durum değildir. Ayrıca hash tablolarının sıralı dolaşıma izin vermediğine de dikkatinizi çekmek istiyoruz. Halbuki ikili ağaçlar sıralı bir biçimde dolaşılabilir.

Heap Veri Yapısı (Heap Ağaçları)

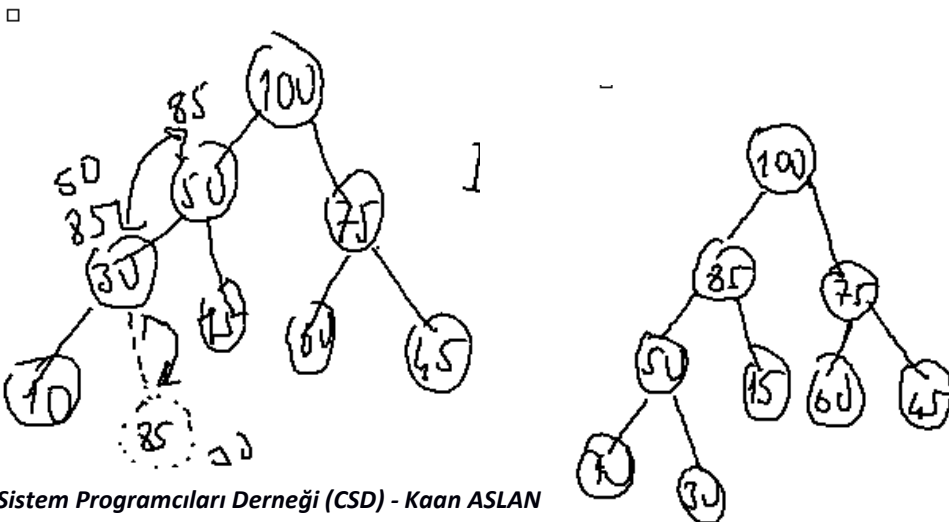
Heap özel bir ağaç türüdür. Öyle ki heap ağacında her zaman üst düğümün anahtarı alt düğümlerinin anahtarlarından ya daha büyüktür (max heap) ya da daha küçüktür (min heap). Heap ağaçları her zaman tam (complete) ağaçlardır. Bunun dışında heap ağacının başka bir koşulu yoktur. Heap ağaçları da n'li olabilir. Ancak en çok kullanılan heap ağaçları ikili heap ağaçlarıdır (binary heap).

Heap ağaçları aslında ağaç olarak gerçekleştirilmek için tasarlanmamıştır dizi olarak gerçekleştirilmek için tasarlanmıştır. Şöyle ki, biz bir heap ağacını breadth-first bir dolaşım ile bir diziye çevirebiliriz (buna terminoloji de "heapify" denilmektedir. Örneğin:

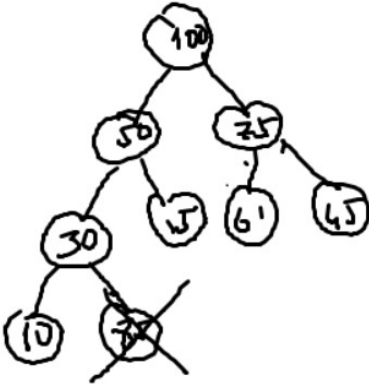
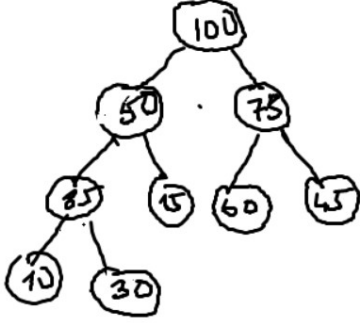


Uygulamada dizinin ilk elemanı hesaplamalar kolay yapılsın diye boş bırakılır (boş bırakılmasa da olur ancak hesaplamaların daha kolay yapılması için bu tercih edilmektedir.) Görüldüğü gibi heap ağacı diziye dönüştürüldüğünde n'inci indeksteki bir düğümün üst düğümü n/2'inci indekste bulunur. Benzer biçimde n'inci indeksteki düğümün alt düğümleri de 2*n ve 2 * n + 1 numaralı indekslerde bulunacaktır.

Peki kuralı bozmadan heap ağacını temsil eden bir diziye nasıl ekleme yapılabilir? Eleman önce her zaman sona yerleştirilir. Sonra bir döngü içerisinde üst düğümlere erişilerek onunla üst düğüm arasındaki ilişkiye bakılır. Kuralın sağlandığı noktada durulur, kural sağlanmıyorsa onunla üst düğümü yer değiştirilir. Örneğin yukarıdaki ağaca biz 85 ekleyelim:



Heap ağacından eleman silmek için bunun tam tersi yapılır. Silinecek eleman alınır. Onun yerine geçecek eleman onun alt düğümlerinden biridir. Bunlardan hangisi büyükse o yukarı alınır. Böyle devam edilir. Ta ki ağacın yapraklarına gelene kadar. Örneğin yukarıdaki ağaçta 85'i silmek isteyelim. İşlemler şu admlardan geçilerek yapılacaktır:



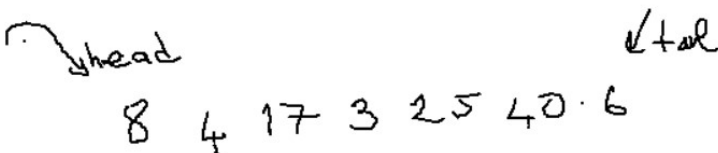
Heap ağacı her zaman tam (complete) bir ağaç görünümündedir. Zaten o nedenle dizi olarak temsil edilmesi uygundur. Tabi buradaki dizinin dinamik bir dizi gibi olması daha uygundur.

Heap Ağacına Neden Gereksinim Duyulmaktadır?

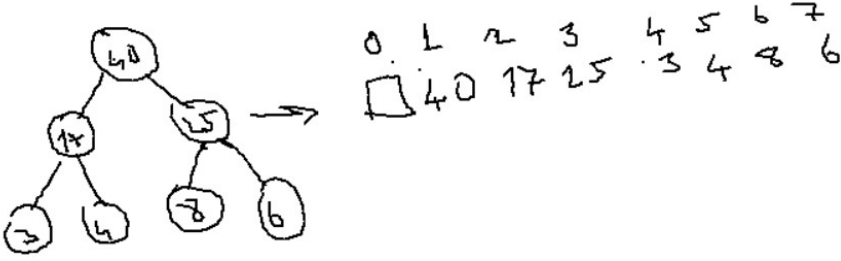
Heap ağacının en önemli uygulama alanı öncelik kuyruklarıdır (priority queues). Öncelik kuyruklarında kuyrukta birtakım elemanlar vardır. Kuyruktan eleman alınacağı zaman en yüksek önceliğe sahip olan eleman alınır. Böyle bir kuyruk sistemi düz bir diziyile gerçekleştirilmeye çalışılırsa her defasında en büyük elemanın bulunması için dizinin dolaşılması gerekir. İşte bunun için en uygun veri yapısı ikili heap ağaçlarıdır. Eleman dizinin sonuna eklenir. Dizi heap kuralını bozmayacak biçimde düzenlenir. Eleman alınacağı zaman baştan alınır yine dizi kuralı bozmayacak biçimde düzenlenir. Heap ağaçları sıraya dizme için de kullanılmaktadır. Buna "heap sort" denilmektedir. Heap sort da $O(N \log N)$ karmaşıklıkta kaliteli bir sort algoritmasıdır.

Öncelik Kuyrukları (Priority Queues)

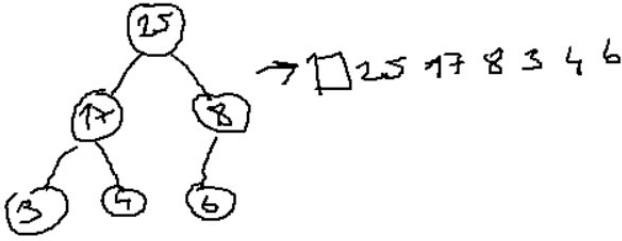
Öncelik kuyrukları kuyruğa yerleştirilen elemanların bir öncelik derecesinin olduğu kuyruk sistemleridir. Dolayısıyla kuyruktan eleman alınırken ilk sıradaki eleman değil öncelik derecesi en yüksek olan eleman alınmaktadır. Örneğin öncelik dereceleri birer sayı ile belirtilmiş olsun. Aşağıdaki gibi bir kuyruk sisteminin bulunduğunu düşünelim:



Burada normal bir kuyruk sistemi söz konusu olduğunda eleman alınacakken kuyruğun önündeki 8 alınır. Ancak öncelik kuyruklarında kuyruğun başındaki eleman değil önceliği en yüksek olan eleman alınmaktadır. Böyle bir kuyruk sistemini daha önce yaptığımız gibi döngüsel biçimde tasarlırsak eleman alma işlemi doğrusal karmaşıklıkta yapılır. Çünkü en büyük elemanın dizi içerisinde aranması gerekecektir. Dizinin sürekli sıralı tutulmaya çalışılması da bu sefer eleman ekleme konusunda zorluk oluşturur. Çünkü sıranın bozulmaması için $O(N)$ karmaşıklıkta bir "insertion" işleminin yapılması gerekir. İşte bu veri yapısını gerçekleştirmek için en etkin yöntem heap ağaçlarıdır. Çünkü heap ağaçlarına eleman eklemek $O(\log N)$ karmaşıklıkta bir işlemdir. Heap ağaçlarının tepesinden eleman alma da yine $O(\log N)$ karmaşıklıkta yapılabilir. Örneğin yukarıdaki değerlerden bir heap ağacı oluşturulmuş olsun:



Burada biz öncelik kuyruğundan eleman almak için tek yapacağımız şey kökteki elemanı almak olacaktır. Anımsanacağı gibi heap ağacından eleman silme işleminde alt düğümlerin en büyüğü ya da en küçüğü ile yer değiştirme yapıyordu.



Heap Veri Yapısının Gerçekleştirilmesi

Heap veri yapısı için bir dizi kullanılır. Tabii dizinin dinamik bir dizi olması tercih edilir. Heap'a eleman insert edilirken insert edilecek yer en aşağıdan hareketle yukarıya doğru çıkılarak yer değiştirmelerle belirlenir. Heap'ın kökündeki elemanın alınması benzer biçimde yapılır. Kök elemanın yerine geçecek eleman aşağıya doğru inilerek belirlenir. Örnek bir heap gerçekleştirimi şöyle olabilir:

```

/* HeapTree.h */

#ifndef HEAP_H_
#define HEAP_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE      0
#define TRUE       1
#define DEF_CAPACITY 8

/* Type declarations */

typedef int DATATYPE;
typedef int BOOL;

typedef struct tagHEAP {
    DATATYPE *pArray;
    size_t capacity;
    size_t lastIndex;
} HEAP, *HHEAP;

```

```

/* Function Prototypes */

HHEAP CreateHeap(void);
BOOL PutHeap(HHEAP hHeap, DATATYPE val);
BOOL GetHeap(HHEAP hHeap, DATATYPE *val);
void ClearHeap(HHEAP hHeap);
void CloseHeap(HHEAP hHeap);

/* Macros */

#define IsEmptyHeap(hHeap) ((hHeap)->lastIndex == 1)
#define GetSizeHeap(hHeap) ((hHeap)->size)
#define GetCountHeap(hHeap) ((hHeap)->lastIndex - 1)
#define GetHeapArray(hHeap) ((hHeap)->pArray + 1)

#endif

/* HeapTree.c */

#include <stdlib.h>
#include "HeapTree.h"

HHEAP CreateHeap(void)
{
    HHEAP hHeap;

    if ((hHeap = (HHEAP)malloc(sizeof(HHEAP))) == NULL)
        return NULL;

    if ((hHeap->pArray = (DATATYPE *)malloc(sizeof(DATATYPE)* DEF_CAPACITY)) == NULL) {
        free(hHeap);
        return NULL;
    }

    hHeap->capacity = DEF_CAPACITY;
    hHeap->lastIndex = 1;

    return hHeap;
}

BOOL PutHeap(HHEAP hHeap, DATATYPE val)
{
    DATATYPE *pTemp;
    size_t i;

    if (hHeap->lastIndex == hHeap->capacity) {
        if ((pTemp = (DATATYPE *)realloc(hHeap->pArray, sizeof(DATATYPE) * hHeap->capacity * 2)) ==
NULL)
            return FALSE;

        hHeap->pArray = pTemp;
        hHeap->capacity *= 2;
    }

    i = hHeap->lastIndex;
    if (i != 1)
        while (val > hHeap->pArray[i / 2] && i > 1) {
            hHeap->pArray[i] = hHeap->pArray[i / 2];
            i /= 2;
        }

    hHeap->pArray[i] = val;
    ++hHeap->lastIndex;
}

```

```

    return TRUE;
}

BOOL GetHeap(HHEAP hHeap, DATATYPE *val)
{
    size_t i, ci;
    DATATYPE lastVal;

    if (hHeap->lastIndex == 1)
        return FALSE;

    *val = hHeap->pArray[1];
    lastVal = hHeap->pArray[hHeap->lastIndex - 1];
    --hHeap->lastIndex;

    i = 1;
    ci = 2;

    while (ci < hHeap->lastIndex) {
        if (ci + 1 < hHeap->lastIndex && hHeap->pArray[ci] < hHeap->pArray[ci + 1])
            ++ci;
        if (lastVal > hHeap->pArray[ci])
            break;
        hHeap->pArray[i] = hHeap->pArray[ci];
        i = ci;
        ci *= 2;
    }
    hHeap->pArray[i] = lastVal;

    return TRUE;
}

void ClearHeap(HHEAP hHeap)
{
    hHeap->lastIndex = 1;
}

void CloseHeap(HHEAP hHeap)
{
    free(hHeap->pArray);
    free(hHeap);
}

/* Test.c */

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

int main(void)
{
    HHEAP hHeap;
    size_t i;
    DATATYPE *pArray;
    DATATYPE val;

    if ((hHeap = CreateHeap()) == NULL) {
        fprintf(stderr, "cannot create heap!..\n");
        exit(EXIT_FAILURE);
    }

    PutHeap(hHeap, 100);
    PutHeap(hHeap, 50);
    PutHeap(hHeap, 75);
    PutHeap(hHeap, 30);
    PutHeap(hHeap, 15);

```

```

PutHeap(hHeap, 60);
PutHeap(hHeap, 45);
PutHeap(hHeap, 10);

pArray = GetHeapArray(hHeap);
for (i = 0; i < GetCountHeap(hHeap); ++i)
    printf("%d ", pArray[i]);
printf("\n-----\n");

while (!IsEmptyHeap(hHeap)) {
    if (!GetHeap(hHeap, &val)) {
        fprintf(stderr, "heap is empty!\n");
        exit(EXIT_FAILURE);
    }
    printf("Value: %d\n", val);

    pArray = GetHeapArray(hHeap);
    for (i = 0; i < GetCountHeap(hHeap); ++i)
        printf("%d ", pArray[i]);
    printf("\n-----\n");
}

CloseHeap(hHeap);

return 0;
}

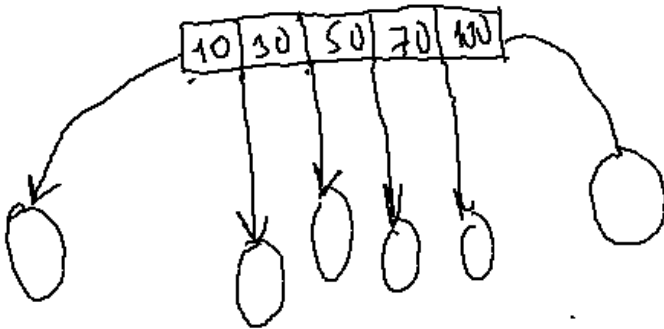
```

Dışsal Aramalar (External Search)

Daha önceden de belirtildiği gibi dışsal aramalar (external search) ikincil belleklerde (yani dosyalarda) yapılan aramaları belirtmektedir. Aslında dışsal aramalar için de benzer algoritmalar söz konusudur. Örneğin ikili arama ağacını dosya üzerindeki kayıtlar için oluşturabilir miyiz? Tabiki evet. Şöyle ki: Dosyanın başına bir başlık kısmı (file header) yerleştiririz. Sonra eklenecek kayıtları hep dosyanın sonuna ekleriz. İkili ağaç düğümlerindeki left ve right göstericiler burada offset'lere karşılık gelir. Böylece kök düğümden gireriz. fseek fonksiyonuyla konumlanarak okumaları yapabiliriz.

Ancak ikili ağaçlar dengelenmiş olsa bile disk aramaları için çok uygun değildir. Çünkü ağacın yüksekliği disk okumalarının sayısını doğrudan artırmaktadır. Çünkü her düğüm geçişinde bir disk okumasının yapılması gerekir. Bilindiği gibi disk erişimleri ana bellek erişimine göre oldukça yavaştır. Bu durumda biz ağacın yüksekliğini düşürmek isteriz. İşte "B-Tree" denilen ağaç yapısı bunu hedeflemektedir.

B-Tree Rudolf Bayer tarafından 1971 yılında tasarlanmıştır. İsminin neden "B-Tree" olduğuna yönelik kesin bir bilgi yoktur. B Tree aslında n'li bir arama ağacıdır Şöyle ki her düğümün iki alt düğümü değil daha fazla alt düğümü vardır. Örneğin B Tree'nin 6'lı arama ağacı olarak oluşturulduğunu varsayalım. Burada her düğümde tek bir anahtar değil 5 anahtar bulunacaktır. Tabi bu 5 anahtar bir dizi içerisinde tutulur:



10'dan küçük olanlar, 10 ile 30 arasında olanlar, 30 ile 50 arasında olanlar, 50 ile 70 arasında olanlar ve 100'den büyük olanlar ayrı düğümlere yönlendirilmiştir. Böylece arama yapılırken daha az düzey geçişi oluşur. Bu da disk okumalarının daha az olması anlamına gelir. Çünkü her düğümden düğüme geçiş bir disk okumasıyla yapılmaktadır. B-Tree dengeli bir ağaç yapısıdır. B-Tree'nin oluşturulması ve dengelenmesi biraz karmaşıktır. Bu konu Sistem Programlama ve İleri C Uygulamaları II Kursunda ele alınmaktadır.

Peki birincil belleklerde B-Tree mi yoksa Binary Tree mi daha etkindir? Aslında birincil bellekler için Binary Tree daha etkindir. Çünkü B-Tree'de düğümler arasındaki geçiş sırasında anahtar karşılaştırmaları daha fazladır. Ve genel olarak yapılan işlemler daha yoğundur. Oysa Binary Tree'de hızlı biçimde düğümden düğüme geçiş sağlanır. Disk söz konusu olduğunda asıl yavaşlık düğümler arası geçişte olduğu için B-Tree çok daha uygun bir yöntem olmaktadır. B-Tree'nin biraz revize edilmiş biçimine B+Tree denilmektedir.

Bugün kullandığımız Sql Server gibi, MySql gibi, Oracle gibi Veritabanı Yönetim Sistemlerinin çekirdek arama algoritmaları hep B-Tree ya da B+ Tree'ye dayanmaktadır. Şimdilik bunun ötesinde disk için daha verimli algoritmalar bulunmamıştır.

Veri Yapılarının Genelleştirilmesi

Yukarıdaki örneklerde biz veri yapılarının içerisinde tutulan türü DATATYPE ile temsil ettik. Bunun birkaç sakıncası vardır. Birincisi bu yöntemde aynı programda programın farklı yerlerinde DATATYPE türü değiştirilemez. Yani örneğin DATATYPE int ise biz yalnızca int türden bağlı listeler oluşturabiliriz. C++ gibi, Java ve C# gibi dillerdeki template (ya da generic) özelliği bu sorunu bir ölçüde çözmektedir. İkinci sorun fonksiyonlara parametre aktarım işleminin değer yoluyla mı adres yoluyla mı yapılacağı ile ilgilidir. Örneğin DATATYPE int ise biz onu fonksiyonlara doğrudan geçirmeyi tercih ederiz. Ancak DATATYPE bir yapı ise biz onu adres yoluyla geçirmeyi tercih ederiz. Bu durumda da yazmış kodların DATATYPE türüne göre değiştirilmesi gerekir.

Veri yapılarını genelleştirerek biz onların kodlarına dokunmadan istediğimiz her türü tutmasını sağlayabiliriz. Veri yapılarının genelleştirilmesi için üç yol izlenebilir:

1) Her tür için o veri yapısına ilişkin fonksiyonları o tür için yeniden yazmak. Şüphesiz bu yöntem kodu büyütme eğilimindedir. Ancak böyle oluşturulmuş veri yapılarının kullanımı kolaydır ve böyle oluşturulmuş veri yapıları görece olarak diğer alternatif yöntemlere göre daha hızlıdır. Zaten template ya da generic mekanizmaları aslında arka planda böyle bir işleme yol açar. Örneğin:

```
list<int> a;  
list<Person> b;
```

Burada list template bir sınıftır. Kendisi şablon bir bildirimdir. Derleyici her tür için onu yeniden yazar.

2) Tek bir fonksiyon grubunun tüm türler için çalışabilmesini sağlamak. Bu yöntemde kod tekrarı olmadığı için toplamda kodlar daha az yer kaplar. Ancak bu yöntemde fonksiyonların parametrik yapıları daha karmaşıktır ve genel kullanım daha zordur.

3) Veri yapısının türden bağımsız olarak yalnızca metadata kısımlarla ilişkili biçimde oluşturmak. Örneğin Linux, BSD ve CSD işletim sistemlerindeki genelleştirme için bu yöntem kullanılmıştır. Bu yöntemin de kullanımı zordur. Ancak kod etkinliği yüksektir.

Her Tür İçin Fonksiyonların O Türe Özgü Olarak Yeniden Yazılması (1. Yöntem)

Her tür için veri yapısına ilişkin fonksiyonların o türe özgü olarak yeniden yazılması C için uygun bir yöntem değildir. Çünkü C'de pek çok nesne yönelimli programlama dilinde var olan "fonksiyonların aşırı yüklenmesi (function overloading)" ya da template gibi özellikler bulunmamaktadır. Her tür için veri yapısındaki fonksiyonları yeniden yazmak için C'de fonksiyonlara değişik isimlerin verilmesi gerekir. Bu ise kodun karmaşık gözükmesine yol açmaktadır. Ayrıca yukarıda da belirttiğimiz gibi her tür için o türe özgü olarak fonksiyonların yeniden yazılması durumunda bir kod büyümesi de kaçınılmazdır.

Veri Yapılarının Türden Bağımsız Olarak Genel Fonksiyonlarla Genelleştirilmesi (2. Yöntem)

Teorik olarak yukarıda sözünü ettiğimiz veri yapılarını oluşturmak için aslında biz DATATYPE türünün ne olduğunu derleme zamanında bilmek zorunda değiliz. O türün byte uzunluğunu bilsek bu bize yeter. Şöyle ki: Örneğin bağlı liste düğümlerinde saklanan elemanların türlerini bilmiyor olalım. Bu durumda bir düğümün biçimi biz göre şöyle olacaktır:



Örneğin ikili ağacı bu biçimde oluşturmak istediğimizde eleman eklerken neye göre karşılaştırma yapacağız? Bu tür durumlarda fonksiyon göstericilerini kullanarak karşılaştırmayı aslında veri yapısını kullanan kişiye bırakabiliriz. Şüphesiz böyle bir sistemde artık fonksiyonların elemana ilişkin parametreleri void * türünden olacaktır. Eleman atamaları da atama operatörüyle değil memcopy gibi bir fonksiyonla yapılacaktır.

Türden bağımsız örnek bir ikili arama ağacı şöyle oluşturulabilir.

```
/* BinaryTree.h */

/* BinaryTree.h */

#ifndef BINARYTREE_H_
#define BINARYTREE_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE 0
#define TRUE 1

/* Type declarations */

typedef int BOOL;
typedef int(*Compare)(const void *, const void *);

typedef struct tagNODE {
    struct tagNODE *pLeft;
    struct tagNODE *pRight;
    unsigned char data[1];
} NODE;

typedef struct tagBINARYTREE {
    NODE *pRoot;
    size_t count;
    size_t dataSize;
    Compare compare;
} BINARYTREE, *HBINARYTREE;

/* Function Prototypes */

HBINARYTREE CreateBinaryTree(size_t dataSize, Compare compare);
BOOL InsertItem(HBINARYTREE hBinaryTree, const void *pVal);
BOOL WalkInOrder(HBINARYTREE hBinaryTree, BOOL(*Proc)(void *));
void *FindItem(HBINARYTREE hBinaryTree, const void *key, int(*Compare)(const void *key, const void *data));
void Clear(HBINARYTREE hBinaryTree);
void CloseBinaryTree(HBINARYTREE hBinaryTree);
```

```

#endif

/* BinaryTree.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "BinaryTree.h"

/* static Function Prototypes */

static BOOL walkInOrder(NODE *pNode, BOOL(*Proc)(void *));
static NODE *createNode(HBINARYTREE hBinaryTree, const void *pVal);
static void clear(NODE *pNode);

/* Function Definitions */

HBINARYTREE CreateBinaryTree(size_t dataSize, Compare compare)
{
    HBINARYTREE hBinaryTree;

    if ((hBinaryTree = (HBINARYTREE)malloc(sizeof(BINARYTREE))) == NULL)
        return NULL;

    hBinaryTree->pRoot = NULL;
    hBinaryTree->count = 0;
    hBinaryTree->dataSize = dataSize;
    hBinaryTree->compare = compare;

    return hBinaryTree;
}

static NODE *createNode(HBINARYTREE hBinaryTree, const void *pVal)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE *) * 2 + hBinaryTree->dataSize)) == NULL)
        return NULL;
    memcpy(pNewNode->data, pVal, hBinaryTree->dataSize);
    pNewNode->pLeft = pNewNode->pRight = NULL;

    return pNewNode;
}

BOOL InsertItem(HBINARYTREE hBinaryTree, const void *pVal)
{
    NODE *pNewNode, *pNode, *pParentNode;
    int result;

    if ((pNewNode = createNode(hBinaryTree, pVal)) == NULL)
        return FALSE;

    if (hBinaryTree->pRoot == NULL) {
        hBinaryTree->pRoot = pNewNode;
        ++hBinaryTree->count;

        return TRUE;
    }

    pParentNode = pNode = hBinaryTree->pRoot;

    while (pNode != NULL) {
        pParentNode = pNode;
        result = hBinaryTree->compare(pVal, pNode->data);
        if (result > 0)

```

```

        pNode = pNode->pRight;
    else if (result < 0)
        pNode = pNode->pLeft;
    else
        return FALSE;
}

if (result > 0)
    pParentNode->pRight = pNewNode;
else
    pParentNode->pLeft = pNewNode;

return TRUE;
}

BOOL WalkInOrder(HBINARYTREE hBinaryTree, BOOL(*Proc)(void *))
{
    return walkInOrder(hBinaryTree->pRoot, Proc);
}

static BOOL walkInOrder(NODE *pNode, BOOL(*Proc)(void *))
{
    if (pNode->pLeft != NULL && !walkInOrder(pNode->pLeft, Proc))
        return FALSE;

    if (!Proc(pNode->data))
        return FALSE;

    if (pNode->pRight != NULL && !walkInOrder(pNode->pRight, Proc))
        return FALSE;

    return TRUE;
}

void *FindItem(HBINARYTREE hBinaryTree, const void *key, int(*Compare)(const void *key, const void *data))
{
    NODE *pNode;
    int result;

    pNode = hBinaryTree->pRoot;
    while (pNode != NULL) {
        result = Compare(key, pNode->data);
        if (result == 0)
            return pNode->data;

        pNode = result < 0 ? pNode->pLeft : pNode->pRight;
    }

    return NULL;
}

void Clear(HBINARYTREE hBinaryTree)
{
    clear(hBinaryTree->pRoot);

    hBinaryTree->pRoot = NULL;
    hBinaryTree->count = 0;
}

static void clear(NODE *pNode)
{
    if (pNode->pLeft != NULL)
        clear(pNode->pLeft);

    if (pNode->pRight != NULL)

```

```

        clear(pNode->pRight);

    free(pNode);
}

void CloseBinaryTree(HBINARYTREE hBinaryTree)
{
    Clear(hBinaryTree);
    free(hBinaryTree);
}

/* Test.c */

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

typedef int KEY;

typedef struct tagPERSON {
    char name[32];
    KEY no;
} PERSON;

int CompPerson(const void *pPerson1, const void *pPerson);
int CompKey(const void *key, const void *pPerson2);
BOOL DispPerson(const void *pPerson);

int main(void)
{
    HBINARYTREE hBinaryTree;
    PERSON persons[] = {
        {"Ali Serce", 123}, {"Ahmet Can", 97}, {"Sibel Aydin", 150}, {"Necati Ergin", 68}, {"Ayse Er",
160}, {"Sami Erdem", 27}, {"Hasan Keskin", 145}, {"Salih Bulut", 72}, {"", 0} };
    int i;
    int key;
    PERSON *pPer;

    if ((hBinaryTree = CreateBinaryTree(sizeof(PERSON), CompPerson)) == NULL) {
        fprintf(stderr, "cannot create binary tree!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; persons[i].no != 0; ++i)
        if (!InsertItem(hBinaryTree, &persons[i])) {
            fprintf(stderr, "cannot insert item!..\n");
            exit(EXIT_FAILURE);
        }

    WalkInOrder(hBinaryTree, DispPerson);

    key = 123;
    if ((pPer = (PERSON *)FindItem(hBinaryTree, &key, CompKey)) != NULL)
        printf("Found: %s, %d\n", pPer->name, pPer->no);
    else
        printf("cannot found person!..\n");

    CloseBinaryTree(hBinaryTree);

    return 0;
}

int CompKey(const void *key, const void *pPerson)
{
    const PERSON *pPer = (const PERSON *)pPerson;

```

```

    return *(const int *)key - pPer->no;
}

int CompPerson(const void *pPerson1, const void *pPerson2)
{
    const PERSON *pPer1 = (const PERSON *)pPerson1;
    const PERSON *pPer2 = (const PERSON *)pPerson2;

    return pPer1->no - pPer2->no;
}

BOOL DispPerson(void *pPerson)
{
    const PERSON *pPer = (PERSON *)pPerson;

    printf("%s, %d\n", pPer->name, pPer->no);

    return TRUE;
}

```

Veri Yapısının Türden Bağımsız Olarak Yalnızca Metadata Kısımlarla İlişkili Biçimde Oluşturulması (3. Yöntem)

Türden bağımsız veri yapısı oluşturmanın diğer bir yolu da veri yapısının metadata kısımlarını asıl veri yapısının içerisine gömmektir. Örneğin Linux, BSD ve CSD işletim sistemlerinin kernel kodlarındaki veri yapıları genel olarak bu biçimde oluşturulmuştur.

Bu yöntemle örneğin PERSON yapı nesnelerini bir çift bağlı liste içerisinde birbirine bağlayacak olalım. Bu çift bağlı listenin pNext ve pPrev göstericileri NODE isimli bir yapı ile temsil ediliyor olsun. Örneğin:

```

struct NODE {
    struct NODE *pNext;
    struct NODE *pPrev;
};

struct PERSON {
    /* ... */
    struct NODE node;
};

```

Bu çift bağlı listenin node içerisindeki pNext ve pPrev göstericileri sonraki PERSON nesnesinin başlangıç adresini göstermezler. Sonraki PERSON nesnesinin içerisindeki node nesnesinin başlangıç adresini gösterirler. Böylece bağlı liste PERSON türüne bağımlı olmaz, NODE türüne bağımlı olur. Örneğin:

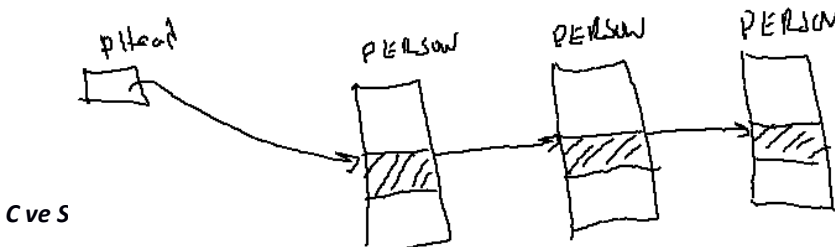
```

struct NODE {
    struct NODE *pNext;
    struct NODE *pPrev;
};

typedef struct tagPERSON {
    char name[64];
    struct NODE node; /* yapının herhangi bir yerinde olabilir */
    int no;
} PERSON;

```

Bu durum şekilsel olarak şöyle gösterilebilir:



Biz bir yapı nesnesinin içerisindeki bir elemanın ismini ve adresini biliyorsak o yapı nesnesinin başlangıç adresini elde edebiliriz. Zaten C'de bu işi yapan standart `offsetof` makrosu vardır. Bu makro aşağıdaki gibi yazılabilir:

```
#define offsetof(st, m) ((size_t)&((st *)0)->m)
```

Bu makronun birinci parametresi yapının ismini (yani tür ifadesini), ikinci parametresi elemanın ismini alır. Örneğin:

```
size_t offset = offsetof(PERSON, node)
```

Yukarıdaki makro C standartlarına göre tamamen geçerlidir. Standartlara göre `p` göstericisi tahsis edilmemiş bir alanı gösteriyor olsa bile (0 adresi de dahil olmak üzere) onun bir elemanın adresini `&p->a` biçiminde elde etmek oraya gerçek bir erişimi gerektirmediği için geçerlidir.

Tabii yukarıdaki bağlı liste örneğinde bizim `node` adresinden `offsetof` kadar eksiltme yapıp sonucu ilgili türe dönüştürmemiz gerekir. Bu da bir makro ile yapılabilir. Örneğin:

```
#define container_of(ptr, type, member) ((type *)((char *)(ptr) - offsetof(type, member))
```

Örneğin `PERSON` yapılarından bir bağlı liste oluşturmuş olalım. Bu bağlı listenin dolaşılması şöyle yapılabilir:

```
NODE pPersonHead;
PERSON *per;

NODE *pNode = pPersonHead;
while (pNode != NULL) {
    per = container_of(pNode, PERSON, node)
    /* ... */
    pNode = pNode->pNext;
}
```

Bu biçimde örnek bir bağlı liste gerçekleştirimi şöyle yapılabilir:

```
#ifndef GENERICLINKEDLIST_H_
#define GENERICLINKEDLIST_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE 0
#define TRUE 1

/* Type declarations */

typedef int BOOL;

typedef struct tagNODE {
    struct tagNODE *pNext;
    struct tagNODE *pPrev;
} NODE;

/* Function Prototypes */

NODE *InsertItemPrev(NODE *pNode, NODE *pNewNode);
NODE *InsertItemNext(NODE *pNode, NODE *pNewNode);
NODE *AddItemHead(NODE *pHead, NODE *pNewNode);
NODE *AddItemTail(NODE *pHead, NODE *pNewNode);
void DeleteItem(NODE *pNode);
```

```

BOOL WalkList(NODE *pHead, BOOL(*Proc)(NODE *));
BOOL WalkListRev(NODE *pHead, BOOL(*Proc)(NODE *));

/* Macros */

#define container_of(ptr, type, member) ( (type *) ( (char *) (ptr) - offsetof(type, member)))

#endif

#include <stdio.h>
#include "GenericLinkedList.h"

NODE *InsertItemPrev(NODE *pNode, NODE *pNewNode)
{
    pNewNode->pNext = pNode;
    pNewNode->pPrev = pNode->pPrev;
    pNode->pPrev->pNext = pNewNode;
    pNode->pPrev = pNewNode;

    return pNewNode;
}

NODE *InsertItemNext(NODE *pNode, NODE *pNewNode)
{
    pNewNode->pPrev = pNode;
    pNewNode->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNewNode;
    pNode->pNext = pNewNode;

    return pNewNode;
}

NODE *AddItemHead(NODE *pHead, NODE *pNewNode)
{
    return InsertItemNext(pHead, pNewNode);
}

NODE *AddItemTail(NODE *pHead, NODE *pNewNode)
{
    return InsertItemPrev(pHead, pNewNode);
}

void DeleteItem(NODE *pNode)
{
    pNode->pPrev->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNode->pPrev;
}

BOOL WalkList(NODE *pHead, BOOL(*Proc)(NODE *))
{
    NODE *pNode = pHead->pNext;

    while (pNode != pHead) {
        if (!Proc(pNode))
            return FALSE;
        pNode = pNode->pNext;
    }
    return TRUE;
}

BOOL WalkListRev(NODE *pHead, BOOL(*Proc)(NODE *))
{
    NODE *pNode = pHead->pPrev;

    while (pNode != pHead) {
        if (!Proc(pNode))

```



```

        return FALSE;
    pNode = pNode->pPrev;
}
return TRUE;
}

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

typedef struct tagPERSON {
    int no;
    char name[32];
    NODE node;
} PERSON;

BOOL WalkProc(NODE *node);

NODE g_head = { &g_head, &g_head };

int main(void)
{
    NODE *pNode;
    int i;

    PERSON persons[] = {
        { 15, "Kaan Aslan" }, { 10, "Ali Serce" }, { 30, "Sacit Hicyilmaz" }, { 8, "Ali Sen" },
        { 12, "Fatih Terim" }, { 20, "Necati Ergin" }, { 52, "Guray Sonmez" }, { 13, "Sami Ercan" }
    };

    for (i = 0; i < 6; ++i) {
        if (i == 3)
            pNode = AddItemTail(&g_head, &persons[i].node);
        else
            AddItemTail(&g_head, &persons[i].node);
    }

    WalkList(&g_head, WalkProc);
    DeleteItem(pNode);
    printf("-----\n");
    WalkList(&g_head, WalkProc);
    printf("-----\n");
    WalkListRev(&g_head, WalkProc);

    return 0;
}

BOOL WalkProc(NODE *pNode)
{
    PERSON *pPerson;

    pPerson = (PERSON *)((char *)pNode - offsetof(PERSON, node));

    printf("%d, %s\n", pPerson->no, pPerson->name);

    return TRUE;
}

```

Ya da handle sistemi kullanılarak aynı gerçekleştirim benzer biçimde şöyle de yapılabilir:

```

#ifndef GENERICLINKEDLIST_H_
#define GENERICLINKEDLIST_H_

#include <stddef.h>

/* Symbolic Constants */

```

```

#define FALSE      0
#define TRUE       1

/* Type declarations */

typedef int BOOL;

typedef struct tagNODE {
    struct tagNODE *pNext;
    struct tagNODE *pPrev;
} NODE;

typedef struct tagLLIST {
    NODE head;
    size_t count;
} LLIST, *HLLIST;

/* Function Prototypes */

HLLIST CreateLList(void);
NODE *InsertItemPrev(HLLIST hLList, NODE *pNode, NODE *pNewNode);
NODE *InsertItemNext(HLLIST hLList, NODE *pNode, NODE *pNewNode);
NODE *AddItemHead(HLLIST hLList, NODE *pNewNode);
NODE *AddItemTail(HLLIST hLList, NODE *pNewNode);
void DeleteItem(HLLIST hLList, NODE *pNode);
BOOL WalkList(HLLIST hLList, BOOL(*Proc)(NODE *));
BOOL WalkListRev(HLLIST hLList, BOOL(*Proc)(NODE *));
void Clear(HLLIST hLList);
void CloseList(HLLIST hLList);

/* Macros */

#define GetCount(hLList)      ((hLList)->count)
#define container_of(ptr, type, member) ( (type *) ( (char *) (ptr) - offsetof(type, member)) )

#endif

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

static NODE *insertItemPrev(NODE *pNode, NODE *pNewNode);
static NODE *insertItemNext(NODE *pNode, NODE *pNewNode);
static void deleteItem(NODE *pNode);

HLLIST CreateLList(void)
{
    HLLIST hLList;

    if ((hLList = (HLLIST)malloc(sizeof(LLIST))) == NULL)
        return NULL;

    hLList->head.pNext = &hLList->head;
    hLList->head.pPrev = &hLList->head;
    hLList->count = 0;

    return hLList;
}

NODE *InsertItemPrev(HLLIST hLList, NODE *pNode, NODE *pNewNode)
{
    ++hLList->count;

    return insertItemPrev(pNode, pNewNode);
}

```

```

NODE *InsertItemNext(HLLIST hLList, NODE *pNode, NODE *pNewNode)
{
    ++hLList->count;

    return insertItemNext(pNode, pNewNode);
}

NODE *insertItemPrev(NODE *pNode, NODE *pNewNode)
{
    pNewNode->pNext = pNode;
    pNewNode->pPrev = pNode->pPrev;
    pNode->pPrev->pNext = pNewNode;
    pNode->pPrev = pNewNode;

    return pNewNode;
}

NODE *insertItemNext(NODE *pNode, NODE *pNewNode)
{
    pNewNode->pPrev = pNode;
    pNewNode->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNewNode;
    pNode->pNext = pNewNode;

    return pNewNode;
}

NODE *AddItemHead(HLLIST hLList, NODE *pNewNode)
{
    return InsertItemNext(hLList, &hLList->head, pNewNode);
}

NODE *AddItemTail(HLLIST hLList, NODE *pNewNode)
{
    return InsertItemPrev(hLList, &hLList->head, pNewNode);
}

void DeleteItem(HLLIST hLList, NODE *pNode)
{
    --hLList->count;

    deleteItem(pNode);
}

void deleteItem(NODE *pNode)
{
    pNode->pPrev->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNode->pPrev;
}

BOOL WalkList(HLLIST hLList, BOOL(*Proc)(NODE *))
{
    NODE *pNode = hLList->head.pNext;

    while (pNode != &hLList->head) {
        if (!Proc(pNode))
            return FALSE;
        pNode = pNode->pNext;
    }
    return TRUE;
}

BOOL WalkListRev(HLLIST hLList, BOOL(*Proc)(NODE *))
{
    NODE *pNode = hLList->head.pPrev;
}

```

```

while (pNode != &hLList->head) {
    if (!Proc(pNode))
        return FALSE;
    pNode = pNode->pPrev;
}
return TRUE;
}

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

typedef struct tagPERSON {
    int no;
    char name[32];
    NODE node;
} PERSON;

BOOL WalkProc(NODE *node);

NODE g_head = { &g_head, &g_head };

int main(void)
{
    int i;

    PERSON persons[] = {
        { 15, "Kaan Aslan" }, { 10, "Ali Serce" }, { 30, "Sacit Hicyilmaz" }, { 8, "Ali Sen" },
        { 12, "Fatih Terim" }, { 20, "Necati Ergin" }, { 52, "Guray Sonmez" }, { 13, "Sami Ercan" }
    };
    HLLIST hLList;
    NODE *pNode;

    if ((hLList = CreateLList()) == NULL) {
        fprintf(stderr, "cannot create linked list!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 6; ++i) {
        if (i == 3)
            pNode = AddItemTail(hLList, &persons[i].node);
        else
            AddItemTail(hLList, &persons[i].node);
    }

    WalkList(hLList, WalkProc);
    DeleteItem(hLList, pNode);
    printf("-----\n");
    WalkList(hLList, WalkProc);
    printf("-----\n");
    WalkListRev(hLList, WalkProc);

    CloseList(hLList);

    return 0;
}

BOOL WalkProc(NODE *pNode)
{
    PERSON *pPerson;

    pPerson = (PERSON *)((char *)pNode - offsetof(PERSON, node));

    printf("%d, %s\n", pPerson->no, pPerson->name);
}

```

```

    return TRUE;
}

void Clear(HLLIST hLList)
{
    hLList->count = 0;

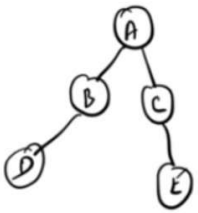
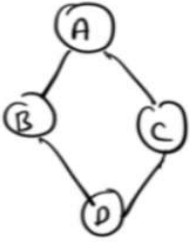
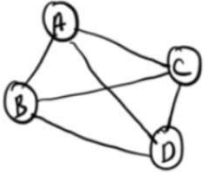
    hLList->head.pNext = &hLList->head;
    hLList->head.pPrev = &hLList->head;
}

void CloseList(HLLIST hLList)
{
    free(hLList);
}

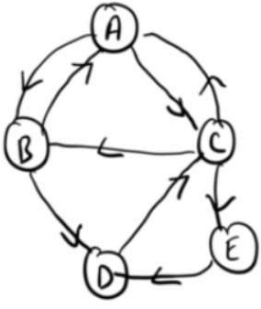
```

Graf Veri Yapısı

Graflar bir düğüme birden fazla düğümden gelebileen düğümlerden (nodes) ve kenarlardan (edges) oluşan bir veri yapısıdır. Aslında ağaçlar grafların özel bir durumudur. Yani bir düğüme tek bir yoldan ulaşılabilen graflara ağaç (tree) denilmektedir. Aşağıda örnek graflar görüyorsunuz:



Buradaki son graf aynı zamanda bir ağaç belirtmektedir. Bir graf yönlü (directed) ya da yönsüz (undirected) olabilir. Yönlü graflarda iki düğüm arasında hangi düğümden hangi düğüme yol olacağı belirtilmektedir. Yönsüz graflarda iki düğüm arasında bir kenar varsa bu kenar gidiş-geliş biçimindedir. Yukarıdaki graflar kenarlarda ok olmadığı yönsüz (undirected) graflardır. Halbuki aşağıdaki graflar yönlüdür:

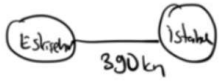


Burada örneğin B'den A'ya, B'den D'ye bir yol olduğu halde B'den C'ye bir yol yoktur. Aslında yönsüz grafları çift yönlü olarak da düşünebiliriz. Gerçek hayatta her iki tür grafla da sıklıkla karşılaşmaktadır. Örneğin kara yollarının belli bir bölümü yönlü grafla modellenir. Ancak örneğin sosyal ağlardaki arkadaşlık bağlantıları yönsüz bir graf biçiminde modellenir. (Yani örneğin Facebook'ta X Y'nin arkadaşıysa Y de X'in arkadaşıdır. Ancak takip sistemi tek yönlü grafla modellenir.)

Bir grafi düğümler ve kenarlar (yollar) oluşturmaktadır. Düğüm terimi grafta hedefteki nesneyi temsil eder. Bu nesne duruma göre herhangi bir kavram olabilir. Örneğin düğümler insanları temsil edebilir, şehirleri temsil edebilir, eşyaları temsil edebilir, ilişkileri temsil edebilir vs. Düğüm terimi İngilizce "node" ya da "vertex" sözcüğüyle ifade edilmektedir. Düğümler birbirlerine yollarla bağlanmıştır. Bu yollara graf terminolojisinde "kenar (edge)" denilmektedir. Bazen kenarlar yalnızca bağlantının olup olmadığını belirtirler. Bazen de kenarlar da başka bilgiler de bulunur. Örneğin bir sosyal ağdaki aşağıdaki bağlantı söz konusu olsun:



Bu bağlantıda biz Ali ile Ayşe arasında yönsüz (ya da çift yönlü) bir bağlantı olduğunu anlıyoruz. Fakat örneğin:



Burada biz Eskişehir ile İstanbul arasında bir bağlantı olduğunu (muhtemelen bir katar yolu bağlantısı) ve aralarındaki bu bağlantının 390 km biçiminde ayrıca bir bilgiye sahip olduğunu anlıyoruz. O halde graf veri yapısı modellenirken düğümler de kenarlar da genel olarak birer yapı ile temsil edilebilirler. Yapının elemanları da bu düğümlerin ya da kenarların özellikleri olacaktır.

```
struct VERTEX {
    ...
};

struct EDGE {
    ...
};
```

Graf Veri Yapısına Neden Gereksinim Duyulmaktadır?

Graflar gerçek hayatta en fazla karşılaşılan veri yapılarından biridir. Dolayısıyla gerçek hayat sorunlarını bilgisayar ortamında çözmek için grafların bir yapısı biçiminde modellenmesi gerekir. Örneğin gerçek hayatta karşılaşılan bazı graf veri yapıları şunlardır:

- Bir navigasyon programında herhangi iki yol arasındaki en kısa mesafenin bulunması istensiz. Aslında o grafin kenarlarını gidilecek merkezler de düğümlerini oluşturmaktadır. Dolayısıyla bu en kısa yol problemi için önce bir graf veri yapısının oluşturulması gerekir.

- Bir dağıtıcı dağıtım yerinden çıkararak belli yerlere dağıtımını yapıp yine dağıtım yerine dönmek ister. Amacı en kısa yol kat edecek biçimde bir rota oluşturmaktır. Bu probleme "gezgin satıcı problemi (traveling salesman problem)" denilmektedir ve ancak bir graf veri yapısı oluşturulduktan sonra çözülebilir.

- Bir delgi makinesinde delinecek yerler bilgisayar ekranında işaretlenir daha sonra otomatik delgi makinesi bunları deler. İşte burada da yine delki makinesinin en az hareket yaparak bu delme eylemlerini gerçekleştirmesi gerekir. Bu problem de ancak bir graf veri yapısı kurularak çözümlür.

- Bir sosyal ağda belli bir kişiye kaç kişiden geçilerek ulaşılabilceği hesaplanmak istenmektedir. Sosyal ağlardaki ilişkiler de graf veri yapısıyla modellenilebilir.

- Bir elektirik devresinde devre elemanları tellerle birbirlerine bağlanmıştır. Bu devre şeması da yine bir graf veri yapısı ile modellenilebilir.

- Akrablık ilişkileri bir graf veri yapısı ile modellenilebilir.

- Kablolarla oluşturulmuş bilgisayar ağları da graflarla modellenilebilir.

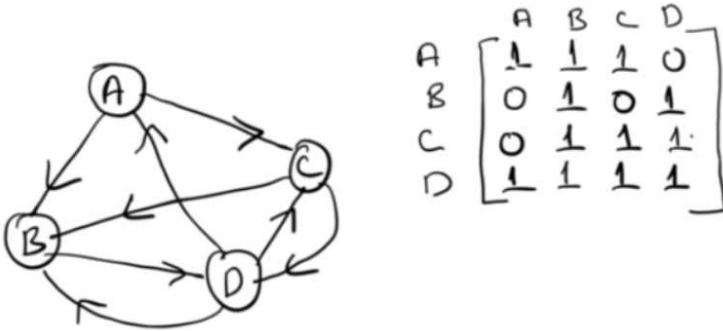
- Ansiklopedik pek çok kavram graflarla modellenmektedir. Çünkü bir kavram pek çok konu ile ilgilidir. Örneğin Wikipedia'da yazılar kategorilerin içerisinde bulunur. Kategoriler de başka kategorilerin içerisinde bulunabilmektedir. Örneğin "dil" bir kavram pek çok kategorinin konusudur. Yani "dil" kavramına ilişkin Wikipedia sayfasına değişik yerlerde gelinebilmektedir. İşte Wikipedia bir "kategorifi grafi" oluşturmaktadır.

Graf Veri Yapısının Oluşturulması

Graf veri yapısı tipik olarak iki biçimde oluşturulmaktadır:

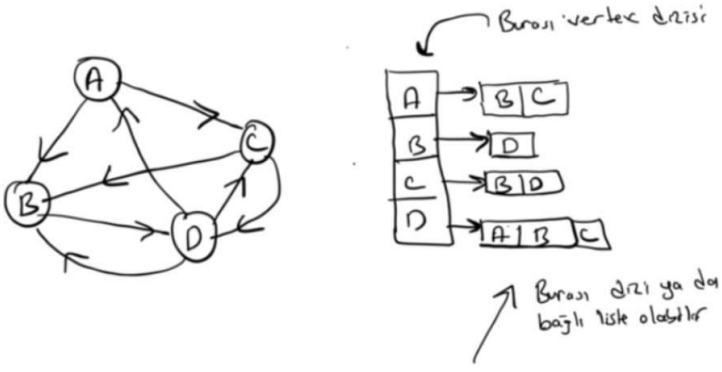
- 1) Komşuluk matrisi (adjacency matrix) yoluyla
- 2) Komşuluk listesi (adjacency list) yoluyla

Komşuluk matrisi graftaki hangi düğümlerin hangi düğümlerle bağlantılı olduğunu temsil eden bir matristir. Eğer graf yönsüzse bu matris simetrik olur. Yönlü ise simetrik olmayabilir. Örneğin:

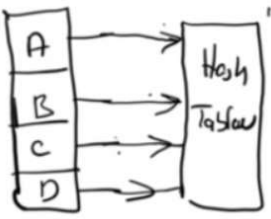


Burada hangi düğümden hangi düğüme bağlantı varsa matriste ilgili eleman 1 yapılmıştır. Böylece matris bize hangi düğümlerden hangi düğümlere bağlantı olduğunu vermektedir. Bu yöntemin en önemli dezavantajı büyük fakat az sayıda kenara sahip olan graflarda matrisin gereksiz biçimde çok yer kaplıyor olmasıdır. Yani bu tür durumlarda matris "seyrek matris (sparse matrix)" biçiminde olur.

Komşuluk listesi yönteminde her düğümün hangi düğümlerle bağlantılı olduğu bir dizide ya da bağlı listede tutulur. Örneğin:



Komşuluk listesi yöntemi komşuluk matrisine göre daha sık tercih edilmektedir. Komşuluk listesi yönteminde hangi düğümün hangi düğümle bağlantılı olduğu bir dizi ya da bağlı liste ile değil bir hash tablosuyla da tutulabilir. Bu durumda belli bir düğümün belli bir düğümle bağlantılı olup olmadığı hash tablosu sayesinde çok daha hızlı tespit edilebilmektedir. Örneğin:



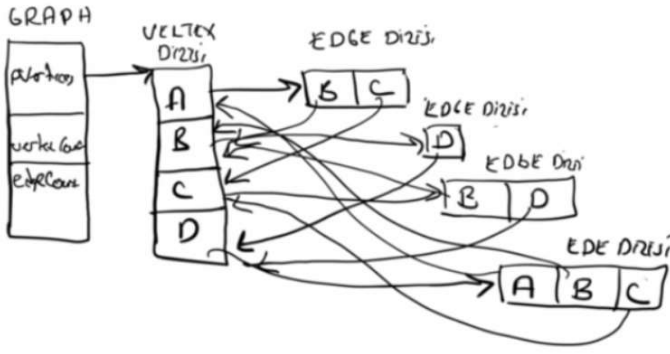
Burada biz komşuluk listesi yönteminin gerçekleştirimini yapacağız. Komşuluk listesi yönteminin gerçekleştirimi çeşitli biçimlerde yapılabilir. Genel bir gerçekleştirim için düğümlerin ve kenarların ayrı yapılar biçiminde oluşturulması uygun olur. Çünkü genel olarak düğümler ve kenarlar başka birtakım bilgiler içerebilmektedir. Graf yaratılırken grafta kaç düğüm olacağı ve hangi düğümlerle hangi düğümlere bağlantı olacağı biliniyorsa gerçekleştirim daha kolay yapılabilir. Ancak bu işin başında bilinmiyorsa graf veri yapısına düğüm ve kenar ekleme fonksiyonlarının yazılması gerekir. Buda veri yapısının biraz değiştirilmesine yol açar. İşin başında düğümler ve kenarlar biliniyorsa ve bunlar bir daha değişmeyecekse aşağıdaki gibi bir veri yapısı uygun olur:

```
typedef struct tagVERTEX {
    char name[32];
    struct tagEDGE *pEdges;
    /* other info */
} VERTEX;

typedef struct tagEDGE {
    VERTEX *pDest;
    /* other info */
} EDGE;

typedef struct tagGRAPH {
    VERTEX *pVertices;
    size_t vertexCount;
    size_t edgeCount;
} GRAPH, *HGRAPH;
```

Burada GRAPH ana veri yapısını temsil etmektedir. GRAPH yapısı içerisinde hem grafın düğümleri hem de graftaki düğüm ve kenar sayıları tutulmaktadır. Bu veri yapısı şekilsel olarak şöyle gösterilebilir:



Peki bu veri yapısında belli bir düğümden hangi düğümlere bağlantı olduğunu nasıl bulabiliriz? Örneğin D düğümden hangi düğümlere bağlantı olduğunu bulmaya çalışalım. Bunun için GRAPH yapısındaki pVertices ile gösterilen dizide D düğümü aranır. (Düğümün ismi yerine indeksi verilirse $O(1)$ karmaşıklıkta düğüm burada bulunacaktır.) D düğümüne ilişkin VERTEX yapısı elde edildikten sonra bu yapının pEdges elemanı ile o vertex'in kenar bilgileri bir EDGE dizisi olarak elde edilecektir. Bu EDGE dizisindeki EDGE nesnelerinin pDest elemanları ilgili düğümün hedefini bize verir. İşte bu tür aramaların daha etkin olması için (örneğin isme dayalı arama yapıyor olabilir) VERTEX dizisinin ve EDGE dizilerinin hash tablolarında tutulması uygun olabilir.

Yukarıda da belirtildiği gibi aslında EDGE dizilerinin birer bağlı liste ya da dinamik dizi biçiminde olması daha uygundur. Çünkü graf yaratıldığında genellikle düğüm sayısı bilinir ancak kenar sayısı baştan bilinmeyebilir. Bu durumda EDGE dizilerinin bağlı liste ya da dinamik dizi biçiminde olması daha uygundur. Eğer işin başında düğüm sayıları da bilinmiyorsa VERTEX dizisinin de benzer biçimde dinamik dizi ya da bağlı liste olması uygun olabilir. Düğüme bağlı kenarların bağlı listede tutulduğu durumda veri yapısı şöyle olacaktır:

```
typedef struct tagVERTEX {
    char name[32];
    struct tagEDGE *pHead;
    /* other info */
} VERTEX;

typedef struct tagEDGE {
    VERTEX *pDest;
    struct tagEDGE *pNext;
    /* other info */
} EDGE;

typedef struct tagGRAPH {
    VERTEX *pVertices;
    size_t vertexCount;
    size_t edgeCount;
} GRAPH, *HGRAPH;
```

Şimdi bu graf veri yapısının fonksiyonlarını yazalım. Handle alanını oluşturan CreateGraph fonksiyonu şöyle yazılabilir:

```
HGRAPH CreateGraph(size_t vertexCount)
{
    HGRAPH hGraph;
    size_t i;

    if ((hGraph = (HGRAPH)malloc(sizeof(GRAPH))) == NULL)
        return NULL;

    if ((hGraph->pVertices = (VERTEX *)malloc(sizeof(VERTEX) * vertexCount)) == NULL) {
        free(hGraph);
        return NULL;
    }

    for (i = 0; i < vertexCount; ++i)
        hGraph->pVertices[i].pHead = NULL;
```

```

hGraph->vertexCount = vertexCount;
hGraph->edgeCount = 0;

return hGraph;
}

```

Şimdi düğümleri veri yapısına iliştiirelim. Örneğimizde düğümlerin yalnızca isimlerinin olduğunu varsayıyoruz:

```

void SetVertex(HGRAPH hGraph, size_t index, char *name)
{
    strcpy(hGraph->pVertices[index].name, name);
}

```

Şimdi de belli bir düğüme bir kenar ekleyelim. Kenarlar EDGE nesnesi ile temsil edilmişlerdir. Örneğimizde kenarların yalnızca hedef düğüm içerdiğini varsayıyoruz:

```

EDGE *AddEdgeByIndex(HGRAPH hGraph, size_t source, size_t dest)
{
    EDGE *pEdge;

    if ((pEdge = (EDGE *)malloc(sizeof(EDGE))) == NULL)
        return NULL;
    pEdge->pDest = &hGraph->pVertices[dest];

    pEdge->pNext = hGraph->pVertices[source].pHead;
    hGraph->pVertices[source].pHead = pEdge;

    ++hGraph->edgeCount;

    return pEdge;
}

EDGE *AddEdgeByName(HGRAPH hGraph, const char *sourceName, const char *destName)
{
    int source, dest;
    size_t i;

    for (i = 0; i < hGraph->vertexCount; ++i)
        if (!strcmp(hGraph->pVertices[i].name, sourceName)) {
            source = i;
            break;
        }
    if (source == hGraph->vertexCount)
        return NULL;

    for (i = 0; i < hGraph->vertexCount; ++i)
        if (!strcmp(hGraph->pVertices[i].name, destName)) {
            dest = i;
            break;
        }
    if (dest == hGraph->vertexCount)
        return NULL;

    return AddEdgeByIndex(hGraph, source, dest);
}

```

Graftaki tüm düğümler de şöyle silinebilir:

```

void Clear(HGRAPH hGraph)
{
    size_t i;
    EDGE *pNode, *pTemp;

    for (i = 0; i < hGraph->vertexCount; ++i) {

```

```

        pNode = hGraph->pVertices[i].pHead;
        while (pNode != NULL) {
            pTemp = pNode->pNext;
            free(pNode);
            pNode = pTemp;
        }
        memset(&hGraph->pVertices[i], 0, sizeof(VERTEX));
        hGraph->pVertices[i].pHead = NULL;
    }

    hGraph->edgeCount = 0;
}

```

Nihayet graf veri yapısı da şöyle yok edilebilir:

```

void CloseGraph(HGRAPH hGraph)
{
    Clear(hGraph);
    free(hGraph->pVertices);
    free(hGraph);
}

```

Graf veri yapısının örnek kodları şöyledir:

```

/* Graph.h */

#ifndef GRAPH_H_
#define GRAPH_H_

#include <stddef.h>

/* Type Declarations */

typedef struct tagVERTEX {
    char name[32];
    struct tagEDGE *pHead;
    /* other info */
} VERTEX;

typedef struct tagEDGE {
    VERTEX *pDest;
    struct tagEDGE *pNext;
    /* other info */
} EDGE;

typedef struct tagGRAPH {
    VERTEX *pVertices;
    size_t vertexCount;
    size_t edgeCount;
} GRAPH, *HGRAPH;

/* Function Prototypes */

HGRAPH CreateGraph(size_t vertexCount);
void SetVertex(HGRAPH hGraph, size_t index, char *name);
EDGE *AddEdgeByIndex(HGRAPH hGraph, size_t source, size_t dest);
EDGE *AddEdgeByName(HGRAPH hGraph, const char *sourceName, const char *destName);
void DispGraph(HGRAPH hGraph);
void Clear(HGRAPH hGraph);
void CloseGraph(HGRAPH hGraph);

/* Macros */

#define GetVertexCount(hGraph) ((hGraph)->vertexCount)
#define GetEdgeCount(hGraph) ((hGraph)->edgeCount)

```

```

#endif

/* Graph.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Graph.h"

HGRAPH CreateGraph(size_t vertexCount)
{
    HGRAPH hGraph;
    size_t i;

    if ((hGraph = (HGRAPH)malloc(sizeof(GRAPH))) == NULL)
        return NULL;

    if ((hGraph->pVertices = (VERTEX *)malloc(sizeof(VERTEX) * vertexCount)) == NULL) {
        free(hGraph);
        return NULL;
    }

    for (i = 0; i < vertexCount; ++i)
        hGraph->pVertices[i].pHead = NULL;

    hGraph->vertexCount = vertexCount;
    hGraph->edgeCount = 0;

    return hGraph;
}

void SetVertex(HGRAPH hGraph, size_t index, char *name)
{
    strcpy(hGraph->pVertices[index].name, name);
}

EDGE *AddEdgeByIndex(HGRAPH hGraph, size_t source, size_t dest)
{
    EDGE *pEdge;

    if ((pEdge = (EDGE *)malloc(sizeof(EDGE))) == NULL)
        return NULL;
    pEdge->pDest = &hGraph->pVertices[dest];

    pEdge->pNext = hGraph->pVertices[source].pHead;
    hGraph->pVertices[source].pHead = pEdge;

    ++hGraph->edgeCount;

    return pEdge;
}

EDGE *AddEdgeByName(HGRAPH hGraph, const char *sourceName, const char *destName)
{
    int source, dest;
    size_t i;

    for (i = 0; i < hGraph->vertexCount; ++i)
        if (!strcmp(hGraph->pVertices[i].name, sourceName)) {
            source = i;
            break;
        }
    if (source == hGraph->vertexCount)
        return NULL;

```

```

    for (i = 0; i < hGraph->vertexCount; ++i)
        if (!strcmp(hGraph->pVertices[i].name, destName)) {
            dest = i;
            break;
        }
    if (dest == hGraph->vertexCount)
        return NULL;

    return AddEdgeByIndex(hGraph, source, dest);
}

void DispGraph(HGRAPH hGraph)
{
    size_t i;
    EDGE *pNode;

    for (i = 0; i < hGraph->vertexCount; ++i) {
        printf("%s ---> ", hGraph->pVertices[i].name);
        pNode = hGraph->pVertices[i].pHead;
        while (pNode != NULL) {
            printf("%s ", pNode->pDest->name);
            pNode = pNode->pNext;
        }
        printf("\n");
    }
}

void Clear(HGRAPH hGraph)
{
    size_t i;
    EDGE *pNode, *pTemp;

    for (i = 0; i < hGraph->vertexCount; ++i) {
        pNode = hGraph->pVertices[i].pHead;
        while (pNode != NULL) {
            pTemp = pNode->pNext;
            free(pNode);
            pNode = pTemp;
        }
        memset(&hGraph->pVertices[i], 0, sizeof(VERTEX));
        hGraph->pVertices[i].pHead = NULL;
    }

    hGraph->edgeCount = 0;
}

void CloseGraph(HGRAPH hGraph)
{
    Clear(hGraph);
    free(hGraph->pVertices);
    free(hGraph);
}

/* Test.c */

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

int main(void)
{
    HGRAPH hGraph;

    if ((hGraph = CreateGraph(4)) == NULL) {
        fprintf(stderr, "cannot create Graph!..\n");
        exit(EXIT_FAILURE);
    }
}

```

```

}

SetVertex(hGraph, 0, "A");
SetVertex(hGraph, 1, "B");
SetVertex(hGraph, 2, "C");
SetVertex(hGraph, 3, "D");

AddEdgeByName(hGraph, "A", "B");
AddEdgeByName(hGraph, "A", "C");

AddEdgeByName(hGraph, "B", "D");

AddEdgeByName(hGraph, "C", "B");
AddEdgeByName(hGraph, "C", "D");

AddEdgeByName(hGraph, "D", "A");
AddEdgeByName(hGraph, "D", "B");
AddEdgeByName(hGraph, "D", "C");

DispGraph(hGraph);

CloseGraph(hGraph);

return 0;
}

```

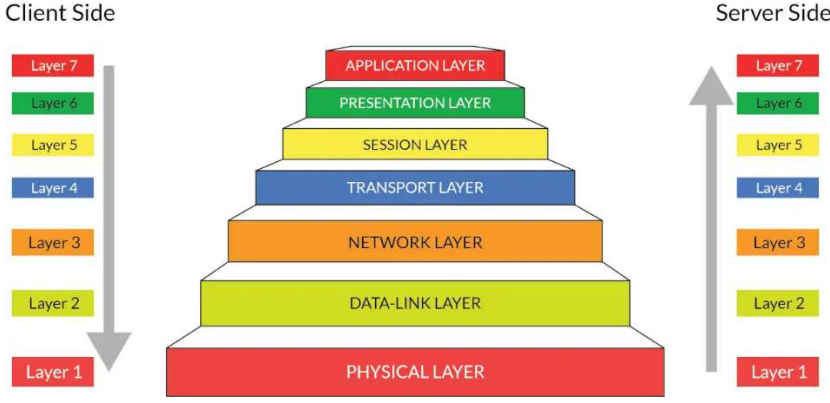
Farklı Makinelerin Prosesleri Arasında Haberleşme

Farklı makineler birbirlerine bir ağ içerisinde bağlanmış olabilir. Biz bir makinede çalışan bir prosesin ağa bağlı başka bir makinedeki prosese bilgi göndermesini ve almasını isteyebiliriz. Böyle bir haberleşmede artık işletim sisteminin dışında başka birtakım aktörler de devreye girecektir. Örneğin ağ alt yapısındaki kablolama sisteminden kullanılan hub'a kadar bazı donanım birimleri işin içine karışmaktadır. Üstelik bu tür haberleşmelerde işletim sistemleri bile birbirlerinden farklı olabilmektedir. İşte böyle heterojen ortamlarda haberleşmenin sağlıklı yürütülmesi için önceden belirlenmiş birtakım kuralların bulunuyor olması gerekir. Örneğin kablo standartları ve konektörler nelerdir? Network kartının özellikleri nasıl olacaktır? Bilgiler nasıl paketlere ayrılıp gönderilecektir? Makineler nasıl birbirlerinden ayrılacaktır gibi... İşte tüm bu belirlemelere protokol denilmektedir.

Tıpkı fonksiyonların birbirlerini çağırarak daha yüksek seviyeli işlemleri yapar hale gelmesi gibi protokoller de üst üste yığılarak birbirlerini kullanacak biçimde oluşturulabilmektedir. Her üst protokol aşağının zaten hazır olduğu fikriyle yalnızca kendi gereksinimlerini tanımlamaktadır. Böyle katmanlı tasarımın pek çok faydası vardır. Örneğin bu sayede üst seviye protokoller detay barındırmazlar ve aşağı düzeydeki protokollerin değişmesinden etkilenmezler. İşte farklı makinelerdeki proseslerin haberleşmesi için bu biçimde oluşturulmuş IP, AppleTalk, NETBIOS gibi pek çok protokol ailesi bulunmaktadır.

Ağ altındaki bilgisayar haberleşmesi için protokol katmanlarının nasıl oluşturulması gerektiğine yönelik ISO ismine OSI (Open System Interconnection) denilen bir doküman yayınlamıştır. Buna OSI model denilmektedir. OSI model bir protokol ailesi değildir. Protokol ailesi oluşturacaklar için bir kılavuz niteliğinde oluşturulmuş bir dokümandır. OSI'nin toplam 7 katmanı vardır:

OSI MODEL



Alıntı Notu: Görsel <https://networkencyclopedia.com/7-layers-osi-model/> adresinden elde edilmiştir.

OSI'nin en aşağı katmanına "Fiziksel Katman (Physical Layer)" deilmektedir. Fiziksel katmanda iletişimin yapılacağı ortam tanımlanmaktadır. Örneğin kullanılacak kablolar, konnektörler, gerilim seviyeleri gibi belirlemeler bu katmana özgü belirlemelerdir. Fiziksel katmanın üzerinde üzerinde "Veri Bağlantı Katmanı (Data Link Layer)" bulunmaktadır. Bu katman fiziksel adresleme gibi, hataların belirlenmesi ve düzeltilmesi gibi, bilgilerin paketlenmesi gibi belirlemelerden oluşmaktadır. Örneğin Ethernet kartlarının kullandığı Ethernet Protokolü bir Veri Bağlantı Katmanı protokolüdür. Network katmanı (Network Layer) mantıksal adreslemenin tanımlandığı, bilginin nasıl paketlere ayrılıp gönderileceğinin belirlendiği en önemli katmanlardan biridir. Örneğin IP protokol ailesinin IP Protokolü (Internet Protocol) OSI'ye göre Network katmanına ilişkindir. Network katmanında ayrıca ağlar arasında paket aktarımı için rotalama belirlemeleri de tanımlanmaktadır. Network katmanı üzerinde "İleti Katmanı (Transport Layer)" bulunmaktadır. İleti katmanı protokolleri paketlerin numaralandırılması gibi, mantıksal port adreslerinin tanımlanması gibi, hata durumunda bunun telafi edilmesi için yapılacak işlemler gibi belirlemelerden oluşur. Örneğin IP protokol ailesindeki TCP ve UDP protokolleri İleti katmanına ilişkin protokollerdir. "Oturum Katmanı (Session Layer)" pek çok ailede bulunmamaktadır. Burada haberleşme için gereken oturum açmaya yönelik izinler gibi kimlik doğrulama gibi belirlemeler bulundurulur. Oturum katmanının üzerinde de "Sunum Katmanı (Presentation Layer)" bulunmaktadır. Sunum katmanında gönderilip alınan bilgilerin sıkıştırılmasına, açılmasına, şifrelenmesine yönelik belirlemeler bulunmaktadır. IP protokol ailesi Sunum Katmanına da sahip değildir. Nihayet OSI'nin en tepesinde "Uygulama Katmanı (Application Layer)" vardır. Bu katman artık belli bir amacı gerçekleştirmek için oluşturulan yazılımların kullanacağı belirlemeleri içerir. Örneğin eposta için kullanılan POP3, dosya transferi için kullanılan FTP birer Uygulama Katmanı protokolüdür.

İnternet'in Kısa Tarihi

Bilgisayarları birbirlerine bağlamak ilk kez 60' yıllarda insanların aklına gelmiştir. Soğuk savaş yıllarında Amerika Savunma Bakanlığına bağlı olan DARPA (Defense Advanced Research Project Agency) kurumu birkaç üniversite ile birlikte 1969 yılında ARPANET isimli bir proje başlattı. ARPANET ilk kez 1969 yılında uzak mesafeden dört üniversitenin birbirlerine bağlanmasıyla hayata geçirilmiş oldu. ARPANET'e daha sonra bazı devlet kurumları ve üniversiteler katılmaya başlamıştır. 70'li yılların sonlarına doğru ARPANET Amerika'da gelişmeye başlamıştır. 1983 yılında ARPANET NCP (Network Control Protocol) protokolünü bırakarak IP ailesine ailesine geçmiştir. Ve artık bu ağ İnternet ismiyle yayılmaya devam etmiştir. İnternet 80 yıllarda Avrupa'ya ve Türkiye'ye de geldi. Ancak tabii kişisel bilgisayarlar daha yeniydi ve İnternet'e ancak Üniversitelerden, bazı devlet kurumlarından ve bazı özel sektör kurumlarından bağlanılabiliyordu. 1990-91 yıllarında HTTP protokö tasarlandı ve ilk Web sayfaları oluşturulmaya başlandı. 90'lı yılların ortalarına doğru tüm dünyada kişisel bilgisayarlarla servis sağlayıcılar sayesinde İnternet'e girmek mümkün hale gelmiştir. Daha sonraları modern modern router'larla evlerden yüksek hızlı erişimler sağlanmıştır.

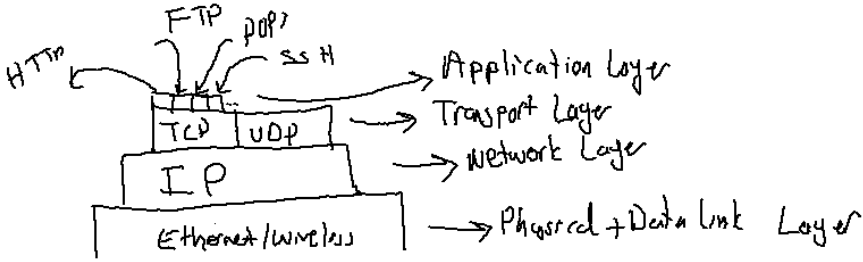
İnternet ismi "internetworking" sözcüğünden gelmektedir. Internetworking "yerel ağların birbirlerine router isimli cihazlarla bağlanmalarıyla oluşturulmaktadır. Internetworking temel bir terimdir ve IP protokol ailesinin ismi buradan gelmektedir. Bugün İnternet denildiğinde herkesin bağlandığı ARPANET'ten evrimleşen dev ağ aklıma gelir. (İnternet yazarken 'ı'yı büyük yazarsak bu ağ anlaşılır.) Şüphesiz mevcut protokoller sayesinde herkes kendi internetini kurabilir. Örneğin biz de birkaç arkadaşınızla ayrı bir İnternet dünyası oluşturabiliriz. Hatta bazı ülkelerin bu biçimde kendilerine özgü İnternet'leri vardır.

IP Protokol Ailesi

IP açık bir protokol ailesidir. Burada açık demekle hiçbir şirketin malının olmaması, protokolün sürdürümünün bağımsız konsorsiyumlar tarafından yapılması kastedilmektedir. Ayrıca açık protokollerde protokol dokümanlarının ücretsiz olarak paylaşılmasıyla ve herkesin önerilerde bulunması mümkün hale getirilmektedir.

IP protokolü Vint Cerf ve Bob Kahn tarafından 1974 yılında önce TCP sonra IP biçiminde tasarlanmıştır. Sonra aileye diğer üyeler katılmıştır. IP protokolünün ilk ciddi gerçekleştirimi BSD sistemleri üzerinde yapılmıştır. 1983 yılında ARPANET'in IP ailesine geçmesiyle bu protokol ailesinin popülaritesi çok artmıştır.

IP protokol ailesinin temel protokolleri dört katmandan oluşmaktadır.

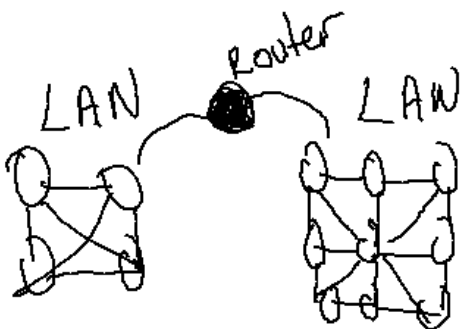


IP protokol ailesi aslında geniş bir ailedir. Ailede pek çok yardımcı protokol vardır. Yukarıdaki şekil yalnızca kursumuzda söz konusu edilen konuları kapsayacak biçimde oluşturulmuştur. Ailenin en önemli taban protokolü IP (Internetworking Protocol) protokolüdür. Zaten IP protokolü aileye ismini veren protokoldür.

IP protokolü paket anahtarlama (packet switching) bir protokoldür. Yani bilgiler paket denilen öbeklere ayrılarak gönderilip alınmaktadır. IP protokolünde adresleme fiziksel değil mantıksaldır. IP protokol ailesinde ağa bağlı her birime "host" denilmektedir. IP protokolünde her host'un ismine IP adresi denilen mantıksal bir adresi vardır. Mantıksal adresler donanımsal olarak değil yazılımsal olarak belirlenmektedir. Ethernet protokolünün kullandığı MAC adreslerinin fiziksel fiziksel adres belirttiğini anımsayınız. Fiziksel adresler donanımsal olarak belirlenen adreslerdir. Örneğin her Ethernet kartının üretim sırasında atanan bir MAC adresi bulunmaktadır. Dolayısıyla mantıksal adresler dinamik, fiziksel adresler statiktir. Mantıksal adresler biz ağa dahil olduğumuzda bize atanmaktadır. Tabii biz ağda bu işten sorumlu olan protokollerde istediğimiz adresin atanması konusunda çeşitli düzeylerde belirleyici olabilmekteyiz.

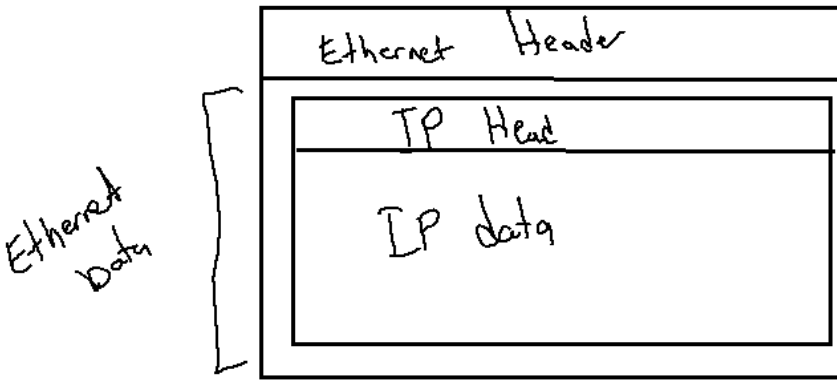
IP protokolünün de versiyonları vardır. Şu anda hala ağırlıklı kullanılan versiyon IPV4'tür. Ancak IPV6 yavaş yavaş daha yaygın kullanılır hale gelmiştir. IPV4te IP adresleri 4 byte uzunluktadır. Ancak IPV6'da IP adresleri 16 byte'tır. 4 byte'lık IP adresleri şu an için artık çok yetersiz kalmaktadır.

Bugün bilgisayarlarımızda fiziksel ve data link katmanı olarak Ethernet ve Wireless Protokolleri kullanılmaktadır. Ethernet protokolü ethernet kartına gereksinim duyar. Bu kart fiziksel olarak bilgileri bilgisayarımızdan dışarı gönderip almakta kullanılır. Ethernet protokolü de paket anahtarlama bir protokoldür. Yani bilgiler paket paket gönderilip alınır. Paket anahtarlama hattın etkin kullanımını sağlar. Biz Ethernet kartlarını bir hub'la birbirine bağlayarak yerel bir ağ (local area network) oluşturabiliriz. Bugün evlerimizdeki ağ da yerel bir ağdır. Yerel ağları birbirlerine bağlamak için "router" denilen aygıtlar kullanılır. Ethernet kartı (yani network kartı) aynı ağdaki bir bilgisayardan diğerine paket haberleşmesi için kullanılmaktadır. Ancak router farklı ağlar arasında paket haberleşmesi için kullanılır. Bugün evlerimizdeki ADSL modemler aynı zamanda birer router görevindedir.



Bizim evimizdeki yerel ağ Internet isimli dev ağa router aracılığıyla tek bir host gibi bağlanmaktadır. Dolayısıyla bizim Internet için dışarıdan kullanılacak tek bir IP adresimiz vardır (Tabi tek bir router ve hattımızın bulunduğunu varsayıyoruz). Bizim evimizdeki yerel ağ ayrı bir IP ağıdır. Yani ayrı bir dünyadır. Biz istersek hiç Internet'te çıkmadan kendi yerel ağımızda tüm Internet uygulamalarını (Yani IP protokol uygulamalarını) çalıştırabiliriz. Buna genellikle "Intranet" denilmektedir. O halde bizim evimizdeki bir bilgisayarın bir yerel IP adresi vardır bir de router'ımızın Internet'ten görülen bir IP adresi vardır. Router dış dünyadan gelen paketleri yerel ağda uygun bilgisayara dağıtmaktadır. Yerel ağdaki paketleri de dış dünyaya ilişkinse dış dünyaya yollamaktadır. Biz yerel ağımızdaki bir host'tan diğerine bilgi gönderirken router devreye girmez.

Ip protokolünde gönderilen bir paketin başında isimli bir başlık kısmı (IP Header) vardır. Burada pakete ilişkin metadata bilgileri bulunur. Örneğin paket hangi IP adresine gönderilmektedir? Checksum bilgisi nedir? Hangi IP versiyonu kullanılmaktadır? vs. Aslında tabii (böyle olmak zorunda değil ama) bilgiler neticede ethernet kartı ile gönderilip alındığı için IP paketi aslında Ethernet protokolünün ethernet paketinin data bölümünde kodlanmaktadır. Ethernet protokolünün de ayrı bir header bölümü vardır. Örneğin:



Ethernet protokolü IEEE 802.3 numaralı standardıyla belirlenmiştir. Wireless protokolü de aynı ailedendir. O da IEEE 802.11 numaralı standarttır.

IP protokü ile birden fazla paketten oluşan bilgi gönderilebilir mi? Evet fakat bunun için paketlere numara vererek bizim de adeta ayrı bir protokol oluşurumuz gerekir. Zaten TCP protokolü buna benzer bir protokoldür.

TCP protokolü güvenli (reliable) bir protokoldür. Burada güvenlik demek alış verişin yolda bozulmasının teleafi edilmesi ve paketlerin düzgün aktarılması anlamına gelir. Çünkü TCP'de bir akış kontrolü (flow control) vardır. Gönderen tarafla alan taraf karşılıklı konuşarak hatalı giden paketlerin telafisini sağlayabilmektedir. TCP stream tabanlı bir protokoldür. Stream tabanlı demekle byte byte okumaya kaldığı yerden devam edebilmek anlaşılır. TCP ile biz daha büyük bilgileri gönderip alabiliriz. TCP bu durumda bu bilgiyi IP paketlerine böler. Onlara numara verir ve onların karşı tarafa güvenli ulaşmasını denetler. Karşı taraf gelen bilgiyi sanki borudan okuma yapıyormuş gibi byte byte elde edebilir.

UDP (User Datagram Protocol) güvenli olmayan paket tabanlı (datagram) bir haberleşme sunar. Yani UDP'de bilgiler IP'deki gibi bağımsız paketler halinde gönderilip alınır. UDP'de bir paket ya alınır ya alınmaz. Byte byte okuma mümkün değildir. Paketin alındığına dair bir geri bildirim yapılmaz. Tabi bu özelliğinden dolayı UDP daha hızlıdır. UDP özellikle periyodik data gönderimlerinde, televizyon yayını gibi işlemlerde tercih edilmektedir.

TCP bağlantılı (connection oriented) bir protokoldür, UDP bağlantısızdır (connectionless). Bağlantılı protokol demek iki taraf haberleşmeden önce birbirlerine bağlanıp karşılıklı konuşma için birbirlerini tanımaları demektir. TCP tipik olarak client-server tarzda bir çalışmayı akla getirmektedir. Client-server haberleşmede bir taraf client bir taraf server olur. Client taraf server tarafa bağlanır, haberleşme bundan sonra yapılır.

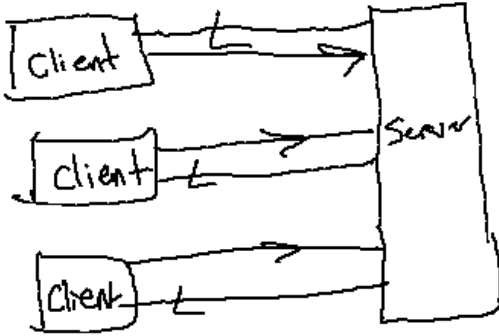
TCP	UDP
Bağlantılı	Bağlantısız

Stream Tabanlı	Datagram Tabanlı
Güvenilir	Güvenilir Değil
Yavaş	Hızlı

TCP ve UDP'de işin içine port numarası kavramı da girmektedir. Port numarası aynı host'taki uygulamaları birbirlerinden ayırmak için düşünülmüştür. Adeta şirketlerdeki içsel (internal) telefon numaralarına benzetilebilir. TCP ve UDP protokollerinde bilgi göndermek için yalnızca gönderilecek host'un IP'sinin bilinmesi yeterli değildir. Aynı zamanda oradaki uygulamanın hangi port ile ilgilendiğinin de bilinmesi gerekir. Genellikle gösterimde ip adresi ve port numarası aralarına ':' karakteri getirilerek "ip:port" biçiminde belirtilmektedir. IPV4'te toplam 65536 port numarası vardır (yani port numarası için iki byte yer ayrılır). IPV6'da ise port numaraları 4 byte uzunluğundadır. IPV4'te ilk 1024 port numarası Internet'in kendi uygulama protokolleri için ayrılmıştır. Bunlara "well known" portlar da denilmektedir. Örneğin FTP 21, SSH 22, Telnet 23, HTTP 80 numaralı portları kullanmaktadır. Biz kendi uygulamalarımız için port numarası belirleyeceksek ilk 1024 portu kullanmamalıyız.

Client-Server Çalışma Modeli

Yukarıda da belirtildiği gibi TCP tipik olarak client-server bir çalışmayı akla getirmektedir. Client-Server modelde ismine client ve server denilen iki ayrı program vardır. Asıl işi server program yapar. Client yalnızca istekte bulunur. Server işi yapar sonuçları client'a gönderir. Bir server birden fazla client'a hizmet verebilmektedir. >



Client-Server modelde önce client server'a bağlanır. Haberleşme ondan sonra başlar. Client-Server uygulamalar her ne kadar TCP'yi çağırıyor olsa da aslında bu bir haberleşme mimarisidir. Yani aslında client-server çalışma için IP ailesinin kullanılması gerekmez. Bu çalışma örneğinin aynı makinadaki prosesler arasında borularla mesaj kuyruklarıyla da sağlanabilir.

Client-Server çalışmanın şu avantajları vardır:

- 1) Server programın çalıştığı makine güçlü olabilir. Biz de onun gücünden yararlanmak istiyor olabiliriz. Örneğin uzun zaman alan bir işlemi el terminalinden yapmak yerine el terminalini client olarak kullanıp asıl işi server'a yaptırmak uygun olabilir.
- 2) Server program kaynak paylaşımı sağlayabilir. Örneğin yazıcı telk bir bilgisayara bağlıdır. Başka bilgisayardaki print programları client gibi çalışarak yazıcının bağlı makinadaki server programa isteği iletir. Server da print işlemini client için yapar. Ya da örneğin server'a bir veritabanı bağlıdır. Client ondan istekte bulunur. Örneğin banka ATM'lerinde veritabanı ATM makinasının içerisinde değildir. ATM'deki program client program gibi davranmaktadır.
- 3) Server program client'lar arasında işbirliği sağlayabilir. Onlar arasındaki iletişime aracılık edebilir. Örneğin bir char programında client'lar birbirini tanımamaktadır. Herkes yalnızca server'ı tanır. Her client server'a bağlanır. Server client arasında haberleşmeye aracılık eder. Ağ üzerinde çalışan oyun programları bu biçimde bir server'ın işbirliği ile gerçekleştirilmektedir.

4) Client-Server çalışma dağıtık uygulamalarda da karşımıza çıkabilmektedir. Yani bir işin belirli parçalarını başka bilgisayarlarda yapıp sonra onu birleştirmek isteyebiliriz.

Soket (Socket) Arayüzü

Soket arayüzü ağ haberleşmesi için kullanılan bir kütüphanedir. Soket kütüphanesi ilk kez 1983 yılında BSD sistemlerinde gerçekleştirilmiştir. Daha sonra başka sistemlere uygulanmıştır. Microsoft'un soket arayüzü BSD soketlerinden alınmadır. Buna Winsock kütüphanesi denilmektedir. Windows'ta iki grup soket API'si vardır. Bunlardan birincisi tamamen BSD uyumlu API'lerdir. (Burada fonksiyon isimleri BSD'deki ile aynıdır.) İkinci olarak başı WSA ile başlayan Windows'a özgü soket API'leridir. Biz Windows'ta da BSD uyumlu soket fonksiyonlarını kullanırsak UNIX/Linux uyumunu da sağlamış oluruz.

Soket arayüzü yalnızca IP ailesi için düşünülmüş bir arayüz değildir. Diğer protokolleri de kapsayan genel bir arayüzdür. Bu nedenle fonksiyonların parametrik yapıları biraz daha karmaşık olma eğilimindedir.

TCP/IP İskelet Client-Server Programların Yazımı

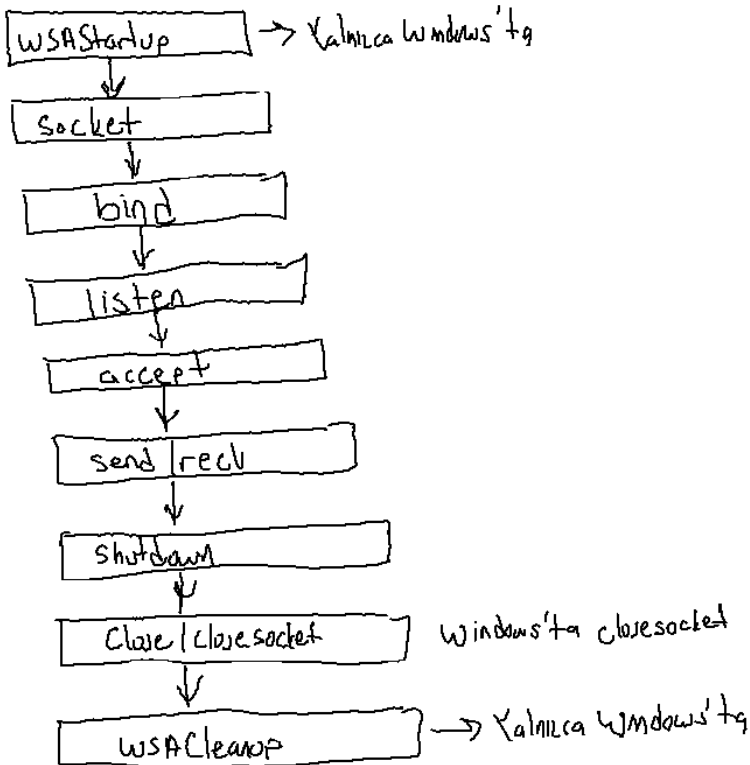
Burada iskelet bir TCP/Ip client-server programın yazımı ele alınacaktır. Uygulamada klasik BSD soket fonksiyonları kullanılacaktır. Windows'a özgü farklılıklara konu içerisinde değinilmektedir.

Anahtar Notlar: Windows'ta soket kütüphanesini ayrıca projeye dahil etmek gerekmektedir. Bunun için proje ayarlarından Linker/Input/Additional Dependencies sekmesinden Ws2_32.lib import kütüphanesi eklenmelidir. Tüm soket API'lerinin prototipleri ve değer önemli bildirimler <Winsock2.h> başlık doyası içerisinde dir. Soket uygulamalarında bu başlık dosyasının include edilmesi gerekmektedir.

Windows'ta soket fonksiyonlarında UNIX/Linux sistemlerinde olmayan çeşitli typedef'ler ve sembolik sabitler de kullanılmaktadır. Windows soket uygulamalarını UNIX/Linux sistemlerine taşıırken kodun bu kısımlarının düzeltilmesi gerekir. Ayrıca Windows'ta son soket fonksiyonu hatayla geri dönmüşse hata kodu GetLastError fonksiyonuyla değil, WSAGetLastError fonksiyonuyla elde edilmelidir.

Server Programın Yazımı

Tipik bir TCP/Ip server programda sırasıyla şunlar yapılmalıdır:



Windows'ta (Fakat UNIX/Linux sistemlerinde değil) soket sistemini işin başında aktive etmek gerekmektedir. Bu işlem soket kullanan her proste bir kez yapılmak zorundadır. WSASStartup fonksiyonunun prototipi şöyledir:

```
int WSASStartup(  
    _In_ WORD wVersionRequested,  
    _Out_ LPWSADATA lpWSADATA  
);
```

Fonksiyonun birinci parametresi Winsock kütüphanesinin versiyon numarasını belirtir. Yüksek bir numara verilirse hata oluşmaz, en yüksek soket versiyonu işleme sokulur. Bu parametre MAKEWORD makrosuyla oluşturulabilir. Halen Winsock kütüphanesinin son versiyonu 2.2'dir. Fonksiyonun ikinci parametresi WSADATA isimli bir yapının adresini almaktadır. Fonksiyon bu yapının içeriğini faydalı bilgilerle doldurur. Fonksiyon başarı durumunda sıfır değerine başarısızlık durumunda hata kodunun kendisine geri döner.

Soketi yaratmak için socket isimli fonksiyon kullanılır. Fonksiyonun prototipi şöyledir:

```
SOCKET WSAAPI socket(  
    _In_ int af,  
    _In_ int type,  
    _In_ int protocol  
);
```

Fonksiyonun UNIX/Linux sistemlerindeki geri dönüş değeri int türündendir. Burada Windows'ta bu tür SOCKET isimli typedef ile temsil edilmektedir. (Yani SOCKET typedef ismi UNIX/Linux sistemlerinde yoktur).

Fonksiyonun birinci parametresi kullanılacak protokol ailesini belirtir. IPV4 protokol ailesi için bu parametre AF_INET biçiminde girilmelidir. İkinci parametre kullanılacak protokolün türünü belirtir. (Yani stream tabanlı mı, datagram mı gibi.) TCP için bu parametre SOCK_STREAM, UDP için SOCK_DGRAM biçiminde girilmelidir. Üçüncü parametre kullanılacak üst protokolü belirtir. TCP için IPPROTO_TCP, UDP için IPPROTO_UDP girilebilir. Fakat IP ailesi için eğer ikinci parametre SOCK_STREAM girilmişse ya da SOCK_DGRAM girilmişse bu üçüncü parametre sıfır geçilebilir. Bu durumda SOCK_STREAM için TCP, SOCK_DGRAM için UDP anlaşılır. Fonksiyon başarı durumunda soketin handle değerine başarısızlık durumunda INVALID_SOCKET değerine geri döner. UNIX/Linux sistemlerinde INVALID_SOCKET isimli bir sembolik sabit yoktur. Bu sistemlerde fonksiyon başarısızlık durumunda -1 değerine geri dönmektedir (Zaten Windows sistemlerinde de INVALID_SOCKET -1 olarak typedef edilmiştir.)

Server program soketi yarattıktan sonra bağlamalıdır (binding). Soketin bağlanması (bind edilmesi) sırasında aslında şu belirlemeler yapılmaktadır:

- 1) Server hangi porttan gelen bağlantı isteklerine yanıt verecektir? Başka bir deyişle server hangi portu kullanacaktır?
- 2) Server hangi network arayüzünden (network kartından) gelen bağlantı isteklerini dikkate alacaktır?

Bind fonksiyonunun prototipi şöyledir:

```
int bind(  
    _In_ SOCKET s,  
    _In_ const struct sockaddr *name,  
    _In_ int namelen  
);
```

Fonksiyonun birinci parametresi bind edilecek soketin handle değerini almaktadır. İkinci parametre IP ailesi için sockaddr_in isimli bir yapının adresini alır. Fonksiyonun üçüncü parametresi genel bir tür olarak struct sockaddr * türündendir. (Eskiden void göstericiler yoktu bu parametre bu nedenle genel bir türü belirtmek için bu biçimde alınmıştır.) Her ne kadar fonksiyonun ikinci parametresi struct sockaddr * türündense de aslında protokole bağlı olarak bir yapı almaktadır. (Örneğin IP ailesinde struct sockaddr_in *, UNIX domain soketlerde struct sockaddr_un * gibi) Fonksiyon üçüncü parametresi ikinci parametrede girilen yapının byte uzunluğunu (yani sizeof'unu) almaktadır.

Fonksiyon başarı durumunda sıfır, başarısızlık durumunda SOCKET_ERROR (UNIX/LINUX sistemlerinde -1) değerine geri döner.

sockaddr_in yapısı şöyledir:

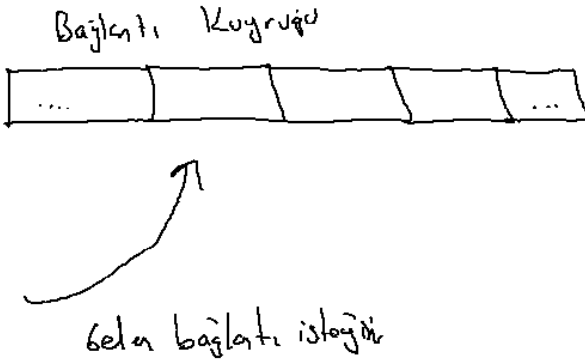
```
struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

Yapının sin_family elemanı kullanılan protokol ailesini belirtir. IP ailesi için bu elemana AF_INET girilmelidir. Yapının sin_port elemanına server'ın dinlemek istediği port numarası girilmelidir. Yapının sin_addr elemanına da server'ın dinlemek istediği network arayüzünün (kartının) IP numarası girilmelidir. Ancak bu parametreye INADDR_ANY özel değeri girilirse bu durumda server tüm network arayüzünden gelen bağlantıları kabul eder.

IP ailesinde ortak bir belirleme olarak bir byte'tan uzun olan bilgilerin "big endian" formata göre depolanmasına ve iletilmesine karar verilmiştir. Bu durumda biz Intel işlemcilerinde olduğu gibi "little endian" formatta çalışıyorsak bir byte'tan uzun değerlerin big endian'a dönüştürülmesi gerekir. İşte bunu yapan iki fonksiyon vardır: htons (host to network byte ordering short) ve htonl (host to byte ordering long) fonksiyonları. Bu fonksiyonlar eğer zaten big endian sistemde çalışılıyorsa hiçbir şey yapmadan aynı değerle geri dönmektedir.

```
struct sockaddr_in sinServer;
...
sinServer.sin_family = AF_INET;
sinServer.sin_port = htons(SERVER_PORT);
sinServer.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(serverSock, (struct sockaddr *)&sinServer, sizeof(sinServer)) == SOCKET_ERROR)
    ExitSys("bind", EXIT_FAILURE, WSAGetLastError());
```

Soket bind edildikten sonra artık server aktif dinleme konumuna geçmelidir. Bu işlem listen fonksiyonuyla yapılır. Listen fonksiyonu blokeye yol açmaz. Dinleme işlemini işletim sisteminin kendisi arka planda yapmaktadır. İşletim sistemi prosese ilişkin bir bağlantı isteği geldiğinde o isteği bir bağlantı kuyruğuna ekler. Artık server program da bağlantı isteklerini kuyruktan alacaktır.



Listen fonksiyonunun bağlantıyı kurmadığına yalnızca gelen bağlantı isteklerini bize ilettiğine dikkat ediniz. Listen fonksiyonunun prototipi şöyledir:

```
int listen(
    _In_ SOCKET s,
    _In_ int backlog
);
```

Fonksiyonun birinci parametresi dinleme soketinin (server soketin) handle değerini alır. İkinci parametre dinleme kuyruğunun eleman uzunluğunu belirtir. Yoğun olmayan server'lar için 8 gibi bir değer uygun olabilmektedir. İşletim

sistemi gelen bağlantı isteğini kuyruğa yerleştirir. Eğer kuyruk dolarsa artık yeni bağlantı istekleri kuyruğa yerleştirilemez doğrudan reddedilir. Fonksiyon başarı durumunda sıfır değerine, başarısızlık durumunda SOCKET_ERROR değerine geri döner. Örneğin:

```
if (listen(serverSock, 8) == SOCKET_ERROR)
    ExitSys("listen", EXIT_FAILURE, WSAGetLastError());
```

Şimdi sıra artık accept işlemine gelmiştir. Accept fonksiyonu kuyrukta sıradaki bağlantı isteğini alır ve bağlantıyı sağlar. Eğer kuyrukta hiçbir bağlantı yoksa blokeli modda bağlantı isteği gelene kadar thread'i blokede bekletmektedir. Blokesiz modda accept kuyrukta bağlantı isteği yoksa başarısızlıkla sonuçlanmaktadır. Default mod blokeli moddur. Fonksiyonun prototipi şöyledir:

```
SOCKET accept(
    SOCKET          s,
    struct sockaddr *addr,
    int             *addrlen
);
```

Fonksiyonun birinci parametresi dinleme socketinin handle değerini alır. İkinci parametre bağlanılacak client'ın bilgilerinin yerleştirileceği sockaddr_in yapısının adresini almaktadır. Üçüncü parametreye ikinci parametredeki yapının byte uzunluğunun yerleştirildiği nesnenin adresi girilmelidir. Fonksiyon duruma göre bunu güncelleyebilir. Fonksiyon başarı durumunda client ile konuşmakta kullanılacak socketin handle değerine geri döner. Başarısızlık durumunda da INVALID_SOCKET (UNIX/Linux sistemlerinde -1) değerine geri dönmektedir.

Görüldüğü gibi her accept işlemi bize bağlanılan client ile konuşmakta kullanılabilecek yeni bir socket vermektedir. Yani server'ın bir tane dinleme socketi vardır. Bununla listen ve accept yapar. Ancak her accept işleminde yeni bir socket elde edilir. Örneğin:

```
printf("waiting client to connect...\n");

addrLen = sizeof(sinClient);
if ((clientSock = accept(serverSock, (struct sockaddr *)&sinClient, &addrLen)) == INVALID_SOCKET)
    ExitSys("accept", EXIT_FAILURE, WSAGetLastError());
```

Server bağlantıyı sağladığında artık bağlandığı client'ın IP numarasını ve port numarasını sockaddr_in yapısından alabilir. Örneğin:

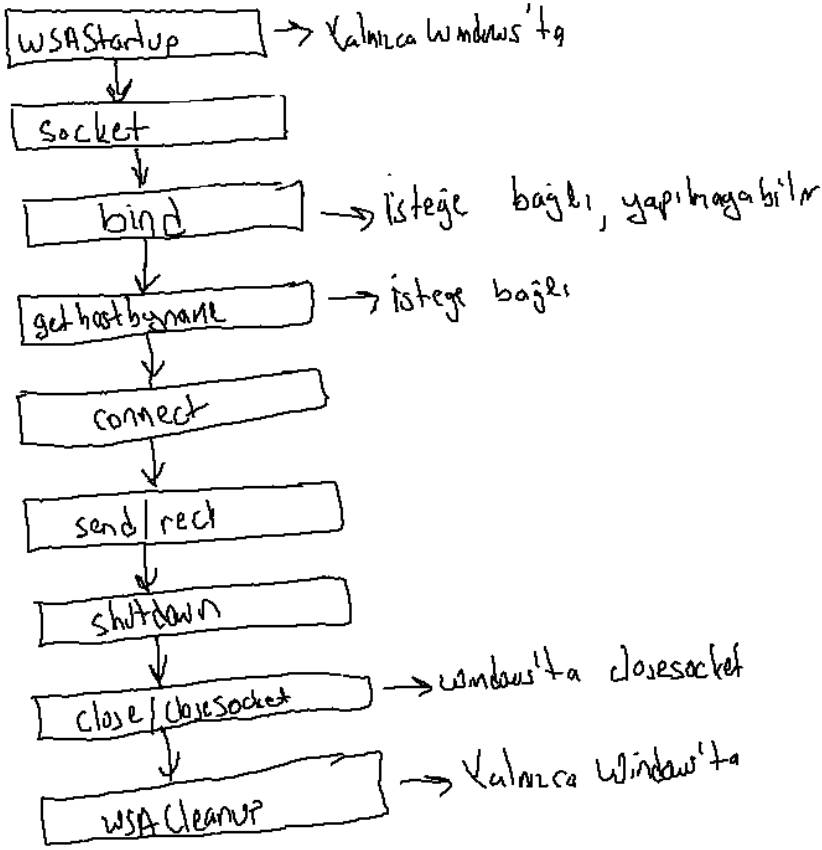
```
printf("Client connected: %s:%d\n", inet_ntoa(sinClient.sin_addr), ntohs(sinClient.sin_port));
```

Burada inet_ntoa, IP adresini noktalı formatta yazıya dönüştürmektedir. ntohs ise htons fonksiyonunun ters işlemini yapar.

Bağlantı sağlandıktan sonra artık send/rcv fonksiyonlarıyla karşılıklı konuşma yapılır. Bu fonksiyonlar client tarafında da kullanıldığı için client tarafı anlatıldıktan sonra ortak biçimde ele alınacaktır.

Client Programın Yazımı

Client program sırasıyla şu aşamalardan geçilerek yazılır:



Görüldüğü gibi client da işin başında bir socket yaratır. Socket'in bind edilmesi server'da zorunludur. Ancak client socketi bind etmeyebilir. Bu durumda işletim sistemi kaynak port olarak rastgele bir port numarası atar. Bağlantıda kaynak socketin de hedef socketin de birer port numarası vardır. Biz kaynak port numarasını belirlemek istiyorsak ya da bağlantı için belli bir network kartından çıkış istiyorsak client tarafta da bind işlemi yapmalıyız.



Örneğin biz 192.168.1.21 IP numaralı host'a 5050 numaralı port'tan bağlanmak isteyelim. Burada hedef port 5050 dir. Ancak client ona herhangi bir porttan bağlanabilir. Yani bizim server'ın 5050 numaralı portuna bağlanmamız için kendi portumuzun 5050 olması gerekmez. İşte client da bind yapılmazsa işletim sistemi ona rastgele (aslında tam olarak rastgele değil, bir kuralı var) bir kaynak port atamaktadır.

Bazı server'lar ancak bazı portlardan gelen bağlantı isteklerini kabul etmektedir. Hatta router'lar buna göre ayarlanabilmektedirler. Örneğin biz evimizde "router'a kaynak port 6300 değilse dış dünyadan gelen bağlantı isteğini hiç içeriye bildirme, doğrudan reddet" diyebiliriz. Fakat kaynak portun bu biçimde sınırlandırılması çok nadir bir uygulamadır.

Client taraf bağlantı için server'ın IP adresini ve port numarasını bilmek durumundadır. Ancak IP adreslerinin akılda tutulması zor olduğu için onlara isimler de karşı düşürülmüştür. Bu isimlere "host ismi (host name)" denilmektedir. İnternet'te host isimleri ile IP adreslerini eşleştiren veritabanları bulunmaktadır. Bunlara "Domain Name Server" denir. Bu server'lara ismine DNS denilen özel bir protokolle erişilmektedir. İşte eğer client server'ın host ismini biliyorsa bu host ismini DNS işlemiyle IP adresine dönüştürmesi gerekir. Bunu gethostbyname isimli socket API fonksiyonu yapar. Bu fonksiyonun ters işlemini yapan gethostbyaddress isimli bir fonksiyon da vardır:

```
struct hostent *gethostbyname(
```

```

const char *name
);

struct hostent *gethostbyaddr(
    const char *addr,
    int len,
    int type
);

```

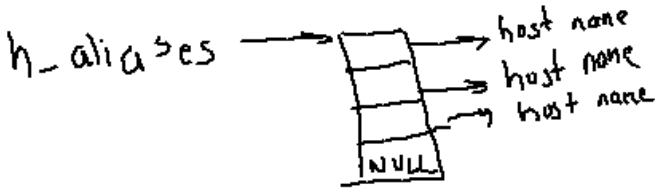
Bu iki fonksiyon da bize static olarak tahsis edilmiş hostent isimli bir yapının adresine geri döner. Hostent yapısı şöyledir:

```

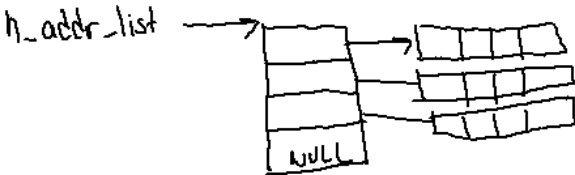
typedef struct hostent {
    char *h_name;
    char **h_aliases;
    short h_addrtype;
    short h_length;
    char **h_addr_list;
} HOSTENT, *PHOSTENT, *LPHOSTENT;

```

Yapının h_name elemanı domain host ismini vermektedir. h_aliases elemanı diğer host isimlerini verir.



Yapının h_addr_list elemanı ip numaralarını vermektedir. IPV4'te bu göstericiyi gösteren gösterici 4'er elemanlık char türden dizileri gösterir:



Fakat bazen kullanıcılar host ismini vermek yerine onun IP adresini noktalı biçimde yazı olarak da verebilirler. İşte bu biçimde verilmiş yazıyı dört byte'lık sayıya dönüştürmek için inet_addr isimli bir fonksiyon kullanılmaktadır.

```

unsigned long inet_addr(
    const char *cp
);

```

Eğer girilen yazı uygun formatta değilse fonksiyon INADDR_NONE değerine geri döner. İşte verilen yazıyı önce bu fonksiyona sokup eğer yazı noktalı biçimde IP belirtmiyorsa gethostbyname fonksiyonuna sokmak iyi bir tekniktir.

Artık bağlanılacak host'un ip adresi de belirlendiğine göre sıra connect işlemine gelmiştir. Conenct fonksiyonun prototipi şöyledir:

```

int connect(
    SOCKET s,
    const struct sockaddr *name,
    int namelen
);

```


Fonksiyonun birinci parametresi socketin handle değerini alır. İkinci parametre bağlanılacak server'ın bilgilerini alan sockaddr_in türünden yapısının adresini almaktadır. Yani programcı server'ın ip numarasını ve port numarasını sockaddr_in türünden bir yapının içerisine yerleştirip bu yapıyı da connect fonksiyonuna verir. sockaddr_in yapısı üzerinde server programda biraz durmuştuk. Yapıyı tekrar hatırlayalım:

```
struct sockaddr_in {
    short    sin_family;
    u_short  sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

Görüldüğü gibi IP adresi yapının sin_addr elemanıdır. Ancak sin_addr in_addr türünden bir yapı belirtmektedir:

```
typedef struct in_addr {
    union {
        struct {
            u_char s_b1,s_b2,s_b3,s_b4;
        } S_un_b;
        struct {
            u_short s_w1,s_w2;
        } S_un_w;
        u_long S_addr;
    } S_un;
} IN_ADDR, *PIN_ADDR, FAR *LPIN_ADDR;
```

Görüldüğü gibi bu yapının içerisinde bir birlik vardır. Birliğin de S_addr elemanı long türünden ip adresini belirtmektedir. O halde biz sin_addr.S_un.S_addr ifadesi ile long türden ip adresine erişebiliriz. İşte bu ifade uzun olduğu için aşağıdaki gibi bir makroyla ifade kısaltılmıştır:

```
#define s_addr S_un.S_addr
```

Bu durumda yukarıdaki ifade sin_addr.s_addr biçiminde kısaltılabilir. sockaddr_in yapısı UNIX/Linux sistemlerinde biraz daha farklı organize edilmiş olabilir. Ancak yapının elemanları aynı isimdedir. Örneğin bu sistemlerde in_addr içerisinde doğrudan s_addr bulunuyor olabilir. Bu durumda connect işlemi şöyle yapılır:

```
struct sockaddr_in sinClient;
...
sinServer.sin_family = AF_INET;
sinServer.sin_port = htons(SERVER_PORT);
if ((sinServer.sin_addr.s_addr = inet_addr(SERVER_NAME)) == INADDR_NONE) {
    if ((host = gethostbyname(SERVER_NAME)) == NULL)
        ExitSys("gethostbyname", EXIT_FAILURE, WSAGetLastError());
    memcpy(&sinServer.sin_addr.s_addr, host->h_addr_list[0], host->h_length);
}
if (connect(clientSock, (struct sockaddr *)&sinServer, sizeof(sinServer)) == SOCKET_ERROR)
    ExitSys("connect", EXIT_FAILURE, WSAGetLastError());
```

connect fonksiyonu uygulandığında server program accept'te beklemiyorsa belli bir zaman aşımı süresi kadar beklenir sonra connect başarısız olur.

Şüphesiz çok client'lı uygulamalarda server bir kez değil döngü içerisinde her client için accept uygulamalıdır. Her accept server'a o client'la konuşmak için ayrı socket verir.

Client İle Server Arasında Bilgi Alış Verişi send ve recv Fonksiyonları

Bağlantı sağlandıktan sonra artık client server'a server da client'a send ve recv fonksiyonlarıyla bilgi gönderip alabilir. Send fonksiyonun prototipi şöyledir:

```
int send(  
    _In_ SOCKET s,  
    _In_ const char *buf,  
    _In_ int len,  
    _In_ int flags  
);
```

Fonksiyonun birinci parametresi soket handle değerini alır. İkinci parametre gönderilecek bilginin yerleştirileceği dizinin adresini belirtir. Üçüncü parametre gönderilecek byte sayısını belirtmektedir. Son parametre bazı gönderim özelliklerini belirlemede kullanılır. Bu parametre sıfır geçilebilir. Fonksiyon blokeli modda network tamponuna aktarılan byte sayısı ile geri döner. Network tamponu doluysa fonksiyon tampona yazabildiği kadar bilgiyi yazar ve bloke olmadan geri döner. Send başarısızlık durumunda SOCKET_ERROR (UNIX/Linux sistemlerinde -1) değerine geri dönmektedir.

send fonksiyonu bilgi karşı tarafa ulaşana kadar beklemez. send bilgiyi network tamponuna yazar hemen geri döner. Network taponundaki bu bilgi paketlenerek karşı tarafa yollanacaktır. Yani send'ten başarılı olarak geri dönülmesi bilgiyi karşı tarafın aldığı anlamına gelmez. Ayrıca network tamponu doluysa send tüm bilginin tampona yazılmasını beklemeyebilir. Yazabildiği kadarını yazıp o değere geri dönebilir. (Bazı sistemlerde send tüm bilgi yazılana kadar blokede kalabilmektedir. O sistemlerde bile çok büyük bilgiler tam olarak tek seferde tampona yazılamayabilir.) Fakat send eğer network tamponu tıka basa doluysa en az bir byte yazana kadar blokede bekler. Bu nedenle send'in geri dönüş değeri yine acaba tüm bilgiler tampona aktarıldı mı diye kontrol edilmelidir.

recv fonksiyonu da çok benzerdir:

```
int recv(  
    _In_ SOCKET s,  
    _Out_ char *buf,  
    _In_ int len,  
    _In_ int flags  
);
```

Fonksiyonun birinci parametresi soketin handle değerini alır. İkinci parametre bilginin yerleştirileceği char türden dizinin adresini almaktadır. Üçüncü parametre kaç byte okunmak istendiğini belirtir. Son parametre ise okuma işleminin biçimini belirtmektedir. Bu parametre sıfır geçilebilir. recv fonksiyonu talep edilen byte'ın hepsinin okunmasını beklemez. O anda sokete ne kadar bilgi gelmişse talep edilen o kadarını okur ve okuyabildiği byte sayısı ile geri döner. Yani biz örneğin recv ile 100 byte okumak isteyelim. Bu yüz byte'ı tek bir recv ile okuyamayabiliriz. recv o anda gelmiş olan 40 byte bilgi varsa bize hemen onu verir. Bizi bekletmez. Recv başarısız olduğunda SOCKET_ERROR değeri ile geri dönmektedir. Ayrıca TCP/IP soket sisteminde karşı tarafın tek bir send gönderdiğini diğer tarafın tek bir recv ile alması garanti değildir. recv fonksiyonu blokeli modda (default durumda) en az bir byte okuyana kadar blokede bekler. Örneğin recv ile biz 1024 byte okumak istemiş olalım. Ancak network tamponunda hazır hiçbir byte olmasın. recv blokede bekler. Sonra örneğin tampona 5 byte gelmiş olsun. recv bu 5 byte'ı alıp hemen geri döner.

shutdown ve close işlemleri

Soket bir handle sistemidir. Dolayısıyla tüm handle sistemlerinde olduğu gibi işimiz bitince soketleri de kapatmalıyız. Tabii biz soketi kapatmamışsak prosesin bitmesiyle işletim sistemi dosyalarda olduğu gibi soketleri de kapatmaktadır. Soketin kapatılması UNIX/Linux sistemlerinde close fonksiyonuyla (anımsanacağı gibi close aynı zamanda dosyaları da kapatmaktadır. Zaten soketler de bu sistemlerde birer dosya gibi ele alınmaktadır) Windows sistemlerinde closesocket fonksiyonuyla yapılır. Soket kapatıldıktan sonra artık o soketten okuma ve yazma yapamayız. Ancak soketi kapatmadan önce shutdown fonksiyonunu çağırmak tavsiye edilir. Önce shutdown sonra close işlemine "graceful close" denilmektedir. shutdown fonksiyonu ile biz soketin kapatılması için hazırlık yaparız. Böylece TCP düzeyinde iletişimin kesilmesi için bir el sıkışma gerçekleşir. shutdown fonksiyonunun prototipi şöyledir:

```
int shutdown(  
    __in SOCKET s,  
    __in int how  
);
```

Fonksiyonun birinci parametresi socketin handle değerini alır. İkinci parametre şunlardan biri olabilir:

SD_RECEIVE
SD_SEND
SD_BOTH

SD_SEND "ben artık bilgi göndermeyeceğim" anlamına gelir. Böylece karşı taraf recv fonksiyonu ile okuma yaptığında sanki socketi kapatılmış gibi görünür. Fakat bu sırada SD_SEND uygulamış taraf socketten okuma yapabilir. SD_RECEIVE ise "ben artık socketten okuma yapmayacağım, fakat gönderme yapabilirim" anlamına gelir. SD_BOTH her iki durumu da kapsamaktadır. İşte socket önce el sıkışmalı biçimde shutdown ile sonra da close uygulanarak kapatılmalıdır. Server'ın dinleme socketi için shutdown uygulamasına gerek yoktur.

Windows sistemlerinde iskelet bir TCP/IP Client server program şöyle yazılabilir:

```
/* Server.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <Winsock2.h>

#define SERVER_PORT    5050

void ExitSys(LPCSTR lpszMsg, int status, DWORD dwLastError);

int main(void)
{
    WSADATA wsaData;
    SOCKET serverSock, clientSock;
    struct sockaddr_in sinServer, sinClient;
    char buf[1024];
    int addrLen;
    int result;

    if ((result = WSASStartup(MAKEWORD(2, 2), &wsaData)) != 0)
        ExitSys("WSASStartup", EXIT_FAILURE, result);

    if ((serverSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == SOCKET_ERROR)
        ExitSys("socket", EXIT_FAILURE, WSAGetLastError());

    sinServer.sin_family = AF_INET;
    sinServer.sin_port = htons(SERVER_PORT);
    sinServer.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(serverSock, (struct sockaddr *)&sinServer, sizeof(sinServer)) == SOCKET_ERROR)
        ExitSys("bind", EXIT_FAILURE, WSAGetLastError());

    if (listen(serverSock, 8) == SOCKET_ERROR)
        ExitSys("listen", EXIT_FAILURE, WSAGetLastError());

    printf("waiting client to connect...\n");

    addrLen = sizeof(sinClient);
    if ((clientSock = accept(serverSock, (struct sockaddr *)&sinClient, &addrLen)) == INVALID_SOCKET)
        ExitSys("accept", EXIT_FAILURE, WSAGetLastError());

    printf("Client connected: %s:%d\n", inet_ntoa(sinClient.sin_addr), ntohs(sinClient.sin_port));

    for (;;) {
        if ((result = recv(clientSock, buf, 1024 - 1, 0)) == SOCKET_ERROR)
            ExitSys("recv", EXIT_FAILURE, WSAGetLastError());
        buf[result] = '\0';
        if (!strcmp(buf, "exit"))
            break;
        puts(buf);
    }
}
```

```

    }

    shutdown(clientSock, SD_BOTH);
    closesocket(clientSock);

    closesocket(serverSock);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status, DWORD dwLastError)
{
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

/* Client.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <Winsock2.h>

#define SERVER_PORT    5050
#define SERVER_NAME    "127.0.0.1"

void ExitSys(LPCSTR lpszMsg, int status, DWORD dwLastError);

int main(void)
{
    WSADATA wsaData;
    SOCKET clientSock;
    struct sockaddr_in sinClient;
    struct sockaddr_in sinServer;
    struct hostent *host;
    char buf[1024];
    int result;

    if ((result = WSASStartup(MAKEWORD(2, 2), &wsaData)) != 0)
        ExitSys("WSAStartup", EXIT_FAILURE, result);

    if ((clientSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == SOCKET_ERROR)
        ExitSys("socket", EXIT_FAILURE, WSAGetLastError());

#if 0

#define CLIENT_PORTNO  5060

    sinClient.sin_family = AF_INET;
    sinClient.sin_port = htons(CLIENT_PORTNO);
    sinClient.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(clientSock, (struct sockaddr *)&sinClient, sizeof(sinClient)) == SOCKET_ERROR)
        ExitSys("WSAStartup", EXIT_FAILURE, WSAGetLastError());
#endif

    sinServer.sin_family = AF_INET;
    sinServer.sin_port = htons(SERVER_PORT);
    if ((sinServer.sin_addr.s_addr = inet_addr(SERVER_NAME)) == INADDR_NONE) {
        if ((host = gethostbyname(SERVER_NAME)) == NULL)

```

```

        ExitSys("gethostbyname", EXIT_FAILURE, WSAGetLastError());
        memcpy(&sinServer.sin_addr.s_addr, host->h_addr_list[0], host->h_length);
    }
    if (connect(clientSock, (struct sockaddr *)&sinServer, sizeof(sinServer)) == SOCKET_ERROR)
        ExitSys("connect", EXIT_FAILURE, WSAGetLastError());

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

    for (;;) {
        printf("Text:");
        gets(buf);
        if (send(clientSock, buf, strlen(buf), 0) == SOCKET_ERROR)
            ExitSys("send", EXIT_FAILURE, WSAGetLastError());
        if (!strcmp(buf, "exit"))
            break;
    }

    shutdown(clientSock, SD_BOTH);
    closesocket(clientSock);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status, DWORD dwLastError)
{
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

```

Yukarıdaki client-server programının Linux sistemlerinde eşdeğeri de şöyle yazılabilir:

Çok Client'lı Uygulamalar

Normal olarak server'lar aynı anda birden fazla client'ta hizmet verebilecek biçimde tasarlanırlar. Bu durum server programlarının yazılmasını biraz karmaşık hale getirmektedir. Öncelikle çok client'lı server programlar yazılırken yalnızca bir tane accept uygulanmaması gerekir. Her accept sıradaki client ile bağlantı kuracağına göre böyle server'larda accept fonksiyonları da bir döngü içerisinde uygulanmalıdır.

Tabii çok client'lı server programlarının asıl zorluğu birden fazla accept uygulamak değildir. Server her bağlandığı client için yeni bir soket elde eder ve o client'la o soketi kullanarak konuşur. İşte birden fazla client ile aynı anda konuşma sırasında server bir client'ta blokede kalabilmektedir. Bu durumda diğer client'larla server'ın konuşması mümkün olmaz. Server ile client'lar arasındaki konuşma aynı anda ve birbirlerini etkilemeden gerçekleştirilmelidir. (Örneğin server client'lardan birine recv uygulamış olsun. Fakat o client henüz server'a birşey göndermemiş olsun. Server akışı blokede kalacağından dolayı server artık diğer client'larla konuşamaz hale gelir.) Pekiyi server'ın birden fazla client ile bloke olmadan aynı anda konuşması nasıl sağlanmaktadır? İşte bunun için birkaç yöntem uygulanmaktadır: Thread oluşturma yöntemi, select ya da poll yöntemi, IO Completion port yöntemi. Şimdi bunları sırasıyla ele alalım:

Thread Yöntemi

Bu yöntemde server her bir client ile bağlantı sağladığında bir thread yaratır ve o client ile o thread yoluyla konuşur. Böylece bir thread bloke olsa bile diğerleri çalışmaya devam edeceği için sorun oluşmaz. Örneğin:

```

for (;;) {
    if ((clientSock = accept(serverSock, (struct sockaddr *) &sinClient, &addrLen)) == INVALID_SOCKET)
        ExitSys("accept", EXIT_FAILURE, WSAGetLastError());
}

```

```

printf("Client connected: %s:%d\n", inet_ntoa(sinClient.sin_addr), ntohs(sinClient.sin_port));
if ((hThread = CreateThread(NULL, 0, ClientThreadProc, clientSock, 0, &dwThreadId)) == NULL)
    ExitSys("CreatrThread", EXIT_FAILURE, GetLastError());
}
}

```

Görüldüğü gibi Thread fonksiyonuna socket parametre olarak geçirilmiştir. Bu durumda her client için aynı fonksiyon çalıştırılır fakat bunlara geçirilen parametre farklı olduğu için farklı işlemler gerçekleşecektir. Yukarıdaki örnekte thread fonksiyonuna yalnızca socket geçirilmiştir. Eğer thread fonksiyonuna socketin yanı sıra başka parametreler de geçirilecekse bunun için bir yapının organize edilmesi uygun olur. Örneğin:

```

typedef struct tagCLIENT_INFO {
    struct sockaddr_in sinClient;
    SOCKET sock;
    int sourcePort;
} CLIENT_INFO;
...
for (;;) {
    addrLen = sizeof(sinClient);
    if ((clientSock = accept(serverSock, (struct sockaddr *)&sinClient, &addrLen)) == INVALID_SOCKET)
        ExitSys("accept", EXIT_FAILURE, WSAGetLastError());

    printf("Client connected: %s:%d\n", inet_ntoa(sinClient.sin_addr), ntohs(sinClient.sin_port));

    if ((pClientInfo = (CLIENT_INFO *)malloc(sizeof(CLIENT_INFO))) == NULL) {
        fprintf(stderr, "cannot allocate memory!..\n");
        exit(EXIT_FAILURE);
    }
    pClientInfo->sinClient = sinClient;
    pClientInfo->sock = clientSock;
    pClientInfo->sourcePort = ntohs(sinClient.sin_port);

    if ((hThread = CreateThread(NULL, 0, ClientThreadProc, pClientInfo, 0, &dwThreadId)) == NULL)
        ExitSys("accept", EXIT_FAILURE, GetLastError());
}

```

Thread modelinin en önemli dezavantajı fazlaca sistem kaynağı kullanma eğiliminde olmasıdır. Her ne kadar IO olaylarına yanıt veren thread'ler zamanının büyük çoğunluğunu uykuda geçiriyorsa da thread'ler yine de önemli ölçüde sistem kaynağına mal olmaktadır. Örneğin her client için bir thread'in yaratıldığı durumda thread'lerin stack'leri bile ciddi bir bellek harcamasına yol açabilir. O halde thread yöntemi çok fazla client'ın söz konusu olduğu server sistemlerinde uygun değildir. (Buradaki "çok fazla" nitelemesi o andaki sisteme bağlı olarak değişebilir. Örneğin bugün kullandığımız güçlü bir donanımda birkaç yüz thread ciddi sistem kaynağı harcar.) Ancak bazı çok client'lı uygulamalardaki client sayısı makul düzeyde olabilir. Örneğin bir kağıt oyununda client'lar az sayıdadır. Bu tür durumlarda thread modeli basitliğinden dolayı ilk tercih edilecek modellerdendir.

Select / Poll Modeli

Bu model özellikle UNIX/Linux sistemlerinde tercih edilmektedir. Fakat Windows sistemlerinde de select fonksiyonu kullanılabilir. select ve poll fonksiyonları aslında aynı işlemi farklı parametrik yapılarla gerçekleştiren iki POSIX fonksiyonudur. select System 5 grubu sistemlerde poll ise BSD türevi sistemlerde ilk kez tanımlanmıştır. Fakat bugün her iki fonksiyon da POSIX standartlarında bulunmaktadır. Tabii aslında UNIX/Linux sistemlerinde select ve poll yalnızca socket işlemleri için kullanılan fonksiyonlar değildir. Bu fonksiyonlar genel olarak asenkron IO işlemlerinde kullanılmaktadır. Örneğin bu fonksiyonlar normal dosyalarla, borularla da çalışabilmektedir. (Ancak Windows sistemlerinde select fonksiyonu bu kadar genel bir fonksiyon değildir. Yalnızca socket işlemlerinde kullanılmaktadır.)

select (ve poll) kullanımı şöyledir: Biz socketleri bir dizide toplayıp select fonksiyonuna veririz. select bu socketleri kendisi izler. Eğer onların hiçbirinde bir IO olayı yoksa bizi blokede bekletir. Eğer socketlerin bir ya da birden fazlasında bir IO olayı gerçekleşmişse select blokeyi çözer. Biz de hangi socketlerde hangi IO olayının gerçekleştiğini sorgularız. Yalnızca onlar üzerinde blokeye yol açmadan işlem yaparız. Örneğin biz select'e 100 client'ın socketini vermiş olalım. O sırada bir

client'tan bilgi gelmiş olsun. select blokeyi çözecektir. Biz de o soketten okuma yaparsak hiç bloke oluşmayacaktır. Tabii select bir kez çağrılacak bir fonksiyon değildir. Genellikle döngü içerisinde bu işlemler hep yinelenir. select kullanımının sembolik kodu şöyle gösterilebilir:

```
for (;;) {
    select(<izlenecek_soketler>);
    if (<hangi sokete bilgi gelmiş>) {
        <o soketten okuma yap>
    }
    ...
}
```

Akış select fonksiyonundan çıktığında biz recv yaptığımızda karşı tarafın gönderdiği bilginin yalnızca bir kısmını elde etmiş olabiliriz. Bu nedenle okunan bilgilerin bir tamponda toplanması ve bilginin tamamı geldiğinde işleme sokulması gerekmektedir. Biz select fonksiyonuna hangi IO olayları ile ilgilendiğimizi veririz. select de yalnızca o olayları bizim için izler.

IO Completion Port ve Asenkron IO Modelleri

"IO Completion Port" Windows'a özgü olan ve Windows'ta en fazla tercih edilen modeldir. Bunun UNIX/Linux sistemlerindeki en yakın karşılığı "Asenkron IO" modelidir. IO completion Port arka planda pek çok framework tarafından da kullanılmaktadır ve Windows sistemlerinde Microsoft tarafından çeşitli zamanlarda optimize edilmiştir. IO Completion Port modeli de aslında genel bir IO modelidir. Yani yalnızca soketlerle değil diğer IO olaylarıyla da kullanılmaktadır. IO Completion Port modelinde biz soketleri sisteme veririz. Bu soketlerde olay gerçekleştiğinde sistem bizim belirlediğimiz fonksiyonları çağırır. Tabii çağırma sistemin kendisinin yarattığı bir akış tarafından asenkron yapılmaktadır. Biz de bu callback fonksiyon içerisinde okuma yazma işlemlerini ve diğer işlemleri yaparız. UNIX/Linux sistemlerindeki aio_xxx fonksiyonları da benzer biçimde kullanılmaktadır.

Client İle Server Arasındaki Mesajlaşmalar

Client ile server arasındaki mesajlaşmalar aslında bir uygulama protokolü oluşturmaktadır. Temel olarak mesajlaşma text tabanlı olarak ya da binary düzeyde yapılabilir. Text tabanlı mesajlaşmanın gerçekleştirilmesi daha kolaydır. Fakat binary mesajlaşma daha hızlı olma eğilimindedir. IP ailesinin üst seviye protokollerinin çoğu text düzeyde mesajlaşma yapmaktadır.

Text düzeydeki mesajlaşmalarda client ve server birbirlerine istekleri ve yanıtları birer yazı olarak gönderir alır. Örneğin dört işlem yapan ve sonucu client'a gönderen bir server söz konusu olsun. Client'tan server'a gönderilen komutların genel yapısı şöyle olabilir;

```
"ADD op1 op2"
"SUB op1 op2"
"MUL op1 op2"
"DIV op1 op2"
```

Server'dan client'a gönderilen yanıtın formatı da şöyle olabilir:

```
"RESULT val"
```

Server aldığı yazıyı parse eder. Sonucu hesaplar ve onu client'a gönderir. Örneğin IRC stili bir chat programı yazacak olalım. Client'tan server'a gönderilecek mesajlar şunlar olabilir:

```
"LOGIN user_name password"
"CHATMSG message"
"LOGOUT"
"GETUSERS"
```

Server'dan client'a gönderilcek mesajlar da şöyle olabilir:

```
"LOGIN_ACCEPTED"  
"NEWMSG message"  
"USERLIST"
```

Örneğin dizin işlemleri yapan bir TCP uygulamasında client'tan server'a gönderilen mesajlar şunlar olabilir:

```
"LOGIN username password"  
"GETDIR"  
"CHDIR path"  
"LISTDIR"  
"GETFILE name"  
"SENDFILE name"  
"LOGOUT"
```

Server'dan client'a gönderilen mesajlar da şöyle olabilir:

```
"LOGIN_ACCEPTED"  
"CURRENT_DIR"  
"OK"  
"DIRCONTENTS file list"  
"FILECONTENTfile_data"  
"ERROR"
```

Görüldüğü gibi bu tür uygulamalarda client taraf server'a soketten bağlandıktan sonra bir de ayrıca kendisini server'a kabul ettirmeye çalışmaktadır. Bu işleme login işlemi diyebiliriz. Yani client'ın server'a fiziksel olarak bir porttan bağlanmış olması server'ın onu kabul edeceği anlamına gelmemektedir. Bazen bu iki kavram "fiziksel bağlantı" ve "mantıksal bağlantı" biçiminde de ifade edilebilmektedir.

Komutlar ve yanıtlar yazısal olarak gönderilip alınırken alan taraf yazının sonunu nasıl bilecektir? İşte bunun için iki yöntem kullanılabilir. Birinci yöntemde yazı özel bir karakterle (örneğin '\n' ya da '\0' gibi) sonlandırılır. Karşı taraf da o karakteri görene kadar portu byte byte okur. Tabii portun böyle byte byte okunması çok verimli olmadığı için okuyan tarafın daha büyük bloğu okuyup bir tampona yerleştirmesi ve o tamponda sonlandırıcı karakteri araması daha uygun olur. İkinci yöntemde her yazının uzunluğunun yazıdan önce gönderilmesidir. Yazı uzunluğu binary olarak gönderilebilir. IP protokol ailesinin uygulama katmanındaki FTP, POP3, TELNET, HTTP gibi protokolleri text tabanlı mesajlaşma yöntemini kullanmaktadır. Bu yöntemde client server'a isteğini yukarıda belirtildiği gibi yazı biçiminde gönderir. Yazıların sonu CR/LF karakterleriyle sonlandırılmaktadır. Soketten CR/LF karakterlerini görene kadar tamponlu okuma yapan bir fonksiyon şöyle yazılabilir:

```
int ReadLineSocket(SOCKET sock, char *buf, size_t len)  
{  
    char *bufx = buf;  
    static char *bp;  
    static int count = 0;  
    static char b[2048];  
    char ch;  
  
    while (--len > 0) {  
        if (--count <= 0) {  
            count = recv(sock, b, sizeof(b), 0);  
            if (count == SOCKET_ERROR)  
                return -1;  
            if (count == 0)  
                return 0;  
            bp = b;  
        }  
        ch = *bp++;  
        *buf++ = ch;  
    }  
}
```



```

        if (ch == '\n') {
            *buf = '\0';
            return buf - bufx;
        }
    }

    return SOCKET_ERROR;
}

```

Fonksiyonun statik nesne kullandığı için thread güvenli olmadığına dikkat ediniz.

Belli bir miktarda bilginin kesin olarak gönderilip alınabilmesi için aşağıdaki iki fonksiyon kullanılabilir:

```

int ReadSocket(SOCKET sock, const void *buf, int count)
{
    int result;
    int left = count, index = 0;

    while (left > 0) {
        if ((result = recv(sock, (char *)buf + index, left, 0)) == SOCKET_ERROR)
            return SOCKET_ERROR;
        if (result == 0)
            break;
        index += result;
        left -= result;
    }

    return index;
}

```

```

int WriteSocket(SOCKET sock, const void *buf, int count)
{
    int result;
    int left = count, index = 0;

    while (left > 0) {
        if ((result = send(sock, (char *)buf + index, left, 0)) == SOCKET_ERROR)
            return SOCKET_ERROR;
        if (result == 0)
            break;
        index += result;
        left -= result;
    }

    return index;
}

```

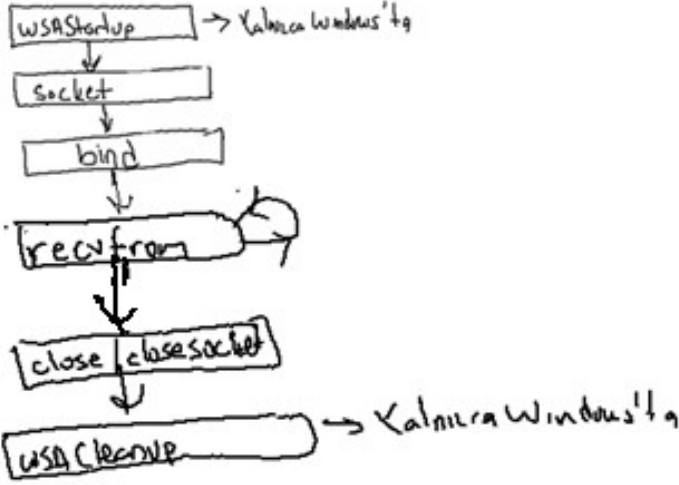
UDP/IP Haberleşme

Daha önceden de belirtildiği gibi UDP bağlantısız ve datagram haberleşme sunmaktadır. UDP'de bir bağlantı olmadığı için TCP'deki gibi bir akış kontrolü yoktur. Gönderen taraf bilgiyi bir paket olarak gönderir. Alan taraf bunu byte byte almaz. Paketi tümünden alır. Gönderen taraf bilginin alıcı tarafından alındığını bilmez. UDP özellikle periyodik birtakım mesajların iletilmesi için tercih edilmektedir. Örneğin dağıtık bir sistemde server'lar belli periyotlarla proxy'ye "ben çalışıyorum, sağlam durumdayım" mesajı iletmek isteyebilirler. Ya da bir oyun programı periyodik olarak arabanın konumunu ve hızını ış dünyaya iletmek isteyebilir. Bu tür durumlarda UDP hızından ve kullanım kolaylığından dolayı tercih edilmektedir. Televizyon yayını gibi "broadcast" işlemlerde de UDP kullanılmaktadır.

UDP'de TCP'de olduğu gibi client ve server kavramları çok belirgin değildir. Ancak yine de UDP'de geleneksel olarak bilgiyi alan tarafa server, bilgiyi gönderen tarafa client denilmektedir.

UDP Server Programının Organizasyonu

Tipik bir UDP server program şöyle oluşturulmaktadır:



Server hangi port'tan gelen bilgileri alacağını yine bind işlemi ile belirler. recvfrom her türlü client'tın gönderdiği bilgileri okumaktadır. Server gönderilen paketin kimin tarafından gönderildiğini recvfrom fonksiyonunun parametresinden elde eder. recvfrom fonksiyonunun parametrik yapısı şöyledir:

```
int recvfrom(  
    SOCKET s,  
    char *buf,  
    int len,  
    int flags,  
    struct sockaddr *from,  
    int *fromLen  
);
```

Fonksiyonun birinci parametresi socketin handle değerini, ikinci parametresi UDP paketinin yerleştirileceği adresi alır. Üçüncü parametre bu dizinin uzunluğunu belirtir. Eğer paket daha büyükse böylece dizi taşmaz, kırılarak diziye yerleştirilir. Dördüncü parametre 0 geçilebilir. Fonksiyonun beşinci parametresine sockaddr_in yapısının adresi girilir. Böylece paketin kimden geldiği anlaşılacaktır. Fonksiyonun son parametresi bizim ve karşı tarafın sockaddr yapısının byte uzunluğunu almaktadır. Biz fonksiyonu çağırmadan önce buraya kendi sockadd_in yapımızın sizeof değerini girmemiz gerekir. Fonksiyon da bu parametreyi güncelleyerek karşı tarafın sockaddr yapısının uzunluğunu elde eder. recvfrom fonksiyonu başarısızlık durumunda SOCKET_ERROR değerine başarı durumunda da alınan byte sayısına geri döner.

UDP socketinin yaratılırken protokolün UDP olarak belirlenmiş olması gerekir. Örneğin:

```
if ((serverSock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == INVALID_SOCKET) {  
    fprintf(stderr, "socket failed: %lu\n", WSAGetLastError());  
    exit(EXIT_FAILURE);  
}
```

Tipik bir iskelet UDP server programı şöyle oluşturulabilir:

```
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
#include <WinSock2.h>  
#include <Windows.h>  
  
#define PORTNO 5050
```

```

int main(void)
{
    WSADATA wsaData;
    int result;
    SOCKET serverSock;
    struct sockaddr_in sinServer;
    struct sockaddr_in sinClient;
    int addrLenClient;
    char buf[8192];

    if ((result = WSASStartup(MAKEWORD(2, 2), &wsaData)) != 0) {
        fprintf(stderr, "WSAStartup failed: %d\n", result);
        exit(EXIT_FAILURE);
    }

    if ((serverSock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == INVALID_SOCKET) {
        fprintf(stderr, "socket failed: %lu\n", WSAGetLastError());
        exit(EXIT_FAILURE);
    }

    sinServer.sin_family = AF_INET;
    sinServer.sin_port = htons(PORTNO);
    sinServer.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(serverSock, (struct sockaddr *) &sinServer, sizeof(sinServer)) == SOCKET_ERROR) {
        fprintf(stderr, "bind failed: %lu\n", WSAGetLastError());
        exit(EXIT_FAILURE);
    }

    for (;;) {
        addrLenClient = sizeof(sinClient);
        result = recvfrom(serverSock, buf, 8192, 0, (struct sockaddr *) &sinClient, &addrLenClient);
        printf("%d byte received from %s:%d\n", result, inet_ntoa(sinClient.sin_addr),
sinClient.sin_port);
    }

    closesocket(serverSock);

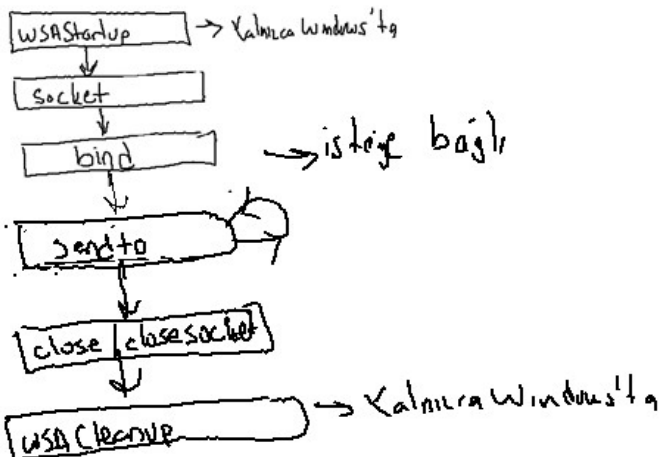
    WSACleanup();

    return 0;
}

```

UDP Client Programın Organizasyonu

UDP/IP haberleşmede geleneksel olarak paketi gönderen tarafa client denilmektedir. İskelet bir client program şöyle oluşturulur:



sendto fonksiyonunun prototipi şöyledir:

```
int sendto(
    SOCKET s,
    const char *buf,
    int len,
    int flags,
    const struct sockaddr *to,
    int tolen
);
```

Fonksiyonun birinci parametresi socketin handle değerini ikinci parametresi gönderilecek paket içeriğinin bulunduğu dizinin adresini alır. Üçüncü parametre paketin byte sayısıdır. Dördüncü parametre yine 0 geçilebilir. Beşinci parametre gönderilecek server'ın ip adresi ve port numarasının bulunduğu sockaadr_in yapısının adresini alır. Son parametre yine bu sockaadr_in yapısının byte uzunluğunu almaktadır. sendto fonksiyonu başarısızlık durumunda SOCKET_ERROR değerine başarı durumunda gönderilen byte sayısına geri döner.

Tipik bir UDP client program şöyle yazılabilir:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <WinSock2.h>
#include <Windows.h>

#define PORTNO 5050
#define HOSTNAME "127.0.0.1"

int main(void)
{
    WSADATA wsaData;
    int result;
    struct sockaddr_in sinClient;
    int addrLenClient;
    SOCKET clientSock;
    struct hostent *host;
    char buf[1024];

    if ((result = WSASStartup(MAKEWORD(2, 2), &wsaData)) != 0) {
        fprintf(stderr, "WSASStartup failed: %d\n", result);
        exit(EXIT_FAILURE);
    }

    if ((clientSock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == INVALID_SOCKET) {
        fprintf(stderr, "socket failed: %lu\n", WSAGetLastError());
        exit(EXIT_FAILURE);
    }

    sinClient.sin_family = AF_INET;
    sinClient.sin_port = htons(PORTNO);
    sinClient.sin_addr.s_addr = inet_addr(HOSTNAME);
    if (sinClient.sin_addr.s_addr == INADDR_NONE) {
        if ((host = gethostbyname(HOSTNAME)) == NULL) {
            fprintf(stderr, "gethosybyname failed: %lu\n", WSAGetLastError());
            exit(EXIT_FAILURE);
        }
        memcpy(&sinClient.sin_addr.s_addr, host->h_addr_list[0], host->h_length);
    }

    for (;;) {
        printf("Text:");
        gets(buf);
    }
}
```

```

    if (sendto(clientSock, buf, strlen(buf) + 1, 0, (struct sockaddr *)&sinClient,
sizeof(sinClient)) == SOCKET_ERROR) {
        fprintf(stderr, "send failed!\n");
        exit(EXIT_FAILURE);
    }
    if (!strcmp(buf, "exit"))
        break;
}

closesocket(clientSock);

WSACleanup();

return 0;
}

```

Kütüphanelerin Oluşturulması ve Kullanılması

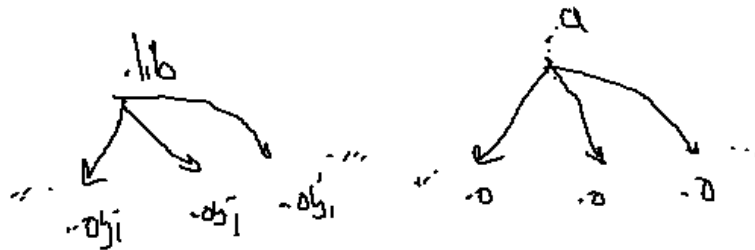
İçerisinde kullanıma hazır halde fonksiyonların ve sınıfların bulunduğu dosyalara "kütüphane (library)" denilmektedir. C/C++ gibi derleme yoluyla kullanılan programlama dillerinde fonksiyonlar da derlenmiş bir biçimde kütüphanelerin içerisinde bulunurlar. Yorumlayıcılar yoluyla kullanılan Python, Ruby, PHP gibi dillerde genel olarak fonksiyonlar ve sınıflar kütüphanelerin içerisinde kaynak kod biçiminde bulunmaktadır. Biz burada sistem programlama bağlamında C ve C++ programlama dillerinde söz konusu olan işletim sistemi ve derleyiciler tarafından aşağı seviyeli bir biçimde desteklenen kütüphaneler üzerinde duracağız. Sınıflar aslında yüksek seviyeli mantıksal bir organizasyondur. Aşağı seviyeli olarak bakıldığında sınıflar aslında yalnızca fonksiyonlardan oluşmaktadır. Sınıfların içerisindeki fonksiyonların sınıf ile ilişkisi mantıksal düzeydedir ve bu mantıksal ilişki derleyiciler tarafından kurulmaktadır.

Kütüphaneler statik ve dinamik olmak üzere ikiye ayrılırlar. Windows'ta statik kütüphane dosyalarının uzantıları .lib (library) biçiminde dinamik kütüphane dosyalarının uzantıları da .dll (dynamic link library) biçimindedir. UNIX/Linux sistemlerinde ise statik kütüphaneler .a (archive), dinamik kütüphaneler ise .so (shared object) biçimindedir.

.NET gibi Java gibi platformlarda ara kod sistemi kullanıldığı için buradaki kütüphanelerin oluşturulması ve kullanılması C/C++ gibi doğal kodlu sistemlerden farklıdır. (Örneğin biz Windows'ta bir DLL dosyası gördüğümüzde bu C ve C++ gibi dillerle oluşturulmuş doğal kod içeren bir DLL olabileceği gibi C#, Visual BASIC gibi dillerle oluşturulmuş bir DLL dosyası da olabilir. Biz kurulumuzda doğal kod içeren kütüphaneleri ele alacağız.)

Statik Kütüphaneler

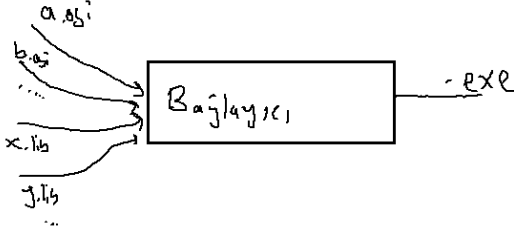
C ve C++'ta derleme işlemi sonucunda elde edilen dosyalara "object modül dosyaları" ya da kısaca "object modüller" denilmektedir. Statik kütüphane dosyaları Windows'ta ve UNIX/Linux sistemlerinde aslında object müdüleri (object modules) tutan bir dizin gibidir. Derleme sonucunda oluşturulan object modül dosyalarının Windows'taki uzantılarının .obj biçiminde, UNIX/Linux sistemlerindeki uzantılarının ise .o biçiminde olduğunu anımsayınız.



Object modül dosyaları içerisinde derlenmiş fonksiyonlar bulunmaktadır. Bu durumda biz C'de bir grup fonksiyonu bir statik kütüphane içerisine yerleştirmek istersek önce onları bir kaynak dosya içerisine yazıp derleyerek bir object modül dosyası elde ederiz. Sonra bu object modül dosyasını kütüphane dosyası içerisine yerleştiririz.

Bağlama (linking) işlemi "bağlayıcı (linker)" denilen programlar tarafından yapılmaktadır. Microsoft firması Windows sistemleri için "link.exe" isimli bağlayıcı program bulundurmaktadır. UNIX/Linux ve MAC sistemlerinde ağırlıklı olarak GNU projesi kapsamında geliştirilmiş olan "ld" isimli bağlayıcı programlar kullanılmaktadır.

Normalde bağlama işlemine girdi olarak birden fazla object modül dosyası ve statik kütüphane dosyası verilebilmektedir. Örneğin:



Biz C'de birbirinden bağımsız olarak derleyerek oluşturduğumuz object modül dosyalarını ve birtakım statik kütüphane dosyalarını bağlayıcı programa girdi olarak verdiğimizde bağlayıcı program verdiğimiz object modül dosyalarını birleştirerek çalıştırılabilen dosyayı (executable file) oluşturmaktadır. Ancak derleyici bu birleştirme süreci içerisinde object modüllerden çağrılmış olan fakat bu object modüller içerisinde bulunmayıp kütüphaneler içerisindeki object modüller içerisinde bulunan fonksiyonları da tespit etmekte ve birleştirme işlemine bu object modülleri de dahil etmektedir. Örneğin Windows sistemlerinde geliştirdiğimiz projemiz "a.c" ve "b.c" isimli iki kaynak dosyadan oluşuyor olsun. Bu kaynak dosyalar içerisinde "mylib.lib" isimli bir statik kütüphanede bulunan fonksiyonları çağırılmış olalım. "mylib.lib" kütüphane dosyasının da "x.obj", "y.obj" ve "z.obj" isimli object modülleri içerdiğini varsayalım. Bizim "a.c" ve "b.c" dosyalarından çağırdığımız fonksiyonların da "mylib.lib" dosyasının içerisindeki "x.obj" modülünde olduğunu kabul edelim. Bu durumda biz bağlayıcı programa "a.obj", "b.obj" ve "mylib.lib" dosyalarını girdi olarak veririz. Bağlayıcı program da "a.obj", "b.obj" ve "x.obj" dosyalarını birleştirerek çalıştırılabilir dosyayı elde edecektir. Bir statik kütüphaneden bir fonksiyon çağırıldığında bağlayıcının çağrılan fonksiyonun statik kütüphanenin içerisindeki hangi object modül içerisinde olduğunu belirlediğine ve o object modülü de birleştirme işlemine soktuğuna dikkat ediniz. Böylece biz bir statik kütüphaneden tek bir fonksiyon çağırılmış olsak bile o fonksiyonun içinde bulunduğu object modülün tamamı çalıştırılabilen dosyaya eklenecektir. Görüldüğü gibi statik kütüphanelerden bir çağrı yapıldığında çağrılan fonksiyonun bulunduğu object modülün tamamı oradan alınarak çalıştırılabilen dosyanın içerisine yerleştirilmektedir. Böylece program çalışırken artık statik kütüphaneye gereksinim duyulmamaktadır.

Statik kütüphaneler bağlama aşamasında devreye girdiği için kullanım ve konuşlandırma (deployment) bakımından bazı durumlarda avantajlar sağlamaktadır. Ancak statik kütüphaneler çalıştırılabilen dosyaları büyütme eğiliminde olduğu için disk kullanımı bakımından dezavantajlara sahiptir.

Anahtar Notlar: Visual Studio IDE'sinde standart C kütüphanesinin default olarak dinamik versiyonu (yani DLL versiyonu) kullanılmaktadır. Ancak komut satırında /MT seçeneği ile ya da Visual Studio IDE'sinde "Project Properties/C-C++/Code Generation/Runtime Library" girişinden statik kütüphane kullanımı sağlanabilmektedir. Linux'ta gcc ile derleme yaparken yine default durumda standart C kütüphanesinin dinamik versiyonu (.so versiyonu) kullanılmaktadır. Fakat biz açıkça derleme yaparken yine standart C kütüphanesinin static versiyonunun kullanılmasını sağlayabiliriz.

Anahtar Notlar: Microsoft'un C ve C++ derleyicisinin ismi cl.exe'dir. Linux ve Mac OS X sistemlerinde C derleyicisi olarak ağırlıklı biçimde gcc clang derleyicileri kullanılmaktadır. Her iki derleyici de esas olarak komut satırından çalıştırılmaktadır. Visual Studio IDE'si ya da Linux'taki IDE'ler derleme yaparken aslında bu komut satırından çalıştırılan derleyicileri kullanırlar. Yani örneğin Visual Studio IDE'sinin ayrı bir C/C++ derleyicisi yoktur. Visual Studio IDE'si derleme sırasında "cl.exe" derleyicisini kullanmaktadır. cl.exe default durumda derlemeyi yaptıktan ürettiği object modülleri ve standart C fonksiyonlarının ve API fonksiyonlarının bulunduğu kütüphaneleri girdi yaparak sonra Microsoft'un bağlayıcı programı olan "link.exe" programını çalıştırmaktadır. Ancak biz istersek /c seçeneği ile cl.exe'nin link.exe'yi çalıştırmasını engelleyebiliriz. Aynı çalışma biçimi Linux'ta ve MAC OS X sistemlerinde de böyledir. gcc ve clang derleyicileri default durumda "ld" isimli bağlayıcı programı çalıştırmaktadır. Eğer biz derleme işleminin ardından bağlayıcı programın çalıştırılmasını istemiyorsak yine gcc ya da clang derleyicilerini -c seçeneği ile (only compile) çalıştırmalıyız.

Object Modül Dosyaları ve Bağlama İşlemi

Derleme sonucunda elde edilen object modül dosyalarının çeşitli biçimleri bulunmaktadır. Microsoft firması DOS sistemlerinde "OMF (Object Module Format)" denilen bir object modül formatı kullanıyordu. Microsoft Windows'un 3.X versiyonlarında da bu formatı kullanmaya devam etti. Fakat Windows'u 32 bit bir işletim sistemi haline getirdiğinde object modül formatını da değiştirdi. Microsoft'un 32 bit ve 64 bit Windows sistemlerinde kullandığı object modül formatına "COFF (Common Object File Format)" denilmektedir. COFF formatı bu sistemlerdeki çalıştırılabilen dosya

formatı olan PE (Portable Executable) formatına çok benzemektedir. Linux sistemlerinde object modül formatı olarak ağırlıklı biçimde "ELF (Executable and Linkable Format)" formatı kullanılmaktadır. İsminden de anlaşılacağı gibi ELF hem bir object modül formatı hem de çalıştırılabilir dosya formatıdır.

Peki object modül dosyalarının içinde ne vardır? İşte object modül dosyaları derlenmiş olan programların makine kodlarının yanı sıra bağlayıcı programların gereksinim duyduğu çeşitli bağlama bilgilerini de içermektedir. Örneğin bir C dosyası içerisinde (translation unit) o C dosyası içerisinde olmayan bir global değişkeni ve bir fonksiyonu kullanmış olalım:

```
/* sample.c */  
  
extern int g_a;  
void bar(void);  
  
void foo(void)  
{  
    g_a = 10;  
  
    bar();  
}
```

Bu programı Windows sistemlerinde Microsoft derleyicileri ile derleyip object modül haline getirelim:

```
cl /c sample.c
```

Şimdi dumpbin programı ile object modülü inceleyelim:

```
dumpbin /ALL sample.obj > test.txt
```

Şöyle bir çıktı elde edilmiştir:

```
Microsoft (R) COFF/PE Dumper Version 14.29.30133.0  
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Dump of file sample.obj
```

```
File Type: COFF OBJECT
```

```
FILE HEADER VALUES
```

```
    14C machine (x86)  
    4 number of sections  
62160D73 time date stamp Wed Feb 23 13:33:23 2022  
    1BB file pointer to symbol table  
    E number of symbols  
    0 size of optional header  
    0 characteristics
```

```
SECTION HEADER #1
```

```
.drectve name  
    0 physical address  
    0 virtual address  
    2F size of raw data  
    B4 file pointer to raw data (000000B4 to 000000E2)  
    0 file pointer to relocation table  
    0 file pointer to line numbers  
    0 number of relocations  
    0 number of line numbers  
    100A00 flags  
        Info  
        Remove  
        1 byte align
```

```
RAW DATA #1
```

```
00000000: 20 20 20 2F 44 45 46 41 55 4C 54 4C 49 42 3A 22    /DEFAULTLIB:"
```

```
00000010: 4C 49 42 43 4D 54 22 20 2F 44 45 46 41 55 4C 54 LIBCMT" /DEFAULT
00000020: 4C 49 42 3A 22 4F 4C 44 4E 41 4D 45 53 22 20 LIB:"OLDNAMES"
```

Linker Directives

```
-----
/DEFAULTLIB:LIBCMT
/DEFAULTLIB:OLDNAMES
```

SECTION HEADER #2

```
.debug$S name
  0 physical address
  0 virtual address
  90 size of raw data
  E3 file pointer to raw data (000000E3 to 00000172)
  0 file pointer to relocation table
  0 file pointer to line numbers
  0 number of relocations
  0 number of line numbers
42100040 flags
  Initialized Data
  Discardable
  1 byte align
  Read Only
```

RAW DATA #2

```
00000000: 04 00 00 00 F1 00 00 00 83 00 00 00 45 00 01 11 ....E...
00000010: 00 00 00 00 43 3A 5C 55 73 65 72 73 5C 61 73 6C ...C:\Users\asl
00000020: 61 6E 5C 44 72 6F 70 62 6F 78 5C 53 74 75 64 79 an\Dropbox\Study
00000030: 5C 57 69 6E 64 6F 77 73 2D 53 79 73 50 72 6F 67 \Windows-SysProg
00000040: 5C 53 61 6D 70 6C 65 5C 73 61 6D 70 6C 65 2E 6F \Sample\sample.o
00000050: 62 6A 00 3A 00 3C 11 00 22 00 00 07 00 13 00 1D bj.:.<..".....
00000060: 00 B5 75 00 00 13 00 1D 00 B5 75 00 00 4D 69 63 .u.....u..Mic
00000070: 72 6F 73 6F 66 74 20 28 52 29 20 4F 70 74 69 6D rosoft (R) Optim
00000080: 69 7A 69 6E 67 20 43 6F 6D 70 69 6C 65 72 00 00 izing Compiler..
```

SECTION HEADER #3

```
.text$mn name
  0 physical address
  0 virtual address
  14 size of raw data
  173 file pointer to raw data (00000173 to 00000186)
  187 file pointer to relocation table
  0 file pointer to line numbers
  2 number of relocations
  0 number of line numbers
60500020 flags
  Code
  16 byte align
  Execute Read
```

RAW DATA #3

```
00000000: 55 8B EC C7 05 00 00 00 00 0A 00 00 00 E8 00 00 U.....]
00000010: 00 00 5D C3 ..]
```

RELOCATIONS #3

Offset	Type	Applied To	Symbol Index	Symbol Name
00000005	DIR32	00000000	B	_g_a
0000000E	REL32	00000000	9	_bar

SECTION HEADER #4

```
.chks64 name
  0 physical address
  0 virtual address
  20 size of raw data
  19B file pointer to raw data (0000019B to 000001BA)
  0 file pointer to relocation table
  0 file pointer to line numbers
  0 number of relocations
```



```

0 number of line numbers
A00 flags
Info
Remove
(no align specified)

```

RAW DATA #4

```

00000000: 23 07 66 15 27 1A BF 1A D4 9A DA 21 BA 3C E5 1E #.f.'.?.?.?.!<?.
00000010: 36 68 66 95 5C 95 74 57 00 00 00 00 00 00 00 6hf.\.tW.....

```

COFF SYMBOL TABLE

```

000 010475B5 ABS notype Static | @comp.id
001 80010191 ABS notype Static | @feat.00
002 00000001 ABS notype Static | @vol.md
003 00000000 SECT1 notype Static | .directve
Section length 2F, #relocs 0, #linenums 0, checksum 0
005 00000000 SECT2 notype Static | .debug$S
Section length 90, #relocs 0, #linenums 0, checksum 0
007 00000000 SECT3 notype Static | .text$mn
Section length 14, #relocs 2, #linenums 0, checksum F992C02D
009 00000000 UNDEF notype () External | _bar
00A 00000000 SECT3 notype () External | _foo
00B 00000000 UNDEF notype External | _g_a
00C 00000000 SECT4 notype Static | .chks64
Section length 20, #relocs 0, #linenums 0, checksum 0

```

String Table Size = 0x0 bytes

Summary

```

20 .chks64
90 .debug$S
2F .directve
14 .text$mn

```

Burada foo fonksiyonun makine kodları şöyle oluşturulmuştur:

RAW DATA #3

```

00000000: 55 8B EC C7 05 00 00 00 00 0A 00 00 00 E8 00 00 U.?.?.?.?.?.?.?.?.?.
00000010: 00 00 5D C3 ..]

```

RELOCATIONS #3

Offset	Type	Applied To	Symbol Index	Symbol Name
00000005	DIR32	00000000	B	_g_a
0000000E	REL32	00000000	9	_bar

Derleyici kodu derlerken eğer kullanılan bir nesne ya da fonksiyon o derleme modülü (translation unit) içerisinde değilse o nesne ya da fonksiyonun adresini boş (yani 0 olarak) bırakmaktadır. Yukarıdaki örnekte foo fonksiyonunun çağrısı için derleyici E8 makine kodunu üretmiştir. Bu E8 makine kodu Intel işlemcilerinde CALL işlemi yapan makine kodudur. Ancak derleyici burada CALL makine komutuyla çağrılacak fonksiyonun adresini 00 00 00 00 biçiminde boş bırakmıştır. Yani derleyicinin foo çağrısı için ürettiği kod şöyledir:

```
CALL 00 00 00 00
```

Benzer biçimde derleyici g_a = 10 işlemi için kod üretirken g_a nesnesinin yerini bilmediği için onun adresini de boş (yani 0 olarak) bırakmıştır. Derleyicinin atama işlemi yapan MOV komutu için ürettiği kod C7 05 biçimindedir. Yani derleyici g_a = 10 işlemi için şöyle bir kod üretmiştir:

```
MOV [00 00 00 00], 10
```

Şimdi object modül içerisindeki makine kodlarını dumpbin ile sembolik olarak görüntüleyelim:

```
dumpbin /DISASM sample.obj
```

Şöyle bir çıktı elde edilmiştir:

Dump of file sample.obj

File Type: COFF OBJECT

```
_foo:
00000000: 55                push     ebp
00000001: 8B EC            mov     ebp,esp
00000003: C7 05 00 00 00 00 mov     dword ptr [ _g_a ],0Ah
          0A 00 00 00
0000000D: E8 00 00 00 00   call    _bar
00000012: 5D                pop     ebp
00000013: C3                ret
```

Summary

```
20 .chks64
90 .debug$S
2F .directve
14 .text$mn
```

Derleyicinin kullanılan bu nesnelerin adresini bilmediği için onların adreslerini boş bıraktığına yeniden dikkat ediniz. Pekiyi bu makine komutlarında boş bırakılan adresler kim tarafından ve nasıl doldurulmaktadır? İşte bu adreslerin düzeltilmesi işlemini bağlayıcı program yapmaktadır. Bağlayıcı program bağlama işlemi sırasında bu nesnelerin bulunduğu modülleri belirleyip onları çalıştırılabilen kodun içerisinde birleştirdikten sonra artık onların nihai adreslerini belirler ve bu adresleri kod içerisinde düzeltir. Görüldüğü gibi bağlayıcı program object modülleri birleştirmenin yanı sıra boş bırakılmış adresleri de gerçek değerleriyle doldurmaktadır. Pekiyi bağlayıcı program kodun nerelerini düzeltereğini nereden bilmektedir? İşte aslında kodun düzeltilecek yerlerini derleyici COFF formatı içerisinde "relocation tablosu" denilen bir tablonun içerisinde tek tek yazmaktadır. Bağlayıcı da düzeltereği yerleri bu tabloya bakarak anlamaktadır. Yukarıdaki şekilde relocation tablosundaki girişler ve düzeltilecek yerler ok işaretleriyle gösterilmiştir.

Statik Kütüphanelerin Windows Sistemlerinde Oluşturulması

Statik kütüphane dosyaları Windows'ta Microsoft'un "lib.exe" isimli yardımcı programıyla oluşturulur. lib.exe ile sıfırdan bir static kütüphane dosyası yaratıp içerisine eklemeleri şöyle yapabiliriz:

```
lib /OUT: MyStaticLib.lib A.obj B.obj C.obj
```

Statik bir kütüphanenin içerisindeki object modülleri /LIST seçeneği ile görüntüleyebiliriz. Örneğin:

```
lib /LIST MyStaticLib.lib
```

/REMOVE seçeneği ile bir object modülü kütüphaneden atabiliriz. Örneğin:

```
lib MyStaticLib.lib /REMOVE C.obj
```

Daha sonra kütüphaneye yeni object modüller ekleyebiliriz:

```
lib MyStaticLib.lib D.obj
```

Windows'ta komut satırından değil de Visual Studio IDE'si ile statik kütüphanelerin oluşturulması da oldukça kolaydır. Tek yapılacak şey proje yaratılırken (File/New/Project/Win32 Console Application) proje türü olarak "Application Settings"te "Static Library" seçeneğini kullanmaktır. Bu durumda build işlemi yaptığımızda projeye eklediğimiz .c

dosyaları derlenir, bunlar object modül haline dönüştürülür. Sonrada lib.exe programı çalıştırılarak .lib dosyası oluşturulur.

Static Kütüphanelerin UNIX/Linux Sistemlerinde Oluşturulması

UNIX/Linux sistemlerinde statik kütüphane dosyaları ile işlemler yapmak için "ar" (archive) isimli program kullanılmaktadır. (Yani UNIX/Linux sistemlerindeki "ar" programını Windows'taki lib.exe'nin karşılığı olarak düşünebiliriz.) ar programı ile bir statik kütüphane dosyasını yaratıp içerisine object modül ekleme işlemi şöyle yapılabilmektedir:

```
ar cr mystaticlib.a a.o b.o c.o
```

Burada "c (create)" seçeneği eğer kütüphane dosyası yoksa yaratmak için kullanılır. Eğer kütüphane dosyası zaten varsa "c" seçeneğinin bir etkisi olmaz. "r (replace)" seçeneği ise kütüphaneye bir object modül eklemek için kullanılır. Eğer aynı isimli object modül zaten varsa kütüphanedeki değiştirilir. Zaten var olan bir kütüphaneye object modül eklemek için "c" seçeneğini kullanmak zorunda değiliz. Örneğin:

```
ar r mystaticlib.a d.o
```

Kütüphanedeki object modülleri görüntülemek için ise "t (display table)" seçeneği kullanılmaktadır:

```
ar t mystaticlib.a
```

Kütüphane içerisindeki bir object modülü silmek için ise "d (delete)" seçeneği kullanılır. Örneğin:

```
ar d mystaticlib.a o.o
```

ar programının diğer pek çok seçeneği vardır. Bu seçenekleri ilgili dokümanlardan inceleyebilirsiniz.

Statik Kütüphanelerin Windows Sistemlerinde Kullanılması

Yukarıda da belirtildiği gibi statik kütüphane dosyalarına "bağlayıcı (linker)" bakmaktadır. Bağlayıcı object modülü inceleyerek derleyici tarafından bulunamayan fonksiyonları kütüphane dosyalarında arar. Eğer onları kütüphane dosyalarında bulursa oradan çekerek çalıştırılabilen (executable) dosyaya yazmaktadır. Ancak bağlayıcı programlar yalnızca standart C fonksiyonlarının bulunduğu kütüphane dosyalarına otomatik biçimde bakmaktadır. Programcı bağlayıcının kendi statik kütüphane dosyalarına da bakmasını istiyorsa bunu açıkça bağlama işleminde belirtmelidir.

Microsoft'un link.exe isimli bağlayıcı programında komut satırı argümanı olarak .lib dosyaları verilirse bu durumda link.exe bağlayıcı programı object modüllerde bulunamayan fonksiyonları burada belirtilen statik kütüphane dosyalarında da aramaktadır. Örneğin projemiz a.c ve b.c biçiminde iki kaynak dosyadan oluşuyor olsun. Bu kaynakdosyalar içerisinde de "mystaticlib.lib" isimli statik kütüphane dosyası içerisindeki fonksiyonlar çağrılıyor olsun. Bu durumda derleme ve bağlama işlemlerini şöyle yapabiliriz:

```
cl /c a.c  
cl /c b.c  
link /OUT:app.exe a.obj b.obj mystaticlib.lib
```

link.exe programında oluşturulacak çalıştırılabilen dosyanın ismi /OUT seçeneği ile belirlenebilmektedir. Eğer bu belirleme yapılmazsa oluşturulacak çalıştırılabilir dosya komut satırında belirtilen ilk dosyanın ismini alır.

Aslında bu işlemi tek satırda da yapabiliydik. Microsoft'un "cl.exe" derleyicisinde komut satırında .lib dosyaları belirtilirse cl.exe derleyicisi bunları "bağlayıcı (link.exe)" programı çalıştırırken ona komut satırı argümanı yapmaktadır. Böylece biz cl.exe ile komut satırında kendi kütüphanemize bakılacak biçimde derlemeyi aşağıdaki yapabiliriz:

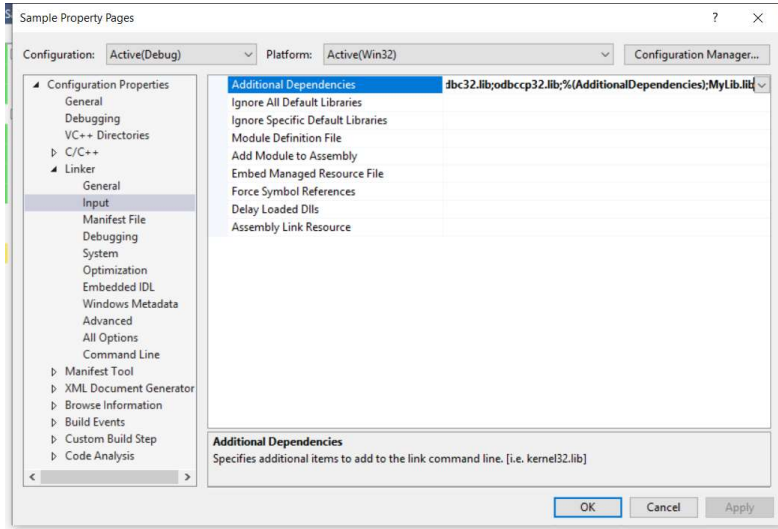
```
cl /Fe:app.exe a.c b.c mystaticlib.lib
```

Burada /Fe komut satırı seçeneği oluşturulacak çalıştırılabilir dosyanın ismini belirlemek için kullanılmaktadır. cl.exe derleyicisi burada belirtilen ismi link programını çağırırken /OUT seçeneğinde kullanılmaktadır. Yine eğer /Fe seçeneği ile çalıştırılabilir dosyanın ismi belirtilmezse çalıştırılabilir dosyanın ismi komut satırında ilk belirtilen dosya ismi olarak alınmaktadır.

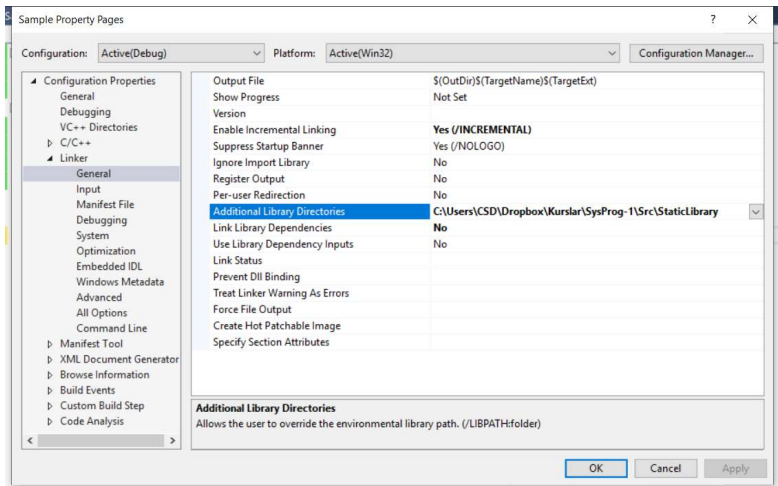
Burada bir noktaya dikkatinizi çekmek istiyoruz. Microsoft'un link.exe isimli bağlayıcı programı otomatik olarak standart C fonksiyonlarının ve API fonksiyonlarının bulunduğu kütüphaneleri de link aşamasına dahil etmektedir. Fakat programcı isterse /NODEFAULTLIB bu default özelliği kaldırabilir. Microsoft'un "link.exe" isimli bağlayıcısının daha pek çok komut satırı argüman seçeneği vardır. Bu seçenekleri dokümanlardan inceleyebilirsiniz.

Statik kütüphane kullanan bir programı komut satırından değil Visual Studio IDE'sini kullanarak da oluşturabiliriz. Bunun için şunların yapılması gerekmektedir:

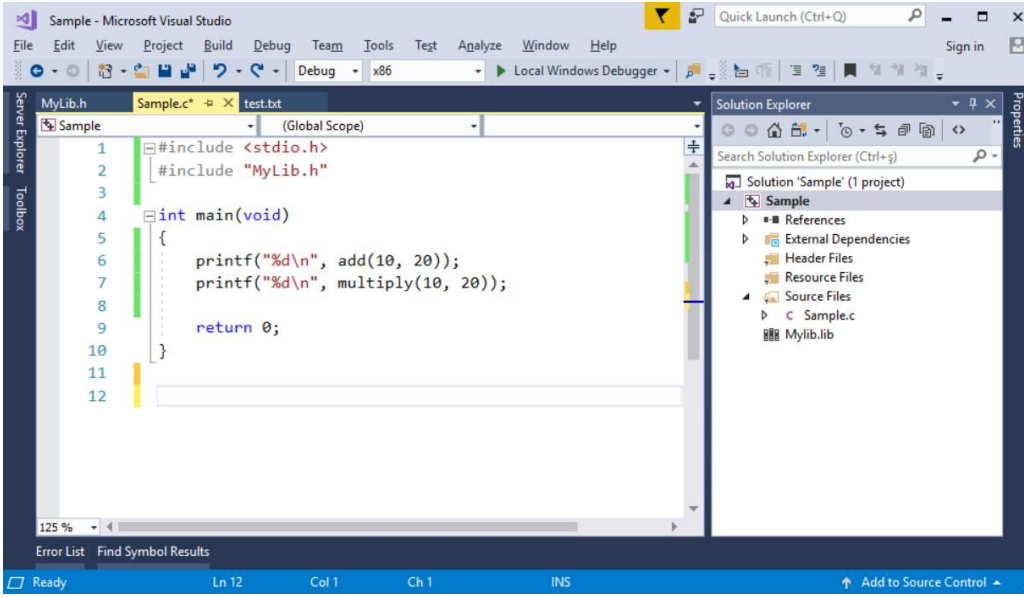
1) Önce kullanılacak kütüphane dosyasının ismi yol ifadesi olmadan proje özelliklerinden Linker/Input/Additional Dependencies sekmesine eklenir:



2) Daha sonra bu kütüphane dosyasının bulunduğu dizin Linker/General/Additional Library Directories sekmesine eklenir:



Visual Studio IDE'sinde bağlayıcının bir statik kütüphane dosyasına bakmasını sağlamanın diğer bir yolu da doğrudan statik kütüphane dosyasının projeye eklenmesidir. Örneğin:



Statik Kütüphanelerin UNIX/Linux Sistemlerinde Kullanılması

UNIX/Linux sistemlerindeki statik kütüphane kullanımı tamamen Windows sistemlerine benzemektedir. gcc derleyicisinde komut satırında ".a" uzantılı statik kütüphane dosyaları belirtilirse gcc bunları benzer biçimde "ld" bağlayıcısına girdi yapmaktadır. Örneğin:

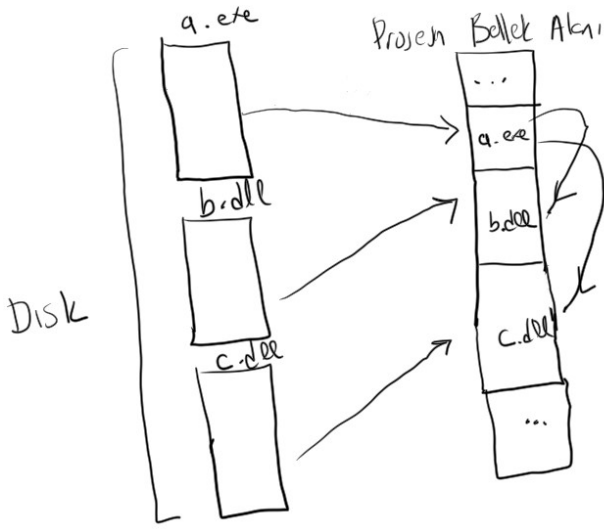
```
gcc -o app app.c mystaticlib.a
```

UNIX/Linux sistemlerinde statik kütüphane dosyaları kaynak dosya listesinin sonunda belirtilmelidir. Aksi takdirde onun sağındakiler için bu kütüphaneler link işlemine sokulmamaktadır.

Dinamik Kütüphaneler

Günümüzde dinamik kütüphaneler statik kütüphanelere göre oldukça yoğun bir biçimde kullanılmaktadır. Windows'ta dinamik kütüphane dosyalarının uzantıları .dll (dynamic link library), UNIX/Linux sistemlerinde ise .so (shared object) biçimindedir. Windows'taki dinamik kütüphanelerin yüklenmesi ve kullanıma hazır hale getirilmesi ile UNIX/Linux'taki dinamik kütüphanelerin yüklenmesi ve kullanıma hazır hale getirilmesi birbirlerinden biraz farklıdır. Fakat genel kullanımları fikir olarak birbirlerine çok benzemektedir.

Bir dinamik kütüphaneden bir fonksiyon çağırdığımızda bağlayıcı onu oradan alarak çalıştırılabilen dosyanın içerisine yazmaz. Bağlayıcı yalnızca çalıştırılabilen dosyanın içerisine o programın hangi dinamik kütüphanelerden hangi fonksiyonları kullandığı bilgisini yazar. Böylece bu fonksiyonlar çalıştırılabilen dosyanın içerisinde yer kaplamazlar. İşte işletim sisteminin yükleyicisi dinamik kütüphane kullanan bir programı yüklerken o programın hangi dinamik kütüphaneleri kullandığını belirler. Sonra çalıştırılabilen dosyayla birlikte dinamik kütüphanenin tamamını prosesin sanal bellek alanına yükler. Örneğin a.exe programı b.dll ve c.dll dosyalarını kullanıyor olsun. İşletim sistemi a.exe çalıştırılmak istendiğinde yalnızca o dosyayı değil b.dll ve c.dll dosyalarını da prosesin sanal bellek alanına yüklemektedir.



Artık program çalışırken dll çağırması yapıldığında akış bellekteki dll'in içerisine geçerek çalıştırılır. Tabii bu sistemde programı başka bir makineye taşımak istediğimizde yalnızca çalıştırılabilen programı değil onun kullandığı dinamik kütüphaneleri de o makineye taşımak gerekir.

Peki dinamik kütüphanelerin statik kütüphanelere göre avantajları ve dezavantajları nelerdir?

1) Dinamik kütüphane kullanan uygulamaları başka bir makineye taşırken o dinamik kütüphanelerin de uygun yerlere (ileride açıklanacak) kopyalanması gerekir. Yani dinamik kütüphane kullanan programların konuşlandırılması (deployment) daha zordur. Halbuki statik kütüphane kullanan programlar yalnızca çalıştırılabilen dosya biçimindedir. Onların hedef makineye kurulması yalnızca çalıştırılabilen dosyanın kopyalanması yoluyla yapılabilir.

2) Statik link işleminde her çalıştırılabilen dosya statik kütüphanelerdeki fonksiyonları tekrar tekrar bulundurmaktadır. Halbuki dinamik kütüphanelerden çağrılar yapıldığında bunlar çalıştırılabilen dosyaların içerisine yazılmazlar. Yani dinamik kütüphanelerin toplamda disk kullanımı konusunda avantajı vardır.

3) Dinamik kütüphane kullanan programlar toplamda sanal bellekte daha fazla yer kaplama eğilimindedir. Çünkü dinamik kütüphaneler bir bütün olarak (yani onların içerisindeki tek bir fonksiyonu çağırarak bile) belleğe yüklenmektedir. Fakat zaten dinamik kütüphanelerin kullanıldığı sistemlerin sanal bellek alanları genişler ve bu sistemlerin neredeyse hepsi sanal bellek kullanmaktadır. Dolayısıyla aslında dinamik kütüphaneler bütünsel olarak fiziksel belleğe yüklenmemektedir.

4) Aslında dinamik kütüphaneler pek çok durumda fiziksel belleğin daha etkin kullanımına yol açmaktadır. Çünkü aynı dinamik kütüphaneyi kullanan programlar söz konusu olduğunda aslında işletim sistemi bu dinamik kütüphaneyi fiziksel belleğe tekrara tekrar yüklemeyi. Dinamik kütüphanenin yalnızca bir kopyasını fiziksel belleğe yükler. (Tabii aslında bu kopyanın tamamı da fiziksel belleğe yüklenmek zorunda değildir.) İşletim sistemi proseslerin sayfa tablolarını organize ederek bunların fiziksel bellekteki aynı tek kopyayı kullanmasını sağlamaktadır. (Tabii proseslerden biri dinamik kütüphane içerisinde değişiklik yaptığında işletim sistemi "copy on write" işlemi ile ortak sayfayı ayıracaktır.) Örneğin hem Windows hem de UNIX/Linux sistemlerinde default durumda standart C fonksiyonları dinamik kütüphanelerden çağrılmaktadır. Pek çok program printf fonksiyonunu çağırdığında aslında bu dinamik kütüphanedeki printf kodları toplamda tek kopya olarak fiziksel belleğe yüklenmiş olacaktır.

5) Dinamik kütüphanelerin kullanıldığı sistemlerde uygulamadaki bazı değişiklikler yeniden link işlemi gerekmeden de yapılabilir. Örneğin bazı resimler, yazılar, fonksiyonlar bir dinamik kütüphanenin içerisindedir. Uygulamayı yazan kişi bu dinamik kütüphanenin yeni versiyonunu oluşturup onu hedef makineye taşıyabilir. Böylece program hiç yeniden derlenip link edilmeden değiştirilmiş olur.

Windows'ta Dinamik Kütüphanelerin Oluşturulması

Aslında dinamik kütüphanelerin oluşturulması bağlayıcı programlar tarafından yapılmaktadır. Windows'ta Microsoft'un "link.exe" bağlayıcısında /DLL komut satırı seçeneği hedef dosyanın .exe değil .dll yapılacağını belirtir. Ayrıca cl.exe derleyicisinde /LD seçeneği zaten "link.exe" bağlayıcısını /DLL seçeneğiyle çalıştırmaktadır. Yani biz doğrudan cl.exe ile derleme yaparken /LD seçeneği ile hedef dosyanın dll olmasını sağlayabiliriz. Örneğin:

```
cl /LD mydll.c
```

Biz bu işlemi derleyici ve bağlayıcıyı ayrı ayrı çalıştırarak da yapabiliriz:

```
cl /c mydll.c  
link /DLL mydll.obj
```

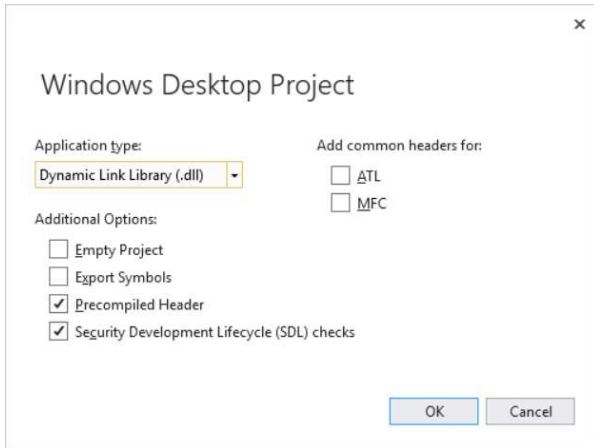
Tabii oluşturulacak hedef dosyanın ismi /Fe seçeneği ile değiştirilebilir. Örneğin:

```
cl Fe:test.dll /LD mydll.c
```

Ya da örneğin:

```
cl /c mydll.c  
link /Fe:test.dll mydll.obj
```

Visual Studio IDE'sinde New Project menüsünde "Windows Desktop Wizard" seçilir. Sonra çıkan diyalog penceresinde "Application Type" DLL olarak seçilir. Artık build yapıldığında IDE zaten cl.exe'yi /LD seçeneğiyle çalıştırır ve ürün olarak .dll dosyası elde edilecektir. (Tabii bir dll dosyasının IDE'den çalıştırılmak istenmesi normal bir durum değildir. Yalnızca build işlemi yapılmalıdır.)



Anahtar Notlar: Windows'ta doğal kod içeren dll'lerin oluşturulması ve kullanılması ile .NET ortamındaki dll'lerin oluşturulması ve kullanılması tamamen farklı biçimlerde yapılmaktadır. Her iki ortam için de hazırlanmış olan dinamik kütüphanelerin uzantıları .dll biçimindedir. Yani her iki ortamdaki dosyaların uzantılarının .dll olması bunların içeriğinin, oluşturulma ve kullanım biçimlerinin aynı olduğu anlamına gelmemektedir. .NET ortamındaki dll'lerin oluşturulması ve kullanılması doğal kod içeren DLL'lerin oluşturulması ve kullanılmasına göre çok daha kolaydır.

Windows'ta dinamik kütüphanelerin içerisindeki fonksiyonların dışarıdan (örneğin başka bir dll ya da exe dosyasından) çağrılabilmesi için fonksiyonun prototipinin ve/veya tanımlamasının önüne __declspec(dllexport) belirleyicisinin yerleştirilmesi gerekir. __declspec Microsoft derleyicilerine özgü bir eklentidir. __declspec(dllexport) belirleyicisi ilgili fonksiyonun adresini DLL dosyasının export tablosuna yazmaktadır. Örneğin:

```
#include <stdio.h>  
  
__declspec(dllexport) int Add(int a, int b)  
{  
    return a + b;  
}  
  
__declspec(dllexport) int Multiply(int a, int b)  
{
```



```
    return a * b;
}
```

DLL'in içerisindeki global değişkenler de dışarıdan kullanılabilirler. Ancak bunların da yine `__declspec(dllexport)` ile bildirilmesi gerekir. Örneğin:

```
__declspec(dllexport) int g_a;
```

Eğer C++'ta çalışıyorsak bir sınıfın yalnızca belirli bir üye fonksiyonunu export edebiliriz. Ya da sınıfın tüm üye fonksiyonlarını export edebiliriz. Sınıfın public, protected ya da private fonksiyonları export edilebilir. Erişim belirleyicisi ile export işleminin bir ilgisi yoktur. Örneğin:

```
class Sample {
    __declspec(dllexport) void Foo();
    //...
};

void Sample::Foo()
{
    //...
}
```

Bir sınıfın tüm üye fonksiyonlarını (private ve protected bölümlükiler de dahil olmak üzere) export edebilmek için `__declspec(dllexport)` belirleyicisi class anahtar sözcüğü ile sınıf isminin arasına getirilir. Örneğin:

```
class __declspec(dllexport) Sample {
    void Foo();
    void Bar();
    //...
};

void Sample::Foo()
{
    //...
}

void Sample::Bar()
{
    //...
}
```

Sınıfın static olmayan veri elemanlarının export edilmesinin bir anlamı yoktur. Çünkü onlar zaten aslında dll'in içerisinde değildir. Fakat sınıfın static veri elemanları dll'in içerisinde dir. Onlar da aynı biçimde export edilebilirler. Örneğin:

```
class Sample {
    __declspec(dllexport) static int ms_a;

    __declspec(dllexport) void Foo();
    __declspec(dllexport) void Bar();
    //...
};

int Sample::ms_a;

void Sample::Foo()
{
    //...
}

void Sample::Bar()
{
    //...
}
```


UNIX/Linux Sistemlerinde Dinamik Kütüphanelerin Oluşturulması

Dinamik kütüphanelerin sanal bellek alanının her hangi bir yerine yüklenebilmesi gerekir. Çünkü o anda sanal bellek alanının nerelerinin boş olduğu önceden bilinemez. İşte Windows sistemlerinde dinamik kütüphanelerin herhangi bir yere yüklenebilmesi için PE dosya formatında "relocation" bilgilerinin bulundurulması gerekmektedir. Böylece Windows'ta işletim sisteminin yükleyicisi dinamik kütüphaneyi uygun yere yükledikten sonra bu "relocation" tablosundan faydalanarak dinamik kütüphane kodları üzerinde gerekli olan değişiklikleri yapabilmektedir. Ancak UNIX/Linux sistemlerinde Windows sistemlerindeki gibi "relocation" tekniği kullanılmamaktadır. Bunun yerine UNIX/Linux sistemleri "Konumdan Bağımsız Kod (Position Independent Code)" denilen tekniği kullanır. Her iki tekniğin de avantajları ve dezavantajları vardır.

UNIX/Linux sistemlerinde dinamik kütüphane oluşturabilmek için öncelikle kodun derleyici tarafından "konumdan bağımsız (position independent)" bir biçimde oluşturulması gerekir. Çünkü bu sistemler Windows sistemlerinden farklı bir yükleme biçimine sahiptir. Konumdan bağımsız (position Independent) kodların anlamı "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır. gcc ya da g++ derleyicileri ile konumdan bağımsız derleme yapmak için komut satırında -fPIC seçeneğinin girilmesi gerekir. Ayrıca derleme işleminden sonra "ld" bağlayıcısı da -shared seçeneği ile çalıştırılmalıdır (Bu Microsoft'taki /DLL seçeneğine benzemektedir). Bu durumda bir dinamik kütüphaneye UNIX/Linux sistemlerinde şöyle oluşturulabilir.

```
gcc -c -fPIC foo.c bar.c
gcc -o libmy.so -shared foo.o bar.o
```

Mademki gcc'nin kendisi "ld" bağlayıcısını da çalıştırmaktadır, o halde -shared seçeneği gcc komut satırında da belirtilebilir. Bu durumda dinamik kütüphane dosyasının oluşturulması aşağıdaki gibi yapılabilmektedir:

```
gcc -o libmy.so -fPIC -shared foo.c bar.c
```

UNIX/Linux sistemlerinde konumdan bağımsız kod tekniği nedeniyle Microsoft sistemlerinde olduğu gibi fonksiyonu export etmeye kavramı yoktur. Yani biz dinamik kütüphanedeki tüm fonksiyonları ve statik dataları dışarıdan kullanabiliriz.

Windows Sistemlerinde Dinamik Kütüphanelerin Kullanılması

Windows'ta bir dinamik kütüphaneden çağırma yapan programı link ederken link aşamasında "o dinamik kütüphanenin import kütüphanesi" denilen bir .lib dosyasını kullanmamız gerekir. Yani biz Windows'ta bir dll oluştururken aslında bize ürün olarak iki dosya verilmektedir: Bir .dll dosyası ve bir de .lib dosyası. Asıl fonksiyon kodları .dll dosyası içerisinde. Buradaki .lib dosyasına "dll'in import kütüphanesi" denir. Bu .lib dosyasının içerisinde fonksiyonların kodları yoktur. O .dll içerisinde hangi fonksiyonların olduğu, onların sıra numaraları (ordinal number) ve dekore edilmiş ve edilmemiş isimleri vs. bulunur. Yani import kütüphanesi adeta dll'in bir dizin dosyası gibidir. O halde örneğin aşağıdaki gibi bir derleme işleminden biz iki dosya elde ederiz:

```
cl /LD mydll.c
```

Bu dosyalar mydll.dll ve mydll.lib dosyalarıdır.

Bir DLL'i kullanmak için kullanan programın link aşamasında DLL'in import kütüphanesini link işlemine dahil etmesi gerekir. Bu dahil etme işlemi tamamen statik kütüphanelerdeki gibidir. Yani statik kütüphaneler konusunda da belirtildiği gibi bu işlem iki biçimde yapılabilmektedir:

1) Import kütüphane dosyası projeye "Add Existing Item" seçeneği ile eklenirse, proje build edilirken bu dosya linker programına girdi olarak verilir.

2) Proje seçeneklerinden "Linker/Input" sekmesine gelinir. "Additional Dependencies" kısmına import kütüphanesinin yalnızca ismi (yol ifadesi olmadan) eklenir. Fakat onun bulunduğu dizin de "Linker/General/Additional Library Directories" kısmına girilir.

Ancak bir dinamik kütüphane kullanılırken dinamik kütüphane içerisindeki fonksiyonların prototipleri bulundurulmalı (tercihen bir başlık dosyasında) ve bu fonksiyonların prototiplerinin başına `__declspec(dllimport)` bildirimini yerleştirilmelidir. Örneğin:

```
__declspec(dllimport) int Add(int a, int b);
```

Aslında `__declspec(dllimport)` belirleyicisi DLL'deki fonksiyonları kullanırken mutlak anlamda gerekli değildir. Fakat bu belirleyicinin yerleştirilmesi daha etkin kod üretilmesine yol açmaktadır.

Anahtar Notlar: Eğer bu belirleyiciyi derleyici görmezse derleyici çağrı sırasında doğrudan CALL komutu kullanır. Bu durumda bağlayıcı bu komutları düzelterek onları dolaylı CALL komutu haline getirmektedir. İşte `__declspec(dllimport)` için başında derleyicinin daha etkin kod üretmesini sağlar.

Sonuç olarak DLL içerisindeki fonksiyonları dışarıdan kullanabilmek için `__declspec(dllexport)`, dışardan kullanırken `__declspec(dllimport)` belirleyicisinin kullanılması gerekir. Bu işlemi kolaylaştırmak için şöyle bir yol tavsiye edilmektedir: DLL'i hazırlayan kişi dışarıdan kullanılacak (export edilecek) fonksiyonların prototiplerini bir başlık dosyasında toplar. Ancak burada `#ifdef` işlemiyle aşağıdaki gibi bir düzenleme yapar:

```
#ifndef MYDLL_H_
#define MYDLL_H_

#ifdef DLLEXPORT
#define DLLSPEC __declspec(dllexport)
#else
#define DLLSPEC __declspec(dllimport)
#endif

/* Function Prototypes */

DLLSPEC int Add(int a, int b);
DLLSPEC int Multiply(int a, int b);

/* extern data Declarations */

DLLSPEC extern int g_a;

#endif
```

Burada görüldüğü gibi DLLEXPORT isimli bizim uydurduğumuz makro eğer daha önce define edilmişse yine bizim uydurduğumuz DLLSPEC yerine önişlemci `__declspec(dllexport)` belirleyicisini yerleştirecektir. Eğer bu makro daha önce define edilmemişse bu sefer DLLEXPORT yerine önişlemci `__declspec(dllimport)` belirleyicisini yerleştirir. Bu durumda bu başlık dosyası hem DLL oluşturulurken hem de kullanılırken include edilir. Örneğin bu durumda DLL'i oluşturan kod şöyle olacaktır:

```
#define DLLEXPORT

#include <stdio.h>
#include "MyDll.h"

DLLSPEC int g_a;

DLLSPEC int Add(int a, int b)
{
    return a + b;
}

DLLSPEC int Multiply(int a, int b)
{
```

```
    return a * b;
}
```

DLL'i kullanan kod da şöyle olacaktır:

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

int main(void)
{
    int result;

    result = Add(10, 20);
    printf("%d\n", result);

    result = Multiply(10, 20);
    printf("%d\n", result);

    return 0;
}
```

Tabii DLL'i kullanan kodda import kütüphanesinin link aşamasına dahil edilmesi gerekir.

Görüldüğü gibi bir DLL oluşturulurken o DLL'i kullanacak kişi üç dosyaya gereksinim duymaktadır:

- 1) DLL dosyasının kendisine (örneğimizdeki MyDll.dll)
- 2) DLL'in import kütüphanesine (örneğimizdeki MyDll.lib)
- 3) DLL içerisindeki fonksiyonların ya da sınıfların bildirimlerinin `__declspec(dllimport)` ile yapıldığı bir başlık dosyasına. (örneğimizdeki MyDll.h)

Dinamik kütüphane kullanan bir program çalıştırılırken kullandığı DLL'in de o makinede bulunuyor olması gerekir. Fakat kullanılan DLL hangi dizinde olmalıdır? İşte Windows sırasıyla DLL'leri bazı dizinlerde aramaktadır. Ancak bu konuda Windows'un versiyonları arasında küçük bazı farklılıklar vardır. Özellikle bu farklılık 32 bit Windows sistemleriyle 64 bit Windows sistemleri arasında daha belirgindir. Burada bazı küçük ayrıntıları göz ardı ederek bu arama dizinlerini sırasıyla aşağıda belirtiyoruz:

- 1) Programın .exe dosyasının bulunduğu dizin
- 2) Windows'un altındaki System32 dizini (bu dizin GetSystemDirectory API fonksiyonuyla elde edilebilir.)
- 3) Windows'un altındaki System dizini.
- 4) Windows'un dizinini kendisi (bu dizin GetWindowsDirectory API fonksiyonuyla elde edilebilir)
- 5) CreateProcess fonksiyonunu uygulayan prosesin (yani exe'yi çalıştıran programın. Bu program Windows'un komut satırı ya da masaüstü (explorer.exe) de olabilir.) çalışma dizini.
- 6) CreateProcess fonksiyonunu uygulayan prosesin PATH çevre değişkeni ile belirlediği dizinler sırasıyla

Windows'ta DLL'lerin yerleştirildiği iki tipik yer vardır: Birincisi geleneksel olarak Windows'un System32 dizinidir. Kurulum programları paylaşılan (yani çok proses tarafından kullanılan DLL'leri buraya çekebilir. Ancak bu yöntem gittikçe daha az tercih edilir olmuştur. Bakınız: "Dll Cehennemi") Microsoft uzunca süredir her uygulamanın kendi DLL'lerini .exe dosyasının bulunduğu kurulum dizinine çekmesini tavsiye etmektedir.

UNIX/Linux Sistemlerinde Dinamik Kütüphanelerin Kullanılması

UNIX/Linux sistemlerinde Windows sistemlerinde olduğu gibi dinamik kütüphanelerin import kütüphaneleri yoktur. Bu sistemlerde dinamik kütüphaneler kullanılırken bizzat dinamik kütüphane dosyasının (yani .so dosyasının) kendisi link işlemine sokulur. (Yani adeta UNIX/Linux sistemlerinde .so dosyasının kendisinin aynı zamanda sanki Windows sistemlerindeki import kütüphanesinin yerine de geçtiği düşünülebilir.) Böylece bağlayıcı hangi fonksiyonların dinamik kütüphane içerisinde olduğunu buradan anlamaktadır. Örneğin dinamik kütüphane aşağıdaki dosyadan oluşuyor olsun:

```
/* foo.c */
```

```
#include <stdio.h>

void foo(void)
{
    printf("foo\n");
}
```

```
/* bar.c */
```

```
#include <stdio.h>

void bar(void)
{
    printf("bar\n");
}
```

Dinamik kütüphanenin derlenmesi şöyle yapılmalıdır:

```
gcc -fPIC -shared -o libmy.so foo.c bar.c
```

Buradaki libmy.so başlık dosyası aşağıdaki olabilir:

```
/* libmy.h */

#ifndef LIBMY_H_
#define LIBMY_H_

/* Function Prototypes */

void foo(void);
void bar(void);

#endif
```

Dinamik kütüphaneyi kullanan program da şöyle olabilir:

```
/* app.c */

#include <stdio.h>
#include "libmy.h"

int main(void)
{
    foo();
    bar();

    return 0;
}
```

Dinamik kütüphaneyi kullanan program da şöyle derlenebilir:

```
gcc -o app app.c libmy.so
```

Peki UNIX/Linux sistemlerinde dinamik kütüphaneyi kullanan programı çalıştırırken dinamik kütüphane dosyası (yani .so dosyası) nerede bulunmalıdır? İşte tıpkı Windows sistemlerinde olduğu gibi UNIX/Linux sistemlerinde de dinamik kütüphaneler yükleyici tarafından sırasıyla bazı dizinlerde aranır. Arama prosedürü özetle şöyledir:

- 1) Dinamik kütüphane dosyasının ELF formatı içerisindeki DR_RPATH dizinine bakılır.
- 2) exec işlemi yapan prosesin LD_LIBRARY_PATH isimli çevre değişkeninde belirtilen dizinlere sırasıyla bakılır.
- 3) /etc/ld.so.cache dosyasında belirtilen dosyalara bakılır.

- 4) /lib dizinine bakılır. (Bazı 64 bit sistemlerde /lib64 dizinine de bakılmaktadır.)
5) /usr/lib dizinine bakılır. (Bazı 64 bit sistemlerde usr/lib64 dizinine de bakılmaktadır.)

Burada eğer dinamik kütüphane paylaşılacaksa onun /usr/lib ya da /lib içerisine çekilmesi daha uygun olur. /lib dizini çok daha temel kütüphaneleri, /usr/lib dizini ise daha çok uygulama programlarının kullandığı kütüphaneleri barındırmaktadır. Eğer dinamik kütüphane paylaşılmayacaksa LD_LIBRARY_PATH çevre değişkenini ayarlamak daha iyi bir yoldur. Örneğin:

```
export LD_LIBRARY_PATH=.  
./app
```

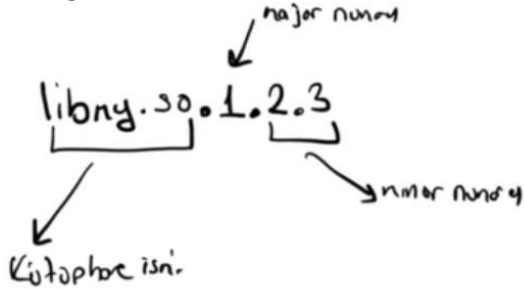
UNIX/Linux Sistemlerinde Dinamik Kütüphanelerin Kullanımına İlişkin Ayrıntılar

UNIX/Linux sistemlerinde Windows'taki DLL cehennemi (DLL Hell) denilen dinamik kütüphanelerin versiyon karmaşası değişik biçimde çözülmüştür. Dinamik kütüphanelerin değişik versiyonları zamanla oluşturulabilmektedir. Bu yeni versiyonlar çoğu kez eski versiyonlarla uyumlu olma iddiasındadır. Ancak bu uyum istense bile her zaman mükemmel biçimde sağlanamayabilmektedir. (Hatta bazen dinamik kütüphanenin eski versiyonunda bazı böcekler bulunabilir, yeni versiyonda bu böcekler düzeltilmiş olabilir. Eski versiyonundaki böceklerden kaçınmak için yazılmış olan (workaround) kodlar düzeltilmiş yeni versiyonda uygun biçimde çalışmayabilir.) Bu nedenle dinamik kütüphanelerin eski ve yeni versiyonları genellikle aynı makinede bulundurulmakta, böylece farklı programlar bu dinamik kütüphanelerin farklı versiyonlarını kullanabilmektedir.

Dinamik kütüphanelerin versiyonlanması UNIX/Linux sisteminde majör ve minör numaralarla yapılmaktadır. Dinamik kütüphanelerin majör ve minör numaraları dosya isminin sonuna eklenmektedir. Majör numara tek basamaktır. Minör numara iki basamaktır. Majör numara ve minör numaranın iki basamağı '.' karakteri ile birbirinden ayrılmaktadır. Böylece versiyonlanmış kütüphane isminin genel biçimi şöyledir:

isim.so.majör.minör1.minör2

Örneğin:



Majör numara genellikle büyük değişikliklerin yapıldığı ve geçmişe doğru uyumun sorunlu ya da mümkün olmadığı versiyonları belirtir. Minör numaralar ise daha küçük değişikliklerin yapıldığı versiyonları belirtmektedir. Böylece zamanla dinamik kütüphanelerin yeni minörlü ve majörlü versiyonları ortaya çıkmaktadır. Örneğin aşağıdaki üç versiyon bir arada bulunabilmektedir:

```
libny.so.1.0.0  
libny.so.1.0.1  
libny.so.1.1.0
```

Versiyonlamadan amaç farklı versiyonları bulunan kütüphanelerin belli versiyonlarının istenildiğinde ya da default durumda kullanılmasını sağlamaktır.

Bir dinamik kütüphane dosyası içingenellikle üç isim bulundurulmaktadır:

- 1) Dinamik kütüphanenin gerçek ismi
- 2) Dinamik kütüphanenin so ismi

3) Dinamik kütüphanenin bağlayıcı (linker) ismi

Yukarıda açıkladığımız majör ve minör numaralarla belirtilmiş olan isme "gerçek isim (real name)" denilmektedir. "so ismi (so name)" yalnızca majör numara içerisinde isimdir. Dolayısıyla so isminin genel biçimi şöyledir:

isim.so.majör

so ismi bir grup minör numara için ortak bir isimdir. Örneğin aşağıdaki minör numaralara sahip değişik versiyonlar bulunuyor olsun:

```
libmy.so.1.0.0
libmy.so.1.0.1
libmy.so.1.1.0
```

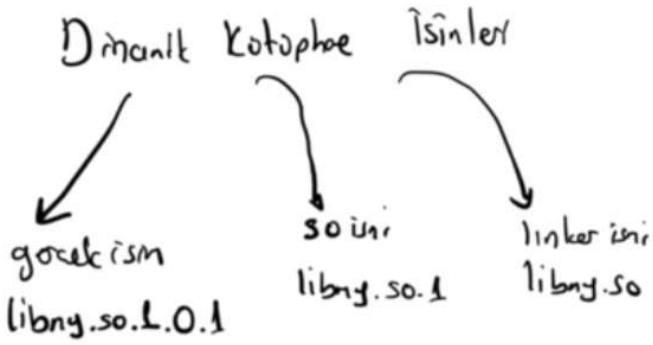
Bunların ortak so ismi şöyledir:

libmy.so.1

Dinamik kütüphanenin "bağlayıcı ismi (linker name)" ise hiçbir majör ve minör numara içermeyen isimdir. Genel biçimi şöyledir:

isim.so

Örneğin:



Peki bu üç isim ne işe yaramaktadır? Aslında dinamik kütüphane dosyası bir tanedir ve o da gerçek isme sahip olan dosyadır. so ismi ve linker ismi aslında sembolik link dosyalarıdır. so ismi gerçek dosya ismine, linker ismi de so ismine sembolik link yapılmıştır:

linker ismi -> so ismi -> gerçek dosya ismi

Örneğin:

```
gcc -o libmy.so.1.0.1 -fPIC -shared foo.c bar.c
```

Biz burada gerçek isimli dinamik kütüphaneyi oluşturmuş olduk. Şimdi so ismi için bu dosyaya işaret eden bir sembolik link dosyası oluşturalım:

```
ln -s libmy.so.1.0.1 libmy.so.1
```

Şimdi de linker ismi için so ismine bir sembolik link oluşturalım:

```
ln -s libmy.so.1 libmy.so
```

mylib.so → mylib.so.1 → mylib.so.1.0.0

Aşağıda oluşturulan dosyaları görüyorsunuz:

```
lrwxrwxrwx 1 csd study 10 Kas 9 11:15 libmy.so -> libmy.so.1
lrwxrwxrwx 1 csd study 14 Kas 9 11:13 libmy.so.1 -> libmy.so.1.0.1
-rwxr-xr-x 1 csd study 8168 Kas 9 11:12 libmy.so.1.0.1
```

Peki gerçek ismin, so isminin ve bağlayıcı isminin anlamı nedir? so isminin amacı bir dinamik kütüphaneyi kullanan programın o andaki dinamik kütüphanenin en son minör versiyonunu default biçimde kullanmasını sağlamaktır. Yani amaç dinamik kütüphaneyi kullanan programda hiçbir değişiklik yapmadan onun istenilen son minör versiyonu kullanmasını sağlamaktır. Bağlayıcı isminden amaç ise bağlama aşamasında bağlama işlemi yapanın kafasının karışmaması için sadece bir ismin kullanılmasıdır. Anımsanacağı gibi statik ve dinamik kütüphane dosyalarının ismi "lib" ile başlatılıyorsa biz bunu gcc ve clang derleyicilerinde ve bağlayıcılarında -l seçeneği ile kullanabilmekteyiz. Yani örneğin:

```
gcc -o app app.c -lmy
```

işlemi ile:

```
gcc -o app app.c libmy.so
```

işlemi benzer anlamdadır fakat aynı anlamda değildir. Çünkü -l seçeneği ile kısa isimli belirlemede bağlayıcı kütüphane dosyasını /lib, /usr/lib gibi bazı özel yerlerde arar. Eğer bağlayıcının istediğimiz bir dizine de bakmasını istiyorsak -L seçeneği ile dizin belirtebiliriz. Örneğin:

```
gcc -o app app.c -lmy -L.
```

Şimdi artık linker bulunulan dizine de bakacaktır. Halbuki dinamik kütüphane ismini aşağıdaki gibi vermiş olalım:

```
gcc -o app app.c libmy.so
```

Burada linker libmy.so yol ifadesi göreliliği için yalnızca bulunulan dizinde onu arayacaktır.

İşte kütüphanelerin linker isimleri link işlemi için pratik yazım amaçlı bulundurulmaktadır. Biz aşağıdaki gibi derleme ve bağlama işlemi yapmış olalım:

```
gcc -o app app.c -lmy
```

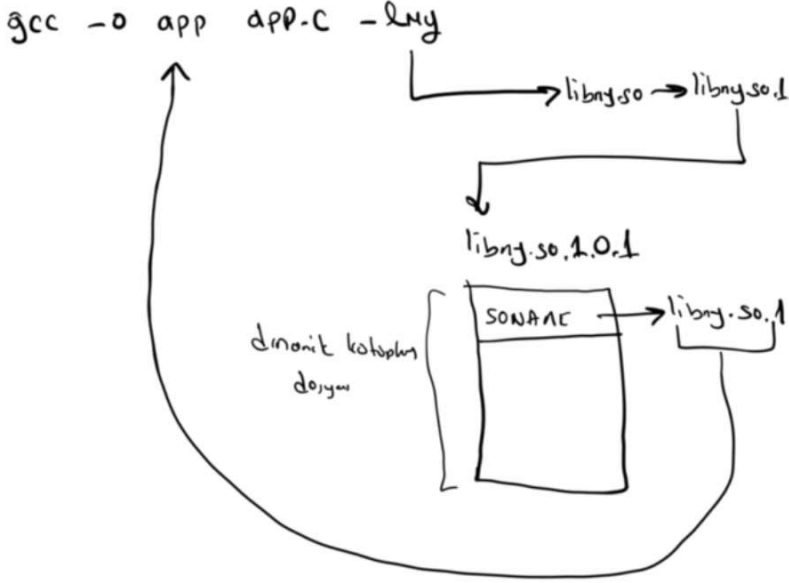
Burada derleyici sistemi ilgili dizinde libmy.so dosyasını arar. Bunu bir sembolik link dosyası biçiminde bulur. Bu dosya da so ismine link vermiştir. Linki izler. so ismi de gerçek isme link vermiştir. Bu durumda aslında yukarıdaki basit biçim sonuçta gerçek dinamik kütüphaneye varılmasını sağlayacaktır.

Bağlayıcı ismi link aşamasında kolay bir isim kullanmaya olanak sağlamaktadır. Peki so isminin amacı nedir? Neden bağlayıcı ismi doğrudan gerçek isimli dosyaya sembolik link yapılmamıştır da önce so ismine oradan gerçek isme sembolik link yapılmıştır? İşte bunun nedeni de programın minör versiyonlardan etkilenmemesini sağlamaktır. Yani biz kütüphanenin belli bir majör versiyonunu hedefleriz. Ancak bu sembolik link daha sonra değiştirilerek başka minör versiyonlara referans edebilir. Örneğin app isimli program libmy.so.1 versiyonunu hedeflemiş olsun. Hali hazırda bu versiyon libmy.so.0.1 gerçek versiyona referans etmektedir. Ancak bir süre sonra kütüphanenin libmy.so.1.0.2 gerçek isimli yeni bir minör versiyonu oluşturulmuş olsun. Şimdi bizim programımız halen libmy.so.1 dosyasına referans etmektedir. İşte biz de komut satırından bu libmy.so.1 sembolik linkine artık yeni minör versiyon olan libmy.so.1.0.2'ye referans edecek duruma getirirsek hiçbir şey yapmamıza gerek kalmadan eskiden link ettiğimiz app programı artık yeni minör versiyonu kullanır duruma gelecektir.

Peki çalıştırılabilen programın içerisine so ismi yerleştirilmektedir? Çünkü program aşağıdaki gibi derlenip link edildiğine göre bu programın içerisinde (yani ELF başlığında) gerçek ismin bulunması gerekmez mi?

```
gcc -o app app.c -lmy
```

Bu durumda gerçekten aslında app isimli çalıştırılabilen dosyanın ELF başlığında bu dosyanın libmy.so.1.0.1 dosyasını kullandığı (yani gerçek dosyayı kullandığı) bilgisi vardır. Halbuki anlatımımızda çalıştırılabilen dosyanın referans ettiği kütüphane dosyasının so isimli dosya olduğunu söylemiştir. İşte bu durum şöyle çözülmüştür: Aslında gerçek kütüphane oluşturulurken buna bir so ismi iliştilirilebilir. Bu isimde gerçek kütüphanenin başlık kısmında özel bir bölüme yerleştirilmektedir. Sonra bu gerçek kütüphaneye bağlama aşamasında referans edildiğinde bağlayıcı aslında çalıştırılabilen programa bu gerçek kütüphaneyi değil onun ELF başlığına gömülmüş olan so ismini çalıştırılabilen dosyaya yerleştirir. Böylece her ne kadar çalıştırılabilen program gerçek dosyaya referans edilerek oluşturulmuşsa da aslında onun içerisinde referans edilen dinamik kütüphaneye so isimli kütüphane olur.



Peki so ismini gerçek kütüphane dosyasının ELF başlığına nasıl yazdırırız? İşte bunun için soname isimli bağlayıcı seçeneği kullanılmaktadır. Anımsanacağı gibi gcc ve g++ derleyicileri ve clang derleyicileri -Wl seçeneği ile belirtilen switch'leri linker'a aktarmaktadır. Bu durumda gerçek kütüphane dosyasının oluşturulması şöyle yapılmalıdır:

```
gcc -o libmy.so.1.0.1 -fPIC -shared -Wl,-soname,libmy.so.1 foo.c bar.c
```

Burada -Wl,-soname,libmy.so.1 virgüllü seçenekleri aslında linker'a "-soname libmy.so.1" seçeneklerini göndermek için kullanılmıştır. Artık burada bizim gerçek kütüphanemizin içerisinde libmy.so.1 dosya ismi bir so ismi olarak gömülmüştür. Dolayısıyla biz artık aşağıdaki gibi bu kütüphaneyi kullandığımızda:

```
gcc -o app app.c -lmy
```

app isimli çalıştırılabilen dosya libmy.so.1.0.1 yerine libmy.so.1 dosyasına referans edecektir.

UNIX/Linux sistemlerinde dinamik kütüphaneler için birkaç yararlı program da değişik gerekçelerle kullanılabilir. Bunların biri ldd isimli programdır. Bu program bir çalıştırılabilen dosyanın ya da dinamik kütüphane dosyasının hangi dinamik kütüphaneleri kullandığını görmek için kullanılır. Örneğin:

```
csd@csd-virtual-machine ~/Study/Havel-san-SysProg/dynamiclib $ ldd app
linux-vdso.so.1 => (0x00007ffe28df9000)
libmy.so.1 => ./libmy.so.1 (0x00007fb56fa49000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb56f657000)
/lib64/ld-linux-x86-64.so.2 (0x000055bc155e9000)
```

Burada ldd bize şu iki sorunun yanıtını vermektedir?

1) app programı hangi dinamik kütüphaneleri kullanmaktadır? Bu bilgi ldd tarafından ELF formatının ilgili bölümünden alınmaktadır.

2) Yükleyici bu dinamik kütüphaneyi bulacak mıdır? Bulacaksa hangi dizinde bulacaktır?

Dinamik Kütüphanelerin Dinamik Kütüphaneleri Kullanması Durumu

Gerek Windows sistemlerinde gerekse UNIX/Linux sistemlerinde bir dinamik kütüphane diğer bir dinamik kütüphaneyi kullanıyor olabilir. Örneğin Windows'ta biz B.c dosyasını derleyerek B.dll oluşturmak isteyelim. Ancak B.c dosyasında da A.dll'deki fonksiyonlardan faydalanmış olalım. O halde komut satırında B.c dosyasını şöyle derleriz:

```
c1 /LD B.c A.lib
```

Burada A.lib A.dll kütüphanesinin import kütüphanesidir. Benzer işlem UNIX/Linux sistemlerinde de benzer biçimde yapılır:

```
gcc -o b.so -fPIC -shared b.c a.so
```

Burada b.c dosyası dinamik kütüphane oluşturacak şekilde derlenmek istenmiştir. Fakat b.c içerisinde a.so'daki fonksiyonlar kullanılmıştır.

Bir dinamik kütüphanenin diğerini kullanması durumunda işletim sistemi yükleme sırasında söz konusu tüm dinamik kütüphaneleri özyinelemeli olarak yükler. Böylece bizim de programı konuştururken tüm bu dinamik kütüphaneleri çalıştırılabilen dosyayla birlikte hedef makineye taşımamız gerekir.

Dinamik Kütüphanelerin Dinamik Yüklmesi

Hem Windows sistemlerinde hem de UNIX/Linux sistemlerinde dinamik kütüphaneler programın çalışma zamanı sırasında dinamik biçimde yüklenebilmektedir. Daha önce gördüğümüz dinamik kütüphane kullanma biçimlerinde dinamik kütüphaneler için başında program dosyası belleğe yüklenirken işletim sisteminin yükleyicisi tarafından yükleniyordu. Pekiyi dinamik kütüphanelerin programın çalışma zamanı sırasında yüklenmesine neden gereksinim duyulabilmektedir? Bunun birkaç nedeni olabilir:

1) Normal dinamik kütüphane kullanımında programın kullandığı tüm dinamik kütüphaneler yükleme sırasında işletim sisteminin yükleyicisi tarafından yüklenmektedir. Bu da yükleme zamanını uzatabilmektedir. Halbuki bazı kütüphanelerin gereksinim duyulduğu zamanda yüklenmesi bu zamanı azaltabilir.

2) Dinamik kütüphaneler prosesin bellek alanında yer kaplamaktadır. Halbuki bazı dinamik kütüphanelerin içerisindeki fonksiyonlara nadiren gereksinim duyuluyor olabilir. (Ya da bazı koşullarda gereksinim duyuluyor olabilir). Bu tür dinamik kütüphanelerin dinamik olarak yüklenmesi daha iyi bir tekniktir. Örneğin IDE'lerde olduğu gibi plug-in ya da add-in içeren mimarilerde bu eklentileri yazan kişiler bunu bir DLL'de toplarlar ana yazılım gereksinim duyulduğunda bu DLL'leri yükleyebilmektedir.

Windows'ta dinamik kütüphanelerin programın çalışma zamanı sırasında yüklenmesi şöyle gerçekleştirilmektedir:

1) Önce LoadLibrary (ya da onun biraz daha geliştirilmiş biçimi olan LoadLibraryEx) API fonksiyonu ile dinamik kütüphane belleğe yüklenir.

```
HMODULE WINAPI LoadLibrary(  
    LPCTSTR lpFileName  
);
```

Fonksiyon yüklenecek dinamik kütüphane dosyasının yol ifadesini parametre olarak alır. (Eğer yol ifadesi '\\' ya da '/' içermiyorsa daha önce bahsedilen yerlerde sırasıyla arama yapılmaktadır.). Fonksiyon geri dönüş değeri olarak DLL'in belleğe yüklendiği adresi bize verir. Geri dönüş değerinin HMODULE türünden olması 16 bit Windows zamanlarından kalmıştır. HMODULE türü void * olarak typedef edilmiştir. Örneğin:

```
HMODULE hModule;
```

```
if ((hModule = LoadLibrary("MyDll.Dll")) == NULL)  
    ExitSys("LoadLibrary", EXIT_FAILURE);
```

Görüldüğü gibi dinamik kütüphanelerin dinamik yüklenmesi için "import kütüphanesi"ne gereksinim duyulmamaktadır.

2) Dinamik kütüphane yüklendikten sonra onun içerisindeki fonksiyonun ya da global bir nesnenin adresi GetProcAddress fonksiyonuyla elde edilir:

```
FARPROC WINAPI GetProcAddress(  
    HMODULE hModule,  
    LPCSTR lpProcName  
);
```

Fonksiyonun birinci parametresi bizden LoadLibrary fonksiyonundan elde edilen kütüphanenin belleğe yüklenme adresini ister. İkinci parametre adresini elde edeceğimiz fonksiyonun ya da global nesnenin ismidir. Burada ismi girilen fonksiyon ya da global nesnenin DLL tarafından export edilmiş olması (yani __declspec(dllexport) bildirimi ile derlenmiş olması) gerekmektedir. Çünkü GetProcAddress ilgili sembolü "DLL'in export tablosu" denilen bir yerde arar. Ancak export edilen semboller "export tablosu"nda bulunmaktadır. Ayrıca burada verilen isimlerin dekore edilmiş isimler olduğuna dikkat edilmelidir. Fonksiyonların ya da global değişkenlerin isimleri C derleyicileri tarafından değiştirilerek (dekore edilerek) amaç koda ve dolayısıyla DLL dosyasına yazılmaktadır. Bu isimlerin dekore edilmiş son halleri DUMPBIN programı ile şöyle elde edilebilir:

```
DUMPBIN /EXPORTS MyDll.dll
```

GetProcAddress fonksiyonu geri dönüş değeri olarak bize DLL içerisindeki fonksiyonun adresini vermektedir. Geri dönüş değerindeki FARPROC bir fonksiyon typedef türüdür. Programcının tür dönüştürmesi ile bu adresi uygun fonksiyon türüne dönüştürmesi gerekir. (Anımsanacağı gibi void * türü C'de ve C++'ta fonksiyon adreslerini tutamamaktadır.) FARPROC türü aşağıdaki gibi typedef edilmiştir (tabii bunun bizim için bir önemi yok):

```
typedef int(__stdcall *FARPROC)();
```

Örneğin:

```
typedef int(*PADD)(int, int);  
  
if ((padd = (PADD)GetProcAddress(hModule, "Add")) == NULL)  
    ExitSys("GetProcAddress", EXIT_FAILURE);
```

Benzer biçimde global değişkenlerin de adresleri GetProcAddress fonksiyonu ile elde edilebilir:

```
typedef int *GDATA;  
GDATA gd;  
...  
if ((gd = (GDATA)GetProcAddress(hModule, "g_a")) == NULL)  
    ExitSys("GetProcAddress", EXIT_FAILURE);  
printf("%d\n", *gd);
```

3) Artık fonksiyonlar ya da global değişkenler program içerisinden kullanılabilir. Örneğin:

```
result = padd(10, 20);  
printf("%d\n", result);
```

4) Kullanım bittikten sonra dinamik kütüphane FreeLibrary API fonksiyonuyla boşaltılabilir:

```
BOOL WINAPI FreeLibrary(  
    HMODULE hModule
```

```
);
```

Ayrıca biz DLL'i yükleme zamanında yüklenecek biçimde kullanıyor olsak bile onu yine istediğimiz zaman FreeLibrary API fonksiyonuyla bellekten boşaltabiliriz.

Fonksiyon parametre olarak DLL'in bellekteki yüklenme adresini almakatdır.

Windows için örnek kullanım programı şöyle olabilir:

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

void ExitSys(LPCSTR lpszMsg, int status);

typedef int(*PADD)(int, int);
typedef int(*PMUL)(int, int);
typedef int *GDATA;

int main(void)
{
    HMODULE hModule;
    PADD padd;
    PMUL pmul;
    GDATA gd;
    int result;

    if ((hModule = LoadLibrary("MyDll.Dll")) == NULL)
        ExitSys("LoadLibrary", EXIT_FAILURE);

    if ((padd = (PADD) GetProcAddress(hModule, "Add")) == NULL)
        ExitSys("GetProcAddress", EXIT_FAILURE);

    result = padd(10, 20);
    printf("%d\n", result);

    if ((pmul = (PMUL)GetProcAddress(hModule, "Multiply")) == NULL)
        ExitSys("GetProcAddress", EXIT_FAILURE);

    result = pmul(10, 20);
    printf("%d\n", result);

    if ((gd = (GDATA)GetProcAddress(hModule, "g_a")) == NULL)
        ExitSys("GetProcAddress", EXIT_FAILURE);
    printf("%d\n", *gd);

    FreeLibrary(hModule);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}
```

UNIX/Linux sistemlerinde de dinamik kütüphanelerin dinamik yüklenmesi Windows sistemlerindekiyle çok benzer biçimde yapılmaktadır. İşlemler sırasıyla şu adımlardan geçilerek yapılabilir:

1) Önce `dlopen` fonksiyonuyla dinamik kütüphane belleğe yüklenir (`dlopen` Windows'taki `LoadLibrary` fonksiyonuna benzetilebilir.)

```
#include <dlfcn.h>
```

```
void *dlopen(const char *filename, int flag);
```

Fonksiyonun birinci parametresi yüklenecek dosyanın yol ifadesini almaktadır. Eğer bu yol ifadesinde dosya ismi içerisinde hiç '/' karakteri geçmeden belirtilmişse fonksiyon onu yukarıda belirtildiği gibi bazı dizinlerde sırasıyla aramaktadır. Eğer dosya isminde en az bir '/' karakteri varsa bu durumda dosya belirtilen dizinde aranır. İkinci parametre yüklemeyle ilgili bazı ayrıntılar için düşünülmüştür. Burada bu ayrıntılar üzerinde durulmayacaktır. Bu parametre `RTLD_LAZY` olarak geçilebilir. (lazy load işlemi "UNIX/Linux Sistem Programlama" kurslarında ele alınmaktadır). Fonksiyonun geri dönüş değeri dinamik kütüphanenin yüklendiği adresi belirtir. Dinamik yükleme fonksiyonları `libdl` kütüphanesinin içerisinde olduğu için link işlemine `-ldl` seçeneğinin eklenmesi gerekir. Örneğin:

```
void *dl;  
  
if ((dl = dlopen("./mysharedlib.so", RTLD_LAZY)) == NULL) {  
    fprintf(stderr, "cannot load library!\n");  
    exit(EXIT_FAILURE);  
}
```

2) Dinamik kütüphane içerisindeki fonksiyonun ya da global değişkenin adresi `dlsym` fonksiyonuyla elde edilir. (Bu fonksiyonu Windows sistemlerindeki `GetProcAddress` fonksiyonuna benzetebiliriz.):

```
#include <dlfcn.h>
```

```
void *dlsym(void *handle, const char *symbol);
```

Fonksiyonun birinci parametresi dinamik kütüphanenin bellekteki yüklenme adresi, ikinci parametresi ilgili sembolün ismini belirtir. Buradaki isimler de derleyici tarafından dekore edilmiş isimler olabilir. İsimlerin dekore edilmiş biçimlerini `objdump`, `readelf` ya da `nm` isimli utility'ler ile elde edebiliriz. Örneğin:

```
readelf -s mysharedlib.so
```

ya da:

```
nm mysharedlib.so
```

Fonksiyonun başarı durumunda sembolün adresine, başarısızlık durumunda `NULL` adrese geri döner. Örneğin:

```
if ((padd = dlsym(dl, "add")) == NULL) {  
    fprintf(stderr, "cannot get address!..\n");  
    exit(EXIT_FAILURE);  
}
```

Anahtar Notlar: Aslında standartlar bağlamında bu fonksiyonun ve `GetProcAddress` fonksiyonlarının tasarımları yanlıştır. Zira C ve C++'ta fonksiyon adreslerinden data adreslerine, data adreslerinden de fonksiyon adreslerine tür dönüştürme operatörüyle bile dönüştürme tanımlanmamıştır.

3) Artık adresi elde edilmiş olan fonksiyon çağrılabilir. Örneğin:

```
result = padd(10, 20);  
printf("%d\n", result);
```

4) Nihayet dinamik kütüphane dlclose fonksiyonuyla kapatılır:

```
#include <dlfcn.h>
```

```
int dlclose(void *handle)
```

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

UNIX/Linux sistemlerinde örnek bir kullanım kodu şöyle olabilir:

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

typedef int(*PADD)(int, int);

int main(void)
{
    void *dl;
    PADD padd;
    int result;

    if ((dl = dlopen("./mysharedlib.so", RTLD_LAZY)) == NULL) {
        fprintf(stderr, "cannot load library!\n");
        exit(EXIT_FAILURE);
    }

    if ((padd = dlsym(dl, "add")) == NULL) {
        fprintf(stderr, "cannot get address!..\n");
        exit(EXIT_FAILURE);
    }

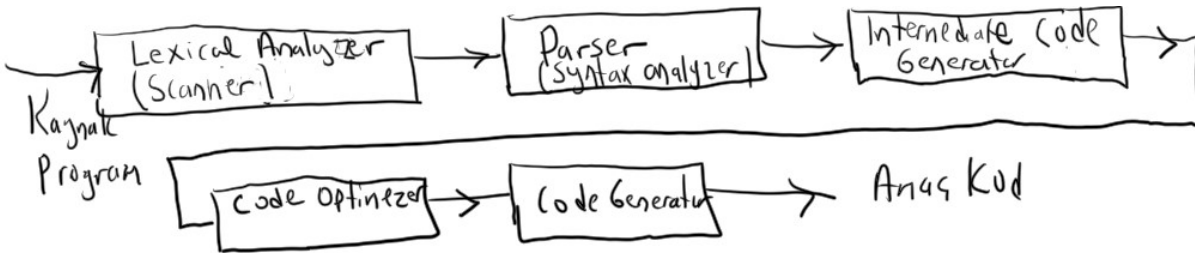
    result = padd(10, 20);
    printf("%d\n", result);

    dlclose(dl);

    return 0;
}
```

Derleyicilerin Çalışma Mekanizması

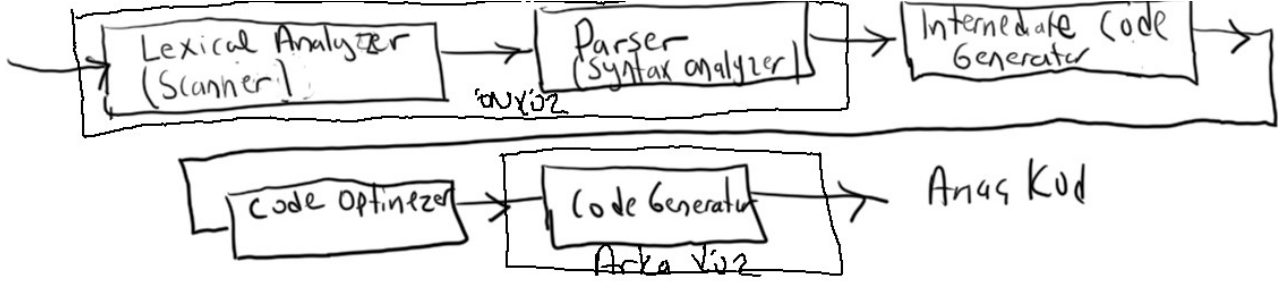
Bir derleyici tipik şu aşamalardan geçerek derleme işlemini yapar:



Kaynak kod derleyicinin "lexical analyzer" ya da "scanner" ya da "tokenizer" denilen modülü tarafından ele alınır. Bu modül kaynak kodu atomlarına ayırmaktadır. Daha sonra "parser" denilen modül devreye girer. Bu modül kaynak programı sentaks bakımından kontrol eder ve bir ağaç veri yapısına dönüştürür. Parser modül tarafından oluşturulan bu ağaç biçimindeki veri yapısına "parse tree" denilmektedir. "Parse tree" programın işlenebilir halidir. Yani artık program bir yazı olmaktan çıkmış bir veri yapısı olarak ifade edilmiştir. Daha sonra bu ağaç özyinelemeli biçimde dolaşılır ve ağaçtaki elemanlar için -gerçek makine komutları değil- onları temsil eden arakodlar (intermediate codes) üretilir. Bu modüle "intermediate code generator" denilmektedir. Bu aşamada gerçek makine komutlarının değil de arakodların üretilmesinin iki önemli nedeni vardır: Üretilen arakodlar gerçek makine kodlarına göre daha kolay işlenebilir biçimdedir.

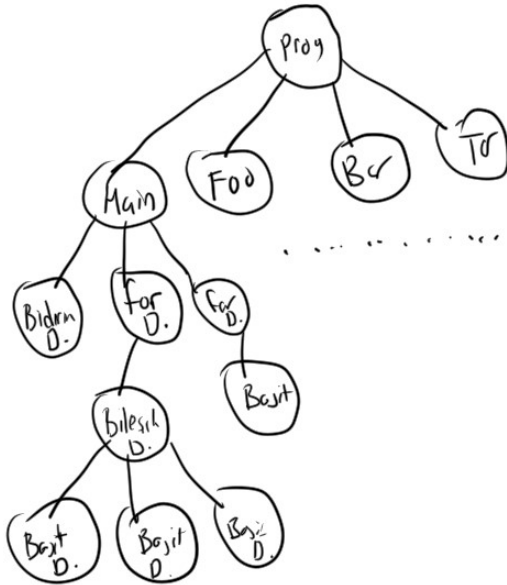
Dolayısıyla optimizasyon gibi işlemler için daha iyi bir girdi oluşturmaktadır. Ara kodlar aynı zamanda derleyiciyi yazanlar için standart bir birleşime noktası oluşturmaktadır. Bu sayede derleyicilerin port edilmesi kolaylaştırılmış olur. Ara kod üretiminden sonra üretilen bu arakodlar optimize edilmektedir. Derleyicilerin bu modellerine "code optimizer" denir. Nihayet optimize edilmiş kodlardan gerçek makine komutları üretilir. Bu modüle de "code generator" denilmektedir.

Derleyicilerin kaynak program üzerinde işlem yaptığı modüllere "ön yüz (frontend)", makine kodu ürettiği modüllere "arka yüz (back end)" ve diğer aradaki modüllere de "orta yüz (middle end)" denilmektedir. Ön yüz kaynak dile arka yüz hedef dile bağlıdır. Yani "ön yüz" yazımı kaynak dili bilmekle "arka yüz" yazımı da makine dilini bilmekle yapılabilecek bir faaliyettir.



Derleyicileri yazanlar önyüzlerin çıktılarını, parse tree çıktılarını vs. standart hale getirmeye çalışırlar. Böylece derleyicinin port edilmesi kolaylaşır. Şöyle ki: Bir firma düşünelim bu firma N tane dili M tane makine dili için derleyen derleyiciler yazmak istesin. Normalde bunun için kaç derleyici yazması gerekir? Yanıt: $N * M$ tane. Halbuki firma N tane dil için önyüz, M tane dil için de arka yüz yazsa toplamda $N + M$ tane faaliyetle bunları birleştirebilir. Örneğin biz bir dil tasarlamış olalım. Ancak bunun derleyicisi için gcc derleyicisinden faydalanmak isteyelim. Mademki gcc modül modül yazılmıştır o halde biz gcc için kendi dilimizi atomlarına ayıran ve gcc'nin parse tree modülüne veren bir önyüz yazabiliriz. Kod optimizasyonunu ve kod üretimini gcc'nin modülleri yapabilir. Bu faaliyete gcc için "frontend" yazma faaliyeti denilmektedir. Şimdi biz yeni bir işlemci için C derleyicisi yazmak isteyelim. Ve bunun için gcc derleyicisinden faydalanmak isteyelim. Bu durumda gcc'nin C frontend'i doğrudan kullanılabilir. Bunun için bizim gcc için bir "backend" yazmamız gerekir.

Peki "parse tree" nasıl bir ağaçtır? Bir program bütünden yalnız doğru özyinelemeli olarak parçalara ayrılabilir. Örneğin şöyle bir dil tasarlamış olalım: Dilimiz fonksiyonlardan oluşsun. Fonksiyonlar deyimlerden oluşsun. Dilimizde bildirim deyimi, bileşik deyim, basit deyim ve for deyimi olsun. Bu durumda yazdığımız program aşağıdaki bir ağaçla temsil edilebilir:



Görüldüğü gibi program bir metin biçiminden çıkartılıp işlenebilir bir ağaç yapısına dönüştürülmüştür. Sonra bu ağaç yapısı dolaşılıp ara kod üretilecek, sonra o kod optimize edilip gerçek kod üretilecektir. Parse tree derleyiciler için önemli veri yapılarından biridir.

Yorumlayıcıların (interpreters) temel aşamaları da derleyicilerle aynıdır. Ancak bunların arkayüzü yoktur. Çünkü yorumlayıcı hedef kod üretmeden programı doğrudan çalıştırır. Fakat yorumlayıcılar da kod optimizasyonu yapabilmektedir. Bazı yorumlayıcılar derleyicilerde olduğu gibi önce bir arakod üretir. Sonra bu arakodu adeta bir sanal makinede çalıştırır. Fakat bu arakodu programcı hiç görmeyebilir. Bazı yorumlayıcılar aynı zamanda istendiğinde hedef kod üretebilme yeteneğine de sahiptir.

Derleyicilerin Kod Optimizasyonları

Derleyiciler arasındaki en önemli kalite göstergelerinden biri "kod optimizasyonu"dur. Kod optimizasyonu "kodun işlevini değiştirmeden onu daha etkin biçimde çalıştırmak için yeniden düzenleme süreci"dir.

Anahtar Notlar: Aslında buradaki optimizasyon terimi yanlış uydurulmuştur. Optimizasyon eniyileme demektir. Halbuki derleyicilerin böyle bir iddiası yoktur. Derleyiciler kodları onların işlevleri değişmemek üzere yeniden düzenleyerek daha iyi hale getirirler. Bu süreç bir optimizasyon süreci değil iyileştirme (improvement) sürecidir. Fakat optimizasyon terimi bu yanlışlık bilinmesine rağmen yaygın olarak kabul görmüştür ve kullanılmaktadır.

Kod optimizasyonu ile ilgili önemli noktalar şunlardır:

- Optimizasyon kaynak kod üzerinde değil üretilen kod üzerinde yapılan bir faaliyettir. Yani optimizasyon ile bizim kaynak kod dosyamızda bir değişiklik olmaz.

- Optimizasyon sırasında kodda hiçbir anlam değişikliği olmaması gerekir. Yani optimize edilmemiş kod ile edilmiş kod tamamen aynı davranışı göstermelidir.

- Optimizasyon genel olarak "hız" ya da "kod büyüklüğü" temel alınarak yapılabilir. Fakat baskın ölçüt ve default durum "hız"optimizasyonudur. Hatta derleyiciler kodun daha hızlı çalışmasını sağlarken kodu büyütebilirler.

- Pek çok derleyicide optimizasyon çeşitli derleme seçenekleri (switch'ler) ile kontrol altına alınabilmektedir. Yani biz programcı olarak derleyicilere hangi tarz optimizasyonu yapması gerektiğini, hangilerini yapmaması gerektiğini söyleyebiliriz. Hatta programcı isterse optimizasyonu tamamen de kapatabilir. Bu durumda derleyici kodu optimize etmeye çalışmaz.

- Kod optimizasyonu C/C++ standartlarının ele aldığı bir konu değildir. Ancak standartlarda "derleyicinin programdaki semantik işlevin değişmemesi şartıyla kodu yeniden düzenleyerek derleyebileceği" belirtilmiştir.

- Derleyicilerin kod optimizasyonları halen tek akışlı ya da tek thread'li sistemlere göre yapılmaktadır. Yani derleyici kodu optimize ederken programda tek bir akış varmış gibi düşünür. Örneğin derleyici global bir değişken kullanılırken onun başka bir thread'te de kullanılıyor olduğuyla ilgilenmez. Dolayısıyla çok thread'li uygulamalarda programcı bu durumu göz önüne almalıdır. (Tabii çok thread'li uygulamalarda yalnızca global değişkenlerin kullanıldığı durumda böyle bir potansiyel tehlike oluşmaktadır. Bunlara da volatile gibi anahtar sözcüklerle ya da diğer bazı yöntemlerle önlem alınabilmektedir.)

Derleyicilerin Yaptığı Tipik Optimizasyonlar

Bu bölümde derleyicilerin tipik yaptığı kod optimizasyonları başlıklar halinde ele alınacaktır. Bu optimizasyonların sistem programcısı tarafından bilinmesi önemlidir. Çünkü böylece derleyicilerin hangi optimizasyonları bizim yapabildiği ve hangi maliyetlerle yapabildiğini anlayabilir. Böylece gerektiğinde bu optimizasyon sürecine katkıda bulunabilir ya da süreci daha iyi kontrol altına alabilir. Optimizasyonlar yerel (local), global (global) ve döngü optimizasyonları olarak da sınıflandırılabilir. Yerel optimizasyonlar kodun küçük bir bölümü üzerinde etkili olur. Global optimizasyonlar birden fazla fonksiyonu içine alacak biçimde geniş bölgeyi etkileyecek biçimde yapılır. Döngü optimizasyonları döngü içlerini optimize etmeyi hedefler. Burada optimizasyonlar karışık sırada ele alınacaktır.

1) Ortak Alt İfadelerin Elimine Edilmesi (Common Subexpression Elimination): Bir grupta işlemde ortak olan bazı alt ifadelerin her defasında yeniden yapılması yerine derleyici bunların sonuçlarını bir yazmaçta ya da geçici değişkende tutup onları kullanabilir. Örneğin:


```
a = x + y + 10;  
b = x + y + 20;  
c = x + y + 30;
```

Burada derleyici her defasında $x + y$ toplamını hesaplamak yerine bu toplamı geçici bir değişkende ya da yazmaçta tutarak oradan alıp kullanabilir. (Örneğin sınıfta Microsoft C derleyicisi ile optimizasyon kapalı ve açık durumda yapılan derlemede üretilen kodun sembolik makine dili çıktısına bakılmıştır. Ve derleyicinin optimizasyon sırasında $x + y$ toplamını ECX yazmacında tutarak oradan kullandığı görülmüştür. Fakat optimizasyon seçeceği kapatıldığında derleyici hep yeniden $x + y$ toplamını hesaplamıştır.)

2) Döngü içerisinde Değişmeyen İfadelerin Döngü Dışına Çıkartılması (Loop Invariant): Bu optimizasyon temasında döngü içerisindeki ifade döngü içerisinde kullanılmıyorsa döngünün dışına alınır. Örneğin:

```
for (i = 0; i < 100; ++i) {  
    total += i;  
    x = 10;  
}
```

Burada $x = 10$ işleminin döngünün çalışmasıyla bir ilgisi yoktur. Yani bu işlemin her yinelemede yeniden yapılmasına gerek yoktur. Derleyici kodu aşağıdaki gibi ele alabilir:

```
for (i = 0; i < 100; ++i) {  
    total += i;  
}  
x = 10;
```

Microsoft'un C derleyicisinde yukarıdaki kod analiz edildiğinde optimizasyon seçeneği kapalı olduğunda derleyicinin $x = 10$ ifadesini döngünün dışına çıkarmadığı açık olduğu durumda çıkardığı görülmüştür.

Fonksiyon çağrıları (hele ki derleyici fonksiyonun içeriğini göremiyorsa) döngü dışına çıkartılamaz. Örneğin:

```
for (i = 0; i < strlen(str); ++i) {  
    ...  
}
```

Burada programcı derleyicinin optimizasyon yaparak `strlen` fonksiyonunu bir kez çağırmasını beklememelidir. Derleyiciler standart C fonksiyonlarının ne yaptığını bilmediği için onlara normal bir fonksiyon muamelesi yaparlar. Bu nedenle derleyici `strlen` fonksiyonunu her defasında çağırarak zorundadır. Döngü içerisinde yapılan çağrıların döngü dışına çıkartılamayabileceğine dikkat ediniz. Yukarıdaki kodu programcı şöyle düzenlemeliydi:

```
len = strlen(str);  
for (i = 0; i < len; ++i) {  
    ...  
}
```

Peki derleyici döngü içerisindeki çağrıları hiç mi döngü dışına çıkartamaz? İşte eğer çağrı yapılan fonksiyon kodun içerisindeyse ve bir yan etkiye yol açmıyorsa derleyiciler onu döngü dışına çıkartabilmektedir.

Bazı C derleyicilerinde (örneğin Microsoft derleyicilerinde ve gcc derleyicilerinde) "intrinsic" fonksiyon kavramı da vardır. Intrinsic fonksiyonların ne yaptıkları derleyici tarafından bilinir. Böylece derleyici o çağrıları optimize edebilmektedir. Gerçekten de Microsoft derleyicilerinde ve gcc derleyicilerinde bazı standart C fonksiyonları "intrinsic" fonksiyonlardır. Bu "intrinsic" fonksiyonlar için prototip bildirimine de gerekmez. Tabii "intrinsic" fonksiyon kavramı C standartlarında yoktur. Bu nedenle taşınabilirlik bakımından bunların "intrinsic" olarak kullanılması taşınabilirliği bozabilir. Intrinsic fonksiyonlar için derleyiciler doğrudan `call` komutunu kaldırarak sanki onlar inline fonksiyonlarmış gibi kod da yerleştirebilmektedir.

3) Sabit İfadelerinin Derleme Aşamasında Ele Alınması (Constant Folding): Bilindiği gibi sabit ifadelerinin sayısal değerleri derleme aşamasında hesaplanabilmektedir. Örneğin:

```
a = 100 + 2 * 5 + b;
```

Böyle bir ifadedeki $100 + 2 * 5$ alt ifadesinin programın çalışma zamanına bırakılmasının bir anlamı yoktur. Bu ifade derleme aşamasında bir kez yapıp kod ona göre üretilebilir. Örneğin:

```
a = 110 + b;
```

Bu kod yukarıdakiyle aynı anlamdadır. Sabit ifadelerinin derleme aşamasında ele alınması çok temel bir optimizasyondur. Derleyicilerin optimizasyon seçenekleri kapalı olsa bile derleyiciler bu optimizasyonu default olarak yapabilmektedir.

4) Etkisiz Kodların Elimine Edilmesi (Dead Code Elimination): Derleyiciler etkisi olmayan kod parçalarını optimizasyon sırasında tamamen koda çıkartabilmektedir. Örneğin:

```
int main(void)
{
    int i;
    int total = 0;
    int result;

    for (i = 1; i <= 100; ++i)
        total += i;

    printf("Ok\n");

    return 0;
}
```

Burada aslında döngünün de total değişkeninin de result değişkeninin de koda bir etkisi yoktur. Yani başka bir deyişle program aşağıdaki gibi olsaydı da yazan için değişen bir şey olmazdı:

```
int main(void)
{
    printf("Ok\n");

    return 0;
}
```

Çünkü her ne kadar total değişkeni üzerinde bir toplama yapılmışsa da bu toplam başka bir yerde kullanılmamıştır. Benzer biçimde result değişkeni de bildirilmiştir fakat kullanılmamıştır. Tabii bir kodun etkisinin olmadığı iyi bir biçimde analiz edilmelidir. Bazı kodlar dolaylı yan etkilere sahip olabilmektedir. Şimdi aklınıza aşağıdaki gibi bir kodun etkisiz olup olmadığı sorusu gelebilir:

```
for (i = 0; i < 100; ++i)
    foo();
```

Bu kod optimizasyon seçenekleri açılmışsa derleyiciler tarafından "etkisiz kod" oldukları gerekçesiyle elimine edilebilirler. Halbuki programcı bu kodu zaman kaybı oluşturmak için yazmış olabilir.

5) Koşul İçeren Döngülerin Yeniden Düzenlenmesi (Loop Unswitching)

Bir döngü içerisinde döngüye bağlı olmayan bir koşul altında farkı işlemler yapılmak istenebilir. Bu durumda döngünün içerisinde her defasında bu koşula bakmak yerine koşulun içerisinde döngü kullanmak daha hızlı çalışmaya yol açabilir. Örneğin flag değişkeni 1 ise döngü içerisinde bir işlem 0 ise başka bir işlem yapıyor olsun:

```
for (i = 1; i < 1000000; ++i) {
    ifade1;
    if (flag)
        ifade2;
    else
        ifade3;
}
```

Döngü şöyle düzenlenirse daha hızlı çalışacak hale getirilebilir:

```
if (flag)
    for (i = 1; i < 1000000; ++i) {
        ifade1;
        ifade2;
    }
else
    for (i = 1; i < 1000000; ++i) {
        ifade1;
        ifade3;
    }
```

Şüphesiz optimizasyonlar gerçek anlamda değecek bir kazanç sağlanıyorsa uygulanmalıdır. Örneğin burada optimize edilmiş biçim kodun daha hızlı çalışmasını sağlamasını karşın hem kodu daha karmaşık gibi göstermekte hem de kodu büyütülmektedir. Buradaki döngü eğer çok az dönüyorsa derleyicinin böyle bir düzenleme yapması toplamda önemli bir hız kazancı sağlamayacaktır.

6) Döngü Açımı (Loop Unrolling): Döngü açımı döngü içerisindeki deyimleri çoklayarak döngünün daha az dönmesini sağlayan ve böylece döngü karşılaştırmasını azaltmayı hedefleyen bir tekniktir. Örneğin:

```
for (i = 0; i < 100; ++i)
    foo();
```

Burada 100 kez foo fonksiyonu çağırılmıştır. Döngü aşağıdaki gibi düzenlenirse daha hızlı çalışacak hale getirilebilir:

```
for (i = 0; i < 100; i += 4) {
    foo();
    foo();
    foo();
    foo();
}
```

Tabi döngü şöyle de açılabilirdi:

```
foo();        // 1
foo();        // 2
...
foo();        // 98
foo();        // 99
foo();        // 100
```

Ancak bu açık kodu çok büyütürdü. İşte derleyiciler bunun iyi bir noktasını bulmaya çalışmaktadır.

7) Döngülerin Ters Çevrilmesi (Loop Inversion): Bu optimizasyon tekniğinde *while* döngüsü ters yüz edilerek *do-while* döngüsü haline getirilir. Örneğin:

```
while (i < 100) {
    foo();
    ++i;
```

```
}
```

Bu döngünün makina kodlarında iki *jump* işlemi vardır. Birincisi döngü kontrolündeki karşılaştırma için uygulanan *jump* ikincisi de döngünün sonuna gelindiğinde yeniden başa geçmek için kullanılan *jump*. Örneğin *Microsoft C (Version 15.00)* derleyicilerinde optimizasyon seçenekleri kapatılarak yapılan derleme işlemi sonucunda aşağıdaki gibi bir kod elde edilmiştir:

```
$LN2@main:
  cmp    DWORD PTR _i$[ebp], 100
  jge    SHORT $LN1@main
  call   _foo
  mov    eax, DWORD PTR _i$[ebp]
  add    eax, 1
  mov    DWORD PTR _i$[ebp], eax
  jmp    SHORT $LN2@main
$LN1@main:
```

Burada *_i\$[ebp]* yerel *i* değişkenini belirtiyor. İşte yukarıda verdiğimiz *while* döngüsünün aşağıdaki ters yüz edilmiş hali daha hızlıdır:

```
if (i < 100)
  do {
    foo();
    ++i;
  } while (i < 100);
```

Bu biçimde döngü içerisinde tek bir *jump* işlemi yapılmaktadır. Derleyiciler optimizasyon seçenekleri açıldığında *while* döngülerini bu biçime getirerek hız kazancı sağlamaya çalışabilirler. Örneğin yukarıdaki birinci kod *Microsoft C (Version 15.00)* derleyicilerinde optimizasyon seçenekleri açıldığında aşağıdaki gibi derlenmektedir:

```
mov    DWORD PTR _i$[esp+4], eax
mov    eax, DWORD PTR _i$[esp+4]
cmp    eax, 100
jge    SHORT $LN7@main
$LL2@main:
call   _foo
inc    DWORD PTR _i$[esp+4]
mov    ecx, DWORD PTR _i$[esp+4]
cmp    ecx, 100
jl     SHORT $LL2@main
```

Jump işlemi pek çok işlemci için *pipeline* mekanizması üzerinde olumsuz etkileri olan bir işlemdir. İşlemci *Jump* işlemini gördüğünde sonraki komutların çalışıp çalışmayacağını bilmediğinden onlar için hazırlık işlemleri yapmayabilir. Bu konuda modern işlemciler arasında çeşitli farklılıklar bulunduğunu belirtelim.

8) Döngü Birleştirmesi (Loop Fusion): Birden fazla peş peşe döngü tek döngü olarak birleştirilebilir. Örneğin:

```
int a[100], i;

for (i = 0; i < 100; ++i)
  a[i] = i;

for (i = 0; i < 100; ++i)
  foo(i);
```

Aşağıdaki döngü daha hızlı çalışabilir:

```
int a[100], i;
```

```
for (i = 0; i < 100; ++i){
    a[i] = i;
    foo(i);
}
```

Ancak cache kullanan bazı işlemcilerde döngüleri birleştirmek yerine tam tersine ayrıştırmak (loop fission) hız kazancı artırabilmektedir. Ya da şöyle diyebiliriz: Cache kullanan bazı işlemcilerde döngü birleştirmesi kar değil zarar oluşturabilir. Bunun nedeni işlemcinin bir bölgeye eriştiğinde oradaki belli miktarda byte'ı cache almasıdır. Bu cache'e alıp bırakma cache tazeleme sorununa yol açabilir ve toplam verimi düşürebilir.

9) Döngü Yer Değiştirmesi (Loop Interchange): Özellikle çok boyutlu dizilerin indekslenmesinde kullanılan bir optimizasyon tekniğidir. Bu teknikte bellekteki atlamaların kısaltılarak *cache* etkinliğinin artırılması hedeflenir. Örneğin:

```
for (k = 0; k < colSize; ++k)
    for (i = 0, i < rowSize; ++i)
        a[i][k] = val;
```

C/C++ gibi programlama dillerinde iki boyutlu diziler satır dizilerinin art arda getirilmesiyle tek boyutlu olarak oluşturulmaktadır. Yani bu dillerde $a[i][k]$ elemanı tek boyutlu dizinin $i * colSize + k$ indeksinde bulunur. Yani bu organizasyon dikkate alındığında yukarıdaki örnekte iç döngüde uzun atlamaların olduğu söylenebilir. Dikkat ediniz, iç döngünün her yinelenmesinde bellekte bir sütun uzunluğu kadar atlama yapılmıştır. Bu atlama modern işlemcilerde fazlaca "cache miss" oluşmasına yol açabilir. İşte bu optimizasyon tekniğinde derleyici döngüleri ters sırada dizerek bu sorunu gidermeye çalışır:

```
for (i = 0, i < rowSize; ++i)
    for (k = 0; k < colSize; ++k)
        a[i][k] = val;
```

Şimdi artık içteki döngünün her yinelenmesinde uzun atlamalar oluşmuyor. Dolayısıyla "cache miss" oranının düşürüleceğini söyleyebiliriz. Ayrıca, döngülerin bu biçimde yer değiştirilmesi vektörel işlem yapan işlemcilerde hız kazancı sağlayabilecek potansiyel bir durumu da oluşturduğunu belirtelim.

10) Göstericilere İlişkin Optimizasyonlar (Pointer Aliasing): Bir göstericinin gösterdiği yerdeki bilgi geçici bir değişkende ya da yazmaçta tutulursa tekrar tekrar o göstericinin gösterdiği yere erişme işlemi elimine edilebilir. Çünkü örneğin C'de *p gibi ya da p[i] gibi işlemler aslında birden fazla makine komutuyla yapılmaktadır. Örneğin:

```
a = *p + 1;
b = *p + 2;
c = *p + 3
```

Burada her defasında *p işlemini yapmak yerine kod aşağıdaki gibi düzenlenirse hız kazancı sağlanabilir:

```
temp = *p;          /* temp bir yazmaç da olabilir */
a = temp + 1;
b = temp + 2;
c = temp + 3
```

Tabii derleyicinin bu optimizasyonu yapabilmesi için kesinlikle o sırada bu göstericinin gösterdiği yerdeki değer değişmeyeceğini garanti altına alması gerekir. Örneğin:

```
a = *p + 1;
foo();
b = *p + 2;
c = *p + 3
```

Burada foo p'nin gösterdiği yerde değişiklik yapıyor olabilir. Bu durumda derleyici foo'nun içini de incelemeyen böyle bir kararı veremez. Örneğin p bir global nesneyi gösteriyor olabilir. foo da o nesneyi değiştiriyor olabilir. Bu durumda foo çağrısından sonra *p değişmiş olacaktır. Böylece derleyici *p değerini yazmaçlarda tutarak onu foo çağrısından sonra kullanamaz.

Eğer bir göstericinin gösterdiği yere yalnızca o gösterici tarafından erişiliyorsa yani o göstericinin dışında oraya erişen başka bir kod yoksa bu yukarıda belirtilen tarzda optimizasyonlara yol açabilir. Örneğin bir sistemde belleğin bir bölgesinden n byte'ı tek hamlede çekerek diğerine tek hamlede aktaran makine komutunun var olduğunu düşünelim. Bu durumda derleyici aşağıdaki gibi bir memcpy fonksiyonunu bu komutu kullanarak optimize edebilir mi?

```
void *memcpy(void *dest, void *source, size_t size)
{
    char *cdest = (char *) dest;
    char *csource = (char *) source;

    while (size-- > 0)
        *cdest++ = *csource++;

    return dest;
}
```

Eğer bloklar çakışıkça derleyici bu makine komutundan faydalanamaz. İşte memcpy'yi çağıran kişi böyle bir duruma yol açabileceği için derleyici bu olasılığı göz ardı edemez.

C99 ile birlikte derleyicilerin gösterici optimizasyonlarını iyileştirmek için restrict gösterici kavramı dile sokulmuştur. restrict anahtar sözcüğü göstericinin kendisi için kullanılabilir, gösterdiği yer için kullanılamaz. Bir gösterici restrict anahtar sözcüğü ile bildirilmişse artık derleyici o göstericinin yaşamı boyunca o göstericinin gösterdiği yere yalnızca bu gösterici yoluyla erişildiği garantisini alır. Şimdi yukarıdaki memcpy fonksiyonundaki göstericileri restrict yapalım:

```
void *memcpy(void * restrict dest, void * restrict source, size_t size)
{
    char * restrict cdest = (char *) dest;
    char * restrict csource = (char *) source;

    while (size-- > 0)
        *cdest++ = *csource++;

    return dest;
}
```

Artık biz burada derleyiciye şunun garantisini veriyoruz: "Derleyici ben sana iki adres vereceğim. Bu göstericilerin faaliyet alanı bitene kadar bu adreslerdeki bilgilere ben başka yolla erişmeyeceğim. Dolayısıyla source ve dest de çakışık blokları göstermeyecek. Sen bu bilgilerden hareketle gösterici optimizasyonları yapabilirsin. Ben bu kurala uymamışsam cezamı ben çekerim."

Şimdi önceki örnekteki optimizasyona geri dönelim:

```
a = *p + 1;
foo();
b = *p + 2;
c = *p + 3
```

Burada eğer p restrict bir göstericiyse derleyici artık aşağıdaki gibi bir optimizasyonu yapabilir:

```
temp = *p;          /* temp bir yazmaç da olabilir */
a = temp + 1;
foo();
b = temp + 2;
```

c = temp + 3

Çünkü artık derleyici p'nin gösterdiği yere foo tarafından da erişilmeyeceğinin sözünü almıştır.

11) Yazmaç Tahsisatları (Register Allocation): İşlemcilerde belli sayıda yazmaç vardır. Değişkenler yazmaçlara çekilerek işleme sokulurlar. Bu durumda derleyiciler hangi değişkenlerin hangi yazmaçlara çekileceği kararını vermek zorundadır. Bu karar verilirken çok kullanılan değişkenlerin mümkün olduğu kadar fazla süre yazmaçta kalmasına çalışılır. Böylece derleyici aynı değişkeni tekrar bellekten yazmaca çekmek zorunda kalmaz. İşte hangi değişkenlerin hangi yazmaçlara çekileceği önemli bir optimizasyon konusudur.

12) Makine Komutlarının Düzenlenmesi ve Yer Değiştirilmesi (Instruction Scheduling and Reordering): Bazı makine komutlarının sırasının değiştirilmesi hız kazancı sağlayabilmektedir. İşte eğer makine komutları arasında girdi çıktı ilişkisi yoksa derleyici bunları yeniden düzenleyerek kodun daha hızlı çalışmasını sağlayabilir. Bu optimizasyon sanıldığı kadar basit değildir ve polinomsal olmayan (NP) tarzda karmaşıklıklara sahiptir. Dolayısıyla derleme zamanını bu tür optimizasyonlar uzatabilmektedir.

Konuyu kapamadan önce şu uyarıyı tekrarlamakta fayda görüyoruz: Derleyicilerin optimizasyonları tek thread'li akışa göre yapılmaktadır. Yani eğer çok thread'li uygulama yapılıyorsa derleyicinin bu optimizasyonları bizim kodumuzun anlamını değiştirebilir. Çok thread'li uygulamalarda bu duruma programcının kendisinin dikkat etmesi gerekmektedir. Örneğin:

```
a = x + y + 10;  
----> Bu noktada x, y değişirse ne olur?  
b = x + y + 20;  
c = x + y + 30;
```

Burada x ve y'nin global değişkenler olduğunu düşünelim ve "ok" ile belirtilen noktada başka bir thread'in x ve y'yi değiştirdiğini varsayalım. Derleyici burada başka bir thread'in x ve y'yi değiştirebileceğini dikkate almaz. Ve optimizasyonunu aşağıdaki gibi yaapabilir:

```
temp = x + y;  
----> Bu noktada x, y değişirse ne olur?  
b = temp + 20;  
c = temp + 30;
```

Görüldüğü gibi burada artık x ve y'nin thread tarafından değiştirilen yeni değeri değil eski değeri sonraki işlemlere sokulmuştur. Biz bu tür optimizasyonlara dikkat etmeliyiz. Burada volatile anahtar sözcüğü hayat kurtarıcı olabilir:

```
volatile x, y;
```

Artık volatile bir nesneye derleyici erişirken her zaman onun o anki değerini bellekten alarak kullanır. Aşağıdaki örneğe dikkat ediniz:

```
while (g_flag) {  
    ...  
}
```

Burada başka bir thread g_flag değişkenini sıfıra çektiğinde döngü sonlanır. Programcı böyle bir sonlanmayı istemiş olabilir. Fakat derleyici g_flag değerini yazmaçta tutarak bir optimizasyon yapabilir. Yani derleyici sürekli olarak g_flag'in değerini yeniden bellekten almak zorunda değildir. Bu nedenle bu koddaki g_flag değişkeni volatile yapılmalıdır.

Derleyiciler eğer multi thread bir ortamı dikkate alarak optimizasyon yapsaydı yukarıda gördüğümüz optimizasyon temalarının çoğu devre dışı kalırdı.

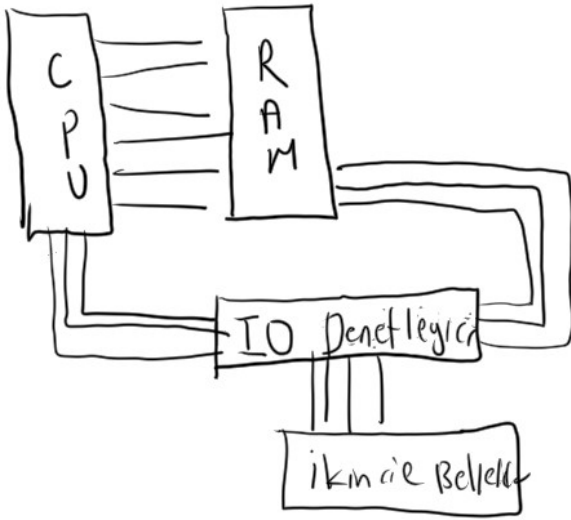
Aşağı Seviyeli Disk İşlemleri

Kursumuzda disk sözcüğü yalnızca manyetik temelli ikincil bellekler için değil tüm ikincil bellekler için kullanılan genel bir terim olarak kullanılmaktadır. Bu bağlamda örneğin SSD'ler ya da flash EPROM'lar yarı iletkenlerle imal edildikleri halde onlara da burada disk denilmektedir.

Bugün kullandığımız ikincil bellekler tipik olarak üç gruba ayrılmaktadır:

- 1) Manyetik tabanlı elektomekanik hard diskler
- 2) Yarı iletkenlerle imal edilmiş SSD'ler, flash EPROM ve EEPROM bellekler
- 3) CDROM, DVD ROM gibi ışınal mekanizmayla çalışan elektormekanik ikincil bellekler

Tipik olarak bir bilgisayar devresinde CPU ile ikincil bellekler elektriksel doğrudan bağlantılı değildir. Bağlantı bir denetleyici yoluyla gerçekleştirilmektedir. CPU'nun en az bir IO denetleyici birimle bağlantısı vardır. CPU bu birimlere elektriksel olarak komutlar gönderir. IO denetleyicileri de bu komutları ikincil bellek birimlerine (yani disklere) yollar. İkincil bellek birimleri de kendi devrelerini harekete geçirerek istenilen işlemi yapar. Böylece örneğin bir hard diskin içerisindeki devreler ve mekanik bileşenler standart değildir. Ancak CPU'nun komut gönderdiği IO denetleyicisi standart bir arayüze sahiptir.



İkincil bellekleri kontrol etmek için IDE, ATA, SATA, SCSI gibi çeşitli denetleyici birimler ve bus arayüzleri bulunmaktadır. CPU bu denetleyicilere komutlar göndererek isteğini bildirir. Bu denetleyiciler elektriksel yollarla asıl ikincil belleğe komutu iletirler. İkincil bellekler de istenilen işlemi yaparlar. Bir okuma işlemi söz konusuysa okunan bilgiler doğrudan RAM'e aktarılır. Bir yazma işlemi söz konusuysa RAM'de oluşturulmuş bilgiler ikincil belleğe doğrudan aktarılır. Bu aktarım işlemi DMA (Direct Memory Access) denilen denetleyiciler tarafından yapılmaktadır.

Bu durumda ikincil bellekten bir okuma ya da yazma isteği iki önemli bilgiyi içermek zorundadır:

- 1) Bu istek ikincil belleğin neresine ilişkindir ve hangi büyüklüktedir?
- 2) Okunacak ya da yazılacak bilgiler için RAM aktarım adresi neresidir?

Modern çok prosesli işletim sistemleri ikincil belleklerden bir talepte bulunduğu anda onun sonucunu beklemezler. Proseslerarası (ya da thread'lerarası) geçiş yaparak istekte bulunan proses ya da thread'i blokede bekletirler. IO denetleyicileri işlem bittiğinde bunu CPU'ya bildirmektedir. İşletim sistemi de işleme yol açan proses ya da thread'i çizelgeye yeniden sokarak onun kaldığı yerden çalışmaya devam etmesini sağlar.

İkincil Belleklerdeki Organizasyon

İkincil belleklerden bilgiler byte byte değil blok blok transfer edilmektedir. Genel olarak ikincil belleklerden transfer edilecek en küçük birime sektör denir. Bir sektör 512 byte uzunluktadır. (Bazı durumlarda bu büyüklük değişebilir. Ancak

512 standart bir değer olarak kabul görmüştür). Sektör klasik elektromekanik disk sistemleri zamanında uydurulmuş bir terimdir. Günümüzde SSD gibi Flash EPROM ve EEPROM temelli belleklerdeki transfer yöntemleri disklere göre değişiklik gösterse de yine sektörel bir transfer uygulanmaktadır.

Hard Disklerin Yapısı

Elektromekanik hard disklerde bilgiler manyetik olarak bir yüzeye kaydedilir ve oradan okunur. Bir hard diskte tipik olarak platter denilen yüzeye bir çubuğa geçirilmiş duurmdadır. Bu yüzeyler belli bir hızda döndürülür. Okuma yazma amacıyla her yüz için bir kafa (head) bulundurulmuştur. Kafa diske çok yaklaşır fakat onunla temas etmez. Bilgi manyetik olarak kodlanır ve geri alınır.

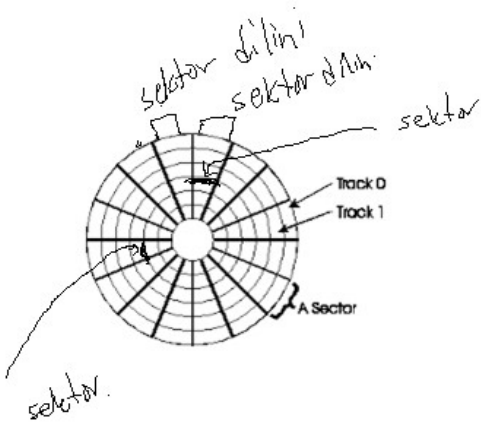


Yukarıdaki resimde üç platter vardır. Disk kafaları aynı eksene monte edilmiştir. Her yüzey için bir kafa bulunmaktadır. Yani yukarıdaki resimde görülen hard diskte toplam 6 kafa vardır. Dodlama ve geri alma disk dönerken kafalar tarafından yapılır. Hard diskler kendi içerisinde küçük bir önbelleğe de sahip olabilirler.

Sektör bir hard diskten okunabilecek ya da hard diske yazılabilecek en küçük bir birimdir. Yani örneğin biz bir diske 10 byte yazamayız. En az bir sektör yazabiliriz. Örneğin biz bir diskteki bir byte'ı değiştirmek isteyelim. Bunun tek yolu şudur: Önce o byte'ın bulunduğu sektör okunur ve RAM'e aktarılır. Değişiklik RAM'de yapılır. Sonra o sektör diske geri yazılır.

Hard diskte bilgilerin yazıldığı yollara track denilmektedir. Track'ler en dıştan başlayarak sıfırdan itibaren numaralandırılmıştır. Kafalar track hizasına getirilir, disk dönerken okuma yazma işlemi gerçekleşir.

Eskiden diskler sektör dilimi denilen pasta dilimlerine ayrılıyordu. Track'lerin sektör dilimleri içerisinde kalan parçaları sektörleri oluşturuyordu:



Şekilden de görüldüğü gibi dış track'ler daha uzundur. Fakat eski sistemde track'lerin sektör dilimleri içerisinde kalan parçalarına hep aynı miktar bilgi (512 byte) depolanıyordu. Ancak dah sonraları (90'lı yıllarda) artık üretici firmalar dış track'lere daha fazla bilgi depolamaya başlamıştır. Dolayısıyla sektör dilimi kavramının pratik bir geçerliliği kalmamıştır.

Bugünkü hard disklerde yine sektör temelinde aktarım yapılır. Ancak dış track'lerdeki sektör sayıları iç track'lerden daha fazladır.

Bir hard disk hızı saniyede okunan ya da yazılan byte sayısı (ya da sektör sayısı) ölçülmektedir. Bir sektörün okunma ya da yazılma hızı üç bileşenin toplamından oluşur:

1) Disk Kafasının Uygun Track'e Konumlanma Zamanı (Seek Time): Disk kafasının uygun track hizasına çekilmesi en önemli zaman kaybını oluşturmaktadır. Dolayısıyla sistem programcısı için en önemli bileşen disk kafa hareketinin optimize edilmesidir.

2) Diskin Dönerek Kafa Hizasına Gelene Kadarki Zaman (Rotational Delay): Diskin hızını etkileyen ikinci bileşen disk dönüş hızıdır. Çünkü sektör kafa hizasından geçerken okuma yazma yapılır. İlgili sektörün kafa hizasından geçme zamanı diskin dönüş hızıyla ilgilidir. Bugünkü diskler tipik olarak 4200, 5400, 7200, 10000, 15000 hızlardadır. Bizim notebook'larımızda tipik olarak 5400 dönüş hızları kullanılmaktadır.

3) Aktarım Zamanı (Transfer Time): Bu zaman en önemsiz zamanı oluşturur. Disk dönerken kafanın bilgiyi okuyup IO denetleyicisine aktardığı zamandır. Bu zaman IO denetleyicisinin tasarımına da hard disk teknolojisine de bağlıdır. Örneğin SCSI denetleyiciler daha hızlı bir aktarım sunabilmektedir.

İşletim sistemi tasarımcıları disk kafa hareketini minimize etmeye odaklanırlar. Çünkü diğer bileşenler sistem programcısının kontrolü dışındadır. Eğer işletim sistemi dosyanın parçalarını birbirine yakın sektörlerde tutarsa o dosyaya erişimdeki kafa hareketleri azaltılmış olur. Örneğin "defrag" isimli disk utility programları dosyaların parçalarını yer değiştirerek birbirlerine yaklaştırmaktadır.

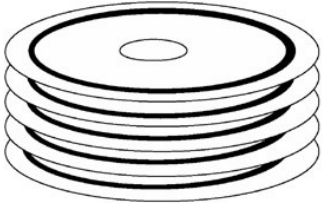
Sektörler İçin Koordinat Sistemi ve Sektör Numaralandırması

İkincil belleğin türü ne olursa olsun bugün ikincil belleklerle ana bellek arasındaki veri aktarımı sektör düzeyinde yapılmaktadır. Sektörlerin de birer numaraları vardır. Sektörler track'lerin üzerinde bulunur. Track'teki sektör sayısı hard diskten hard diske değişebilmektedir. Ayrıca disk yüzeyleri dairesel olduğu için dış track'lerdeki sektör sayıları iç track'lerdekine göre daha fazladır. Eskiden sektör dilimlerinin anlamlı zamanlarda bir sektörün koordinatı yüzey numarası (kafa numarası da denilmektedir), track numarası ve sektör dilimi numarasıyla belirleniyordu. Yani koordinat h:t:s bileşenlerinden oluşmaktaydı. Bugün hala BIOS geçmişe doğru uyumu korumak için bu koordinat sistemini de kullanmaktadır. Ancak daha sonra sektör dilimlerinin fiziksel bir anlamı kalmamaya başlayınca daha sade bir koordinat sistemine geçilmiştir. Bu koordinat sisteminde sektörler tek bir sayıyla belirtilmektedir. Bu sistemde ilk sektörün numarası sıfır olmak üzere her sektöre artan sırada bir numara karşılık düşürülmüştür. Buna Mantıksal Blok Adreslemesi (Logical Block Addressing ya da kısaca LBA) denilmektedir. Numaralandırma en dış yüzeyin en dış track'inden başlayarak (yani sıfıncı track) önce yüzler sonra trackler dolanılarak yapılmaktadır. Yani aşağıdaki sahte koddaki gibi bir numaralandırma söz konusudur:

```
logical_address = 0;
for (track = 0; track < MAX_TRACK; ++track)
  for (head = 0; head < MAX_HEAD; ++head) {
    <logical_address numarasını oluştur>
  }
```

Görüldüğü gibi mantıksal numaralandırma kafa hareketini minimize edecek biçimde yapılmıştır. Numaralandırmada önce en dış yüzeyin ilk track'i dolanılır, sonra diğer yüze geçilip onun en dış track'i dolanılır. Böyle böyle tüm yüzlerin en dış track'leri dolandıktan sonra bir sonraki track'e geçilir. Yani mantıksal mantıksal sektör numaraları birbirine yakınsa bunlar daha küçük kafa hareketiyle erişilecek sektörlerdir. Artık IO denetleyicileri de sistem programcısından sektörleri mantıksal sektör numarasıyla istemektedir. Sonuç olarak diskteki her sektörün bir numarası vardır. Ardışıl sektör numaraları aynı kafa hizasına karşılık gelir.

Sektör numaralandırmasında bir de silindir (cylinder) terimi ile karşılaşılmaktadır. Tüm yüzeylerin n numaralı track'lerinin kümesine 'n' numaralı silindir denilmektedir. Örneğin 6 yüzeyli (3 platter'lı) bir hard diskte 10 numaralı silindir demek tüm yüzeylerin 10 numaralı track'lerinin kümesidir.



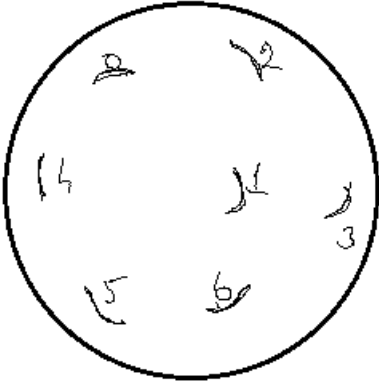
A cylinder is the set of the same tracks across all platters.

SSD'Diskler ve Sektörler

Günümüzde hard diskler gitgide azalmakta ve yerini SSD disklere bırakmaktadır. SSD diskler mekanik bir parçaya sahip değildir. Bizim kullandığımız flash EPROM'larla (memort stick'lerle) aynı teknolojiye sahiptirler. Dolayısıyla SSD'lerden okuma ve yazma işlemi hard disklere göre çok daha hızlı yapılabilmektedir. Ancak genel olarak SSD'lerden okuma çok hızlı olmasına karşın yazma görece olarak daha yavaştır. SSD'ler mekanik bir birim olmadığı için onlarda okuma ve yazma için kafalar bulunmaz. Dolayısıyla SSD'lerde track kavramı yoktur. SSD'lerde her sektörün okunması ve yazılması yaklaşık aynı sürede yapılmaktadır. (Halbuki hard disklerde kafanın o anki konumuna ve okunacak sektörün yerine göre sektör okuma ve yazma hızları değişebilmektedir.) Her ne kadar SSD'ler mekanik bir parça içermiyorsa da bunlardan transfer yine 512 byte'lık sektörler yoluyla yapılmaktadır. Yani sistem programcısı için arayüz bakımından hard disklerle SSD'ler arasında bir fark yoktur.

Dosya Kavramı İle Sektörler Arasındaki İlişki

Aslında dosya (file) kavramı uydurma bir kavramdır. İkincil belleklerdeki tek gerçek, sektör gerçeğidir. İşletim sistemi birtakım bilgileri diskin sektörlerinde saklar buna da bir isim karşılık düşürür. Kullanıcılara bunu dosya kavramı biçiminde gösterir. Aslında dosyalar ardışıl byte'lardan oluşmamaktadır. Diskteki sektörlerde bulunan bilgilerden oluşmaktadır. İşletim sistemi bize o sektörlerdeki bilgiyi sanki ardışıl byte topluluğuymuş gibi gösterir. Örneğin 7 sektörden oluşan bir dosya aşağıdaki gibi disk yüzeyine (hatta yüzeylerine) dağılmış olabilir.



İşletim sistemi dosyaların parçalarını sektörlerle neden ardışıl yerleştirmemektedir? İşte daha önceden de gördüğümüz gibi ardışıl yerleştirme zorunluluğu bölünme (fragmentation) denilen olguya yol açar. Bölünme tahsis etme ve silme işlemleri sonucunda bellekte ardışıl olmayan çok sayıda küçük blokların oluşması durumudur. Bu diskin kullanım verimini çok düşürür. İşte sistemleri dosyaların parçalarını disk kafa hareketlerini azaltmak için mümkün olduğu kadar ardışıl yerleştirmeye çalışmaktadır. Ancak ardışıl yerleştirme bir zorunluluk biçiminde değildir.

Mademki işletim sistemi dosyaların parçalarını ardışıl olmayan sektörlerde tutabilmektedir, o halde dosyanın hangi parçalarının diskin neresinde olduğu bir biçimde işletim sistemi tarafından tutulmalıdır.

Cluster ve Block Kavramları

Aslında sektörler dosya parçalarını saklamak için küçük birimlerdir (bir sektörün 512 byte olduğunu anımsayınız). İşte işletim sistemleri dosya parçalarını sektörler yerine "cluster" ya da "block" denilen ardışıl sektörlerde saklarlar. Cluster ile block kavramları bu bağlamda eş anlamlıdır. Microsoft "cluster" terimini tercih ederken UNIX/Linux sistemlerinde block terimi tercih edilmektedir. Bir cluster ya da blok ardışıl n tane sektörden oluşturulmaktadır (burada n sayısı

genellikle 2'nin bir kuvvetidir). İşletim sistemleri diski cluster ya da bloklara ayırır ve her cluster ya da bloğa bir numara verir. Tabii cluset ve block kavramları aslında yapay kavramlardır. Diske sektör numaralattıyla erişilir.

İşletim sistemleri dosyaların parçalarını cluster ya da block'larda saklar. Bir cluster ya da block bir dosyanın parçası olabilecek en küçük birimdir. Örneğin dosya 1 byte bile uzunlukta olsa o diskte bir cluster ya da block yer kaplar. Dosyanın parçalarının sektörlerde değilde ardışıl sektör bloklarında tutulmasının iki nedeni vardır:

1) Bu sayede dosyaların hangi parçalarının nerede tutulduğuna ilişkin meta-data alanı küçültülmüş olur (çünkü dosya daha az parçadan oluşacaktır)

2) Bu sayede dosya diskte daha az yayılmış bir biçimde tutulur. Bu da dosyaya erişmek için gereken disk kafa hareketlerinin azalmasına yol açar.

Peki bir cluster ya da block kaç sektörden oluşmalıdır? İşte eğer bir cluster ya da block az sayıda sektörden oluşursa diskteki meta-data alanları büyür, dosya erişmek için gereken kafa hareketleri artar. Fakat dosyaların son cluster ya da bloklarında kullanılmayan alanlar (yani içsel bölünme) azaltılır. Eğer bir cluster ya da block çok sektörden oluşturulursa bu durumda disk meta-data alanları küçülür, kafa hareketleri azalır fakat içsel bölünme miktarı artar. İşte işletim sistemleri disk büyüdükçe cluster ya da bloğun daha fazla sektörden oluşturmayı, disk küçüldükçe daha az sektörden oluşturmayı tercih etmektedir. Bazı işletim sistemlerinde disk formatlanırken bunu sistem yöneticisi de belirleyebilmektedir.

Disk Editörleri

Nasıl text editörler varsa, audio ve video editörleri varsa diski analiz etmek ve görüntülemek için de disk editörleri vardır. Bazı disk editörleri paralıdır. Bazıları ise açık kaynak kodlu ve/veya bedavadır. Bedava iki önemli disk editörü "Active Disk Editor" ve "HxD" editörüdür. Bu editörler sayesinde istediğimiz disk sektörlerini inceleyip onlar üzerinde değişiklikler ve dosya sistemi üzerinde çeşitli incelemeler yapabiliriz.

Dosya Sistemleri

Bir işletim sisteminin dosyalarla ilgili işlemler yapan alt sistemine dosya sistemi (file system) denilmektedir. Dosya sisteminin gerçekleştirilme (implementation) bakımından iki tarafı vardır:

1) Ana Bellek Tarafı: İşletim sistemi açılan her dosya için kernel alanı içerisinde o dosyaya yönelik kayıtlar tutar. Sonra bu kayıtlardan faydalanarak dosya işlemini gerçekleştirir. Bu kayıtlar dosya sisteminin ana bellek tarafını oluşturmaktadır.

2) Disk Tarafı: İşletim sistemleri diskin sektörlerini organize ederek dosyaları diskte düzenli bir biçimde saklarlar. İşte işletim sistemlerinin disk üzerindeki organizasyonuna o dosya sisteminin disk tarafı denilmektedir. Biz kursumuzda "bir dosya sisteminin disk organizasyonu" dediğimizde o dosya sisteminin hangi disk sektörlerini hangi amaçla kullandığını anlamalıyız.

Bazı işletim sistemlerine ilişkin dosya sistemlerinin ana bellek tarafı birden fazla disk organizasyonunu destekleyebilmektedir. Örneğin Linux sistemleri Microsoft'un FAT ve NTFS olarak düzenlenmiş disk organizasyonunu tanıyıp o diskler üzerinde dosya işlemleri yapabilmektedir. Ancak Microsoft, Linux sistemlerinde kullanılan EXT-2 EXT-3 gibi disk organizasyonlarını desteklememektedir.

Çok Kullanılan Bazı Dosya Sistemleri

Microsoft'un ilk işletimi olan DOS ismine FAT (File Allocation Table) denilen bir dosya sistemi kullanıyordu. FAT sisteminin 12 bit, 16 bit ve 32 bitlik versiyonları vardır. Bunlar FAT12, FAT16, FAT32 biçiminde isimlendirilmektedir. Microsoft daha sonra NTFS (New Technology File System) isminde yeni bir dosya sistemi daha geliştirmiştir. Bugün ağırlıklı olarak Windows sistemlerinde NTFS kullanılmaktadır. Linux sistemleri geleneksel I-Node tabanlı dosya sistemlerini kullanır. Ext-2, Ext-3, Ext-4 sistemleri Liux'ta en çok kullanılan dosya sistemleridir. Mac OS sistemleri geleneksel olarak HFS (Hierarchical File System) ve bunun biraz daha geliştirilmiş versiyonu olan HFS+ sistemlerini kullanmıştır. Ancak 2017 yılında Apple ismine APFS (Apple File System) denilen yeni bir dosya sistemine geçmiştir. Bu dosya sistemi IOS işletim

sistemlerinde de benzer biçimde kullanılmaktadır. APFS HFS+ sistemlerinin daha ileri bir versiyonu gibi düşünülebilir. CD-ROM'lar ve DVD-ROM'lar yalnızca okunabilen medyalar oldukları için onlara yönelik ISO 9660 denilen ayrı bir dosya sistemi tasarlanmıştır.

Windows'ta Aşağı Seviyeli Disk İşlemlerinin Yapılması

Bilindiği gibi disk sektörlerinin okunup yazılması aşağı seviyeli olarak Disk IO denetleyicisinin (IDE, SATA gibi) programlanmasıyla yapılmaktadır. Ancak Windows ve Linux gibi işletim sistemleri bu işlemi yapan kodları zaten kendi içlerinde bulundurlar. User moddan bizim bu kodları kullanabilmemizi aygıt sürücülerle sağlarlar. Bu aygıt sürücüler diski sanki bir dosyaymış gibi bize göstermektedir. Biz bu aygıt sürücülerini Windows sistemlerindeki dosya açan CreateFile API fonksiyonuyla açarız. Sonra SetFilePointer API fonksiyonuyla ilgili sektörün başladığı pozisyona dosya göstericisini konumlandırırız. Oradan ReadFile API fonksiyonuyla okuma yapıp, WriteFile API fonksiyonuyla oraya yazma yapabiliriz. Bu işlemleri yapan iki dosya DiskIO.h ve DiskIO.c isimliyle kurs için hazırlanmıştır:

```
/* DiskIO.h */

#ifndef DISKIO_H_
#define DISKIO_H_

#include <Windows.h>

/* Symbolic Constants */

#define BYTE_PER_SECTOR      512

/* Function Prototypes */

HANDLE OpenDisk(int disk, DWORD dwDesiredAccess);
HANDLE OpenVolume(int drive, DWORD dwDesiredAccess);
BOOL ReadSector(HANDLE hDrive, __int64 sector, DWORD count, void *buf);
BOOL WriteSector(HANDLE hDrive, __int64 sector, DWORD count, const void *buf);

#endif

/* DiskIO.c */

#include <stdio.h>
#include <Windows.h>
#include "DiskIO.h"

/* static Function Prototypes */

static BOOL SetFilePosition(HANDLE hDrive, __int64 distance, DWORD MoveMethod);

/* Function definitions */

HANDLE OpenDisk(int disk, DWORD dwDesiredAccess)
{
    char driverName[32];
    DWORD shareMode = 0;

    sprintf(driverName, "\\\\.\\PhysicalDrive%d", disk);

    if (dwDesiredAccess & GENERIC_READ)
        shareMode |= FILE_SHARE_READ;
    if (dwDesiredAccess & GENERIC_WRITE)
        shareMode |= FILE_SHARE_WRITE;

    return CreateFile(driverName, dwDesiredAccess, shareMode, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
}

HANDLE OpenVolume(int drive, DWORD dwDesiredAccess)
```

```

{
    char driverName[32];
    DWORD shareMode = 0;

    sprintf(driverName, "\\.\%c:", drive + 'A');

    if (dwDesiredAccess & GENERIC_READ)
        shareMode |= FILE_SHARE_READ;
    if (dwDesiredAccess & GENERIC_WRITE)
        shareMode |= FILE_SHARE_WRITE;

    return CreateFile(driverName, dwDesiredAccess, shareMode, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
}

BOOL ReadSector(HANDLE hDrive, __int64 sector, DWORD count, void *buf)
{
    __int64 location = (__int64)sector * BYTE_PER_SECTOR;
    DWORD dwBytes = count * BYTE_PER_SECTOR;
    DWORD dwRead;

    if (!SetFilePosition(hDrive, location, FILE_BEGIN))
        return FALSE;

    if (!ReadFile(hDrive, buf, dwBytes, &dwRead, NULL))
        return FALSE;

    return TRUE;
}

BOOL WriteSector(HANDLE hDrive, __int64 sector, DWORD count, void *buf)
{
    __int64 location = (__int64)sector * BYTE_PER_SECTOR;
    DWORD dwBytes = count * BYTE_PER_SECTOR;
    DWORD dwRead, status;

    if (!SetFilePosition(hDrive, location, FILE_BEGIN))
        return FALSE;

    if (!DeviceIoControl(hDrive, FSCTL_DISMOUNT_VOLUME, NULL, 0, NULL, 0, &status, NULL))
        return FALSE;

    if (!WriteFile(hDrive, buf, dwBytes, &dwRead, NULL))
        return FALSE;

    return TRUE;
}

static BOOL SetFilePosition(HANDLE hDrive, __int64 distance, DWORD MoveMethod)
{
    LARGE_INTEGER li;

    li.QuadPart = distance;
    li.LowPart = SetFilePointer(hDrive, li.LowPart, &li.HighPart, MoveMethod);

    if (li.LowPart == 0xFFFFFFFF && GetLastError() != NO_ERROR)
        return FALSE;

    return TRUE;
}

```

Burada OpenDisk tüm diski (disk bölümleri dikkate alınmaksızın) tek bir dosya gibi açmak için, OpenVolume ise yalnızca ilgili disk bölümünü bir dosya açmak için kullanılır. Örneğin biz G volümünün boot sektörünü şöyle ekrana yazdırabiliriz:

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

```

#include <Windows.h>
#include "DiskIO.h"

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)
{
    BYTE buf[512];
    HANDLE hVolume;
    int i;

    if ((hVolume = OpenVolume(6, GENERIC_READ)) == INVALID_HANDLE_VALUE)
        ExitSys("OpenVolume", EXIT_FAILURE);

    if (!ReadSector(hVolume, 0, 1, buf))
        ExitSys("ReadSector", EXIT_FAILURE);

    for (i = 0; i < 512; ++i) {
        printf("%02X ", buf[i]);

        if (i % 16 == 15)
            printf("\n");
        else if (i % 8 == 7)
            printf(" ");
    }
    printf("\n");

    CloseHandle(hVolume);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    exit(status);
}

```

Linux Sistemlerinde Aşağı Seviyeli Disk İşlemlerinin Yapılması

Linux sistemlerinde diskler ve volümler /dev dizininin altında aygıt sürücü biçiminde bulunmaktadır. Yani bu sistemlerde disk işlemleri tamamen dosya işlemleri gibi yapılmaktadır. Disk ya da volüme ilişkin aygıt sürücü open fonksiyonuyla açılır. lseek fonksiyonuyla konumlandırma yapılarak read ve write fonksiyonlarıyla da okuma ve yazma yapılmaktadır. Örneğin tipik olarak sata disklerde tüm diske ilişkin bu aygıt sürücü dosyaları /sda, /sdb, /sdc biçimindedir. Volümlere ilişkin dosyalar da /sda1, /sda2, /sda3 biçiminde organize edilmiştir. Bu aygıt sürücü dosyaları "other" için read ve write haklarına sahip değildir. Bu nedenle bu aygıt sürücüleri kullanan programlar "sudo" ile çalıştırılmalıdır.

```

csd@csd-vm:~/Study/SysProg-2019$ ls -l /dev/sd*
brw-rw---- 1 root disk 8, 0 Nov  3 11:25 /dev/sda
brw-rw---- 1 root disk 8, 1 Nov  3 11:25 /dev/sda1
brw-rw---- 1 root disk 8, 2 Nov  3 11:25 /dev/sda2

```

Örneğin fiziksel hard diskin ilk sektörünü okumak isteyelim:

```

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

```

```

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

void exit_sys(const char *msg);

int main(void)
{
    int fd;
    unsigned char buf[512];
    int i;

    if ((fd = open("/dev/sda", O_RDONLY)) == -1)
        exit_sys("open");

    if (read(fd, buf, 512) != 512)
        exit_sys("read");

    for (i = 0; i < 128; ++i)
        printf("%02X%c", buf[i], i % 16 == 15 ? '\n' : ' ');

    close(fd);

    return 0;
}

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

    exit(EXIT_FAILURE);
}

```

FAT Dosya Dosya Sistemlerinin Disk Organizasyonu

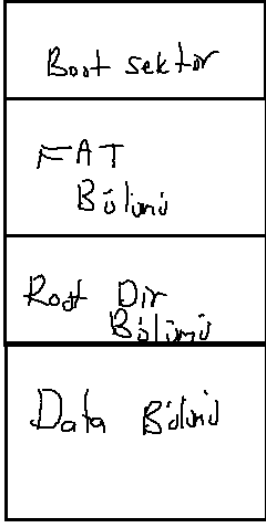
FAT dosya sistemleri Microsoft'un ilk DOS işletim sisteminden beri kullandığı temel dosya sistemidir. FAT dosya sistemi ilk yıllarda önemli bir yenilik olarak değerlendirilmiştir. Bu dosya sistemlerinde ileride ele alınacağı gibi ismine FAT (File Allocation Table) denilen önemli bir meta-data bölümü vardır. Bu bölüm dosya sistemine ismini vermiştir.

FAT dosya sisteminin FAT12, FAT16 ve FAT32 olmak üzere üç değişik biçimi vardır. FAT12'de FAT elemanları 12 bit, FAT16'da 16 bit ve FAT32'de 32 bittir. Ayrıca FAT32'deki meta-data organizasyonunda bazı önemli değişiklikler de yapılmıştır. FAT32 daha büyük disk bölümlerini kullanabilmektedir. Genel kapasite olarak FAT12 ve FAT16 sistemlerinin geliştirilmiş bir biçimidir.

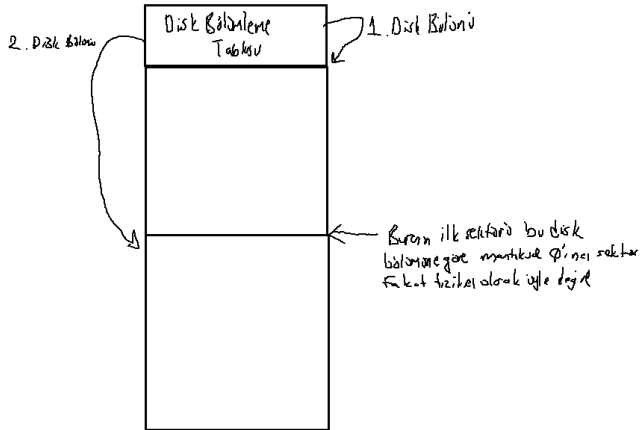
FAT dosya sistemi hakkında sistem programcısına yönelik teknik bilgiler çeşitli sitelerde dağınık biçimde bulunabilmektedir. Bunun için Microsoft'un Technical Reference dokümanlarına aşağıdaki linkten ulaşılabilir:

[https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc758586\(v=ws.10\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc758586(v=ws.10))

FAT12 ve FAT16 dosya sistemiyle formatlanmış bir disk bölümü aşağıdaki organizasyonel bölümlerden oluşur:



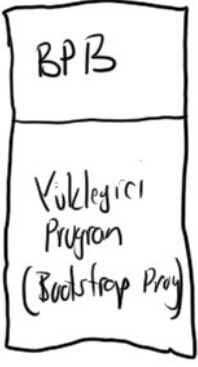
Hard disk'ler, SSD'ler ve Flash EPROM bellekler birden fazla disk bölümüne sahip olabilirler ve bu disk bölümlerine farklı dosya sistemleri yüklenebilir. Bu medyalarda hangi disk bölümlerinin hangi sektörden başlayıp ne kadar uzunlukta olduğu medyanın içerisinde saklanmaktadır. Disk bölümlerinin nereden başladığı ve ne uzunlukta olduğunu tutan meta-tablosuna genel olarak Disk Bölümleme Tablosu (Disk Partition Table) denilmektedir. Disk bölümlerine ayrılmış bir diskteki sektör numaralandırmaları iki biçimde yapılabilmektedir: Fiziksel numaralandırma ve mantıksal numaralandırma. Fiziksel numaralandırma kontrol kartının uyguladığı gerçek numaralandırma. Burada medyanın tamamı tek bir medya olarak değerlendirilir ve ilk sektör 0'dan başlatılarak her sektöre artan bir numara karşılık düşürülmüştür. Mantıksal numaralandırmada ilgili disk bölümünün başı orijin noktası olarak kabul edilir. Yani her disk bölümü adeta ayrı bir diskmiş gibi 0'dan başlayarak numaralandırılmaktadır. Örneğin:



Bir diskteki kullanıma hazırlanmış disk bölümlerine (partitions) "volume" de denilmektedir. Microsoft "volume" sözcüğünü bu bağlamda çok kullanmaktadır.

Boot Sektör

FAT dosya sistemlerinde volüm'ün ilk sektörüne (yani mantıksal 0'ıncı sektörüne) boot sektör sektör denilmektedir. Boot sektör kendi içerisinde iki bölümden oluşmaktadır: "BPB (Bios Paramter Block)" ve "Yükleyici Program (Bootstrap Program)"



BPB FAT dosya sisteminde volümün parametrik bilgilerinin tutulduğu bölümdür. Boot sektörün başından başlar. Yaklaşık 30-40-50 byte uzunluğundadır (uzunluğu FAT türüne göre ve versiyona göre değişebilmektedir.)

BPB bölümü FAT dosya sisteminin kalbidir. Volümün tüm parametrik bilgileri, meta-data alanlarının yerleri vs. bu bölümde tutulmaktadır. Boot sektörün bölümleri iyi anlaşılın diye FAT16 bir volümün boot sektörü aşağıda verilmiştir:

```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 02 08 00  <.MSDOS5.0....
00000010 02 00 02 00 00 F8 A0 00 3F 00 FF 00 00 B8 DC E8  ....ø.?.ÿ..Üè
00000020 00 40 01 00 80 00 29 49 73 D6 26 4E 4F 20 4E 41  .(..€.)IsÖ&NO NA
00000030 4D 45 20 20 20 20 46 41 54 31 36 20 20 20 33 C9  ME  FAT16 3É
00000040 8E D1 BC F0 7B 8E D9 B8 00 20 8E C0 FC BD 00 7C  .Ñ+ç{.Ü.. .Äü%|
00000050 38 4E 24 7D 24 8B C1 99 E8 3C 01 72 1C 83 EB 3A  8N$)$(Á"è<.r.fè:
00000060 66 A1 1C 7C 26 66 3B 07 26 8A 57 FC 75 06 80 CA  f;.|&f;.&šWüu.€È
00000070 02 88 56 02 80 C3 10 73 EB 33 C9 8A 46 10 98 F7  .˘V.€Ä.sè3ÈšF."÷
00000080 66 16 03 46 1C 13 56 1E 03 46 0E 13 D1 8B 76 11  f..F..V..F..Ñ<v.
00000090 60 89 46 FC 89 56 FE B8 20 00 F7 E6 8B 5E 0B 03  `%Fü%Vç, .÷æ<^..
000000A0 C3 48 F7 F3 01 46 FC 11 4E FE 61 BF 00 00 E8 E6  ÅH÷ó.Fü.Nçaz..èæ
000000B0 00 72 39 26 38 2D 74 17 60 B1 0B BE A1 7D F3 A6  .r9&8-t.`±.%;|ó|
000000C0 61 74 32 4E 74 09 83 C7 20 3B FB 72 E6 EB DC A0  at2Nt.fç ;ûræëÜ
000000D0 FB 7D B4 7D 8B F0 AC 98 40 74 0C 48 74 13 B4 0E  û)`<ç-"@t.Ht.`.
000000E0 BB 07 00 CD 10 EB EF A0 FD 7D EB E6 A0 FC 7D EB  »..Í.èi 1)èæ ü)è
000000F0 E1 CD 16 CD 19 26 8B 55 1A 52 B0 01 BB 00 00 E8  áí.í.è<U.R°.»..è
00000100 3B 00 72 E8 5B 8A 56 24 BE 0B 7C 8B FC C7 46 F0  ;.rè[šVš%|<üçFç
00000110 3D 7D C7 46 F4 29 7D 8C D9 89 4E F2 89 4E F6 C6  =)çFó)çÜ%Nò%NòÈ
00000120 06 96 7D CB EA 03 00 00 20 0F B6 C8 66 8B 46 F8  .-)Èè... .çÈf<Fø
00000130 66 03 46 1C 66 8B D0 66 C1 EA 10 EB 5E 0F B6 C8  f.F.f<çfÁè.è^çÈ
00000140 4A 4A 8A 46 0D 32 E4 F7 E2 03 46 FC 13 56 FE EB  JJšF.2ä÷â.Fü.Vçè
00000150 4A 52 50 06 53 6A 01 6A 10 91 8B 46 18 96 92 33  JRP.Sj.j.`<F.-'3
00000160 D2 F7 F6 91 F7 F6 42 87 CA F7 76 1A 8A F2 8A E8  Ò÷ò'÷òB÷È÷v.šòšè
00000170 C0 CC 02 0A CC B8 01 02 80 7E 02 0E 75 04 B4 42  Àì..ì,..€~..u.'B
00000180 8B F4 8A 56 24 CD 13 61 61 72 0B 40 75 01 42 03  <òšVšÍ.aar.@u.B.
00000190 5E 0B 49 75 06 F8 C3 41 BB 00 00 60 66 6A 00 EB  ^.Iu.øÄA»..`fj.è
000001A0 B0 42 4F 4F 54 4D 47 52 20 20 20 20 0D 0A 52 65  °BOOTMGR ..Re
000001B0 6D 6F 76 65 20 64 69 73 6B 73 20 6F 72 20 6F 74  move disks or ot
000001C0 68 65 72 20 6D 65 64 69 61 2E FF 0D 0A 44 69 73  her media.ÿ..Dis
000001D0 6B 20 65 72 72 6F 72 FF 0D 0A 50 72 65 73 73 20  k errorÿ..Press
000001E0 61 6E 79 20 6B 65 79 20 74 6F 20 72 65 73 74 61  any key to resta
000001F0 72 74 0D 0A 00 00 00 00 00 00 00 00 AC CB D8 55 AA  rt.....-ÈØU*

```

Şimdi BPB bölümünü byte byte inceleyelim:

Offset (Hex)	Uzunluk	İçerik
00	3 byte	Jmp Code
03	8 byte	OEM Yorum
0B	WORD	Sektördeki Byte Sayısı
0D	BYTE	Cluster'ın Sektör Uzunluğu
0E	WORD	Ayrılmış Sektörlerin Sayısı
10	BYTE	FAT kopyalarının Sayısı
11	WORD	Root Girişlerinin Sayısı
13	WORD	Volümdeki Toplam Sektör Sayısı

15	BYTE	Ortam Belirleyicisi (Media Descriptor)
16	WORD	Fat'ın Bir Kopyasındaki Sektör Sayısı
18	WORD	Track Başına Düşen Sektör Sayısı
1A	WORD	Diskteki Kafa (yüzey) Sayısı
1C	DWORD	Saklı Sektörlerin Sayısı
20	DWORD	Yeni Toplam Sektör Uzunluğu
24	BYTE	Fiziksel Sürücü Numarası
25	BYTE	Kullanılmıyor
26	BYTE	Genişletilmiş Boot Id'si (Extended Boot Signature)
27	DWORD	Volümün Seri Numarası (Volume Serial Number)
2B	11 Byte	Volüm Etiket (Volume Label)
36	8 byte	Dosya Sisteminin Türü

Jmp Code: Boot sektörün hemen başında BPB bloğunu atlamak için kullanılan jmp makine komutu vardır.

OEM Yorum: Burada format programlarının yazdıkları 8 byte'lık bir yazı bulunur. Bu yazının ne olduğunun bir önemi yoktur. Silinse de sorun oluşmaz. Genellikle bu alana Microsoft işletim sisteminin versiyon numarasını yazmaktadır. (Burada MSDOS 5.0 görürseniz şaşırmayınız. Çünkü FAT dosya sistemlerinde BPB'nin son gelişmiş hali MSDOS 5.0 zamanında yapılmıştır)

Sektördeki Byte Sayısı: Bu alanda volümün bir sektöründe kaç byte olduğu bilgisi vardır. Burada her zaman 512 değerini görürüz. Gerçi formatla parametreleriyle eskiden flopy'ler sektördeki byte sayısı 512'den farklı olacak biçimde formatlanabiliyordu. Fakat 512 artık standart bir değerdir.

Cluster'ın Sektör Uzunluğu: Burada bir cluster'ın kaç sektörden oluştuğu bilgisi vardır. Küçük volümlerde bir cluster az sektörden büyük volümlerde çok sektörden oluşur.

Ayrılmış Sektörlerin Sayısı: Bazen boot sektörden hemen sonra FAT bölümü gelmeyebilir. Örneğin hard disklerde FAT'ın bir sonraki track başına hizalanması işleyişi hızlandırabilmektedir. Bazen de boot sektör yükleyici programı daha büyük olabilir. Bu durumda da FAT'ın yeri ötelenebilir. İşte bu alanda FAT'ın hangi mantıksal sektörden başladığı bilgisi vardır. Bu alandaki sayı 1 ise FAT hemen boot sektörden sonra başlar. Örneğin bu alandaki sayı 100 ise boot sektörden sonra 99 sektör başka bir amaçla kullanılmıştır ve FAT 100'üncü mantıksal sektörden başlamaktadır.

FAT Kopyalarının Sayısı: Microsoft FAT'ın iki kopyasını bulundurmaktadır. Aslında bu iki kopyanın hata düzeltme bakımından bir faydasının dokunduğunu söylemek de çok zordur. Ancak Microsoft geleneksel olarak hala FAT için iki kopya kullanmaktadır. Hatta Microsoft bu iki sayısını o kadar içselleştirmiştir ki boot sektör BPB bloğundaki değeri artık dikkate bile almamaktadır. O halde bu alanda biz hemen her zaman 2 değerini görürüz. FAT kopyalarının arasında hiç sektörel boşluk yoktur. Birinci kopyayı hemen ikinci kopya izler. Normal olarak işletim sistemi FAT'ın bir kopyası üzerinde güncelleme yapar yapmaz ikinci kopyası üzerinde de güncelleme yapar. Yani bu kopya normal durumda aynı içeriğe sahiptir. Bizim de iki kopyaya birden bakmak gibi bir gereksnimimiz olmaz. Burada yeniden vurgulamak gerekirse: Microsoft FAT'ın iki kopyasını biri bozulursa diğerinden faydalanılsın diye oluşturmuştur. Ancak pratikte bu durumun faydası olduğu söylenemez.

Root Girişlerinin Sayısı: Yukarıdaki şekilden de görüldüğü gibi FAT bölümünü "Root Dir" bölümü izlemektedir. "ROOT Dir" bölümü kök dizindeki "dizin girişlerinden oluşur". Bir dizin girişi FAT12 ve FAT16'da 32 byte uzunluğundadır. Bu alanda kaç tane dizin girişi olduğu bilgisi vardır. Buradan Root Dir bölümünün sektör sayısı dolaylı olarak hesaplanabilmektedir. Yani Root Dir bölümünün sektör uzunluğu şöyle hesaplanır:

Root Dir SektörUzunluğu = Root girişlerinin sayısı * 32 / 512

FAT12 ve FAT16 sistemlerinde kök dizindeki dosya sayısının en fazla bu alanda belirtilen miktarda olabileceğine dikkat ediniz. Ancak alt dizinlerdeki dosyalar için bir kısıtlama yoktur.

Volümdeki Toplam Sektör Sayısı: Bu alanda ilgili disk bölümündeki toplam sektör sayısı bulunur. Maalesef Microsoft ilk dönemlerde bu alanı WORD olarak ayırmıştır. Böylece volümdeki toplam sektör sayısı en fazla 2^{16} tane olabilmektedir. Volümün kapasitesi de $2^9 * 2^{16} = 2^{25} = 32\text{MB}$ ile sınırlıdır. Bu durum DOS 3.30'a kadar böyle devam etmiştir. DOS 3.30'da Microsoft BPB'nin sonuna (0x20 offsetine) DWORD olarak yeni bir toplam sektör sayısı alanı yerleştirmiştir. Böylece yeni DOS versiyonları (ve tabii ki günümüz Windows'larına kadarki tüm Windows versiyonları) önce 0x13 offset'inden WORD çekerler. Eğer orada 0 görürlerse bu kez 0x20 offset'inden DWORD çekerek toplam sektör sayısını elde ederler.

Ortam Belirleyicisi (Media Descriptor): Bu alanda medyanın nasıl bir medya olduğu bilgisi yer almaktadır. Bu alan artık kullanılan bir alan değildir. Eskiden bu alanın bir işlevi vardı. Fakat artık ciddi bir işlevi olduğu söylenemez. Bu alanda eğer F8 varsa ortam hard disk'tir. F0 varsa ortam floppy'dir. Biz burada artık hep F8 görürüz.

Fat'ın Bir Kopyasındaki Sektör Sayısı: Bu alanda FAT'ın bir kopyasındaki sektör sayısı bulunmaktadır. Böylelikle FAT'ın toplam sektör uzunluğu hesaplanabilir. FAT'ın iki kopyasının bulunduğunu anımsayınız.

Track Başına Düşen Sektör Sayısı: Bu alanın şimdilerde bir önemi kalmamıştır. Ancak fiziksel koordinat sisteminin kullanıldığı (INT 13h) DOS sistemlerinde bu alanda bir track'in kaç sektör diliminden oluştuğu bilgisi bulunuyordu.

Diskteki Kafa (yüzey) Sayısı: Bu alanın şimdilerde bir önemi kalmamıştır. Ancak fiziksel koordinat sisteminin kullanıldığı (INT 13h) DOS sistemlerinde bu alanda diskteki toplam kafa sayısı bulunmaktadır.

Saklı Sektörlerin (Hidden Sectors) Sayısı: Burada boot sektörün ilgili diskin başından itibaren kaçınıcı sektörde olduğu bilgisi vardır. Örneğin buradaki değer 0xE8DCB800 (3906779136) olsun. Bu FAT bölümünün boot sektörü diskin başından itibaren 3906779136'ncı sektöründedir.

Yeni Toplam Sektör Uzunluğu: Bu alanda yukarıda da belirtildiği gibi daha uzun bir toplam sektör sayısı bilgisi DWORD olarak tutulmaktadır.

Fiziksel Sürücü Numarası: Burada IO kontrol kartlarının arayüzleri için fiziksel sürücüye erişmekte kullanılan numara bulunmaktadır. Birinci hard disk için bu numara 0x80, ikinci hard disk için 0x81 biçiminde devam etmektedir. Yani volüme bir hard diskteyse o hard diskin kaçınıcı hard disk olduğu bu alanda yazmaktadır.

Genişletilmiş Boot Id'si (Extended Boot Signature): Buradaki değer ilgili disk bölümündeki dosya sisteminin türü hakkında bilgi verir. 0x29 FAT dosya sistemi olduğu anlamına gelmektedir.

Volümün Seri Numarası (Volume Serial Number): Volüm formatlanırken ona 4 byte'lık rastgele bir seri numarası verilir. Eskiden bu seri numarası floppy zamanlarında bazı programlar tarafından floppy'nin değiştirilip değiştirilmediğini kontrol amacıyla kullanılırdı. Bugün pek bir kullanım amacı kalmamıştır. (Bu değer tekil bir değer değildir. Kopya koruma gibi amaçlarla bu değerden faydalanılması doğru yaklaşım değildir.)

Volüm Etiketini (Volume Label): Bu alanda 11 byte'lık volüme etiketi bulunmaktadır. Ancak volüm etiketi FAT dosya sistemlerinde kök dizinde "Volume Label" özniteliğine (attribute) sahip bir dizin girişi olarak da tutulmaktadır. DOS ve Windows uzun süredir volüm etiketini BOOT sektör BPB bloğundan değil bu dosyadan almaktadır.

Dosya Sisteminin Türü: Burada dosya sisteminin türü yazı olarak tutulmaktadır (örneğin "FAT32", "FAT16", "FAT12" gibi). Dolayısıyla dosya sisteminin türü hemen disk editörle görsel olarak belirlenebilir.

BPB bloğundaki bilgilerden hareketle tüm data alanlarının hangi mantıksal (relative) sektörden başlayıp ne uzunlukta sürdüğü belirlenebilir:

1) Boot sektör volümün ilk sektörüdür. Onun yeri bellidir.

2) FAT bölümü BPB'deki 0x0E offsetinden çekilen "Ayrılmış sektörlerin Sayısı" mantıksal sektöründen başlar. Uzunluğu BPB'deki FAT'ın Bir Kopyasının Sektör Sayısı ile FAT Kopyalarının Sayısı alanındaki bilgilerin çarpımıyla bulunabilir.

3) Root Dir bölümü FAT'ın bitişinden başlar ve uzunluğu BPB'deki Root Dir Girişlerinin Sayısı alanındaki değerden faydalanılarak bulunur.

4) Data bölümü Root Dir bölümünün bitişinden başlar.

Boot Sektör Yükleyici Programı

Boot Sektör yükleyici programı boot sektör'de hemen BPB'nin bittiği yerden başlar ve boot sektörün sonuna kadar devam eder. Boot sektör yükleyici programı sembolik makine dilinde yazılıp buraya yerleştirilmiştir. Boot işlemi sırasında eğer boot edilecek disk bölümü bu bölümse boot yükleyici programı (boot loader) FAT dosya sisteminin boot sektörünü diskten okuyarak kontrolü buradaki programa aktarır. Bu program işletim sisteminin yüklenme sürecini başlatır.



Eğer bu FAT disk bölümü bootable değilse yani yüklenecek bir işletim sistemine sahip değilse buradaki küçük yükleyici program bir error mesajı ekrana basarak sistemi yeniden başlatır.

FAT Dosya Sisteminde BPB Bilgilerinin Elde Edilmesi

Yukarıda da belirtildiği gibi FAT dosya sisteminin kalbi boot sektörün başındaki BPB (Bios Parameter Block) alanıdır. O alan okunup yorumlanmadıktan sonra bu dosya sistemi üzerinde işlemler yapılamaz. Bu nedenle sistem programcısı öncelikle BPB alanını okuyarak bir yapıya yerleştirmelidir. BPB alanını okuyup bir yapıya yerleştiren bir fonksiyon şöyle yazılabilir:

```
/* FatSys.h */

#ifndef FATSYS_H_
#define FATSYS_H_

#include <Windows.h>

/* Symbolic constants */

#define FILE_INFO_LENGTH 32

/* Type Declaration */

typedef struct tagBPB {
    HANDLE hVolume;          /* volume */
    WORD fatLen;             /* number of sectors of FAT (A) */
    WORD rootLen;           /* number of sectors of ROOT */
    WORD fatCopyNum;        /* number of copies of FAT (A) */
    DWORD totalSec;        /* total sector (A) */
    WORD bps;               /* byte per sector(A) */
    WORD spc;               /* sector per cluster(A) */
};
```

```

WORD reservedSect;    /* reserved sector(A) */
BYTE medDes;         /* media descriptor byte(A) */
WORD spt;            /* sector per track(A) */
WORD rootEntryNum;   /* root entry(A) */
WORD headNum;        /* number of heads(A) */
WORD hidNum;         /* number of hidden sector(A)*/
WORD tph;            /* track per head */
WORD fatOrigin;      /* fat directory location */
WORD rootOrigin;     /* root directory location */
WORD dataOrigin;     /* first data sector location */
DWORD serialNumber;  /* Volume Serial Number (A) */
BYTE volumeName[12]; /* Volume Name (A) */
} BPB;

/* Function Prototypes */

BOOL GetBPB(HANDLE hVolume, BPB *pBPB);

#endif

/* FatSys.c */

#include <stdio.h>
#include <windows.h>
#include "DiskIO.h"
#include "FatSys.h"

BOOL GetBPB(HANDLE hVolume, BPB *pBPB)
{
    BYTE bootSect[BYTE_PER_SECTOR];

    if (!ReadSector(hVolume, 0, 1, bootSect))
        return FALSE;

    pBPB->hVolume = hVolume;
    pBPB->bps = *(WORD *)(bootSect + 0x0B);
    pBPB->spt = *(BYTE *)(bootSect + 0x0D);
    pBPB->reservedSect = *(WORD *)(bootSect + 0x0E);
    pBPB->fatLen = *(WORD *)(bootSect + 0x16);
    pBPB->rootLen = *(WORD *)(bootSect + 0x11) * FILE_INFO_LENGTH / pBPB->bps;
    pBPB->fatCopyNum = *(BYTE *)(bootSect + 0x10);
    if (*(WORD *)(bootSect + 0x13))
        pBPB->totalSec = *(WORD *)(bootSect + 0x13);
    else
        pBPB->totalSec = *(DWORD *)(bootSect + 0x20);
    pBPB->medDes = *(bootSect + 0x15);
    pBPB->spt = *(WORD *)(bootSect + 0x18);
    pBPB->rootEntryNum = *(WORD *)(bootSect + 0x11);
    pBPB->headNum = *(WORD *)(bootSect + 0x1A);
    pBPB->hidNum = *(WORD *)(bootSect + 0x1C);
    pBPB->tph = (WORD)(pBPB->totalSec / pBPB->spt / pBPB->headNum);
    pBPB->fatOrigin = pBPB->reservedSect;
    pBPB->rootOrigin = pBPB->reservedSect + pBPB->fatLen * pBPB->fatCopyNum;
    pBPB->dataOrigin = pBPB->rootOrigin + pBPB->rootLen;
    pBPB->serialNumber = *(DWORD *)(bootSect + 0x27);
    memcpy(pBPB->volumeName, bootSect + 0x2B, 11);
    pBPB->volumeName[11] = '\\0';

    return TRUE;
}

```

Şöyle bir kodla test işlemi yapılabilir:


```

/* Test.c */

#include <stdio.h>
#include <Windows.h>
#include "DiskIO.h"
#include "FatSys.h"

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)
{
    BYTE buf[512];
    HANDLE hVolume;
    BPB bpb;

    if ((hVolume = OpenVolume(6, GENERIC_READ)) == INVALID_HANDLE_VALUE)
        ExitSys("OpenVolume", EXIT_FAILURE);

    if (!GetBPB(hVolume, &bpb))
        ExitSys("GetBPB", EXIT_FAILURE);

    printf("%ld\n", bpb.totalSec);
    printf("%d\n", bpb.fatOrigin);
    printf("%d\n", bpb.rootOrigin);
    /* .... */

    CloseHandle(hVolume);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

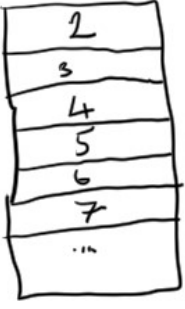
    exit(status);
}

```

Cluster'ların Numaralandırılması

FAT dosya sisteminde Data bölümü clusterlar'dan oluşmaktadır. Her cluster'a ilki 2'den başlamak üzere (0 ve 1 numaralı cluster'lar kullanılmamaktadır) bir cluster numarası karşılık düşürülmüştür. Yani data bölümünün hemen başında 2 numaralı cluster bulunmaktadır. Örneğin bir cluster'ın 4 sektörden oluştuğunu varsayalım. Bu durumda data bölümünün ilk 4 sektörü 2 numaralı cluster'a, sonraki 4 sektörü 3 numaralı cluster'a karşılık gelecektir.

Data Bölümü



Bir cluster'ı okuyup yazan fonksiyonlar oluşturabiliriz. Örneğin:

```
BOOL ReadCluster(const BPB *pBPB, DWORD clu, void *buf)
{
    DWORD sect = pBPB->dataOrigin + (clu - 2) * pBPB->spc;

    return ReadSector(pBPB->hVolume, sect, pBPB->spc, buf);
}

BOOL WriteCluster(const BPB *pBPB, DWORD clu, const void *buf)
{
    DWORD sect = pBPB->dataOrigin + (clu - 2) * pBPB->spc;

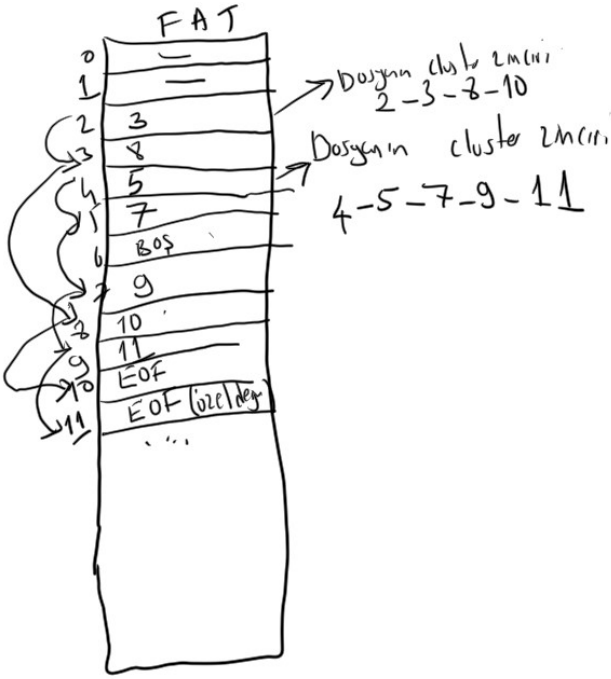
    return WriteSector(pBPB->hVolume, sect, pBPB->spc, buf);
}
```

Burada görüldüğü gibi önce okunacak ya da yazılacak cluster'ın data bölümünün kaçıncı sektöründen başladığı hesaplanmıştır. Sonra da okuma ve yazma işlemleri ReadSector ve WriteSector fonksiyonlarıyla yapılmıştır.

FAT Bölümü

Daha önceden de belirtildiği gibi dosyaları oluşturan cluster'lar data bölümünde dağınık olarak bulunabilirler. İşte hangi dosyaların data bölümünün hangi cluster'larında olduğu bilgisi FAT tablosunda tutulmaktadır. FAT tablosunda bütün dosyaların parçalarının tek tek hangi cluster'larda olduğu bilgisi vardır. Örneğin "a.dat" isimli dosyanın cluster'ları sırasıyla 12, 23, 24, 36, 41 olabilir. Bu numaralara ilgili dosyanın cluster zinciri (cluster chain) denilmektedir.

FAT bağlı listelerden oluşan bir yapıdır. Bir dosyanın ilk cluster bilgisi dizin girişinden alınır (bu konu ileride ele alınmaktadır). FAT tablosu FAT elemanlarından oluşmaktadır. Her FAT elemanının ilki sıfırdan başlamak üzere (ancak 0 ve 1'in kullanılmadığını anımsayınız) bir numarası vardır. Bir dosyanın FAT zinciri şöyle bulunur: Öncelikle dosyanın ilk cluster'ının numarası bilinmek zorundadır. Bu bilgi dizin girişlerinden elde edilmektedir. Sonra FAT'te o fat elemanına gidilir. Orada bulunan değer dosyanın sonraki cluster'ını belirtmektedir. Bu biçimde zincir bağlı listede takip edilir. Özel bir değer görüldüğünde (FFFF gibi) durulur. Örneğin aşağıdaki çizimde iki dosyanın FAT zinciri gösterilmektedir.



FAT bölümü FAT sektörlerinden FAT sektörleri de FAT elemanlarından oluşmaktadır. Pekiyi bir FAT elemanı kaç bit uzunluğundadır? İşte FAT dosya sisteminin FAT12, FAT16 ve FAT32 olmak üzere üç değişik biçimi vardır. FAT12’de bir FAT elemanı 12 bit, FAT16’da 16 bit ve FAT32’de 32 bittir. Bu durumda örneğin FAT16 olan bir sistemin FAT bölümünün bir kısmı aşağıda verilmiş olsun:

	0	1	2	3	4	5	6	7
00001000	F8 FF	FF FF	FF FF	FF FF	05 00	06 00	07 00	08 00
00001010	09 00	0A 00	0B 00	0C 00	0D 00	0E 00	0F 00	10 00
00001020	11 00	12 00	13 00	14 00	15 00	16 00	17 00	18 00
00001030	19 00	1A 00	1B 00	1C 00	1D 00	1E 00	1F 00	20 00
00001040	21 00	22 00	23 00	24 00	25 00	26 00	27 00	28 00
00001050	29 00	2A 00	2B 00	2C 00	2D 00	2E 00	2F 00	30 00
00001060	31 00	32 00	33 00	34 00	35 00	36 00	37 00	38 00
00001070	39 00	3A 00	3B 00	3C 00	3D 00	3E 00	3F 00	40 00
00001080	41 00	42 00	43 00	44 00	45 00	46 00	47 00	48 00
00001090	49 00	4A 00	4B 00	4C 00	4D 00	4E 00	4F 00	50 00
000010A0	51 00	52 00	53 00	54 00	55 00	56 00	57 00	58 00
000010B0	59 00	5A 00	5B 00	5C 00	5D 00	5E 00	5F 00	60 00
000010C0	61 00	62 00	63 00	64 00	65 00	66 00	67 00	68 00
000010D0	69 00	6A 00	6B 00	6C 00	6D 00	6E 00	6F 00	70 00
000010E0	71 00	72 00	73 00	74 00	75 00	76 00	77 00	78 00
000010F0	79 00	7A 00	7B 00	7C 00	7D 00	7E 00	7F 00	80 00
00001100	81 00	82 00	83 00	84 00	85 00	86 00	87 00	88 00
00001110	89 00	8A 00	8B 00	8C 00	8D 00	8E 00	8F 00	90 00
00001120	91 00	92 00	93 00	94 00	95 00	96 00	97 00	98 00
00001130	99 00	9A 00	9B 00	9C 00	9D 00	9E 00	9F 00	A0 00
00001140	A1 00	A2 00	A3 00	A4 00	A5 00	A6 00	A7 00	A8 00
00001150	A9 00	AA 00	AB 00	AC 00	AD 00	AE 00	AF 00	B0 00
00001160	B1 00	B2 00	B3 00	B4 00	B5 00	B6 00	B7 00	B8 00

FAT’ın ilk iki elemanında özel değerler bulunmaktadır. İşletim sistemi bu özel değerleri (magic numbers) kontrol edebilir. (Eskiden DOS ve Windows bunu kontrol ediyordu. Artık Windows kontrol etmemektedir.) Bu değerler yanlış ise FAT bölümünün bozulmuş olduğu sonucu çıkarılabilmektedir. FAT12’de özel değer F8 FF FF (Burada F8 ortam betimleyicisidir), FAT16’da F8 FF FF FF ve FAT32’de F8 FF FF FF FF FF FF FF biçimindedir. Yukarıdaki FAT örneğinde 2 ve 3 numaralı cluster’larda tek cluster’lık küçük iki dosya vardır. Çünkü bu FAT elemanlarındaki FF FF değeri özel bir değerdir ve EOF anlamına gelir. Yani FAT16’da cluster zincirini izlerken biz en sonunda FF FF ile karşılaşırız. Bu durum zincirin bittiğini gösterir.

FAT elemanlarındaki bazı özel değerler bazı özel anlamlara gelmektedir. Bu özel değerler ve anlamları aşağıda tabloda betimlenmektedir:

FAT Elemanındaki Değer (HEX)	Özel Anlamı
FAT12: 000 FAT16: 0000 FAT32: 00000000	Boş cluster. Bu cluster boştur. Dolayısıyla işletim sistemi tarafından tahsis edilebilir. Volüm formatlandığında FAT elemanlarında sıfır değerleri vardır.
FAT12: 001 FAT16: 0001 FAT32: 00000001	1 numaralı cluster kullanılmamaktadır. Dolayısıyla FAT tablosu içerisinde normal olarak böyle bir FAT elemanı bulunmaz
FAT12: [002:FEF] FAT16: [0002:FFEF] FAT32: [00000002:0FFFFFFEF]	Sonraki cluster numarası. Bir FAT elemanında bulunan normal değer cluster zincirindeki sonraki elemanın değeridir.
FAT12: [FF0:FF6] FAT16: [FFF0:FFF6] FAT32: [0FFFFFF0:0FFFFFF6]	Reserved. Bu değerler bazı sistemler tarafından onlara özgü bazı amaçlarla kullanılsın diye ayrılmıştır.
FAT12: FF7 FAT16: FFF7 FAT32: 0FFFFFF7	Bad cluster. Aşağı seviyeli formatlama yapılırken eğer bir sektörde sorun çıkarsa o sektörün içinde bulunduğu cluster "bad cluster" olarak işaretlenir. İşletim sistemi de oraya dosya yerleştirmes.
FAT12: [FF8:FFF] FAT16: [FFF8:FFFF] FAT32: [0FFFFFF8:0FFFFFFF]	Son cluster. Bu değerleri dosyanın cluster zincirindeki son cluster'ı belirtmektedir.

FAT32'de sanıldığı gibi aksine her FAT elemanı 32 bit değil 28 bittir. Dolayısıyla FAT tablosunda FAT elemanlarının yüksek anlamlı 4 biti (yani yüksek anlamlı hex dijiti) her zaman sıfırdır.

FAT Elemanlarının Elde Edilmesi

FAT tablosunun aşağıdaki gibi unsigned char türünden bir dizi içerisine okunduğunu varsayalım:

```
unsigned char fat[FAT_SIZE];
```

Şimdi clu numaralı FAT elemanını elde eden bir fonksiyonu 16 bit, 12 bit ve 32 bit FAT için yazalım.

```
WORD GetNextCluster16(BYTE *pFat, WORD clu)
{
    return *(WORD*)(pFat + clu * 2);
}
```

FAT16'da bir FAT elemanı 2 byte olduğu için cluster numarası 2 ile çarpılmıştır. Şimdi aynı fonksiyonu FAT12 için yazalım. FAT12'de bir FAT elemanı 12 bit yani 3 hex digit'tir. Fat elemanının FAT'teki konumuna cluster numarasının 1.5 ile çarpılmasıyla gidilir. Ancak oradan çekilen WORD değerinin hangi 4 bitlik kısmının atılacağı cluster numarasının tek mi çift mi olduğuna bağlıdır. Örneğin aşağıda 12 bit için girişler verilmektedir:


```

static WORD chain[MAX_FAT16_CHAIN];
WORD clu = firstClu, i;

for (i = 0; clu < 0xFFF8; ++i) {
    chain[i] = clu;
    clu = GetNextCluster16(pFat, clu);
}
chain[i] = 0;

return chain;
}

DWORD *GetClusterChain32(BYTE *pFat, DWORD firstClu)
{
    static DWORD chain[MAX_FAT32_CHAIN];
    WORD clu = firstClu, i;

    for (i = 0; clu < 0xFFF8; ++i) {
        chain[i] = clu;
        clu = GetNextClu32(pFat, clu);
    }
    chain[i] = 0;

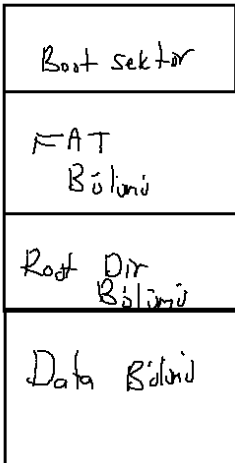
    return chain;
}

```

FAT12 ve FAT16 Dosya Sistemlerinde Dizin Girişleri ve Dizinlerin Organizasyonu

Bir dosyanın tüm bilgileri FAT12 ve FAT16'da 32 byte'lık dizin girişlerinde saklanmaktadır. FAT32'de uzun dosya isimleri de devreye sokulmuştur. Bu nedenle FAT32'nin dizin giriş formatı biraz daha farklıdır. Uzun dosya isimlerinin 32'lik girişlerle nasıl saklandığı daha sonra ele alınacaktır. Örneğin Windows'taki FindFirstFile ve FindNextFile API fonksiyonları dosya bilgilerini bu 32 byte'lık girişlerden elde eder.

Volümün RootDir bölümünde kök dizindeki dosyalara ilişkin 32'lik girişler tutulmaktadır. Bu girişlerin sayısının BPB alanı içerisinde tutulduğunu anımsayınız. RootDir bölümünün sektör sayısı sınırlıdır. Bu sayı formatlama sırasında belirlenmektedir.



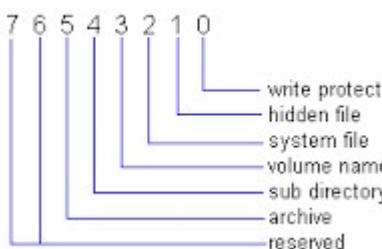
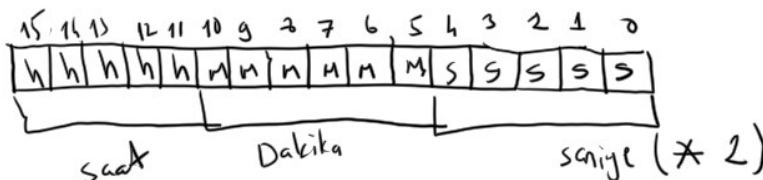
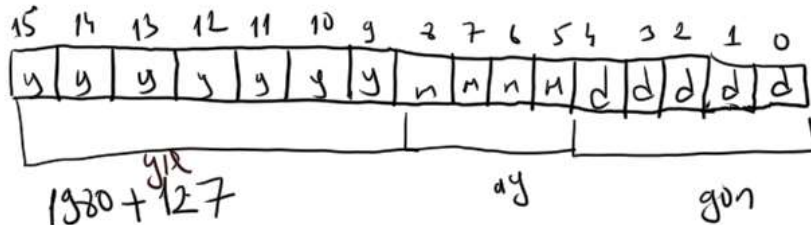
Böylece FAT12 ve FAT16'da kök dizinde belli miktarda dizin girişi bulunabilmektedir. Ancak FAT32'de organizasyon biraz değiştirilmiştir. FAT32'de kök dizinde istenildiği kadar çok dizin girişi bulundurulabilmektedir

FAT dosya sistemlerinde dosya ile dizin arasında organizasyonel bir fark yoktur. Dizinlerin de tıpkı dosyalar gibi cluster zincirleri vardır. Fakat normal dosyaların cluster'larında o dosyaların içindeki bilgiler bulunurken dizin

dosyalarının cluster'larında o dizindeki dosyalara ilişkin 32'lik dizin girişleri bulunmaktadır. Örnek bir 32'lik dizin girişini aşağıda görüyorsunuz:

```
54 45 53 54 20 20 20 20 54 58 54 20 18 04 D0 65 | TEST TXT ..Ğe
90 48 90 48 00 00 D6 65 90 48 06 00 46 00 00 00 | .H.H..Öe.H..F...
```

32 byte'lık dizin girişinin formatı şöyledir:

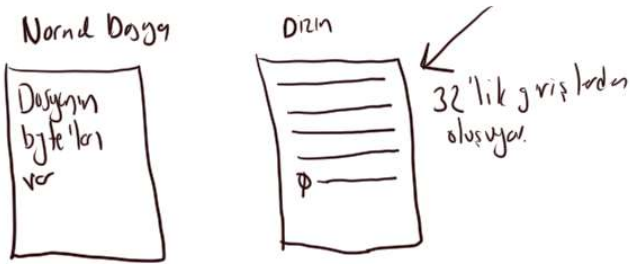
Offset (Hex)	Uzunluk	Açıklama
[00:0A]	11 byte	Dosyanın ismi ve uzantısı. FAT12 ve FAT16 sistemlerinde bir dosyanın ismi 8 karakteri, uzantısı da 3 karakteri geçemiyordu. Eğer isim ya da uzantı 8 ve 3'ten kısaysa geri kalan alanlar boşluk karakteriyle (0x20) doldurulmaktadır.
0B	BYTE	Dosyanın özelliği. Dosya özelliği kursun başlarında da ele alındığı gibi bit bit anlamlı bir alandır.  <p>Görüldüğü gibi girişin bir dosya mı yoksa bir dizin mi olduğu yalnızca özellik byte'ının 4'üncü bitinden anlaşılmaktadır.</p>
[0C:15]	10 byte	Reserved. Bu alan FAT 12 ve FAT16'da reserved bırakılmıştır. Ancak uzun dosya isimlerinin desteklendiği VFAT sisteminde bu alan başka amaçla kullanılmaktadır.
16	WORD	Dosyanın son değiştirildiği zamanı belirtir. Burası bir bit alanı biçiminde aşağıdaki gibi kodlanmıştır. Saniye olarak yalnızca çift saniyelerin tutulduğuna dikkat ediniz. Saniye kısmındaki değer iki ile çarpılmalıdır. 
18	WORD	Burada dosyanın son değiştirildiği tarih bilgisi bulunmaktadır. Bu alandaki bilgi de aşağıdaki gibi bit bit kodlanmıştır. Yıl için orijin noktası 1980'dir.  <p>Yani yıl alanında yazan değere 1980 toplamak gerekir.</p>

2A	WORD	Dosyanın ilk cluster numarası. Bilindiği gibi dosyanın FAT zincirinin izlenebilmesi için onun ilk cluster numarasının bilinmesi gerekir. İşte dosyaların ilk cluster numaraları dizin girişlerinde bulunmaktadır.
2C	DWORD	Dosyanın byte uzunluğu. Bu alanda dosyanın DWORD olarak byte uzunluğu bulunmaktadır.

Dosya silindiğinde dosyanın FAT zinciri sıfırlanır. Ancak dizin girişi sıfırlanmaz. Yalnızca onun ilk byte'ı 0xE5 yapılır. Artık 0xE5 ile başlayan dizin girişlerine yeni bir dosya çekilebilecektir. Pekiyi işletim sistemi dizin girişlerinin listesini almak için 32'lik dizin girişlerine bakarken nerede duracaktır? İşte 32'lik dizin girişlerinin ilk byte'ı eğer 0 ise bu durum dizin listesinin sonuna geldiği anlamına gelmektedir.

FAT Dosya Sisteminde Alt Dizinlerin Organizasyonu

Aslında dizinlerle dosyalar arasında organizasyonel olarak hiçbir fark yoktur. Dizinler de tamamen dosyalar gibi 32'lik girişlere ve FAT zincirine sahiptir. Dizinlerle dosyalar arasındaki en önemli fark dosyaların içerisinde gerçek dosya bilgileri varken dizinlerin içerisinde o dizinlerdeki dosyaların hangi dosyalar olduğunu gösteren 32'lik dizin girişlerinin olmasıdır. Dizinler de çok fazla dosya içerebilirler. Bu durumda onların da cluster zincirleri birden fazla cluster'dan oluşabilmektedir.

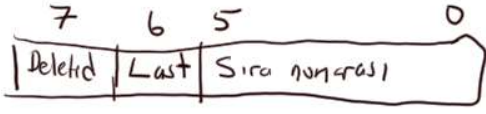


Windows ve UNIX/Linux sistemlerinde kullanılan dosya sistemlerinde bir alt dizin yaratıldığında işletim sistemi o alt dizinin içerisinde "." ve ".." isimli iki alt dizin girişini otomatik olarak oluşturmaktadır. Gerçekten de FAT dosya sisteminde bir alt dizin yarattığımızda onun ilk iki 32'lik girişinde "." ve ".." dizinlerini görürüz. "." dizini bulduğumuz dizinin 32'lik girişinin aynısını, ".." girişi de üst dizinin dizin girişinin aynısını barındırır.

VFAT Dizin Girişleri

Klasik FAT12 ve FAT16 sistemlerinde yukarıda da gördüğümüz gibi dizin girişleri 32 byte uzunluğundadır. 32 byte'lık dizin girişlerinde dosya isimleri ve uzantıları 8 + 3 karakter olabilmektedir. Ayrıca 32'lik dizin girişlerinde dosyalar için yalnızca son değiştirilme tarihi ve zamanının tutulduğunu anımsayınız. Microsoft Windows sistemleriyle birlikte FAT'teki 32'lik dizin girişlerini geçmişe doğru uyumu koruyacak biçimde genişletmiştir. Bu özelliğe VFAT (Virtual FAT) denilmektedir. VFAT yeni bir dosya sistemi değildir. Uzun dosya isimlerinin dizin girişlerinde bulunabilmesi için 32'lik dizin girişlerinin genişletilmiş bir halidir. VFAT geçmişe doğru uyumludur. Ayrıca VFAT'teki dosya isimleri UNICODE karakterlerden oluşturulabilmektedir. Dosya isimlerinde boşluk karakterleri de VFAT ile birlikte bulunabilir hale getirilmiştir. Ayrıca dosya isminde birden fazla '.' karakteri de bulundurulabilmektedir. VFAT sisteminde her uzun dosya ismi için bir de onun kısaltması olan 32'lik ayrı bir giriş de bulundurulmaktadır. Bu kısa isme ilişkin 32'lik giriş uzun isimlere ilişkin 32'lik girişlerin sonundadır.

Uzun dosya girişleri birden fazla 32'lik girişlerde tutulur. 32'lik girişler ters dizilmiştir. Yani dosya isimlerinin sonraki karakterlerine karşı gelen 32'lik girişler önde bulunur. Uzun dosya girişlerinin 32'lik bir parçasının formatı şöyledir:

Offset (Hex)	Uzunluk	Açıklama
0	BYTE	32'lik girişlerin sıra numarası ve tahsisat durumunu belirtir. Buradaki byte'ın bitleri şu biçimde organize edilmiştir: 
[01:0A]	10 byte	Burada her biri 2 byte olmak üzere dosya isminin beş karakteri bulunur.
0B	BYTE	Burada klasik eskiye uyumlu 32'lik girişte olduğu gibi dosya özelliği bilgisi vardır.
0C	BYTE	reserved
0D	BYTE	Checksum. Burada bilgilerin bozulup bozulmadığının anlaşılması için bir checksum değeri tutulmaktadır.
[0E:19]	12 byte	Bu alanda dosya isminin sonraki 6 karakteri UNICODE olarak tutulmaktadır.
1A	BYTE	Burada her zaman 0 değeri bulunur.
[1C:1F]	4 byte	Dosyanın sonraki UNICODE olarak iki karakteri bulunur.

Görüldüğü gibi bir 32'lik giriş uzun dosya isminin 13 karakterini tutabilmektedir. Örneğin "THIS_IS_A_LONG_FILE_NAME.TXT" isimli bir dosya ismi toplam 28 karakterdir (nokta isme dahildir ve herhangi bir karakter gibidir.) Bu durumda bu dosya için 13 + 13 + 2 biçiminde toplam 32'lik 3 giriş oluşturulacaktır. Bu üç girişin hemen aşağısında bu dosyanın kısa ismine ilişkin bir 32'lik giriş daha bulundurulur.

43 58 00 54 00 00 00 FF	FF FF FF 0F 00 FB FF FF	CX.T...ÿÿÿÿ..ûÿÿ
FF FF FF FF FF FF FF FF	FF FF 00 00 FF FF FF FF	ÿÿÿÿÿÿÿÿÿÿÿÿ..ÿÿÿÿ
02 47 00 5F 00 46 00 49	00 4C 00 0F 00 FB 45 00	.G._.F.I.L...ûE.
5F 00 4E 00 41 00 4D 00	45 00 00 00 2E 00 54 00	.N.A.M.E.....T.
01 54 00 48 00 49 00 53	00 5F 00 0F 00 FB 49 00	.T.H.I.S._...ûI.
53 00 5F 00 41 00 5F 00	4C 00 00 00 4F 00 4E 00	S._.A._.L...O.N.
54 48 49 53 5F 49 7E 31	54 58 54 20 00 00 70 62	THIS_I~1TXT ..pb
98 48 98 48 00 00 71 62	98 48 09 00 1B 00 00 00	~H~H..qb~H.....

Buradaki 32'lik girişlerin sıra numaralarının (32'lik girişlerin ilk byte'ları) şöyle oluşturulduğına diikat ediniz:

```

43      XT
02      GFILE_NAME_T
01      THIS_ISALON
???     THIS_IS~1TXT

```

Yol İfadelerinin Çözümlemesi (Pathname Resolution)

Bir yol ifadesi verildiğinde işletim sisteminin yol ifadesiyle hedeflenen dosyaya ilişkin bilgileri ele geçirmesi gerekir. Bu sürece yol ifadelerinin çözümlemesi (pathname resolution) denilmektedir. Anımsanacağı gibi yol ifadeleri mutlak (absolute) ve görelî (relative) olmak üzere ikiye ayrılmaktadır. Eğer yol ifadesinin başında (sürücü ismi varsa bundan sonrasında) ters bölü varsa (UNIX/Linux sistemlerinde düz bölü) bu tür yol ifadelerine mutlak yol ifadeleri denilmektedir. Eğer yol ifadelerinin başında bir ters bölü (ya da düz bölü) yoksa böyle yol ifadelerine de görelî yol ifadeleri denilmektedir. Mutlak yol ifadeleri kök dizinden itibaren

çözülür. Göreli yol ifadeleri ise prosesin kontrol bloğunda tutulan "çalışma dizininden (current working directory)" itibaren çözülür. Şimdi burada FAT dosya sistemi için mutlak bir yol ifadesinin nasıl çözümleneceğine ilişkin bir örnek verelim. Yol ifademiz "\\A\\B\\C\\D.TXT" olsun. İşletim sistemi burada hedef olarak D.TXT dosyasının bilgilerine erişecektir. Bunun için önce işletim sistemi kök dizine gider. FAT dosya sisteminde kök dizinin yeri bellidir. Oradaki 32'lik girişlerde sıralı arama yöntemiyle A dizin girişini arar. Bu dizin girişini bulduğunda onun bir dizine ilişkin olduğunu "özellik byte'ına" bakarak doğrular. A dizin girişinin cluster zincirini elde eder. A'nun cluster'ları içerisindeki bilgileri 32'lik dizin girişi gibi değerlendirir. Bu sefer A'nın cluster'ları içerisinde B girişini arar. Bulursa bunun da bir dizin olduğunu doğrular. Bu sefer de B'nin cluster zincirini elde eder. B'nin cluster'larında da C'yi arar. C'nin cluster zincirinde de benzer biçimde D.TXT'yi arayıp bulacaktır.

Modern işletim sistemleri yol ifadesinin çözümlenmesi sırasında eriştikleri dizin girişlerini bir cache sistemi içerisinde saklamaktadır. Bu cache sistemine sıklıkla "dentry cache" denilmektedir. "Dentry Cache" genellikle bir hash tablosu biçiminde oluşturulmaktadır. Hash tablosuna anahtar olarak yol ifadesi verilir. Değer olarak da dizin girişi bilgileri elde edilir. Böylece son erişilen dosyalara ilişkin girişler cache'te saklandığı için onlara erişilmek istendiğinde artık hiç disk okuması yapılmadan doğrudan giriş elde edilebilmektedir.

Yol ifadelerinin çözümlenmesine ilişkin bir örnek aşağıdaki gibi olabilir. Aşağıdaki örnekte hiçbir iyileştirme yapılmamıştır. Örneğin her defasında FAT bölümünün tamamı okunmaktadır. Ancak yol ifadelerinin çözümlenmesi için bir fikir vermesi bakımından örnek incelenmelidir:

```
/* PathNameResolution.h */

#ifndef PATHNAMERESOLUTION_H_
#define PATHNAMERESOLUTION_H_

#include <stdio.h>
#include <Windows.h>
#include "FatSys.h"

/* Function Prototypes */

BOOL PathNameLookup(const char *path, DIRENTRY *dirEntry);

#endif

/* PathNameResolution.c */

#include <string.h>
#include <ctype.h>
#include "DiskIO.h"
#include "PathNameResolution.h"

/* static Function Prototypes */

DIRENTRY *findDirEntry(void *cluster, size_t size, const char *fileName);

char g_cwd[MAX_PATH] = "g:\\";

BOOL PathNameLookup(const char *path, DIRENTRY *dirEntry)
{
    int drive;
    char absPath[MAX_PATH];
    char *pathPos, *pathComp, *dirEntries, *cluCont;
    size_t dirEntrySize, cluSize;
    HANDLE hVolume;
    BPB bpb;
    DIRENTRY *dentry;
    BOOL foundFlag;
```

```

WORD clu;
BYTE *fat;

if (path[1] == ':') {
    strcpy(absPath, path);
    drive = toupper(path[0]) - 'A';
    if (path[2] == '\\')
        pathPos = absPath + 3;
    else
        pathPos = absPath + 2;
}
else if (path[0] == '\\') {
    strcpy(absPath, path);
    drive = toupper(g_cwd[0]) - 'A';
    pathPos = absPath + 1;
}
else {
    sprintf(absPath, "%s\\%s", g_cwd, path);
    drive = toupper(absPath[0]) - 'A';
    pathPos = absPath + 3;
}

if ((hVolume = OpenVolume(drive, GENERIC_READ)) == INVALID_HANDLE_VALUE)
    return FALSE;

if (!GetBPB(hVolume, &bbp))
    return FALSE;

dirEntrySize = bpb.rootLen * bpb.bps;
if ((dirEntries = (char *)malloc(dirEntrySize)) == NULL)
    return FALSE;
if (!ReadSector(hVolume, bpb.rootOrigin, bpb.rootLen, dirEntries))
    return FALSE;

if ((pathComp = strtok(pathPos, "\\")) == NULL)
    return FALSE;
if ((dentry = findDirEntry(dirEntries, dirEntrySize, pathComp)) == NULL) {
    printf("cannot find!..\n");
    return FALSE;
}

if ((fat = (BYTE *)malloc(bpb.fatLen * bpb.bps)) == NULL)
    return FALSE;

if (!ReadSector(hVolume, bpb.fatOrigin, bpb.fatLen, fat))
    return FALSE;

cluSize = bpb.spc * bpb.bps;
if ((cluCont = (char *)malloc(cluSize)) == NULL)
    return FALSE;

while ((pathComp = strtok(NULL, "\\")) != NULL) {
    clu = dentry->firstClu;

    foundFlag = FALSE;
    while (clu < 0xFFF8) {
        if (!ReadCluster(&bbp, clu, cluCont))
            return FALSE;
        if ((dentry = findDirEntry(cluCont, cluSize, pathComp)) == NULL)
            clu = GetNextClu16(fat, clu);
        else {
            foundFlag = TRUE;
            break;
        }
    }
}

```



```

        if (!foundFlag)
            return FALSE;
    }

    *dirEntry = *dentry;;

    return TRUE;
}

DIRENTRY *findDirEntry(void *cluster, size_t size, const char *fileName)
{
    DIRENTRY *dirEntry = (DIRENTRY *)cluster;
    unsigned int i, k;
    char fname[11] = { 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20 };

    k = 0;
    for (i = 0; fileName[i] != '\\0'; ++i) {
        if (fileName[i] != '.')
            fname[k++] = fileName[i];
        else
            k = 8;
    }

    for (i = 0; dirEntry->fileName[0] != 0 && i < size / 32; ++i) {
        if (!_strnicmp(dirEntry->fileName, fname, 11))
            return dirEntry;
        ++dirEntry;
    }

    return NULL;
}

/* Test.c */

#include <stdio.h>
#include <Windows.h>
#include "DiskIO.h"
#include "FatSys.h"
#include "PathNameResolution.h"

void ExitSys(LPCSTR lpszMsg, int status);

int main(void)
{
    DIRENTRY dentry;
    int i;

    if (!PathNameLookup("g:\\x\\y\\z\\t.txt", &dentry))
        ExitSys("PathNameLookup", EXIT_FAILURE);

    printf("%.11s\n", dentry.fileName);

    return 0;
}

void ExitSys(LPCSTR lpszMsg, int status)
{
    DWORD dwLastError = GetLastError();
    LPTSTR lpszErr;

    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }
}

```

```
    exit(status);  
}
```

Peki yol ifadelerinin çözümlenmesi işleminde nasıl iyileştirmeler yapılabilir? Bazıları şunlardır:

- FAT'ın tamamı her defasında (yani örneğimizde PathNameLookup) fonksiyonunda okunmayabilir. Toplamda bir kez için başında okunup global bir alana yerleştirilebilir.

- FAT'ın tamamının okunması yerine FAT için bir cache sistemi oluşturulabilir. Çünkü büyük volümlerde FAT çok büyük olabilmektedir. Cache'leme FAT elemanı temelinde yapılabilir (örneğin Linux böyle yapmaktadır) ya da sektör temelinde yapılabilir (örneğin Windows böyle yapmaktadır). Her iki durumda da önce aranan bilginin cache'te olup olmadığına bakılmalıdır. Bunun için hash tabloları kullanmak uygundur.

- Dosyalara ilişkin izin girişleri için de bir cache sisteminin oluşturulması uygun olur. Böylece aynı dosya ya da izinler aranırken boşuna tekrar tekrar aynı işlemler yapılmaz. Bu cache sistemine Linux sistemlerinde "dentry cache" denilmektedir.

- Göreli yol ifadelerinin çözümlenmesi için prosesin çalışma dizininin (current working directory) proses kontrol bloğunda saklanması gerekir. Ancak orada çalışma dizininin yalnızca yol ifadesi olarak değil izin girişi olarak da saklanması uygun olur. Böylece doğrudan göreli yol ifadeleri o izin girişinden itibaren çözümlenebilir. (Yukarıdaki örnekte prosesin geçerli dizininin g_cwd dizisi içerisinde yalnızca yol ifadesi olarak saklandığına dikkat ediniz. Bu nedenle o örnekte göreli yol ifadeleri her defasında baştan itibaren çözümlenecektir.)

FAT'ın sektör temelinde cache'lenmesine ilişkin bir örnek aşağıdaki gibi olabilir. Bu örnekte FAT'ın son kullanılan sektörleri bir cache'te saklanmaktadır. Bu cache'ten eleman çıkartmada LRU (Least Recently Used) algoritması kullanılmaktadır. Örnek yalnızca FAT16 sistemi için gerçekleştirilmiştir. GetNextClu16 fonksiyonunun cache'e bakarak çalıştığına dikkat ediniz.

```
/* Fat.h */  
  
#ifndef FAT_H_  
#define FAT_H_  
  
#include <Windows.h>  
#include "BPB.h"  
  
#define FAT_CACHE_TABLE_SIZE    31  
#define FAT_CACHE_SIZE          4  
  
typedef struct tagFATCACHE_NODE {  
    DWORD sect;  
    BYTE *data;  
  
    struct tagFATCACHE_NODE *next;  
    struct tagFATCACHE_NODE *prev;  
  
    struct tagFATCACHE_NODE *lruNext;  
    struct tagFATCACHE_NODE *lruPrev;  
  
} FATCACHE_NODE;  
  
DWORD GetNextFat16(const BPB *pBPB, DWORD clu);  
  
#endif  
  
/* Fat.c */
```

```

#include <stdio.h>
#include <Windows.h>
#include "DiskIO.h"
#include "BPB.h"
#include "Fat.h"

static FATCACHE_NODE *g_cache[FAT_CACHE_TABLE_SIZE];

BYTE g_fatCache[FAT_CACHE_TABLE_SIZE];
DWORD g_count;
FATCACHE_NODE *g_lruHead;
FATCACHE_NODE *g_lruTail;

static DWORD HashFunc(DWORD dw)
{
    return dw % FAT_CACHE_TABLE_SIZE;
}

DWORD GetNextFat16(const BPB *pBPB, DWORD clu)
{
    DWORD sect;
    BYTE *buf;
    DWORD hash;
    FATCACHE_NODE *fcu, *newNode;

    sect = pBPB->fatOrigin + clu * 2 / 512;
    hash = HashFunc(sect);
    fcu = g_cache[hash];

    while (fcu != NULL) {
        if (fcu->sect == sect) {
            if (g_lruTail != fcu) {
                if (fcu->lruPrev != NULL)
                    fcu->lruPrev->lruNext = fcu->lruNext;
                else
                    g_lruHead = fcu->lruNext;
                if (fcu->lruNext != NULL)
                    fcu->lruNext->lruPrev = fcu->lruPrev;

                fcu->lruPrev = g_lruTail;
                g_lruTail->lruNext = fcu;
                g_lruTail = fcu;
                fcu->lruNext = NULL;
            }

            return *(WORD *)(fcu->data + (clu * 2 % 512));
        }
        fcu = fcu->next;
    }

    if (g_count == FAT_CACHE_SIZE) {
        FATCACHE_NODE *temp;

        temp = g_lruHead;
        g_lruHead->lruNext->lruPrev = NULL;
        g_lruHead = g_lruHead->lruNext;

        if (temp->prev != NULL) {
            temp->prev->next = temp->next;
            if (temp->next != NULL)
                temp->next->prev = temp->prev;
        }
        else {
            g_cache[HashFunc(temp->sect)] = temp->next;
        }
    }
}

```

```

    buf = temp->data;
    newNode = temp;
}
else {
    if ((buf = (BYTE *)malloc(BYTE_PER_SECTOR)) == NULL)
        return 0;
    if ((newNode = (FATCACHE_NODE *)malloc(sizeof(FATCACHE_NODE))) == NULL)
        return 0;
    ++g_count;
}

newNode->data = buf;
newNode->sect = sect;
newNode->prev = NULL;
newNode->next = g_cache[hash];
g_cache[hash] = newNode;

if (!ReadSector(pBPB->hVolume, sect, 1, buf))
    return 0;

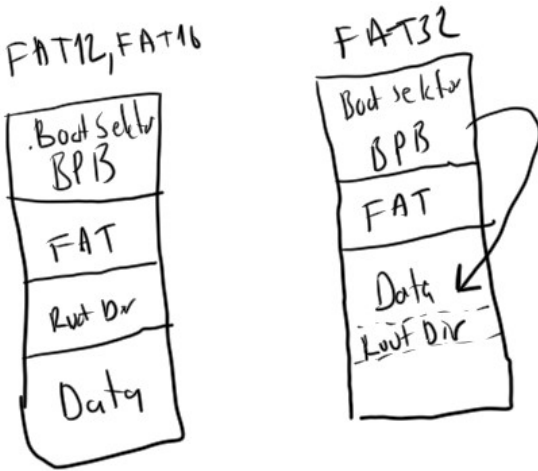
if (g_lruTail == NULL) {
    g_lruHead = g_lruTail = newNode;
    newNode->lruPrev = NULL;
}
else {
    g_lruTail->lruNext = newNode;
    newNode->lruPrev = g_lruTail;
    g_lruTail = newNode;
}
newNode->lruNext = NULL;

return *(WORD *)(buf + (clu * 2 % 512));
}

```

FAT32 Sistemlerinin Farklılıkları

FAT32 sistemlerinde FAT elemanlarının 32 bit (aslında 28 bit) olmasının yanı sıra aynı zamanda boot sektör BPB formatında da farklılıklar vardır. FAT32 sistemlerinde "Root Dir" bölümü data bölümünün içerisine taşınmıştır. Yani kök dizin de sanki bir alt dizin gibi data bölümünde bulunur. Tabii BPB bloğu içerisinde kök dizinin hangi cluster'da olduğu bilgisi tutulmaktadır.



"Root Dir" bölümünün Data bölümüne taşınmasıyla kök dizindeki dosya sayısı sınırı da ortadan kalkmış olmaktadır.

FAT32'deki Boot Sektör BPB Bloğu FAT12 ve FAT16'dakinin biraz değiştirilmiş biçimindedir.

Offset (Hex)	Uzunluk	İçerik
00	3 byte	Jmp Code
03	8 byte	OEM Yorum
0B	WORD	Sektördeki Byte Sayısı
0D	BYTE	Cluster'ın Sektör Uzunluğu
0E	WORD	Ayrılmış Sektörlerin Sayısı
10	BYTE	FAT kopyalarının Sayısı
11	WORD	Root Girişlerinin Sayısı
13	WORD	Volümdeki Toplam Sektör Sayısı
15	BYTE	Ortam Belirleyicisi (Media Descriptor)
16	WORD	Fat'in Bir Kopyasındaki Sektör Sayısı
18	WORD	Track Başına Düşen Sektör Sayısı
1A	WORD	Diskteki Kafa (yüzey) Sayısı
1C	DWORD	Saklı Sektörlerin Sayısı
20	DWORD	Yeni Toplam Sektör Uzunluğu
24	DWORD	FAT'in bir kopyasındaki sektör sayısı
28	WORD	FAT Aynalama Bilgisi
2A	WORD	FAT32 Versiyon Numarası
2C	DWORD	Root Dir Bölümünün Cluster Numarası
30	WORD	FSINFO Yapısının Hangi Sektörde Olduğu Bilgisi
32	WORD	Boot Sektör Kopyasının Yeri
34	12 Byte	Reserved
40	BYTE	Fiziksel Drive Numarası
41	BYTE	Reserved
42	BYTE	Genişletilmiş Boot ID'si
43	DWORD	Volümün Seri Numarası (Volume Serial Number)
47	11 Byte	Volüm Etiketi (Volume Label)
52	8 byte	Dosya Sisteminin Türü

Yukarıdaki tabloda normal yazılı alanlar tüm FAT sistemlerindeki ortak alanları belirtmektedir. Kalın (bold) yazılmış alanlar ise FAT32'deki fazlalıkları göstermektedir. FAT32 BPB Bloğunun ilk 32 byte'ı (0x20) FAT12 ve FAT16 BPB'sinin aynısı olduğuna dikkat ediniz.

Burada önemli bir nokta daha önce yazmış olduğumuz boot sektördeki BPB'yi okuyan GetBPB fonksiyonun FAT32 için artık geçerli olmadığıdır. FAT32 için BPB'yi okuyan ayrı bir fonksiyon yazılmalıdır.

Fat'in Bir Kopyasındaki Sektör Sayısı: FAT12 ve FAT16'da FAT zaten çok büyük olamıyordu. FAT sektörlerinin sayısı WORD ile (0x16'ıncı offset'e bakınız) belirtilebiliyordu. Ancak FAT32'de FAT çok büyük olduğu için bu alan yenilenmiş ve DWORD'e yükseltilmiştir.

FAT Aynalama Bilgisi: Bu alan bitsel olarak yorumlanmaktadır. FAT32'de FAT kopyaları beraber güncellenebilir ya da güncellenmeyebilir (buna İngilizce "mirroring" denilmektedir). Eğer aynalama kapalıysa hangi kopyanın aktif olduğu da burada belirtilmektedir. Ayrıntılı bilgi için "FAT32 File System Specification" dokümanını inceleyiniz.

FAT32 Versiyon Numarası: Burada FAT32 sisteminin versiyon numarası bulunur.

Root Dir Bölümünün Cluster Numarası: Burada Root Dir bölümünün Data bölümünün kaçınıcı cluster'ından başladığı bilgisi bulunmaktadır. (Anımsanacağı gibi Data bölümünün ilk cluster'ı 2 numaralı cluster'dır. 0 ve 1 cluster değerleri kullanılmamaktadır.)

FSINFO Yapısının Hangi Sektörde Olduğu Bilgisi: FAT32 sistemlerinde FAT12 ve FAT16 sistemlerinde bulunmayan bazı bilgiler FSINFO isimli bir yapı biçiminde tutulmaktadır. Bu alanda FSINFO yapısının hangi sektörde bulunduğu bilgisi vardır. Buradaki değer genellikle 1 biçiminde bulunur (yani boot sektörden hemen sonraki sektörde FSINFO bilgileri vardır.) FSINFO yapısındaki iki önemli eleman FSI_Free_Count ve FSI_Next_Free elemanlarıdır. FSI_Free_Count elemanı FAT'te boş olan cluster'ların sayısını, FSI_Next_Free ise ilk boş olan cluster numarasını tutmaktadır. Anımsanacağı gibi FAT12 ve FAT16'da volümdeki boş cluster'ların sayısı herhangi bir yerde tutulmamaktadır. FAT bölümündeki 0 olan FAT elemanları sayılarak boş alan miktarı hesaplanmaktadır.

Boot Sektör Kopyasının Yeri: FAT32'de boot sektörün bir kopyası (backup'ı) da ayrıca bir yerde tutulmaktadır. İşte burada boot sektörün kopyasının nerede olduğu bilgi vardır.

Fiziksel Drive Numarası: Burada INT 13 kesmesi için volümün fiziksel sürücü numarası bulunmaktadır (Aynı bilgi FAT12 ve FAT16 BPB'sinin 0x24'üncü offsetinde bulunmaktadır.)

Genişletilmiş Boot Id'si (Extended Boot Signature): Bu bilginin aynısı FAT12 ve FAT16'nın BPB'sinde de vardır. Burada 0x28 ya da 0x29 değeri bulunur. Bu değer ilgili disk bölümündeki dosya sisteminin türü hakkında bilgi verir. 0x29 FAT dosya sistemi olduğu anlamına gelmektedir.

Volümün Seri Numarası (Volume Serial Number): Bu bilgi de FAT12 ve FAT16 BPB'sinde vardır. Volüm formatlanırken ona 4 byte'lık rastgele bir seri numarası verilir. Eskiden bu seri numarası floppy zamanlarında bazı programlar tarafından floppy'nin değiştirilip değiştirilmediğini kontrol amacıyla kullanılırdı. Bugün pek bir kullanım amacı kalmamıştır. (Bu değer tekil bir değer değildir. Kopya koruma gibi amaçlarla bu değerden faydalanılması doğru yaklaşım değildir.)

Volüm Etiketisi (Volume Label): Bu bilgi de FAT12 ve FAT16 BPB'sinde vardır. Bu alanda 11 byte'lık volüme etiketi bulunmaktadır. Ancak volüm etiketi FAT dosya sistemlerinde kök dizinde "Volume Label" özniteliğine (attribute) sahip bir dizin girişi olarak da tutulmaktadır. DOS ve Windows uzun süredir volüm etiketini BOOT sektör BPB bloğundan değil bu dosyadan almaktadır.

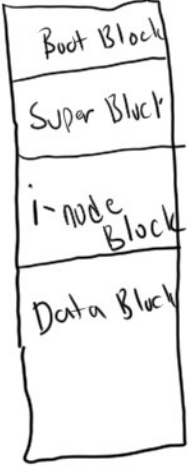
Dosya Sisteminin Türü: Bu bilgi de FAT12 ve FAT16 BPB'sinde vardır. Burada dosya sisteminin türü yazı olarak tutulmaktadır (örneğin "FAT32", "FAT16", "FAT12" gibi). Dolayısıyla dosya sisteminin türü hemen disk editörle görsel olarak belirlenebilir.

UNIX/Linux Sistemlerinde Kullanılan I-Node Tabanlı Dosya Sistemlerinin Disk Organizasyonu

UNIX/Linux sistemlerinde kullanılan dosya sistemleri birbirlerine yapı bakımından oldukça benzemektedir. Bu aileye "i-node tabanlı dosya sistemi ailesi" denebilir. Örneğin Ext2, Ext3, Ext4 gibi sistemler i-node tabanlı ailenin birer üyesidir. (Yani örneğin FAT sistemleri nasıl birbirlerine benziyorsa fakat aralarında bazı farklılıklar da varsa UNIX/Linux sistemlerindeki i-node tabanlı dosya sistemleri de yapı olarak birbirlerine benzemektedir. Fakat bunların aralarında da bazı farklılıklar vardır.)

UNIX/Linux sistemlerindeki i-node tabanlı dosya sistemlerinin disk organizasyonları temel olarak şöyledir:

i-node Tabanlı,
Disk Organizasyonu



Yukarıdaki organizasyonda "block" terimi ardışıl sektör topluluklarını ifade etmektedir. Diskin başında "boot blok" vardır. Bu blok 1024 byte (iki sektör) uzunluğundadır.

Anahtar Notlar: Linux sistemlerinde bir dosya sanki bir volümmüş gibi sisteme gösterilebilmektedir. Bu tür aygıtlara "loopback device" denilmektedir. Böylece i-node temelli dosya sistemleri üzerinde denemeler bu biçimde çok daha kolay yapılabilmektedir. Bunun sırasıyla şu adımlar izlenmelidir:

1) Öncelikle volümü temsil eden içi sıfırlarla dolu bir dosya oluşturmak gerekir. Bu işlem komut satırında dd komutuyla kolay biçimde yapılabilir. Örneğin:

```
dd if=/dev/zero of=mydisk.dat bs=512 count=2880
```

Bu komut if ile belirtilen dosyadan of ile belirtilen dosyaya $bs * count$ kadar byte'ı kopyalamaktadır. /dev/zero aygıt sürücüsü her okunduğunda sıfır değerini veren bir dosya gibi davranır. O halde bu komutla "mydisk.dat" dosyası oluşturulacak, bu dosyanın içi de $512 * 2880$ byte sıfır ile doldurulacaktır.

2) Daha sonra oluşturulan bu dosya "loopback device" olarak tanıtılır:

```
sudo losetup /dev/loop0 mydisk.dat
```

Artık /dev/loop0 aygıt sürücüsü mydisk.dat dosyasını kullanacak hale getirilmiştir.

3) Oluşturduğumuz volümü mkfs ile formatlayabiliriz:

```
sudo mkfs -t ext2 /dev/loop0
```

Burada aygıt ext2 dosya sistemi ile formatlanmıştır. Bu aygıtı Microsoft'un FAT tabanlı dosya sistemleriyle de formatlayabiliriz. Örneğin FAT16 olarak formatlamak için şu komut uygulanmalıdır:

```
sudo mkfs.fat -F 12 -I /dev/loop0
```

4) Artık oluşturulan dosya sistemi mount edilebilir. Tabii bunun için önce mount noktasında içi boş bir dizin açmak gerekir. Geleneksel olarak mount edilecek aygıtlar için /mnt dizininin altı tercih edilmektedir. Buradaki dizin için erişim haklarının uygun biçimde ayarlanması gerekmektedir. İlgili dizini /mnt dizinin altında şöyle açabiliriz:

```
sudo mkdir -m 777 /mnt/mydisk
```

Burada mount edilecek yeri istediğimiz gibi belirleyebildik. Artık mount işlemini yapabiliriz:

```
sudo mount -t ext2 /dev/loop0 /mnt/mydisk
```

Eğer FAT12, FAT16, FAT32 sistemlerini mount edeceksek -t msdos kullanmalıyız:

```
sudo mount -t msdos /dev/loop0 /mnt/mydisk
```

mount işlemi yapıldığında default durumda mount edilen dizinin kullanıcı ve group id'si root olmaktadır. Biz mount komutunda -o uid=csd -o gid seçenekleriyle bu durumu değiştirebiliriz. Örneğin:

```
sudo mount -t ext2 /dev/loop0 /mnt/mydisk -o uid=csd -o gid=study
```

Artık bu /mnt/mydisk dizini bizim yarattığımız volümün kök dizini olacaktır. Tabii aslında buradaki volüm tamamen "mydisk.dat" dosyasını kullanır. Başka bir deyişle biz bu volümde birtakım dosyalar yarattığımızda aslında bu dosyalar "mydisk.dat" dosyasının içerisinde oluşturulmuş olurlar.

5) İşlemler bittikten sonra loopback device yok aşağıdaki gibi yok edilebilir:

```
sudo losetup -d /dev/loop0
```

Tabii mount edilen dizin de unmount edilmelidir:

```
sudo umount /mnt/mydisk
```

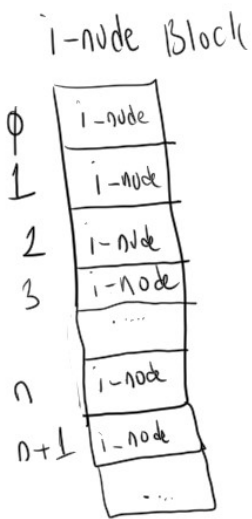
Boot blokta -tıpkı FAT dosya sisteminde olduğu gibi- bir yükleyici program vardır. Bu program işletim sisteminin yüklenmesine ön ayak olur. Süper blok da 1024 byte (yani 2 sektör) uzunluğundadır. urada ilgili i-node dosya sisteminin metadata bilgileri bulunmaktadır. Süper blok FAT dosya sistemindeki BPB bloğuna benzetilebilir. Pekiyi süper blokta nasıl bilgiler vardır? Örneğin Ext-2 dosya sisteminde süper blok aşağıdaki elemanlara sahiptir:

Offset (bytes)	Size (bytes)	Description
0	4	s_inodes_count
4	4	s_blocks_count
8	4	s_r_blocks_count
12	4	s_free_blocks_count
16	4	s_free_inodes_count
20	4	s_first_data_block
24	4	s_log_block_size
28	4	s_log_frag_size
32	4	s_blocks_per_group
36	4	s_frags_per_group
40	4	s_inodes_per_group
44	4	s_mtime
48	4	s_wtime
52	2	s_mnt_count
54	2	s_max_mnt_count
56	2	s_magic
58	2	s_state
60	2	s_errors
62	2	s_minor_rev_level
64	4	s_lastcheck
68	4	s_checkinterval
72	4	s_creator_os
76	4	s_rev_level
80	2	s_def_resuid
82	2	s_def_resgid
-- EXT2_DYNAMIC_REV Specific --		
84	4	s_first_ino
88	2	s_inode_size

90	2	s_block_group_nr
92	4	s_feature_compat
96	4	s_feature_incompat
100	4	s_feature_ro_compat
104	16	s_uuid
120	16	s_volume_name
136	64	s_last_mounted
200	4	s_algo_bitmap
-- Performance Hints --		
204	1	s_prealloc_blocks
205	1	s_prealloc_dir_blocks
206	2	(alignment)
-- Journaling Support --		
208	16	s_journal_uuid
224	4	s_journal_inum
228	4	s_journal_dev
232	4	s_last_orphan
-- Directory Indexing Support --		
236	4 x 4	s_hash_seed
252	1	s_def_hash_version
253	3	padding - reserved for future expansion
-- Other options --		
256	4	s_default_mount_options
260	4	s_first_meta_bg
264	760	Unused - reserved for future revisions

Bu süper blok yapısı Linux kaynak kodlarında başlık dosyalarında bulunmaktadır.

I-node Blok i-node elemanlarından oluşmaktadır. Yani i-node Blok, i-node elemanlarından oluşan bir dizi biçimindedir. I-Node bloktaki her i-node elemanının ilki 0 olmak üzere bir numarası vardır.



Bir dosyanın ismi dışındaki tüm meta data bilgileri i-node elemanında tutulmaktadır. Yani örneğin dosya ne zaman yaratılmıştır, uzunluğu nedir, erişim özellikleri nelerdir, hard link sayısı kaçtır, dosyanın parçaları data bölümünün hangi bloklarındadır (yani cluster'larındadır)? Bu bilgilerin hepsi dosyaya ilişkin i-node elemanında

bulmaktadır. Aslında kursumuzda daha önce görmüş olduğumuz UNIX/Linux sistemlerindeki stat fonksiyonu bilgileri bu i-node elemanından alıp bize vermektedir. stat fonksiyonunun parametrik yapısını yeniden anımsatmak istiyoruz:

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

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

struct stat {
    dev_t    st_dev;    /* ID of device containing file */
    ino_t    st_ino;    /* inode number */
    mode_t   st_mode;   /* protection */
    nlink_t  st_nlink;  /* number of hard links */
    uid_t    st_uid;    /* user ID of owner */
    gid_t    st_gid;    /* group ID of owner */
    dev_t    st_rdev;   /* device ID (if special file) */
    off_t    st_size;   /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks; /* number of 512B blocks allocated */
    time_t   st_atime;  /* time of last access */
    time_t   st_mtime;  /* time of last modification */
    time_t   st_ctime;  /* time of last status change */
};
```

Tabii stat fonksiyonu i-node elemanındaki birçok değeri bize verse de -çok teknik olduğu gerekçesiyle- bazı bilgileri de bilgileri bize vermemektedir. Ext-2 dokümanlarında i-node elemanın içeriği şöyle dokümante edilmiştir:

Offset (bytes)	Size (bytes)	Description
0	2	i_mode
2	2	i_uid
4	4	i_size
8	4	i_atime
12	4	i_ctime
16	4	i_mtime
20	4	i_dtime
24	2	i_gid
26	2	i_links_count
28	4	i_blocks
32	4	i_flags
36	4	i_osd1
40	15 x 4	i_block
100	4	i_generation
104	4	i_file_acl
108	4	i_dir_acl
112	4	i_faddr
116	12	i_osd2

Görüldüğü gibi Ext-2'de bir i-node elemanı 128 byte uzunluğundadır.

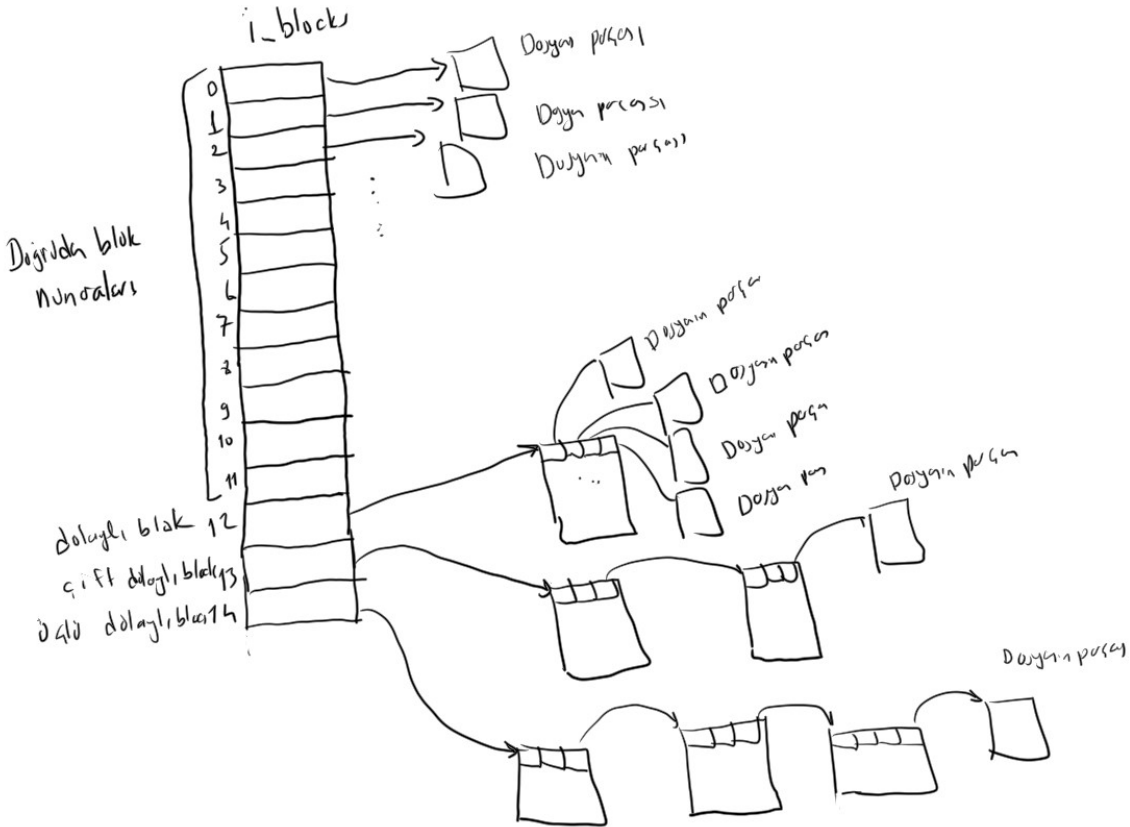
I-node tabanlı dosya sistemlerinde volümdeki toplam dosyaların sayısı i-node blok'ta ayrılmış i-node elemanlarının sayısından fazla olamaz. Çünkü her dosya için (dosyanın büyüklüğü ne olursa olsun) i-node blok'ta bir tane i-node elemanı tahsis edilmek zorundadır.

Bir dosyanın i-node numarası o dosyanın bilgilerinin i-node blok'taki hangi i-node elemanında bulunduğunu belirtir. Is komutunda -i seçeneği ile dosyaların i-node numaraları elde edilebilir. Örneğin:

```
csd@csd-VirtualBox /mnt/mydisk $ ls -li
total 14
12 -rwxr-xr-x 1 root root 97 May 7 12:40 a.c
11 drwx----- 2 root root 12288 May 7 12:33 lost+found
13 -rw-r--r-- 1 csd csd 16 May 7 12:42 y.txt
```

Data bölümünde her bloğa sıfırdan başlayarak bir numara karşı düşürülmüştür. (UNIX/Linux sistemlerinde "cluster" yerine "block" teriminin kullanıldığını anımsayınız). Dosyaların parçaları da data bölümünün çeşitli bloklarında bulunmaktadır. Anımsanacağı gibi FAT dosya sisteminde dosyaların parçalarının hangi cluster'larda bulunduğu FAT bölümünde tutuluyordu. Pekiyi I-Node sistemlerinde dosyanın parçalarının hangi bloklarda olduğu nasıl ve nerede tutulmaktadır? İşte yukarıda da belirttiğimiz gibi dosyanın bütün bilgileri (buna onun parçalarının hangi bloklarda olduğu bilgisi de dahildir) dosyanın i-node elemanında tutulmaktadır. Örneğin dosya toplamda 150 blok yer kaplıyor olsun. i-node elemanında tek tek bu 150 bloğun hangi numaralı bloklar olduğu nasıl tutuluyor olabilir? I-Node elemanlarının 128 byte uzunlukta olduğunu anımsayınız.

İşte i-node sisteminde bunun için doğrudan (direct), dolaylı (indirect), çift dolaylı (double indirect) ve üçlü dolaylı (triple indirect) blok kavramları kullanılmaktadır. Örneğin Ext-2 dosya sisteminde bir i-node elemanının içerisinde 40'inci (decimal) offset'ten başlayan her biri 4 byte uzunlukta olan 15 tane blok numaralarını tutan bir dizi vardır. (Bu dizi yukarıdaki Ext-2 dokümanlarından alınan kısımda i_block ile temsil edilmektedir.) Bu 15 blok numaralarına ilişkin dizinin ilk 12 elemanı doğrudan blok numaralarını, 13'üncü elemanı dolaylı blok numarasını, 14'üncü elemanı çift dolaylı blok numarasını ve 15'inci elemanı üçlü dolaylı blok numarasını tutar.



Ext-2'de bir bloğun kaç sektörden oluştuğu bilgisi süper blok'ta belirtilmektedir. Süper bloğun s_log_block_size elemanındaki değer 0 ise blok uzunluğu 1024 byte, 1 ise 2048 byte 2 ise 4096 byte, ... biçimindedir. Başka bir deyişle s_log_block_size elemanındaki değer ile blok uzunluğu arasındaki ilişki şöyledir:

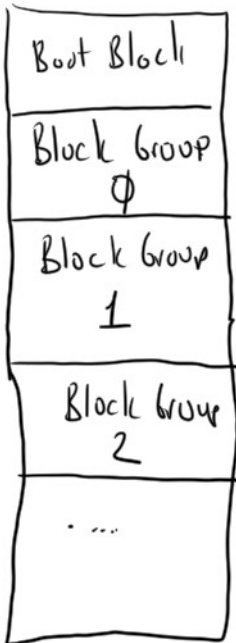
```
block size = 1024 << s_log_block_size;
```

Bir bloğun 1024 byte (yani 2 sektör) uzunluğunda olduğunu varsayalım. Bu durumda eğer dosya $12 * 1024$ byte'tan küçük ya da ona eşitse dosyanın bloklarının numaraları doğrudan i-node elemanının `i_block` dizisi içerisindeki ilk 12 elemandan alınır. Eğer dosyanın uzunluğu $12 * 1024$ (yani 12K)'dan büyükse dosyanın ilk 12K'sının blok numaraları doğrudan bloklarda geri kalanları dolaylı bloklarda bulunur. Dolaylı blok numarası `i_block` dizisinin 13'üncü elemanındadır. Dolaylı blok numarasının ne olduğunu şöyle açıklayabiliriz: Dizinin bu 13'üncü elemanında bir blok numarası vardır. Ancak o blok numarası dosyanın bir parçasını oluşturan bir bloğun numarası değildir. Dosyanın parçalarının hangi bloklarda tutulduğunu gösteren bloğun numarasıdır. Yani biz `i_block` dizisinin 13'üncü elemanında yazılan bloğa gittiğimizde o bloğun içerisinde 4 byte'lık blok numaraları görürüz. Bir bloğun 1024 byte olduğunu varsayarsak blok numaraları 4 byte yer kapladığına göre bir blok içerisinde $1024 / 4 = 256$ blok numarası bulunur. İşte eğer dosya 12K + 256K'dan daha büyükse artık dolaylı bloklar da yetersiz kalır. Bu durumda çift dolaylı bloklar kullanılmaktadır. Çift dolaylı blok numarası `i_block` dizisinin 14'üncü elemanındadır. Yani bu 14'üncü elemenda belirtilen numaralı bloğa gittiğimizde onun içerisinde 256 tane blok numarası görürüz. O 256 blok numarası dosyanın parçalarını belirtmez. Bilakis dosyanın parçalarının hangi bloklarda olduğunu tutan blokları belirtir. Bu durumda çift dolaylı bir blok içerisinde (bir blok 1K uzunluktaysa) bir dosyanın $256 * 256 * 1024$ (yaklaşık 67 MB) kadar byte'lık kısmı tutulmuş olur. Üçlü dolaylı bloklar da benzer biçimdedir.

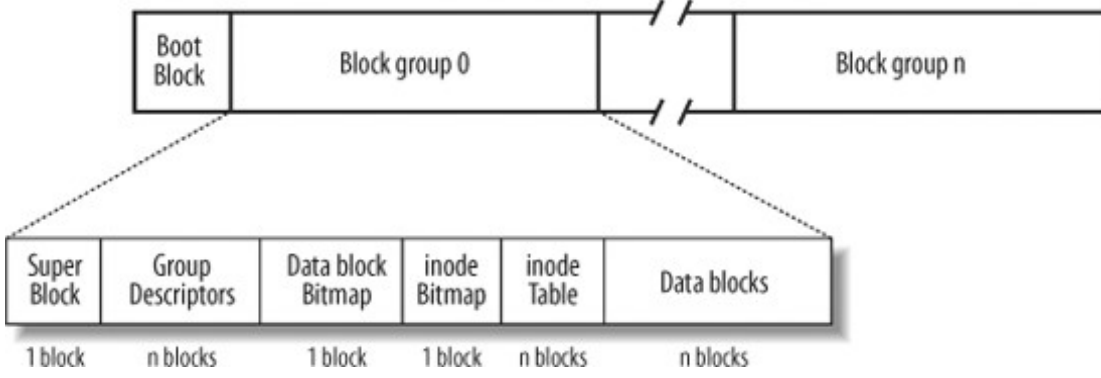
i-node tabanlı dosya sistemleri mimari olarak birbirlerine çok benzese de ayrıntılarda farklılıklar söz konusudur. Bu nedenle burada biz spesifik olarak Ext-2 dosya sistemine özgü disk organizasyonu üzerinde de durmak istiyoruz.

Ext-2 Dosya Sisteminin Disk Organizasyonu

Ext-2 Dosya sistemi resmi olarak "Ext-2 File System Specification" dokümanında ayrıntılarıyla açıklanmaktadır. Bu dosya sisteminin disk organizasyonu kabaca aşağıdaki gibidir:



Görüldüğü gibi aslında ext-2 disk organizasyonu blok gruplardan oluşmaktadır. Her blok grubunun kendi i-node tablosu ve data bölümü vardır. (Ayrıca süper blok ve "block group descriptor" denilen veri yapılarının backup amaçlı birer kopyası da blok gruplarının başında tutulmaktadır. Ext-2 Revision 1'de bu backup kopyalar her blok grubunda değil yalnızca belirli blok gruplarında tutuluyor). Bir blok grubunun formatı şöyledir:



Tabii blok gruplardaki i-node tablolarındaki i-node elemanlarına yine ardışıl numaralar verilmiştir. Yani örneğin ilk blok grubunun son i-node elemanı n numaralı ise ikinci blok grubunun ilk i-node elemanı n + 1 numaralı elemandır.

Anahtar Notlar: Ext-2 dosya sistemindeki meta data alanlarında belirtilen blok numaraları her zaman volümün başından itibaren ve süper blokta belirtilen blok uzunluğu dikkate alınarak verilmiş olan blok numaralarıdır.

Bir blok grubun hemen başında Süper Bloğun olduğuna dikkat ediniz. Tabii volümde toplam bir tane süper blok vardır. Diğer blok grupları aslında 0'ıncı (yani ilk) blok gruptaki süper bloğun kopyalarını tutar. Blok grubun ikinci elemanı "Grup Betimleyicilerinden (Group Descriptors)" oluşmaktadır. Grup betimleyicisi bir blok grubundaki elemanların bazı meta data bilgilerini tutmaktadır. Dolayısıyla yukarıdaki şekildeki "Grup Betimleyicileri (Group Descriptor)" alanı "Grup Betimleyicisi (Group Descriptor)" dizisi biçimindedir. Blok gruplarının kaç tane olduğu "Block Group Descriptor Table" uzunluğundan anlaşılmaktadır. Burada görüldüğü gibi her blok grubunda aslında backup amaçlı blok betimleyicilerinin de kopyası bulundurulmaktadır. Yani blok gruplardaki grup betimleyicilerinin hepsi aynı bilgiye sahiptir. Bir grup betimleyicisinin formatı aşağıda verilmektedir:

Offset (bytes)	Size (bytes)	Description
0	4	bg_block_bitmap
4	4	bg_inode_bitmap
8	4	bg_inode_table
12	2	bg_free_blocks_count
14	2	bg_free_inodes_count
16	2	bg_used_dirs_count
18	2	bg_pad
20	12	bg_reserved

Her blok gruptaki i-node tablosunun yeri vs. "block group descriptor" tablosunun ilgili elemanlarında tutulmaktadır.

Peki bir i-node numarasına ilişkin i-node elemanın hangi blok grubunda olduğu nasıl tespit edilmektedir? İşte süper blok içerisinde blok gruplarına ilişkin bazı bilgiler de vardır. Örneğin bir blok grubunda kaç tane blok

bulunduğu süper bloğun `s_blocks_per_count` elemanında saklanır. Böylece blok gruplarının volümün neresinden başladığı ve hangi uzunlukta olduğu tespit edilebilmektedir. Ayrıca her blok grubunda kaç tane i-node elemanın bulunduğu da yine süper bloğun `s_inodes_per_group` elemanında saklanmıştır. Bu durumda biz `n` numaralı bir i-node elemanın hangi blok grupta olduğunu şöyle tespit edebiliriz:

$n / s_inodes_per_group$

Her blok grubunda eşit sayıda i-node elemanın bulunduğuna dikkat ediniz. Bu durumda bir i-numarasına ilişkin i-node elemanın tam olarak nerede bulunduğu da tespit edilebilir. Çünkü `n` inode elemanın numarası olmak üzere $n \% s_inodes_per_group$ ifadesi i-node elemanın ilgili blok gruptaki yerini bize verecektir.

Ext-2 dosya sisteminde ilk i-node elemanın numarası 1'dir. Yani i-node elemanları 1'den başlayarak numaralandırılmaktadır. 1 numaralı i-node elemanı tamamen 0'larla doldurulmuştur (null eleman). 2 numaralı i-node elemanı kök dizine ilişkindir. (`n` numaralı i-node elemanın i-node tablosunun `n - 1`'inci elemanında olacağına dikkat ediniz.)

Örneğin loopback aygıt olarak mount edilmiş bir ext-2 dosya sisteminin kök dizininde aşağıdaki dosyalar bulunuyor olsun:

```
kaan@kaan-virtual-machine /mnt/mydisk $ ls -li
total 20
11 drwx----- 2 root root 12288 May  8 11:51 lost+found
12 -rwxr-xr-x 1 root root  7922 May  8 11:52 mysharedlib.so
```

Burada `mysharedlib.so` dosyasının i-node numarası 12'dir. Bu dosya ilk blok grubundadır. "Block group descriptor" tablosunun içeriği şöyledir:

```
00000800 04 00 00 00 05 00 00 00 06 00 00 00 D0 01 34 00 .....[06]Ğ.4.
00000810 02 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000820 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000830 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

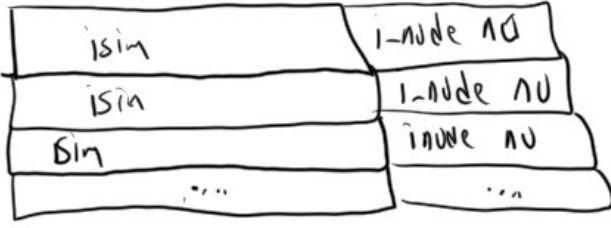
Burada volümün bir tek blok grubundan oluştuğunu görüyorsunuz. Seçilmiş olan alan (8'inci offsetten çekilen DWORD) blok grubunun i-node tablosunun yerini göstermektedir. Bu volümde 1 blok 1024 byte olduğu için i-node tablosunun yeri $6 * 1024$ (0x1800) offset'indedir. 12 numaralı i-node elemanı da i-node tablosunun başından $11 * 128$ byte ileride olacaktır: $6 * 1024 + 11 * 128$ (0x1D80). Bu i-node elemanını aşağıda görmekteyiz:

```
00001D80 ED 81 00 00 F2 1E 00 00 63 FE 2E 57 63 FE 2E 57 i...ò...çş.Wçş.W
00001D90 63 FE 2E 57 00 00 00 00 00 00 01 00 10 00 00 00 çş.W.....
00001DA0 00 00 00 00 01 00 00 00 1C 00 00 00 1D 00 00 00 .....
00001DB0 1E 00 00 00 1F 00 00 00 20 00 00 00 21 00 00 00 .....!...
00001DC0 22 00 00 00 23 00 00 00 00 00 00 00 00 00 00 00 "...#.....
00001DD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00001DE0 00 00 00 00 A5 CB 87 37 00 00 00 00 00 00 00 00 ...#E#7.....
00001DF0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

I-Node Tabanlı Dosya Sistemlerinde Dizin Organizasyonu

I-node tabanlı dosya sistemlerinde dizinler de tamamen dosyalar gibidir. Yani onlar için de bir i-node elemanı vardır. Ancak dizinlerin blokları içerisinde o dizinde bulunan dosyaların isimleri ve i-node numaraları bulunur.

Aslında i-node tabanlı dosya sistemleri arasında izin içeriğinin formatı bakımından farklılıklar vardır. Ancak burada biz bu formatı genel olarak aşağıdaki biçimdeymiş gibi gösterebiliriz:



Ext-2 dosya sisteminde izin elemanlarının formatı şöyledir:

Offset (bytes)	Size (bytes)	Description
0	4	inode numarası
4	2	girişin toplam uzunluğu
6	1	dosya isminin uzunluğu
7	1	dosyanın türü
8	0-255	dosyanın ismi

Görüldüğü gibi Ext-2'de izin içerisindeki dosya listesi sabit uzunluklu kayıtlar biçiminde değildir. Ext-2'de dosya isimleri 256 karaktere kadar uzayabildiği için her kaydın bu kadar büyük olmasına gerek görülmemiştir. Çünkü dosya isimlerinin çoğu aslında kısadır. Tablodaki inode isimli eleman ilgili dosyanın i-node numarasını tutar. rec_len ilgili izin girişinin kaç byte uzunlukta olduğunu belirtir. Yani sonraki giriş burada belirtilen değer kadar ileridedir. name_len değişken olan dosya isminin kaç karakter uzunluğunda olduğunu tutar. file_type dosyanın türünü belirtir. Gerçi bu tür bilgisi aynı zamanda i-node elemanın içinde de vardır fakat "yol ifadesinin çözülmesi (path name resolution)" işleminin hızlı bir biçimde yapılabilmesi için bu bilgi ayrıca Ext-2'de izin girişinde de bulundurulmaktadır.

Peki i-node temelli bir dosya sisteminde bir yol ifadesi nasıl çözülür? İşte kök dizine ilişkin i-node elemanın yeri bellidir. Örneğin Ext-2 dosya sisteminde 2 numaralı i-node elemanı (anımsanacağı gibi Ext-2'de i-node numaraları 1'den başlar ve ilk i-node elemanı 0'larla doludur) kök dizine ilişkindir. Sonra kök izin içerisinde diğer yol bileşeni, onun içerisinde de diğeri bulunarak süreç ilerletilir.

Aşağıda kök dizine ilişkin bir izin içeriğini görüyorsunuz:

```

00003800  b2 00 00 00 0C 00 01 02 2E 00 00 00 02 00 00 00 00 .....
00003810  0C 00 02 02 2E 2E 00 00 0B 00 00 00 14 00 0A 02 .....
00003820  6C 6F 73 74 2B 66 6F 75 6E 64 00 00 0C 00 00 00  lost+found.....
00003830  18 00 0E 01 6D 79 73 68 61 72 65 64 6C 69 62 2E  ...mysharedlib.
00003840  73 6F 00 00 0D 00 00 00 BC 03 08 01 74 65 73 74  so.....4...test
00003850  2E 64 61 74 00 00 00 00 00 00 00 00 00 00 00 00  .dat.....
00003860  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00003870  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00003880  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00003890  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000038A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000038B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
000038C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....

```

Burada kayıtların yerleri aşağıdaki gibidir:

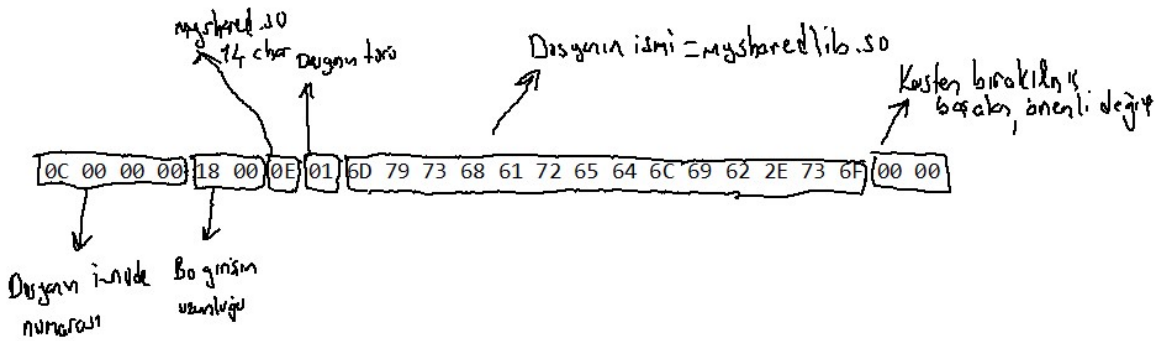
```

00003800 02 00 00 00 0C 00 01 02 2E 00 00 00 02 00 00 00 0.....
00003810 0C 00 02 02 2E 2E 00 00 0B 00 00 00 14 00 0A 02 .....
00003820 6C 6F 73 74 2B 66 6F 75 6E 64 00 00 0C 00 00 00 lost+found.....
00003830 18 00 0E 01 6D 79 73 68 61 72 65 64 6C 69 62 2E ...mysharedlib.
00003840 73 6F 00 00 0D 00 00 00 BC 03 08 01 74 65 73 74 so.....4...test
00003850 2E 64 61 74 00 00 00 00 00 00 00 00 00 00 00 00 .dat.....
00003860 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00003870 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00003880 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00003890 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000038A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000038B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000038C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Burada örneğin mysharedlib.so dosyasının dizin girişi şöyledir:

```
0C 00 00 00 18 00 0E 01 6D 79 73 68 61 72 65 64 6C 69 62 2E 73 6F 00 00
```

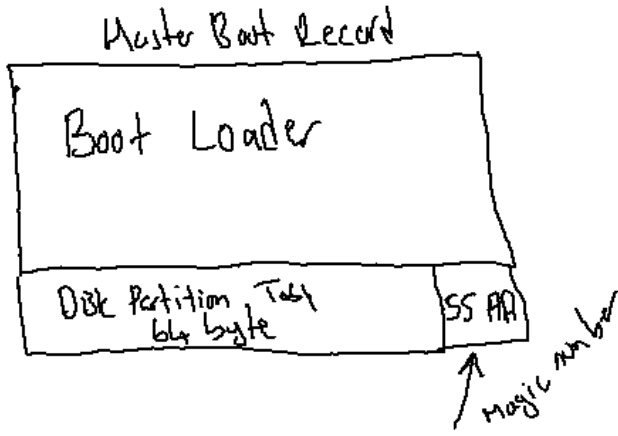


Dizin girişi sıralı olarak taranmaktadır. Dizin listesinin bittiği sonraki kaydın i-node elemanının 0 olmasından anlaşılır. Anımsanacağı gibi Ext-2 dosya sisteminde 0 numaralı i-node elemanı kullanılmamaktadır. i-node elemanları 1'den başlar.

Disk Bölümlere Ayrılması ve Disk Bölümleme Tablosu

Bir hard diske, SSD'ye ya da flash belleğe birden fazla dosya sistemi yerleştirilebilir. Örneğin biz bir diske hem Windows'u hem Linux'u kurabiliriz. Windows NTFS sistemini kullanırken, Linux Ext-2 sistemini kullanıyor olabilir. İşte bunu sağlamak için disk bölümesi kavramı uydurulmuştur. Diskin başında bir disk bölümleme tablosu (disk partition table) bulunur. Burada hangi disk bölümünün hangi sektörden başlayıp hangi uzunlukta olduğu bilgisi yazılıdır. Böylece işletim sistemleri kendi bölümleri dışına çıkmaz. Yani adeta bir disk daha küçük parçalara ayrılmış birden fazla diskmiş gibi kullanılmaktadır.

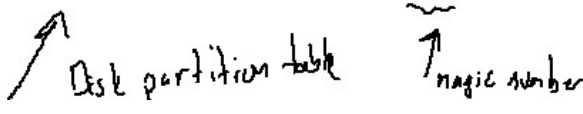
Disk bölümleme tablosu nerededir? Diskin ilk sektörüne (yani 0'ıncı sektörüne) "Master Boot Record" denilmektedir. Bu ilk sektörün yapısı şöyledir:



```

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000000000 33 C0 8E D0 BC 00 7C 8E C0 8E D8 BE 00 7C BF 00 3A.G+.|.À.Ø%.|¿.
0000000010 06 B9 00 02 FC F3 A4 50 68 1C 06 CB FB B9 04 00 .'.úó*Ph..Èù'.
0000000020 BD BE 07 80 7E 00 00 7C 0B 0F 85 0E 01 83 C5 10 *%.€~..|.....fÄ.
0000000030 E2 F1 CD 18 88 56 00 55 C6 46 11 05 C6 46 40 00 añí.´V.UEF..EF..
0000000040 B4 41 BB AA 55 CD 13 5D 72 0F 81 FB 55 AA 75 09 ´A»*Uí.|r..úU«u.
0000000050 F7 C1 01 00 74 03 FE 46 10 66 60 80 7E 10 00 74 ±Ä..t.gF.f'€~.t
0000000060 26 66 68 00 00 00 00 66 FF 76 08 68 00 00 68 00 &fh....fÿv.h..h.
0000000070 7C 68 01 00 68 10 00 B4 42 8A 56 00 8B F4 CD 13 |h..h..´BŠV.<óí.
0000000080 9F 83 C4 10 9E EB 14 B8 01 02 BB 00 7C 8A 56 00 ÝfÄ..é..|.»..|ŠV.
0000000090 8A 76 01 8A 4E 02 8A 6E 03 CD 13 66 61 73 1C FE Šv.ŠN.Šn.í.fas.g
00000000A0 4E 11 75 0C 80 7E 00 80 0F 84 8A 00 B2 80 EB 84 N.u.€~.€..Š.«Eé„
00000000B0 55 32 E4 8A 56 00 CD 13 5D EB 9E 81 3E FE 7D 55 U2äŠV.í.|é..>g}U
00000000C0 AA 75 6E FF 76 00 E8 8D 00 75 17 FA B0 D1 E6 64 *unÿv.é..u.ú°Ñæd
00000000D0 E8 83 00 B0 DF E6 60 E8 7C 00 B0 FF E6 64 E8 75 èf.°Bæ`è|.°ÿædèu
00000000E0 00 FB B8 00 BB CD 1A 66 23 C0 75 3B 66 81 FB 54 .ú..»í.f#Au;f.úT
00000000F0 43 50 41 75 32 81 F9 02 01 72 2C 66 68 07 BB 00 CPAu2.ù..r,fh.».
0000000100 00 66 68 00 02 00 00 66 68 08 00 00 00 66 53 66 .fh....fh....fSf
0000000110 53 66 55 66 68 00 00 00 66 68 00 7C 00 00 66 SfUfh....fh.|..f
0000000120 61 68 00 00 07 CD 1A 5A 32 F6 EA 00 7C 00 00 CD ah...í.Z2öè..|.í
0000000130 18 A0 B7 07 EB 08 A0 B6 07 EB 03 A0 B5 07 32 E4 . .é. ¶.é. u.2ä
0000000140 05 00 07 8B F0 AC 3C 00 74 09 BB 07 00 B4 0E CD ...<g-<t.»..´.í
0000000150 10 EB F2 F4 EB FD 2B C9 E4 64 EB 00 24 02 E0 F8 .éóóéí+Éädé.$.áz
0000000160 24 02 C3 49 6E 76 61 6C 69 64 20 70 61 72 74 69 $.ÄInvalid parti
0000000170 74 69 6F 6E 20 74 61 62 6C 65 00 45 72 72 6F 72 tion table.Error
0000000180 20 6C 6F 61 64 69 6E 67 20 6F 70 65 72 61 74 69 loading operati
0000000190 6E 67 20 73 79 73 74 65 6D 00 4D 69 73 73 69 6E ng system.Missin
00000001A0 67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74 g operating syst
00000001B0 65 6D 00 00 00 63 7B 9A 06 BA 67 85 00 00 80 20 em...c{š.°g...E
00000001C0 21 00 07 BE 12 2C 00 08 00 00 00 F0 0A 00 00 BE !..%. ,...g...%
00000001D0 13 2C 07 FE FF FF 00 F8 0A 00 00 30 D9 1B 00 FE .,şÿÿ.ø...Öÿ.ş
00000001E0 FF FF 27 FE FF FF 00 28 E4 1B 00 10 0E 00 00 00 Ÿÿ'şÿÿ.(ä.....
00000001F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA .....U²

```



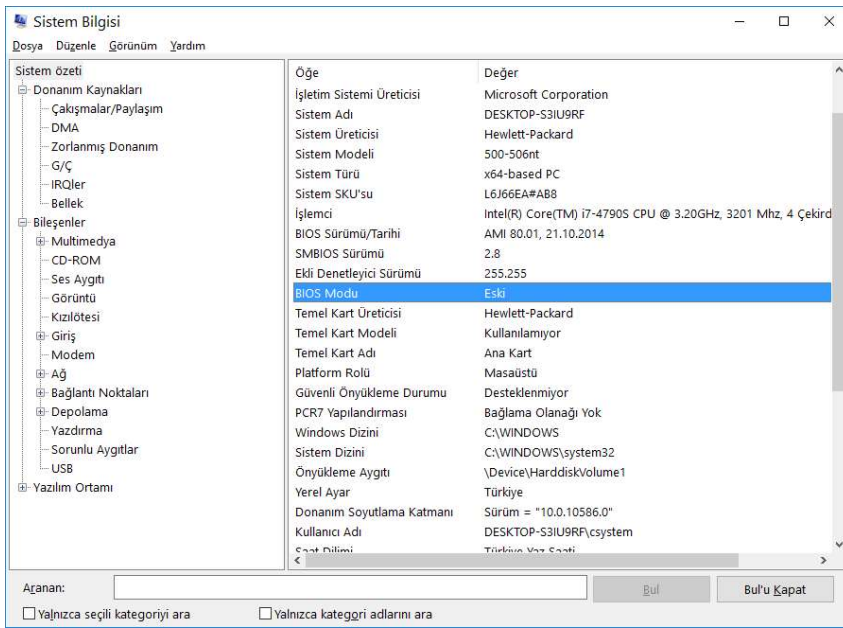
Disk Bölümlene Tablosu her biri 16 byte olan 4 girişe sahiptir. Bu durumda diskte en fazla 4 ana disk bölümü bulunabilmektedir. Daha fazla disk bölümü oluşturabilmek için "uzatılmış disk bölümü (extended partition)" kavramı uydurulmuştur. 16 byte'lık disk bölümlene tablosu elemanın formatı şöyledir:

Structure of a 16-byte Partition Table Entry		
Relative Offsets (within entry)	Length (bytes)	Contents
0	1	Boot Indicator (80h = active)
1 - 3	3	Starting CHS values
4	1	Partition-type Descriptor
5 - 7	3	Ending CHS values
8 - 11	4	Starting Sector

Table 4.

İlk byte disk bölümünün aktif olup olmadığını belirtir. Burada 0x80 değeri varsa o disk bölümü aktiftir, 0x00 varsa aktif değildir. Dört girişten yalnızca bir tanesi aktif olabilir. Dördüncü offset'te o disk bölümünde (partition type descriptor) hangi dosya sisteminin olduğu bilgisi bulunmaktadır. Her dosya sistemi onu belirten ayrı bir numara ile temsil edilmiştir. (Buradaki değer 0xEE ise diskin UEFI BIOS desteği ile organize edildiği anlaşılır. UEFI tarzı disk organizasyonunda disk bölümlenme tablosuna "GUID Bölümlenme Tablosu (GUID Partition Table)" denilmektedir. Kursumuzda GUID Bölümlenme Tablosu ele alınmayacaktır.) Görüldüğü gibi her disk bölümü için o disk bölümünün hangi sektörden başladığı ve ne uzunlukta olduğu bilgisi de disk bölümlenme tablosu elemanında belirtilmektedir. Tablodaki diğer elemanların artık önemi kalmamıştır.

Disk bölümlenme tablosu UEFI Bios kullanan sistemlerde "GUID Partition Table" ismiyle modernize edilmiştir. Pekiyi sistemimizde UEFI BIOS olduğunu nasıl anlarız? Bunun için Windows'ta "msinfo32" programı çalıştırılıp "System Summary (Sistem Özeti)" kısmındaki "BIOS Modu"na bakılabilir:



Linux'ta sistemin UEFI ile boot edilip edilmediğini anlamak için "/sys/firmware/efi" dosyasının var olup olmadığına bakılabilir. Eğer bu dosya varsa sistem UEFI BIOS ile başlatılmıştır. Bilgisayarımız UEFI'yi desteklediği halde boot işlemi klasik BIOS'tan da yapılıyor olabilir. Bunun için "CMOS Setup"a girişte UEFI desteğini açmak gerekebilir.

MBR sektörünün başında küçük bir "boot loader" programı bulunur. Bu program disk bölümlenme tablosunun dört bölümünden hangisinin aktif olduğunu tespit eder. Sonra aktif disk bölümünün ilk sektörünü belleğe yükleyerek oraya jump eder. Her disk bölümünün ilk sektörü kendi işletim sistemini yüklemekle yükümlüdür. (Anımsanacağı gibi FAT dosya sisteminde ilk sektöre zaten "boot sector" deniliyordu)

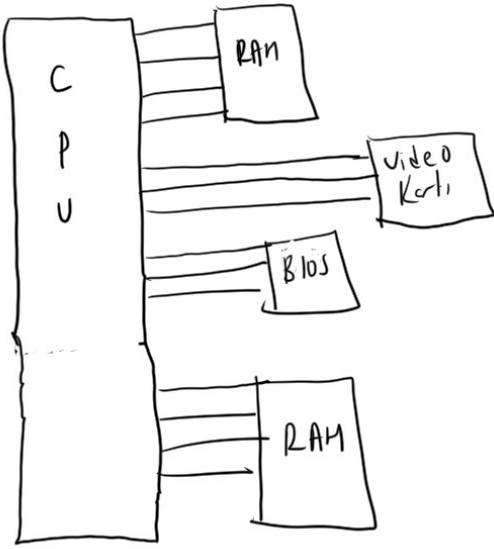
Biz burada disk bölümlenme tablosundaki genişletilmiş (extended) disk bölümlerinin nasıl oluşturulduğu ve GUID disk bölümlenme tablosunun yapısını ele almayacağız. Bu konu "Sistem Programlama ve İleri C Uygulamaları II" numaralı kursun konusu içerisindedir.

Intel Tabanlı Bilgisayarların Boot Süreci

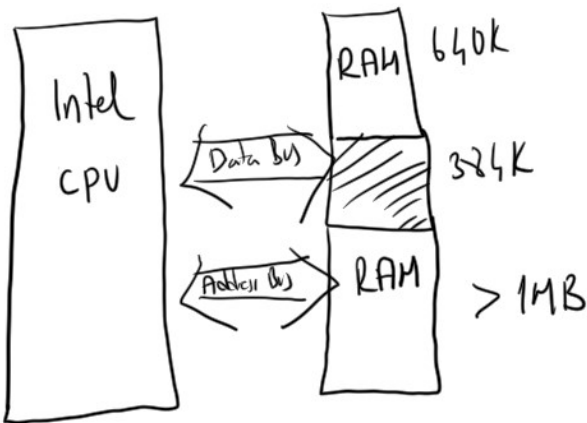
Bugün kullandığımız Intel tabanlı masaüstü ve dizüstü bilgisayarlarımızı açtığımızda işletim sisteminin otomatik olarak yüklendiğini görürüz. Pekiyi bu süreç nasıl gerçekleşmektedir? İşte çok ayrıntılarına girmeden PC'lerimizdeki tipik boot süreci şöyledir:

1) Her işlemci reset edildiğinde belli bir adresten çalışmaya başlayacak biçimde tasarlanmıştır. Bu adrese "reset vektörü" denilmektedir. İşlemci RAM'e bağlı olduğuna göre ve bilgisayarımızı açtığımızda RAM de sıfırlandığına göre işlemci hangi komutlarla çalışmaya başlayacaktır? İşte bilgisayar sistemlerinde CPU'nun reset vektörüne bilgisayarı kapattığımızda da bilgileri tutan tarzda bir bellek yerleştirilmektedir. Eskiden bu tür bellekler EPROM olarak üretiliyordu. Bugün artık bu tür bellekler için EEPROM teknolojisi kullanılmaktadır.

Biz PC'mizi reset ettiğimizde çalışma EEPROM içerisindeki bir yerden başlar. PC açıldığında hazır bulunan ve kalıcı bellekte bulunan bu kodlara PC terminolojisinde "BIOS (Basic Input Output System)" denilmektedir. Aslında bir bilgisayar sisteminde RAM sürekli olmak zorunda değildir. RAM'in bazı bölgeleri başka aygıtların içerisindeki belleklere yönlendirilmiş olabilir (Memory Mapped IO). Bu yönlendirme elektriksel düzeyde kolay bir biçimde yapılabilir. Örneğin:



Tabii bizim sistem programcısı olarak adres alanının hangi kısımlarının hangi aygıtlar içerisindeki belleklere bağlı olduğunu bilmemiz gerekebilir. Örneğin PC mimarisinde ilk 1MB belleğin son 384K'sı başka aygıtlara yönlendirilmiştir. (Bu nedenle DOS işletim sistemi 1024K - 384K = 640K kullanabiliyordu). Yani PC RAM'i aslında sürekli değildir:



İşte PC sisteminde klasik BIOS ilk 1MB'nin son 64K'sında bulunur. Bugünkü 32 bit ve 64 bit Intel işlemcileri reset edildiğinde çalışma reset vektöründen başlar. Oradan BIOS'a jump edilmektedir ve BIOS'taki boot kodu çalışmaktadır. BIOS'taki boot kodunun ilk yaptığı şey POST (Power On Self Test) işlemidir. Bu işlem sırasında sisteme hangi aygıtların bağlı olduğu tespit edilmektedir.

2) BIOS'taki POST kodundan sonra sistemin boot edilmesi için gereken kodlar çalıştırılmaktadır. BIOS'taki "bootstrap" kod "CMOS Setup"ta belirdeğimiz sıraya göre aygıtlara bakar. Örneğin C sürücüsünün (yani birinci hard diskin) ilk boot aygıtı olarak seçildiğini düşünelim.

3) BIOS'taki bootstrap kod boot aygıtının (örneğin C sürücüsünün) ilk sektörünü belleğe yükler (7C00 adresine) ve oradaki koda jump eder. Böylece diskin MBR sektöründeki "yükleyici program" çalıştırılmış olur.

4) MBR'deki yükleyici program da aktif disk bölümünü tespit ve onun ilk sektörünü belleğe yükleyip oraya jump eder. Böylece ilgili disk bölümünün ilk sektöründeki program çalıştırılmış olur. Her dosya sisteminde ilk sektörler boot işlemi için ayrılmıştır. Örneğin FAT dosya sisteminde ilk sektöre boot sektör denilmektedir ve boot sektörün organizasyonu şöyledir:



Burada BPB'nin başındaki jmp komutu FAT dosya sistemindeki yükleyici programa atlamaktadır. Benzer biçimde Ext-2 dosya sisteminde de boot blokta işletim sistemini yükleyen bir kod vardır. Görüldüğü gibi işletim sistemini asıl yükleyen kod ilgili disk bölümünün ilk sektöründeki yükleyici koddur.

Boot işleminin özeti şöyledir: Makina açıldığında akış BIOS'tan başlar. BIOS diskin ilk sektörünü (MBR) belleğe okuyarak oraya jump eder. Buradaki program aktif disk bölümünü tespit ederek onun ilk sektörünü belleğe okur ve oraya jump eder. İşletim sistemini de oradaki kod yükler.

Örnek Çalışma: Bilgisayar Açıldığında Bizim Yerleştirdiğimiz Bir Programın Çalıştırılması

Bunu sağlamanın en pratik yolu sembolik makine dilinde bir program yazıp onu hard diskin, flash belleğin ya da CD'nin ilk sektörüne yerleştirmektir. Ve makineyi o sürücüden boot edilecek biçimde "CMOS Setup"tan ayarlamaktır. Ekran Intel gerçek modda merhaba dünya yazısını çıkartan bir nasm programı şöyle yazılabilir:

```
[BITS 16]
mov     ax, 07C0h
mov     ds, ax
mov     si, message
REPEAT:
mov     al, [si]
test    al, al
jz      EXIT
```

```

mov     ah, 0x0e
mov     bl, 7
int     0x10
inc     si
jmp     REPEAT

```

```

EXIT:
jmp $

```

```

message db "this is a test", 0

```

```

times 510 - ($ - $$) db 0
dw 0xAA55

```

Derleme işlemi şöyle yapılabilir:

```

nasm -f bin -o boot.bin boot.asm

```

Elde edilen boot.bin dosyası flash belleğe "Win32DiskImager" gibi bir programla yazılıp makine boot edilebilir.

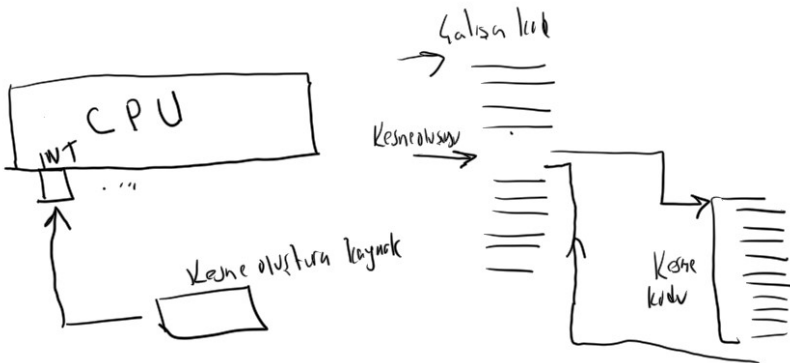
Kesmeler (Interrupts)

Bir kodun çalışmasına ara verilip başka bir kodun çalıştırılması ve bu bitince önceki kodun kaldığı yerden çalışmaya devam etmesi sürecine "kesme (interrupt)" denilmektedir. "Interrupt" sözcüğü İngilizce "araya girme (özellikle konuşmalarda vs.) anlamına gelmektedir".

Kesme işlemciye özgü bir kavramdır. İşlemcilerin hemen hepsinde (mikrodenetleyiciler de dahil olmak üzere) çeşitli biçimlerde kesme kavramları vardır. Kesmeler fonksiyon çağrılarına (yani CALL işlemine) benziyor olsa da aslında çeşitli bakımlardan fonksiyon çağrılarından farklıdır.

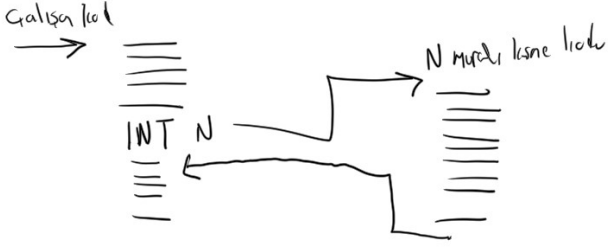
Kesmeler oluş biçimine göre üçe ayrılmaktadır:

1) Donanım kesmeleri (hardware interrupts): Kesme dendiğinde default olarak donanım kesmeleri anlaşılır. Neredeyse hemen her işlemcide ve mikrodenetleyicide donanım kesmeleri vardır. Donanım kesmeleri işlemcinin özel bir pini (bu pine genellikle INT pini denir) dışarıdan elektriksel olarak uyarılarak oluşturulmaktadır. (Yani donanım kesmeleri asenkron olarak dışarıdan başka bir donanım birimi tarafından işlemcinin kesme uyarılarak oluşturulur):



2) Yazılım Kesmeleri (Software Interrupts): Yazılım kesmeleri programcı tarafından makine koduyla koda dahil edilerek oluşturulan kesmelerdir. Örneğin Intel işlemcilerinde INT makine komutu yazılım kesmesi oluşturmak için kullanılmaktadır. Yazılım kesmelerinde kesmeyi oluşturan kaynak bizzat programcının kendisidir. Zaten yazılım kesmelerinin mekanizma olarak fonksiyon çağrılarında (CALL işleminden) çok büyük farkları yoktur.

Pek çok işletim sisteminde sistem fonksiyonları CALL makine komutu yerine yazılım kesmeleriyle (örneğin Intel'de INT makine komutuyla) çağrılmaktadır.



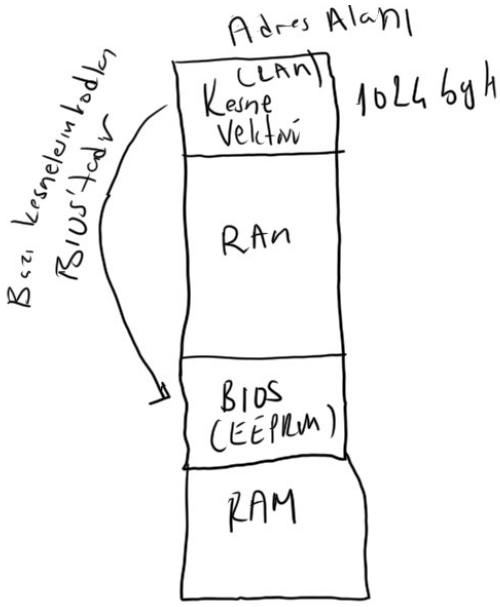
3) İçsel kesmeler (internal interrupts): Bu kesmeler bizzat işlemcinin kendisi tarafından oluşturulmaktadır. Intel terminolojisinde bunlara "fault" ya da "exception" da denir. Sayfalama mekanizması ve koruma mekanizması hep bu tür içsel kesmelerle yönetilmektedir. Örneğin işlemci yetkisi olmayan kod bir bellek alanına erişilmeye çalışıldığında "page fault" denilen bir içsel kesme oluşturur. Bu içsel kesme sonucunda işletim sisteminin kesme kodu çalışır. O kod da prosesi sonlandırır. Ya da örneğin biz Windows gibi Linux gibi sistemlerde bir göstericiye rastgele bir adres yerleştirip onun gösterdiği yere erişmek istersek bu biçimde bir içsel kesme oluşacaktır. Yani içsel kesmeler işlemcinin bir makine komutunu çalıştırırken bazı uygunsuzluklar yüzünden kendisinin oluşturduğu kesmelerdir.

Kesmenin cinsi ne olursa olsun bir kesme oluştuğunda araya girilip çalıştırılan koda "kesme kodu (interrupt handler)" denilmektedir. Pek çok işlemcide kesmelerin numaraları vardır. Yani numaralar sayesinde çok sayıda kesme varmış gibi bir etki oluşturulabilmektedir. Örneğin Intel işlemcilerinde toplam 256 kesme vardır. Zaten örneğin INT makine komutu kesme numarasını argüman olarak alır. Örneğin:

```
....  
int          0x10          ; 10h numaralı kesme yazılımsal olarak çağrılıyor  
....
```

Geleneksel olarak kesme numaraları Intel'de 16'lık sistemde belirtilmektedir.

Peki bir kesme oluştuğunda kimin kodu çalıştırılmaktadır? İşte Intel işlemcilerinde bir kesme oluştuğunda işlemci hangi kodu çalıştıracığına RAM'de "kesme vektörü (interrupt vector)" denilen bir yere bakarak karar verir. Kesme vektörü gerçek modda belleğin tepesindeki ilk 1024 byte'tır. Her kesme kodunun segment:offset değerleri (adresleri) burada yazılıdır. İşlemci n numaralı bir kesme oluştuğunda vektörün n numaralı elemanına bakarak kesme koduna dallanır. Tersten gidersek Intel işlemcilerinde gerçek modda n numaralı bir kesme oluştuğunda kendi kodumuzun çalıştırılabilmesini sağlamak için kodumuzun adresini kesme vektörünün n'inci elemanına yazmamız gerekir. Geçrk modda kesme vektörü yukarıda da belirtildiği gibi RAM'in tepesindeki ilk 1024 byte'tır. Buraya bir değer yazmak gerek C'de gerekse sembolik makine dilinde çok kolaydır.



Burada özellikle bir kesme oluştuğunda sistem programcısının kesme vektörüyle oynayarak kendi kodunun çalışmasını sağlayabildiğine dikkat ediniz. Kullandığımız PC'ler reset edildiğinde kesme vektörü de sıfırlanmış durumdadır. Çalışma BIOS'ta başlar buradaki kod kesme vektörünü de doldurmaktadır. Pek çok kesmenin kodu (örneğin 10h, 13h, 16h gibi) BIOS'tadır. Dolayısıyla BIOS'taki başlangıç kodu kesme vektörünü bu kodları gösterecek biçimde düzenlemektedir.

Bir kesmenin orijinal kodunu vektörden alıp saklayabiliriz. Sonra vektörü kendi kodumuzu gösterecek biçimde değiştirebiliriz. Kendi kodumuz içerisinde de orijinal kesme kodunu çağırabiliriz. Bu sürece kesmenin kancalanması (hook edilmesi) denilmektedir. DOS zamanındaki pek çok virüs 8h gibi 13h gibi BIOS kesmelerini kancalıyor ve bulaşmayı böyle sağlıyordu.

Intel işlemcileri reset edildiğinde "gerçek mod (real mode)" denilen bir modda çalışmaya başlar. Gerçek mod işlemcinin 16 bit 8086 gibi (yani DOS'un kullanıldığı işlemci) çalıştığı moddur. Intel işlemcilerinin "korunmalı mod (protected mode)" denilen diğer bir çalışma modu daha vardır. Örneğin Windows, Linux korunmalı modda çalışmaktadır. Bu işletim sistemleri yüklenirken işlemci gerçek moddan korunmalı moda geçirilmektedir.

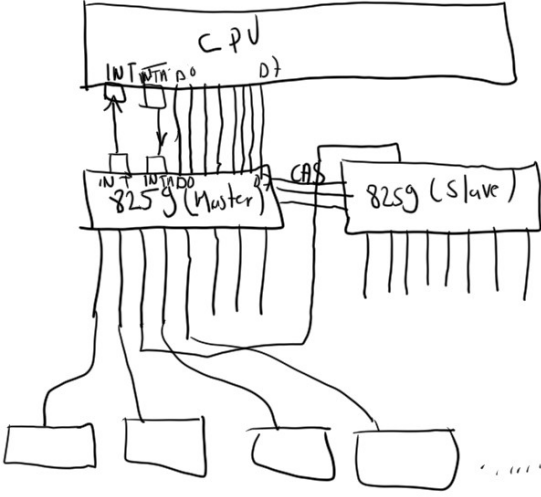
Korunmalı modda kesme vektörünün formatı ve yeri değişmektedir. Korunmalı modda artık kesme vektörüne "kesme betimleyici tablosu (interrupt descriptor table)" denir. Kesme betimleyici tablosu belleğin herhangi bir yerinde oluşturulabilmektedir. İşlemci kesme betimleyici tablosunu "IDTR (Interrupt Descriptor Table Register)" yazmacının gösterdiği yerde aramaktadır. İşletim sistemlerinin yükleyicileri korunmalı moda geçmeden önce kesme betimleyici tablosunu oluşturur. IDTR yazmacının burayı göstermesini sağlar.

Peki korunmalı modda Windows ya da Linux'ta biz bir kesme oluştuğunda kendi kodumuzun çalışmasını sağlayabilir miyiz? Bunu doğrudan yapamayız. Çünkü Windows ve Linux sistemlerinde kesme vektörüne ya da kesme betimleyici tablosuna "user mode"tan erişemeyiz. Bunun için aygıt sürücü oluşturmak gerekir. Tabii aygıt sürücülerin belli mimarileri vardır. Biz aygıt sürücünün içerisinde kesme vektörüne ya da kesme betimleyici tablosuna yine doğrudan erişmemeliyiz. İşletim sisteminin bizim için uygun gördüğü bir biçimde kesme işlemleri için devreye girebiliriz. İşletim sistemleri kesmelerden pek çok aygıt sürücü faydalanabilsin diye genellikle bir zincir oluşturma yöntemi izlemektedir.

PC'lerde Donanım Kesmeleri

PC'lerde işlemcinin kesme ucuna (INT ucuna) doğrudan donanım kaynakları bağlanmamıştır. Çünkü birden fazla kaynağın elektriksel olarak bu ucu uyarması sorunlara yol açabilmektedir. İşte bunu düzene sokmak için

ilk PC'lerde (IBM PC ve uyumlu makineler) Intel'in 8259 Kesme Denetleyicisi (Programmable Interrupt Controller) kullanılmıştır. 8259 kesme denetleyicisinin 8 giriş ucu vardır. Bu uçlara donanım birimleri bağlanabilir. Bu birimler bu uçları elektriksel olarak uyardığında kesme denetleyicisi de CPU'nun INT ucunu uyaracaktır. AT'lerle (PC'lerin 80286 işlemcisiyle yapılmış biçimi) birlikte kesme denetleyicisi sayısı ikiye çıkartılmıştır. Intel'in 8259 kesme denetleyicisinin birbirlerine bağlanabilmesi için birinci kesme denetleyicisinin (master) bir giriş ucunun ikinci kesme denetleyicisine (slave) bağlanması gerekmektedir. İşte IBM birinci kesme denetleyicisinin iki numaralı ucunu bu amaçla kullanmıştır. Bugün hala kullandığımız PC ve notebook'lardaki kesme sistemi bu mimariye dayanmaktadır. 8259 bir chip set'in içerisinde bizim görmeyeceğimiz biçimde yerleştirilmiş durumdadır. Bugünkü PC'lerde kesme sistemi aşağıdaki şekilde görüldüğü gibidir:



Tipik donanım kesmesi oluşturma protokolü de şöyledir:

- 1) Donanım birimleri kesme denetleyicisinin bir ucunu (elektriksel olarak) uyarır.
- 2) Kesme denetleyicisi CPU'nun INT ucunu uyarır.
- 3) CPU bunu kabul ederse bunu INTA ucunu aktive ederek kesme denetleyicisine bildirir.
- 4) Kesme denetleyicisi kesmenin kabul edildiğini INTA ucundan anlar kesme numarasını D0-D7 uçlarından CPU'ya bildirir.
- 5) CPU o anda çalıştırdığı koda ara verir. Kesme vektörüne ya da kesme betimleyici tablosuna bakarak kesme koduna dallanır.

CPU INT ucu uyarıldığında donanım kesmesini kabul etmek zorunda değildir. CPU donanım kesmelerine FLAGS yazmacının IF (Interrupt flag) set edilerek kapatılabilir. Bu işlem CLI (Clear Interrupt Flag) makine komutuyla tek hamlede de yapılabilmektedir. CPU'yu yeniden donanım kesmelerine açmak için ise STI (Set Interrupt Flag) makine komutu kullanılabilir.

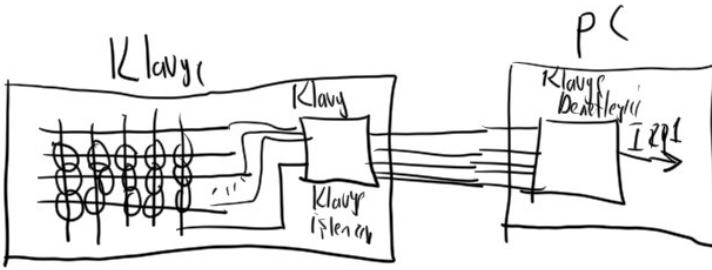
Bugünkü PC mimarisinde kesme denetleyicisinin giriş uçlarından her birine IRQ denilmektedir. Birinci kesme denetleyicinin ilk ucu IRQ0 olmak üzere her IRQ'ya bir numara verilmiştir. IRQ2 hattı kaskat bağlantı için kullanıldığından bu hatta bir donanım birimi bağlı değildir.

Şimdi IRQ hatlarına bağlı donanım birimlerini tek tek inceleyelim:

IRQ0: Bu hatta Intel'in 8254 zamanlayıcısı (Programmable Interval Timer) bağlıdır. Bu denetleyici programlanarak saniyede belli sayıda pals üretmesi sağlanabilmektedir. İşte her pulse kesme denetleyicisinin IRQ0 hattını aktive eder. Böylece saniyede belli bir periyotta donanım kesmesi oluşturulmuş olur. Bilgisayar açıldığında BIOS bu denetleyiciyi saniyede 18.2 kere darbe üretecek biçimde programlanmaktadır.

Windows gibi Linux gibi işletim sistemlerinin thread'ler arası geçiş (context switch) mekanizması tamamen IRQ0 ile yapılmaktadır. Bu işletim sistemleri artık makine hızlıysa bu denetleyiciyi saniyede 1000 kere darbe üretecek biçimde programlarlar (yani 1 milisaniye periyoda). Böylece işletim sistemi her timer kesmesi geldiğinde bazı ayarlamalar yapar. Bu kesmeler belli bir sayıya gelince (örneğin 60 tanede bir. Yani 60 milisaniyede bir) thread'lerarası geçiş uygular. 8254 timer kesmesi kapatılırsa sistem hemen çökebilir. İşte bu nedenle CLI gibi makine komutları koruma mekanizması altında yalnızca kernel moddaki kodlar tarafından kullanılabilir.

IRQ1: Bu hatta Intel'in 8042 (ya da benzeri) klavye denetleyici birimi bağlıdır. Klavye denetleyicisi de klavyenin içerisindeki klavye işlemcisi ile (keyboard encoder) bağlantı halindedir. Bu durum şekilsel olarak şöyle gösterilebilir:



Klavye devesi bir çeşit matris gibidir. Tellerin kesişim noktasında tuşlar vardır. Böylece elektriksel olarak hangi tuşa basıldığı anlaşılabilir. Bu matrisin uçları klavye içerisindeki klavye işlemcisine (keyboard encoder) giridi olarak verilir. Klavye işlemcisi hangi tuşa basılmış olduğunu tespit eder ve bunu klavye kablosu yoluyla bilgisayar tarafındaki klavye denetleyicisine iletir. Klavye denetleyicisi de IRQ1 hattından kesme oluşturur. İşletim sisteminin kesme kodu (ya da default olarak BIOS'taki kod) devreye girer. Klavye denetleyicisinden basılan tuşu alır.

Klavye denetleyicisi basılan tuşun numarasını tespit etmektedir. Tuşların üzerindeki sembollerin bir önemi yoktur. Klavye üzerindeki her tuşa bir numara verilmiştir. Buna ilgili tuşun "tarama kodu (scan code)" denilmektedir. Klavye içerisindeki işlemci de klavye denetleyicisine bu tarama kodunu göndermektedir. Kesme kodu bu tarama kodunu aldığı anda tuşun hangi tuş olduğunu anlar. İşletim sistemleri bölgesel ayarlara bakarak o tuşun aslında hangi karakterle (klavye üzerinde gördüğümüz sembol) ilişkili olduğunu tespit eder. Sanki klavyeden o tuşa basılmış gibi bir etki yaratır. Yani klavyedeki karakterlerin yer değiştirmesi (klavyenin transpoze edilmesi) tamamen yazılım yoluyla yapılmaktadır. (Örneğin Türkçe ve İngilizce klavyenin devreleri arasında bir fark yoktur.

Klavyede yalnızca tuşa basıldığında klavye denetleyicisine bildirmde bulunulmaz. Aynı zamanda el tuştan çekildiğinde de bildirimde bulunulmaktadır. Klavyenin tuşuna basıldığında gönderilen koda "make code", el tuştan çekildiğinde gönderilen koda ise "break code" denilmektedir. Bilgisayar tarafındaki klavye denetleyicisi hem make code için hem de break code için kesme oluşturmaktadır. Örneğin önce A tuşuna basmış olalım sonra parmağımızı çekmeden B tuşuna basalım sonra parmağımızı önce A tuşundan sonra B tuşundan çekelim. Toplam dört kesme oluşacaktır ve klavyeden gönderilen kodlar şunlar olacaktır:

- A tuşu için make code
- B tuşu için make code

- A tuşu için break code
- B tuşu için break code

Bugün kullandığımız PC klavyelerinde klavye içerisindeki işlemci 1 byte make code'u seri olarak bilgisayar tarafına iletmektedir. El tuştan çekildiğinde klavye içerisindeki işlemci bu sefer önce bir F0 byte'ını sonra da make code'un aynısını (toplam 2 byte) break code olarak gönderir.

Klavye üzerindeki ışıklı tuşların ışıkları klavye devresi tarafından yakılıp söndürülür ancak bu işlem otomatik yapılmamaktadır. Biz örneğin NUM-LOCK tuşuna bastığımızda bu tuşa basıldığı bilgisayar tarafına iletilir. Oluşan kesmenin kodu bu tuşun NUM-LOCK tuşu olduğunu anlar bu kez klavye içerisindeki işlemciyi programlayarak bu ışığın yakılmasını sağlar.

Klavyedeki tuşlara yönelik önemli bir durum da "typematic" sürecidir. Bir tuşa basıp parmağımızı o tuşa beklettiğimizde sürekli o tuşa basılıyormuş gibi oluşan işleme "typematic" denilmektedir. Typematic periyodu ve ilk basıştan ne kadar zaman sonra typematic'e başlanacağı klavye içerisindeki işlemci tarafından belirlenir. Bu işlemci bilgisayar tarafındaki klavye denetleyici yoluyla (tabii dolayısıyla yazılım yoluyla) programlanabilmektedir.

IRQ2: Bu hat kaskat bağlantısından dışsal bir donanım birimine bağlı değildir.

IRQ3 ve IRQ4: Bu hatlara 8250 ya da 16550 UART işlemcisi bağlıdır. UART işlemcileri kendilerine bir bilgi geldiğinde kesme oluşturabilme (isteğe bağlı olarak) yeteneğine sahiptir. Böylece seri porta bir bilgi geldiğinde bilgisayar bundan haberdar olabilir. Artık kişisel bilgisayarlarımızda seri portlar bulunmuyor. Biz seri portlar yerine çok daha yetenekli USB portlarını kullanıyoruz. Dolayısıyla bu IRQ hatları da artık pek kullanılmamaktadır. Ancak seri port kullanmaya devam eden eski aygıtlar bulunabilmektedir. İşte bu tür durumlarda dönüştürücülerle USB portu seri port gibi kullanılabilir.

IRQ5: Bu eski PC'lerdeki LPT2 olarak bilinen paralel port işlemcisine bağlıdır. Paralel portlar eskiden bilgisayar yazıcı haberleşmesi için kullanılıyordu. Artık uzunca bir süredir paralel portlar da bilgisayarlarımızda bulunmamaktadır. Artık bilgisayar yazıcı arasındaki bağlantılar için yine USB portlarını kullanmaktayız. IRQ5 günümüzde artık ses kartları tarafından kullanılmaktadır. Ses kartları da bilgisayarda kesme oluşturabilmektedir.

IRQ6: Eskiden floppy disketler vardı. Bu disketler floppy sürücüsüne takılarak çalıştırılırdı. İlk floppy sürücüsü için Microsoft A harfini ikinci floppy sürücüsü için B harfini kullanılıyordu. Hard disklere de C harfi verilmişti. Bugün artık floppy disketler kullanılmamaktadır. İşte floppy disketleri yönetmek için "floppy controller (INTEL 8272)" denilen denetleyiciler vardı. Floppy bir sektörün okunmasını ya da yazılmasını bitirdiğinde floppy denetleyicisi bunu işletim sistemine bildirmek için donanım kesmesi oluşturuyordu. İşte IRQ6 hattı floppy denetleyicisine bağlıydı. Bugün artık bu hat kullanılmamaktadır.

IRQ7: Bu hatta da eskiden LPT1 isimli paralel port işlemcisi bağlıydı. Bugün artık bu hatta eğer varsa ikinci ses kartının işlemcisi bağlıdır.

IRQ8: Bu hat ikinci (slave) kesme denetleyicisinin ilk ucudur. Bu hatta "Real Time Clock" denilen saat işlevini gören devre bağlıdır. Eskiden bilgisayarlarımızda saat yoktu. 80'li yılların sonlarına doğru "real time clock" denilen işlemcilerin PC'lere eklenmesiyle bilgisayar kapatıldığında zamanın tutulması sağlanmıştır. Real Time Clock denetleyicisinin içerisinde CMOS RAM bölgesi de vardır. Yani bilgisayar açılırken Del ya da F1 tuşuna basıldığında çıkan setup ekranındaki bilgiler aynı işlemcinin üzerindeki belleğe yazılmaktadır. Tabii bu Real Time Clock denetleyicisi bir pille beslenmektedir. Genellikle bu pil bilgisayarın ticari ömründen daha fazla

dayanabilmektedir. İşte Real Time Clock denetleyicisi bir çeşit alarm mekanizması gibi belli bir tarih zamana gelindiğinde işlemciye kesme gönderebilmektedir.

IRQ9: Bu hat genellikle Intel'in ACPI denetleyicisine bağlıdır.

IRQ10-IRQ11: Bu IRQ hatları kullanıcılar için boş bulundurulmaktadır. Yani donanım tasarımcıları eğer kesme oluşturmak istiyorlarsa bu IRQ hatlarını kullanırlar. Bugün masaüstü bilgisayarlarımızda kullandığımız PCI slotlar aslında IRQ hatlarına da bağlıdır. İşte PCI kartları tasarlayan donanımcılar birtakım olaylarda PCI slot yardımıyla IRQ oluturabilmektedir.

IRQ12: Fare dediğimiz aygıt her hareket ettiğinde IRQ oluşturmaktadır. Bu IRQ kesme kodu da (Interrupt handler) işletim sistemi tarafından fare okunu hareket ettirmede kullanılır. İşte eski klasik PS/2 fare portları bu hatta bağlıydı. Bugün artık fareler de USB portlarla kullanılmaktadır.

IRQ13: Bu IRQ hattına Intel'in 80387, 80487 matematik işlemcileri bağlıdır. Matematik işlemci sorunla karşılaştığında ana işlemciye bu IRQ hattını kullanarak kesme gönderiyordu. Intel'in 80486 DX modeliyle modeliyle birlikte matematik işlemciler de ana işlemci ile aynı entegre devre içerisine yerleştirilmişlerdir.

IRQ14-IRQ15: Bu IRQ hatları birincil ve ikincil IDE denetleyicisine bağlıdır. Yani hard disklerde ve CD/DVD ROM'larda okuma yazma işlemleri yapıldığında IDE denetleyicisi bu hattı kullanarak işlemciye kesme yollamaktadır.