

Sistem Programlama ve İleri C Uygulamaları

Kurs Notları

Kaan ASLAN

C ve Sistem Programcılar Derneği

Güncellemeye Tarihi: 16/12/2020

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", programmanın 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 aygit 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 yazılmaktadır. 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 aygit 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. Şöyled 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 ispatlanılmış olsa bile böyle atıf vardır.) Daha sonra 1800'lü yılların oratalarından itibaren elektronikte hızlı bir ilerleme yaşanmıştır. Bool cebri ortaya atılmış, çeşitli devre elemanları kullanılmaya başlanmış ve mantık devreleri üzerinde çalışmalar başlatılmıştır. 1900'lü yılların başlarında artık yavaş yavaş elektromekanik bilgisayar fikri belirmeye başlamıştır. 1930'lu yıllarda Alan Turing konuya matematiksel açıdan yaklaşmış ve bugünkü bilgisayar benzeri bir makinenin hangi matematik problemleri çözebileceği üzerine kafa yormuştur. Turing bir şerit üzerinde ilerleyen bir kafadan oluşan ve ismine "Turing Makinesi" denilen soyut makine tanımlamıştır ve bu makinenin neler yapabileceği üzerinde kafa yormuştur. ACM Turing'in anısına bilgisayarın Nobel ödülü gibi kabul edilen Turing ödülleri vermektedir.

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

Modern bilgisayar tarihi üç döneme ayrılarak incelenebilir:

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

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

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

Entegre devreye benzer ilk çalışma aslında ilk olarak 1949 yılında Alman mühendis Werner Jacobi tarafından yapıldı. Ancak entegre devre fikri 1952 yılında İngiliz Geoffrey Dummer tarafından ortaya atıldı. Fakat gerçek anlamda ilk gerçekleştirmesi 1958 yılında Texas Instruments şirketi çalışanı Jack Kilby tarafından yapıldı. Kilby'den habersiz olarak yaklaşık altı ay sonra benzer entegre devre gerçekleştirmesi Fairchild Semiconductor firmasında Robert Noyce tarafından da yapıldı. Kilby ile Noyce patent konusunda mahkemelik olmuşlarsa da sonra anlaşma sağlanmış ve her iki kişi adına patentleme yapılmıştır. Robert Noyce aslında transistörü bulan ekipteki William Schockley'nin yanında çalışıyordu. Bu ekipte Gordon Moore da vardı. Schockley'nin yönetiminden memnun olmayan bu ekip Fairchild Semiconductor şirketine geçmiştir. Noyce şirketin genel müdürü, Moore da arge müdürü olmuştur. Daha sonra 1968 yılında Robert Noyce, Gordon Moore Fairchild Semiconductor firmasından ayrılarak Intel'i kurdu. İki 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. İlana o zaman Harward'ta öğrenci olan Bill Gates ve Paul

Allen başvurdular. Böylece Altair daha sonra Basic ile piyasaya sürüldü. Gates ve Allen okuldan ayrıldılar ve 1975 yılında Microsoft firmasını kurdular. (O zamanlar bu yeni kişisel bilgisayarlarla mikrobilgisayarlar denilmektedir). 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.

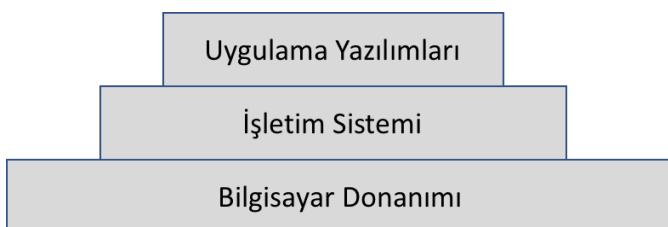
IBM kişisel bilgisayar konusunu hafife aldı. Fakat yine de bir ekip kurarak bugün kullandığımız PC'lerin donanımını tasarlamıştır. Ancak IBM küçük iş olduğu gereğiyle bunlara işletim sistemini kendisi yazmadı, taşeron bir firmaya yazdırma isted. Bu süreç içerisinde Microsoft IBM ile anlaşarak DOS işletim 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 ihmali 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 büydü.

1.3. İşletim Sistemleri

Bu bölümde işletim sistemleri giriş düzeyinde ele alınmaktadır.

1.3.1. İşletim Sistemi Nedir?

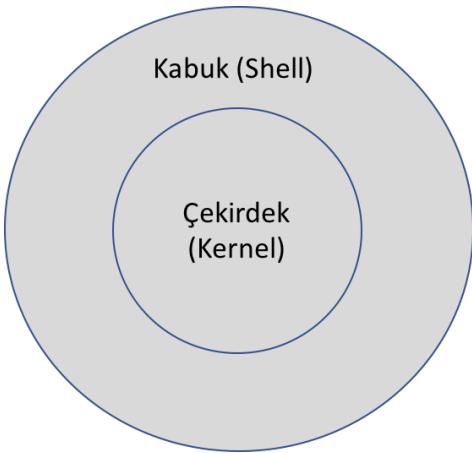
İşletim sistemleri bilgisayar donanımının kaynaklarını yöneten, bilgisayar donanımı ile kullanıcı arasında arayüz oluşturan temel bir sistem programıdır. Pek çok teorisyen işletim sistemini bir kaynak yöneticisi (resource manager) olarak tanımlamıştır.



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 kısımdir. Örneğin UNIX/Linux sistemlerinde komut satırı, Gnome, KDE gibi pencere yöneticileri, Windows'taki masasüstü (Explorer) kabuk kısımdir. Kullanıcılar genellikle çekirdeğin görmezler.



İşletim sistemleri çeşitli biçimlerde sınıflandırılabilir.

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

Kullanıcı Sayısına Göre: Birden fazla farklı kullanıcının çalışabildiği sistemlere "çok kullanıcılı (multiuser)", tek bir kullanıcının çalışabildiği sistemlere "tek kullanıcılı (singleuser)" sistemler denilmektedir. Genellikle çok prosesli işletim sistemleri aynı zamanda çok kullanıcılı sistemlerdir. Birden fazla kullanıcının söz konusu olduğu sistemlerde kullanıcıların yetkilerinin yarlanması, birbirlerinin alanlarına erişememesi, sistem kaynaklarını belli oranlarda paylaşması gerekmektedir. Örneğin DOS tek kullanıcılı bir sistemdi. Halbuki Windows, UNIX/Linux ve Mac OS X sistemleri çok kullanıcılı 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 kendis aralarında "hard realtime" ve "soft realtime" işletim sistemleri olmak üzere ikiye ayrılmaktadır. Hard realtime sistemlerde dışsal olaylara yanıt verme bakımından çok güvenilir olma iddiasındadır. Soft realtime sistemler ise bu konuda daha gevşektir.

Dağıtıklık Durumuna Göre: İşletim sistemleri dağıtıklık durumuma göre "dağıtık olan (distributed)" ve "dağıtık olmayan (nondistributed)" sistemler biçiminde ikiye ayrılabilir. Dağıtık işletim sistemlerinde sistem birden fazla bilgisayardan oluşan tek bir sistem gibi davranışmaktadır. Örneğin 10 tane makineyi tek bir sistem olarak düşünübilirsiniz. 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 yapılmaktadır.

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

aklılı 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#'tir. 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 ayıralım. 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 ayıralım. Bazı işletim sistemlerinin istemci versiyonları birbirlerinden ayrılmıştır. Bazılarında ise bu ayırm yapılmamıştır. Örneğin Windows 7, 8, 10 sistemleri bu bakımdan istemci (client) sistemleridir. Halbuki Windows Server 2012, 2016, 2019 sunucu sistemleri oalrak piyasaya sürülmüştür. Eskiden Mac OS X'in istemci ve sunucu 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 kullanılabilmektedir. 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 Doğusu 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 şimdilik mikrodenetleyicilere kod yazanlarda olduğu gibi.) Transistor bulunuktan 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 ihti. 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 girdi 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 güçlüydi 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 çalışırabilirdi 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 paylaşımı (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 paylaşımı 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 versyonunu piyasaya sürmüştür. DEC'in PDP-8'inin mini bilgisayar devrimini başlattığı söylenebilir. Bu model 50000'in üzerinde satışa ulaşmıştır. UNIX işletim sistemi 1969 yılında ilk kez DEC'in PDP-7 modeli üzerinde yazılmıştır. DEC PDP -7 18 bitlik bir makineydi. Makine DECsys denilen işletim sistemi benzeri bir yönetici programla beraber satılıyordu. DEC'in PDP-10 modelinde DEC işletim sistemi olarak TOPS-10 isimli bir sisteme geçti. PDP-10 26 bitlik bir makineydi.

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

1970'li yılların ikinci yarısında entegre devrelerin de geliştirilmesiyle "ev bilgisayarları (home computer)" ortaya çıkmaya başladı. Bunlarda genellikle BASIC yorumlayıcıları ile iç içe geçmiş CP/M tıa 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 gergisayarlarındandı. 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 Macintosh bilgisayarını çıkarttı. Lisa ile birlikte Apple grafik tabanlı işletim sistemlerine geçiş yaptı. Lisa ve sonraki Apple bilgisayarlarının hepsi grafik bir arayüze sahiptir. Macintosh markası daha sonra Mac olarak telefuz edilmeye başlandı. Lisa bilgisayarlarında kullanılan işletim sistemi LisaOS ismindeydi. Apple daha sonra Macintosh 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 sonda bu sistem açık hale getirildi ve OPENSTEP ismini aldı. Dünyanın ilk Web tarayıcısı Tim Berners Lee tarafından Cern'de NeXT bilgisayarları üzerinde gerçekleştirılmıştır.

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 kararlı aldı. Mac'ler Mac OS X ile birlikte yeni bir çekirdeğe geçtiler.

DOS işletim sistemi text ekranda çalışıyordu. Microsoft da geleceğin grafik tabanlı işletim sistemlerinde olduğunu gördü ve yavaş yavaş DOS'u bırakarak grafik tabanlı bir sisteme geçmeyi planladı. Bunun için Windows isimli grafik arayüzün birinci versyonunu 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. Microsoft 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 bölgelerde ele alınmaktadır.

1.3.3. UNIX Türevi İşletim Sistemlerinin Tarihsel Gelişimi

UNIX işletim 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 devralılmıştır.

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 programlamalı 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 lisanslanmıştır. UNIX işletim sistemi bir araştırma projesi olarak organize edilmiştir. 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 versyonunun 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 bilgiyalarlarında 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 sistemi idi. 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 NetBSD ve 1993 yılında, OpenBSD ise 1996 yılında piyasaya çıkmıştır. Sürdürülmeye 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 taşınabilirdir ve geniş bir porta sahiptir. Daha çok bilimsel çalışmalarında tercih edilmektedir. OpenBSD güvenliğin önemli olduğu alanlarda tercih edilmektedir.

SunOS (Solaris): Sun firmasının BSD kodlarıyla oluşturduğu UNIX türevi işletim sistemi idi. 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 ettiirmektedir. 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 ilk 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 öğrenciyen bir işletim sistemi yazmaya niyetlenmiştir. O zamanlarda telif uygulanmayan UNIX türevi bir işletim sistemi kalmamıştı ve GNU projesinin işletim sistemi de (GNU Hurd) bitirememiştir. MINIX sisteminin lisansı yalnızca akademik kullanıcılar için sınırlanmıştır. 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 zamandakı 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 genelk 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 görsünüz. Çekirdeğin dışındaki tüm yazılımlar (örneğin init prosesinden başlayarak, kabuk yazılımları, paket yöneticileri, pencere yöneticileri vs.) hepç 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, ZYPPER gibi), içeren 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 Dağıtımları: En önemli ve en eski Linux dağıtımlarından biridir. Red Hat Enterprise Linux en önemli Fedora türevidir. Knoppix, Mint, Ubuntu Debian türevi dağıtımlarıdır.

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 dağıtımlarıdır. 2000 yılında ilk sürümü yapılan Red Hat Enterprise Linux (RHEL) en önemli Fedora türevidir. Ondan da CentOS, Scientific Linux gibi dağıtımlar türetilmiştir. CentOS server makinelerde en yaygın kullanılan Linux versiyonudur.

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

Slackware: En eski Linux dağıtımdır. 1993 yılında oluşturulmuştur. Südürümü 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 kalımı 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 Rhapsody'yi çıkardı. 1998'de de yeni işletim sisteminin Mac OS X olacağını açıkladı. Daha sonra Rhapsody'den Darwin projesi türedi. Darwin projesi ayrı bir işletim sistemi olarak da yüklenemektedir. Ancak Darwin grafik arayüzü olmadığı için Mac programlarını çalıştırılamaz.

Darwin'den çeşitli projeler türetilmiştir. Bunlardan biri Apple tarafından 2002'de başlatılan OpenDarwin'dır. 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 ("Karneji" diye okunuyor) üniversitesi'nin Mach 3 mikrokernel çekirdeği ile 4.3BSD karışımı hibrit bir sistemdir.

Mac OS X sistemlerinin versyonları 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 (Mavericks, 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#'tir. 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

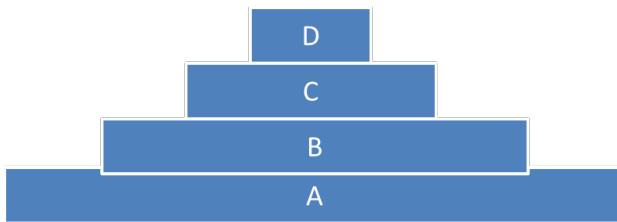
Mac OS X'in çekirdeği olan Darwin Carnegie Mellon Üniversitesi tarafından yazılan Mach çekirdeği ile BSD çekirdeğinin kodlarını hala kullanmaktadır. Bu anlamda Mac OS X orijinal bir kod temeline sahip değildir. Benzer biçimde Android ve IOS da orjinal kod temeline sahip işletim sistemleri değildir.

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.

Pekiyi "bir işletim sistemi yazdım" diyebilmek için ne gereklidir? Yanıt: Yapılan katığın çekirdek düzeyinde ve belli bir yoğunlukta olması gereklidir. 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 ayırip onları çağrılmaktır. Örneğin C'de proje içerisinde bir kod parçasının çeşitli yerlerde yinelendiğini düşünelim. Bu kod parçasını bir fonksiyon olarak bildirip tekrarlanan yerlerde o fonksiyonu çağrılabılır. 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 sınıfında toplanır, bu iki sınıf da o taban sınıfından 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 programclarının doğrudan çağrılabileceği, 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 programclarının 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 kullanımının 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 çağrıriz. 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 kullanılabilmektedir.

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

Bazı firmalar ayrı birimler olarak tasarlanmış mikroişlemcileri, RAM'leri, ROM'ları ve diğer bazı üniteleri tek bir entegre devrenin içerisine sıkıştırmaktadır. Bunlara genel olarak "SOC (System On Chip)" denilmektedir. SOC mikrodenetleyicilere benzese de aslında onlardan farklıdır. SOC'lar içerisindeki işlemcilerin ve belleklerin kapasiteleri yüksektir. Bunlar özel amaçlı üretilirler ve bunların devrelerde kullanılması 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:

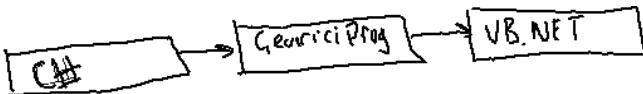


1.8. Gömülü Sistemler (Embedded Systems)

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 ucuzlaşmıştır ve artık gömülü sistemlerde doğrudan kullanılır hale gelmiştir. Bu kartlar tamamen bir bilgisayarın işlevsellüğüne sahiptir. Bunlara genellikle Linux işletim sistemi ya da Android işletim sistemi yüklenir. Böylece gömülü yazılımların güçlü donanımlarda ve bir işletim sistemi altında çalışması sağlanabilmektedir. Örneğin Raspberry Pi'a biz Mono'yu yükleyerek C#ta program yazıp onu çalıştırabiliriz.

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

Bir programlama dilinde yazılmış olan programı eşdeğer olarak başka bir dile dönüştüren programlara çeviriçi programlar (translators) denilmektedir. Çeviriçi programlarda dönüştürilmek 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 gereklidir. 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 gereklidir. Bu da kaynak kod güvenliğini bozur.

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

1.10. Decompiler'lar ve Disassembler'lar

Alçak seviyeli dillerden yüksek seviyeli dillere dönüştürme yapan (yani derleyicilerin yaptığından tam tersini yapan) yazılımlara "decompiler" denilmektedir. Örneğin C#'ta yazılmış derlenmiş olan .exe dosyasından 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 programcılar kendileri bazı önlemler almaktadır. Ancak C, C++ gibi doğal kod üreten derleyicilerin ürettiği kodlar geri dönüştürülememektedir.

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 yapılmırken derlemeyi IDE yapmaz. IDE derleyiciyi çalıştırır. IDE yardımcı bir araçtır, mutlak gereklili bir araç değildir.

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

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

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 dönüştürülmemesi gereklidir. Zaten derleyiciler de bunu yapmaktadır. Eğer bir çevirici program (yani derleyici) o anda çalışılmakta olan makinenin CPU'sunun işletebileceği makine kodlarını üretse CPU da bunları çalıştırırsa 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 işletilemeyecek) yapay bir kod üretmektedir. Bu yapay kodlara genel olarak "ara kodlar (intermediate codes)" denilmektedir. Bu arakodlar doğrudan CPU tarafından çalıştırılamazlar. Arakodlu çalışma Java ve .NET dünyasında ve daha başka ortamlarda kullanılmaktadır. Java dünyasında Java derleyicilerinin üretikleri 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 süreç (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 çalıştırın alt sisteme "Java Sanal Makinesi (Java Virtual Machine)", .NET ortamında ise CLR (Common Language Runtime)" denilmektedir.

Şüphesiz doğal kodlu çalışma arakodla çalışmadan daha hızlıdır. Pek çok benchmark testleri aradaki hız farkının %20 civarında olduğunu göstermektedir. Pekiyi arakodlu çalışmanın avantajları nelerdir? İşte bu çalışma biçimini 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 gereklidir. 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 dokümantete etmek koşuluyla derleyicileri yananlara 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 olacağı böyledir.

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

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 öngörlülmüştür. 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 kullanılabilmektedir.

1) Cygwin Ortamı ile 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 olandaklarıyla UNIX/Linux ortamını emüle ettiğine dikkat ediniz.

2) Sanallaştırma Yoluyla: Bugün VmWare, VirtualBox, Xen gibi sanallaştırma ve hypervisor yazılımlarla orijinal işletim sistemi tamamen sanallaştırma yoluyla çalıştırılabilmektedir. Artık Cygwin kullanımı Windows sistemlerinde bu nedenle çok azalmıştır. Sanallaştırmada "host" ve "guest" sistemler arasında "copy-paste" işlemleri de yapılmaktadır. 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 yazılmış bir dosya biçiminde diskte saklanması gereklidir. 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 yazılmış saklandıktan sonra derleme işlemi komut satırından şöyle yapılmaktadır:

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

ya da :

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

Örneğin:

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

ya da örneğin:

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

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

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

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

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

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

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

UNIX/Linux sistemlerinde bulunulan dizindeki bir programı komut satırından çalıştırabilmek için yalnızca dosyanın ismi yazılmaz. Onun dizini de belirtilmelidir. Tipik 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 gerekmektedir. 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 kullanılabilmektedir. 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 yalnız 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 kullanılabilmektedir.

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 çalışmaktadır. cl programı kaynak dosyayla aynı isimli çalıştırılabilir dosya oluşturur. Tabii biz cl ile yine derleme ve bağlama işlemlerini ayrı ayrı yapabiliriz. cl programının /C seçeneği "yalnızc 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>
```

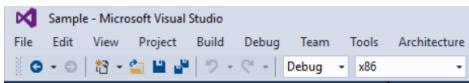
cl derleyicisinde Komut satırı seçenekleri '/' karakteriyle verilmektedir. Yine tipki gcc'de olduğu gibi cl derleyicisi de hiç derleme yapmadan doğrudan bağlayıcıyı çalıştırabilmektedir. Örneğin:

```
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 çalıştırduğumızda 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 sağlayabiliyoruz. 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 içerisindeinedir. Ancak Visual Studio IDE'si ile yapıyorsanız 64 bit derleme basit bir biçimde konfigürasyon seçeneklerinden X64 seçilerek de yapılabilir:



X64

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 ayırtırmakta (parse etmekte) kullanılan standart bir C fonksiyonudur. Örneğin aşağıdaki gibi bir yazı olsun:

"ankara,adana,izmir,kars"

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

"10/12/2007"

Burada yazı '/' karakterlerinden ayırtırlarak 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 ayrıstırılacak yazıyı bozmaktadır. Bu nedenle ayrıstırılacak 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 yazılabılır:

```

#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 fonksiyonun asıl yazı üzerinde değişiklik yapmayan bir biçimi şöyle yazılabılır:

```

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

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

    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 ((dynstr = (char *)malloc(pos - beg + 1)) == NULL)
    return NULL;
strncpy(dynstr, beg, pos - beg);
dynstr[pos - beg] = '\0';

if (*pos != '\0')
    ++pos;

return dynstr;
}

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

    for (str = mystrtok(text, " ,"); str != NULL; str = mystrtok(NULL, " ,")) {
        puts(str);
        free(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, baqşarızlık 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") == -1) {
        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 işlettirir. 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 çağrılabılır. Bu durumda fonksiyon sıfır dışı bir değere geri dönerse ilgili sistemde kabuk programı vardır, sıfırda 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 modda çalıştırılmaktadır: "interaktif mod" ve "interaktif olmayan (yani tek komutluk)" mod. Interaktif 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 modda tek bir komut çalıştırılıp kabuktan çıkmaktadır. 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 bugünlere en çok kullanılan dolayısıyla "default" durumda olan "bash (Bourne Again Shell)" kabuğudur. Tek komutluk çalışma için Windows sistemlerinde "cmd.exe" programı "/C" seçeneği ile, UNIX/Linux sistemlerinde ise "bash" programı "-c" seçeneği ile çalıştırılmaktadır.

Anahtar Notlar: Linux asında bir çekirdek projesidir. Bir Linux dağıtıımıyla 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 içeresine 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 önişlemci modülü ilgili kaynak dosyayı okur, onu #'lı ifadelerden arındırır ve bir geçici dosyaya yazar. Derleme modülü de asında önişlemcinin oluşturduğu bu dosyayı alarak derler. Derleme işlemi bittiğinde de önişlemcinin 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 kapatıldığından 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 kapatmazsa 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 diziye 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 deneyecek kadar zayıftır.) tmpnam fonksiyonunun dosyayı açmadığına yalnızca geçici dosyanın ismini bize verdiğine dikkat ediniz. Dosyayı açmak programcının sorumluluğundadır.

Aşağıdaki örnekte bir dosyadaki #'li satırlar silinmektedir:

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

#define LINE_LEN      4096

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

    return *str == '#';
}

int main(int argc, char *argv[])
{
    FILE *f, *ftemp;
    char *ftempnam;
    char buf[LINE_LEN];

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

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

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

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

    while ((fgets(buf, LINE_LEN, f)) != NULL) {
        if (!issharp(buf) && fputs(buf, ftemp) == EOF) {
            fprintf(stderr, "cannot write file!..\\n");
            exit(EXIT_FAILURE);
        }
    }

    fclose(f);
    fclose(ftemp);

    if (remove(argv[1]) == -1) {
        fprintf(stderr, "cannot delete file!..\\n");
        exit(EXIT_FAILURE);
    }

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

```
    return 0;  
}
```

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 dizin listesini elde etmek için "ls" komutu kullanılmaktadır. Ancak bu komut default durumda yalnızca dizindeki dosyaların isimlerini yazdırır. Fakat örneğin bu komut "-l" seçeneği ile kullanıldığında (long form) dosyaların yalnızca isimleri değil ayrıntılı bilgileri de yazdırılmaktadır. İşte genel olarak UNIX/Linux sistemlerinde programlar bu biçimde komut satırı argümanları alırlar.

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

- GNU stiline komut satırı argümanları üç biçimde bulunabilmektedir:

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

```
cat sample.c
```

Burada "cat" programını, "sample.c" ise seçeneksiz argümanı belirtmektedir.

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

```
ls -l
```

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

```
ls -l -i
```

ile:

```
ls -li
```

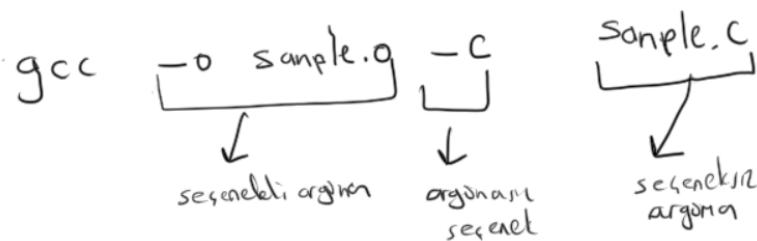
aynı anlamdadır. Eğer seçenek birden fazla karakterden oluşuyorsa bu durumda '-' yerine '--' kullanılmaktadır. Örneğin:

```
ls --version
```

Komut satırı seçeneklerinin büyük harf küçük harf duyarlılığı olduğuna dikkat ediniz. İki tane '-' karakteri ile uzun seçeneklendirme daha sonraları GNU stiline dahil edilmiştir. Klasik GNU stiline 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'lin -o seçeneği ile argümanı bitişik de yazılabildiği:

```
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 biçimde vardır.

1.18.1. getopt Fonksiyonunun Kullanımı

getopt yukarıda da belirtildiği gibi GNU stilindeki komut satırı argümanlarını ayırtmak 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 programının ayrı bir extern bildirimi yapmasına gerek kalmamaktadır.

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

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

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

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

Yukarıda da belirttiğimiz gibi getopt fonksiyonu her çağrılığında kullanıcının girmiş olduğu bir seçenekle geri dönmektedir. Dolayısıyla tipik olarak getopt fonksiyonun geri dönüş değeri switch içerisinde alınarak işlenmelidir. getopt programının üçüncü parametreyle belirlemediği bir seçenekle karşılaşrsa ya da argümanlı bir seçenekte argümanın girilmediğini görürse '?' karakteriyle geri döner. Tabii programının 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 yazdırılmayacağını belirlemek için kullanılmaktadır. Eğer işin başında opterr değişkenine 0 değeri atanırsa artık getopt hata durumlarında stderr dosyasına herhangi bir hata mesajı yazmaz. getopt değişkeninin işin balında sıfır dışı bir değere sahip olduğuna dikkat ediniz.

Argümanlı seçeneklerde seçenek bulunduğuanda optarg isimli global char türden gösterici seçenekin argümanını gösterecek biçimde (tabii argümanın sonu '\0' ile bitmektedir) ayarlanmaktadır. Programcı bu nedenle argümanlı bir seçenekle karşılaşlığında 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 bakarak 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 argument 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 bittiğinden sonra da bu flag değişkenlerine bakarak istediğiniz işlemleri yapabilirsiniz.

getopt fonksiyonun her çağrılarında bize bir seçeneği verdigini söylemiştim. Peki 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 söylemişik. Pekiyi bu geçersiz seçenekle ilişkin karakter nedir? İşte optopt isimli global değişken bu geçersiz seçenekle 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 argümanı 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);
            break;
        }
    }

    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;
}

```

1.18.2. getopt_long Fonksiyonun Kullanımı

getopt_long fonksiyonu getopt fonksiyonunun uzun seçenekleri de dikkate alan yeni birimidir. 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 arımsız kısa seçenek, --color argümasız uzun seçenek, --country ve --type da argümanlı uzun seçeneklerdir. Uzun seçeneklerin isteğe bağlı olarak argüman alabilmesi mümkünür. Tabii bu durumda uzun seçeneğin isteğe bağlı argümanları '=' ile girmek zorundadır. (Çünkü isteğe bağlı argüman alan uzun seçeneklerde argüman boşlukla ayrılsa bu durumda bu argümanın seçeneksız argüman mı yoksa uzun seçeneğin argümanı mı olduğu anlaşılır.) 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. Programının doldurduğu yapı dizisinin son elemanı 0'lardan olmalıdır. 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 seçenekin argüman almadığını, required_argument aldığı belirtmektedir. optional_argument ise uzun seçenekin 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 iyine 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 degere geri dönmektedir. Özellikle kısa seçenekle eşdeğer olan uzun seçenekler söz konusu olduğunda programcı yapının bu flag elemanına NULL değerini, val elemanına da kısa seçenekin karakter değerini yerleştirip durumu daha kolay ele alabilmektedir.

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

getopt_long fonksiyonunun geri dönüş değerinin beş biçimde olabileceği dikkat ediniz:

- 1) Eğer getopt_long komut satırı argüman incelemesinin sonuna geldiyse -1 değerine geri döner.
- 2) Eğer getopt_long uzun bir seçenek bulduysa ve o seçenekle ilişkin option yapı nesnesinin flag elemanı NULL ise bu durumda o yapı nesnesinin val elemanındaki değerle geri döner.
- 3) Eğer getopt_long uzun bir seçenek bulduysa ve o seçenekle ilişkin option 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 seçenek bulduysa tipki getopt gibi ilgili seçenek karakterinin sayısal değerine geri döner.
- 5) Eğer getopt_long fonksiyonu olmayan bir seçenek ile karşılaşırsa ya da argümanlı bir seçenekle karşılaştığı halde argüman belirtmediyse tipki getopt fonksiyonunda olduğu gibi '?' karakterine geri dönmektedir. Uzun bir seçenekin argümanı olmadığından fonksiyon '?' karakteri ile geri döndüğü zaman programcı yine optopt değişkenine bakarak hangi uzun seçenekin argümanının girilmediğini anlayabilir. Bu durumda getopt_long fonksiyonu optopt değişkenine option yapısının val elemanındaki değeri atamaktadır.

getopt_long için değişik kullanımları içeren aşağıdaki örneği verebiliriz:

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

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

```

int i;

opterr = 0;
while ((result = getopt_long(argc, argv, "abc:d:", options, &option_index)) != -1) {
    switch (result) {
        case 2:
            if (option_index == 2) {
                ccc_flag = 1;
                ccc_arg = optarg;
            }
            break;
        case 'a':
            a_flag = 1;
            break;
        case 'b':
            b_flag = 1;
            break;
        case 'c':
            c_flag = 1;
            c_arg = optarg;
            break;
        case 'd':
            d_flag = 1;
            d_arg = optarg;
            break;
        case '?':
            if (optopt == 'c' || optopt == 'd')
                fprintf(stderr, "-%c option given without an argument\n", optopt);
            else if (optopt == 2)
                fprintf(stderr, "--%s option given without an argument\n",
options[optopt].name);
            else
                fprintf(stderr, "invalid switch: -%c\n", optopt);
            break;
    }
}

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

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

/* ... */

```

```

    return 0;
}

```

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

```

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

```

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

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

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

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

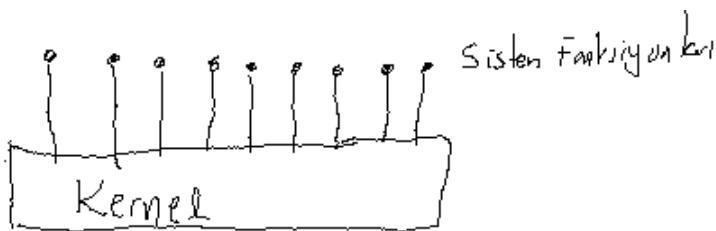
İşletim sistemleri genellikle prosedürel teknik kullanılarak C Programlama Diliyle yazılmaktadır. İşletim sistemleri kabaca çekirdek ve kabuk kısımlarından oluşur.



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 yönetici sistemler) bulunmaktadır.

İşletim sistemlerinin çekirdeklерinde binlerce fonksiyon bulunur. Bunların küçük bir kısmı dışarıdan da (kullanıcı modundan) önemli bazı işleri yapmak için çağrılabilmektedir. Bunlara sistem fonksiyonları (system call) denilmektedir.

Her işletim sisteminin sistem fonksiyonlarının isimleri, parametrik yapıları farklı olabilmektedir. Biz de C Programcısı olarak bu sistem fonksiyonlarını doğrudan çağrılabiliriz. Ancak her işletim sisteminin sistem fonksiyonları farklı olduğu için sistem fonksiyonları taşınabilir değildir.

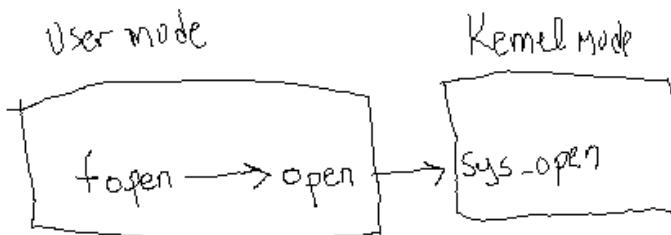


Programlarda sık karşılaşılan bazı işlemler aslında aşağı seviyede tamamen işletim sisteminin kontrolü altındadır. O faaliyetleri gerçekleştirmek isteyen herkes aslında eninde sonunda işletim sisteminin ilgili sistem fonksiyonunu çağrılmak zorundadır. Örneğin bir dosyayı silmek için bir sistem fonksiyonu 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 çağrıarak bu işi yaparlar.

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

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 çağrılmış olalım. Bu sistemlerde fopen fonksiyonu birtakım işlemlerden sonra open POSIX fonksiyonunu çağrılmaktadır. open POSIX fonksiyonu da sys_open isimli sistem fonksiyonunu çağrırlar. Dosyanın açılması aslında sys_open isimli istem fonksiyonu tarafından yapılmaktadır.



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

Pekiyi bazen işletim sisteminin sistem fonksiyonlarını doğrudan çağrılmamız gerekebilir mi? Taşınabilirlik sağlamak için ortak özelliklere hitap etmek gerekmektedir. Yani örneğin Linux'ta olan fakat BSD'de olmayan bir özellik POSIX fonksiyonunun konusu olamaz. Çünkü POSIX fonksiyonları tüm UNIX türevi sistemler için düşünülmüştür. İşte biz bazen belirli bir sisteme özgü işlemler yapmak isteyebiliriz. Bunun için doğrudan o sistemin sistem fonksiyonlarını çağrılmak

zorunda kalabiliriz. Şimdi şöyle bir soru soralım: Linux için fopen mı, open mı, yoksa sys_open mı daha geniş olanaklara sahiptir? İşte Linux'un sys_open fonksiyonu Linux'a özgü yazılmıştır. open ise UNIX türevi tüm sistemleri hedef alacak biçimde tasarlanmıştır. Halbuki fopen fonksiyonu tüm sistemlerde olabilecek ortak özelliklere göre tanımlanmıştır.

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

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

Aslında Proses Kontrol Bloğunu içerisinde göstereklerin de bulunduğu bir ağaç gibi (daha doğru bir deyişle graf gibi) düşününebiliriz. Yani proses kontrol bloğunun içerisindeki bazı elemanlar başka birtakım yapıları gösteriyor olabilir. O yapılar da başka yapıları gösteriyor olabilir. Biz kursumuzda bir bilginin Proses Kontrol Bloğunda olduğunu söylediğimizde o bilginin doğudan ya da dolaylı olarak Proses Kontrol Bloğu Yoluyla erişilebilecek bir yerde olduğunu kastetmiş olacağız.

Bir proses işletim sisteminin sistem fonksiyonuyla yaratılır ve işletim sisteminin bir sistem fonksiyonuyla sonlandırılır. Prosesler arasında altlık-üslük (parent-child) ilişkisi de vardır. Prosesi yatan prosese üst-proses (parent process), yeni oluşturulan prosese 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 çalıştırılmış olalım. Bu "sample.exe" prosesi de "notepad.exe"

programını çalıştırıp yeni bir prosesin oluşmasına 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 içerisindeindedir. 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) 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 büyütür. Örneğin `CreateFile`, `FindFirstFile` gibi.) Genellikle önce eylem sonra onun nesnesi gelir. (Örneğin `CreateWindow`, `CreateProcess`, `OpenProcess`, `CloseHandle` gibi.)
- Değişken isimleri onların türelerini belirten küçük harfli öneklerle başlatılmıştır. Tipik kullanılan önekler 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 önek kullanılmaz.

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

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

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

```
dwNumberOfSectors  
rectWindow  
szName  
studentId
```

Global değişkenlerin farkedilmesi için "g_" gibi bir ekle öneklendirilmesi çok kullanılan bir yöntemdir. Örneğin:

```
g_dwNumberOfSectors  
g_dwCount
```

- **typedef** tür isimleri büyük harflerle isimlendirilmektedir.

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

BYTE	Bir byte'lık işaretetsiz tamsayı türü (unsigned char)
WORD	İki byte'lık işaretetsiz tamsayı türü (tipik olarak unsigned short int)
DWORD	Dört byte'lık işaretetsiz tamsayı türü (duruma göre unsigned long int ya da unsigned int olabilir)
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 işaretetsiz 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

API fonksiyonlarının prototiplerinde parametre değişkenlerinin önünde __in, __out ve __in_out sözcüklerini görebilirsiniz. Bunlar okunabilirliği artırmak için düşünülmüştür. Aslında bunlar aşağıdaki gibi define edilmiş makrolardır:

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

Yani bu makrolar önişlemci tarafından silinmektedir. __in makrosu fonksiyonun parametre değişkenindeki bilgiyi kullanacağı fakat ona bir değer yerleştirmeyeceği anlamına gelir. __out tam tersine fonksiyonun parametre değişkenindeki değeri değiştireceği anlamına gelmektedir. __in_out ise fonksiyonun hem parametre değişkenindeki değeri kullanacağı hem de ona yeni bir değer yerlestireceği anlamına gelir. Fakat bu makroların gerekliliği tartışmalıdır. Zaten gösterici olmayan parametre değişkenleri __in olmak zorundadır. Gösterici parametre değişkenlerinde __in ya da __out durumu göstericinin const olup olmamasıyla zaten anlaşılmaktadır. O halde bunun tek faydası __in_out durumu için olabilir. Microsoft yeni başlık dosyalarında artık bu makroları kullanmamaktadır.

1.21. Fonksiyonlar İçin Hata Kontrolleri

Hata kontrolü bakımından fonksiyonları iki gruba ayıralım:

- 1) Her zaman hata kontrolünün yapılması gereği fonksiyonlar: Bunlar sistemin o anki durumuyla ilgili biçimde başarısız olabilecek fonksiyonlardır. Bu tür fonksiyonlar çağrılrken kesinlikle hata kontrolü yapılmalıdır. (Örneğin fopen, malloc, ... gibi fonksiyonlar)
- 2) Eğer programcı her şeyi düzgün yapmışsa, başarısız olma olasılığı olmayan fonksiyonlar: Bu tür fonksiyonlar için hata kontrolü yapılmayabilir. Örneğin biz bir dosyayı fopen fonksiyonu ile düzgün bir biçimde açmışsak bu dosyanın fclose ile kapatılamamasının makul bir nedeni olamaz. (Zaten böyle bir durumda bizim yapabileceğimiz birsey 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 görürsünüz:

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 çağrılmaması tavsiye edilmektedir. Fonksiyonun başarılı olduğu durumda bu fonksiyonu çağrıdığımızda hata kodu olarak 0 değerini almak zorunda değiliz. Ayrıca bazı API fonksiyonları başarılı olduğunda bile "last error" değerini set etmemektedir. Ancak bu durum çok seyrektr. 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 kullanacağımız:

```
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 xxxx 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 bildirimi <errno.h> dosyası içerisinde yapılmıştır. Bu durumda UNIX/Linux sistemlerinde hata şöyle ele alınabilir:

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

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

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

Ya da örneğin şöyle bir kod da yazabilir:

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

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

UNIX/Linux sistemlerinde bir POSIX fonksiyonu başarısız olursa errno değişkeninde hangi değerlerin bulunabileceği kesin olarak listelenmemiştir. 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 değişkendeki değeri strerror fonksiyonuyla elde edip onu stderr dosyasına yazdırılan 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");

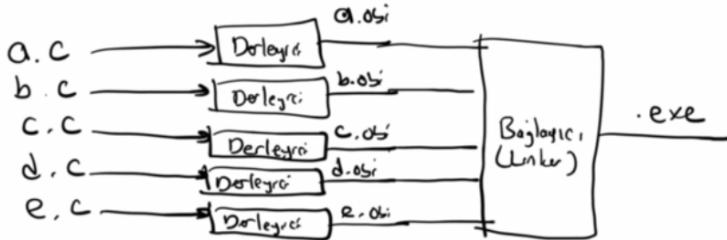
    return 0;
}

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 dosyaya 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 faydalılmaktadır. Bu konu ileride ele alınacaktır.



Eğer biz IDE ile çalışıyoruzsa 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 önişlemci komutlarının bulunduğu bir başlık dosyası ve fonksiyon kodlarının bulunduğu bir kaynak dosya. Bu çeşit çalışma özellikle kütüphanelerin oluşturulması sürecinde de sıkılıkla uygulanmaktadır.

Başlık dosyalarının içerisinde önişlemci 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 ismidir. Başlık dosyası ilk kez önişlemci tarafından ele alındığında bu makro define edilmediği için koşul içerisindeki kodlar derleme işlemine sokulacaktır. Ancak dosya önişlemci tarafından daha sonra ele alınırsa artık bu makro tanımlanmış olduğu için dosyanın içeriği derleme işlemine sokulmayacaktır.

include korumaları için Microsoft ve gcc derleyicisi 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 olmayacağından emin oluyoruz.

static fonksiyonların prototipleri başlık dosyalarında bulundurulmamalıdır. Bunlar modüle özgü oldukları için kaynak dosya içerisinde bulundurulmalıdır. Başlık dosyalarında asla nesne yaratan tanımlamaların bulunmaması gerektiğini biliyorsunuz. (Çünkü bu durumda başlık dosyası farklı modüllerden include edildiğinde aynı nesnenin birden fazla kez tanımlaması 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 değerdir. 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 kapatılan 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 gerekmektedir. (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. fgetc, fputc, fread gibi fonksiyonlar handle sistemini kullanan fonksiyonlardır. fclose da handle sistemini kapatılan 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 ilk değerlerini 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 bildirime **typedef** belirleyicisi getirilebilir. Bir bildirime **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 yazılabilir 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 bildirimin 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 çağrıma biçimleri C standartlarında olan bir konu değildir. Derleyicilerde birer eklenti (extension) biçiminde bulunmaktadır. Hem Microsoft derleyicilerinde hem de gcc ve clang derleyicilerinde fonksiyonların çağrıma biçimleri ek birtakım anahtar sözcüklerle temsil edilmektedir.

Fonksiyon çağrıma biçimi fonksiyon parametlerinin fonksiyona nasıl aktarılacağı ve geri dönüş değerlerinin nasıl elde edileceği hakkındaki belirlemelerden oluşmaktadır. Her çağrıma biçimi bir isimle temsil edilmektedir. Windows sistemlerinde çağrıma biçimleri fonksiyon isminin hemen soluna (yani geri dönüş değeri ile fonksiyon isminin arasına) getirilen anahtar sözcüklerle belirlenir. gcc ve clang derleyicilerinde ise çağrıma biçimleri geri dönüş değerlerinin solunda `__attribute__((..))` sentaksıyla belirtilmektedir. Genel olarak Intel işlemcileri için Windows sistemleriyle UNIX/Linux sistemlerindeki çağrıma biçimleri benzerdir. Biz burada yalnızca her çağrıma biçiminin 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.

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

stdcall Çağırma Biçimi: Windows'un bütün API fonksiyonları ve özel birtakım fonksiyonlar hep bu çağrıma biçimile derlenmişlerdir. Bu çağrıma biçimi Windows derleyicilerinde `__stdcall`, gcc ve clang derleyicilerinde ise `__attribute__((stdcall))` anahtar sözcükleriyle belirtilmektedir. stdcall çağrıma biçiminde yine parametreler sağdan sola stack'e atılır. Geri dönüş değerleri yine yazmaçlardan elde edilir. Ancak stack'in dengelenmesi çağrılan fonksiyon (callee) tarafından yapılmaktadır. Genel olarak bu çağrıma biçimi Intel mimarisinde cdecl çağrıma 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.

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

Yukarıdakilerin dışında başka çağrıma biçimleri de vardır.

2. Aşağı Seviyeli Fonksiyonlarla Dosya İşlemleri

Bilindiği gibi dosya sistemi tamamen işletim sisteminin kontrolü altındadır. Biz hangi dille ya da hangi kütüphaneyle çalışır 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 aygit 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 tasarımlı 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 kapatın bir sistem fonksiyonu.

Biz kursumuzda işletim sisteminin sistem fonksiyonlarını doğrudan çağrılmayacağız. 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 açın ama okuma modunda açın" 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_SHAREREAD|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 (ayrintılar 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 yaratılacağsa anlamlıdır. Eğer olan dosya açılacaksa bu parametre sıfır olarak da geçilebilir. Son parametre zaten açık bir dosyanın handle'ı 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 miktdan daha az sayıda byte'ı okuyabilir. Bu durum bir başarısızlık gerekçesi olusturmaz. 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>

void ExitSys(LPCSTR lpszMsg);

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");

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

        buf[dwRead] = '\0';
        printf(buf);
    } while (dwRead > 0);

    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);
}

```

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. Örneğin:

```
int main(void)
{
    HANDLE hFile;
    DWORD dwWritten;
    char buf[] = "bu bilgiler dosyaya yazdırılacak";

    if ((hFile = CreateFile("test.txt", GENERIC_WRITE, FILE_SHARE_READ, NULL, CREATE_ALWAYS,
                           FILE_ATTRIBUTE_NORMAL, NULL)) == INVALID_HANDLE_VALUE)
        ExitSys("CreateFile");

    if (!WriteFile(hFile, buf, strlen(buf), &dwWritten, NULL))
        ExitSys("WriteFile");

    /* ... */

    return 0;
}
```

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 üzerinden 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'inci 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 DWORD değerlerini almaktadır. Konumlandırma offset'inin fonksiyona iki ayrı DWORD parametreyle aktarıldığına dikkat ediniz. Offset'in yüksek anlamlı DWORD değeri bir gösterici yoluyla aktarılmaktadır. Fonksiyon bu üçünü parametreye aynı zamanda (eğer NULL geçilmemişse) konumlandırılan offset'in yüksek anlamlı DWORD değerini yerleştirmektedir. Tabii bu üçüncü parametre istenirse NULL geçilebilir. Son parametre konumlandırma orijinini belirtmektedir. Bu parametre FILE_BEGIN, FILE_CURRENT ya da FILE_END biçiminde üç seçenekten biri olarak girilir. FILE_BEGIN konumlandırmayı dosyanın başından itibaren yapılacaklığını 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 göreli konumlandırma yapılabileceği anlamına gelir. Bu durumda konumlandırma offset'i pozitif, negatif ya da sıfır olabilir. FILE_END ise konumlandırmayı EOF konumundan itibaren yapılacak anlamına gelir. Bu durumda konumlandırma offset'i 0 ya da negatif olabilir. Fonksiyonun geri dönüş değeri dosya göstericisinin yeni offset'inin düşük anlamlı DWORD değeridir. (Yukarıda da belirtildiği gibi yüksek anlamlı DWORD 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. Ayrıca GetLastError fonksiyonunun NO_ERROR dışında bir değere geri dönüyor olması gereklidir. Örneğin:

```

int main(void)
{
    HANDLE hFile;

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

    SetFilePointer(hFile, 100, NULL, FILE_BEGIN);

    /* ... */
    return 0;
}

```

Bu örnekte dosya CreateFile fonksiyonuyla açılmış ve hemen SetFilePointer fonksiyonuyla dosya göstericisi konumlandırılmıştır.

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 kapatmazsa proses sonlandığında işletim sistemi dosyayı kendisi kapatmaktadır. Yani dosyanın kapatılmaması genellikle bir soruna yol açmaz. Tabii kullanılmayan dosyaların ilk fırسatta 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
);

```

Aşağı seviyeli dosya kullanımına bir örnek şöyle verilebilir:

```

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

void ExitSys(LPCSTR lpszMsg);

int main(void)
{
    HANDLE hFile;
    DWORD dwRead, dwWritten;
    char bufRead[512];
    char bufWrite[512] = "This is a test!..";

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

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

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

    if (SetFilePointer(hFile, 0, NULL, FILE_END) == INVALID_SET_FILE_POINTER)
        ExitSys("SetFilePointer");

    if (!WriteFile(hFile, bufWrite, strlen(bufWrite), &dwWritten, NULL))
        ExitSys("ReadFile");

    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);
}

```

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 çağrımaktadır. 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 çağrılmaktadır. 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 zorudadır:

```

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 ekilmesiyle 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 kullanılabilmektedir.

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

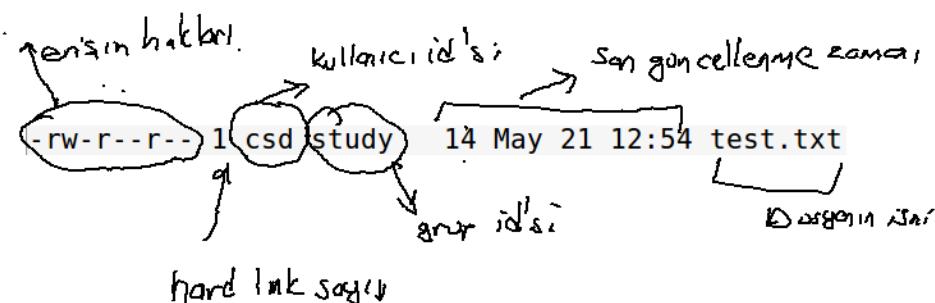
Gördüğü gibi erişim hakları için toplam 9 karakter kullanılmaktadır. En soldaki karakter dosyanın türünü belirtir. Bu karakter şunlar biri olabilir:

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

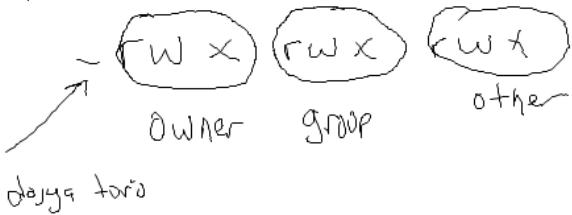
Diğer erişim hakları 3'lük 3 grup oluşturmaktadır:

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 prosten 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 varsayılabilirsiniz.

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). ls -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:



ls -l komutunda ilgili erişim hakkı varsa r, w, x sembollerinden biri yoksa '-' simbolü 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 çalıştırılamaz. Dosyanın grupluk hakları ise "r--" biçimindedir. Dosyanın sahibi olmayan ancak dosyayla aynı grupta olan prosesler (yani etkin griп id'leri dosyanın grup id'si ile aynı olan prosesler) bu dosyadan yalnızca okuma yapabilirler. Dosyanın diğer haklarının "--" biçiminde olduğunu görürsünüz. 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 çalışma hakkı ilgili üçlü gruba verilmemiş de olabilir.

Erişim hakları open fonksiyonu tarafından kontrol edilmektedir. Yani dosya işin başında açılırken open fonksiyonu (aslında open fonksiyonunun çağrıdığı 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 çağrıran prosesin etkin kullanıcı id'si 0 mı diye bakar. Sıfır numaralı kullanıcı id'sine sahip prosese "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 grubuya 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 prosteştir. 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 oluştururulurken 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 da USR, GRP ya da OTH karakterleri izlemektedir.

S_I R USR
W G P
X O T H

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 gereklidir.)

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 çağırmaktadır:

```

#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 karıştıracağınız. Aslında open fonksiyonunun üçüncü parametresinde girilen erişim hakları nihai durumu belirtmemektedir. Bu erişim hakları prosesin "umask" denilen bir maskeleme değerinin bitsel tersi ile bitsel AND işlemeye sokularak gerçek erişim hakları belirlenmektedir. Başka bir deyişle umask değerde 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 prosesten 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.

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 çalıştırın üst prosesten almaktadır. Tabii kullanıcı isterse kabuğun çalıştığı çeşitli script dosyaları içerisinde 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 anlamındadır. O halde bir dizin içerisinde bizim bir dosya yaratabilmemiz için o dizine "w" hakkına sahip olmamız gereklidir. 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 gereklidir. Dizin içeriğinin elde edilmesi ise o dizinden okuma yapma anlamına gelmektedir. Bizim bir dizinin listesini alabilmemiz için o

dizine "r" hakkımızın olması gereklidir. Gerçekten de dizini opendir fonksiyonu ile açabilmemiz için bizim dizine "r" hakkımızın olması gerekmektedir. Dizinlerde 'x' hakkı "çalıştırma" anlamına gelmez. Başka bir anlamda 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 gereklidir. Bu nedenle bu dizinlere "x" hakkımızın olması gerekmektedir. Eğer yol ifadesi göreli olarak "test.txt" biçiminde verilseydi bile bizim yine prosesin çalışma dizinine "x" hakkımızın olması gereklidir. 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 aacı dış dünyadan gizlemek istediğimizde tek yapacağımız şey aacı o kök kısmındaki dizinden "x" hakkını kaldırırmamız 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 gereklidir. Ancak bu dizinlere "r" hakkına sahip olması gerekmektedir. Örneğin bizim open fonksiyonuyla "/home/csd/a/b/test.txt" yol ifadesiyle belirtlen "test.txt" dosyasını O_RDONLY modunda açabilmemiz için yol ifadesinde belirtlen tüm dizinler için "x" hakkına sahip olmamız gereklidir. Ancak "r" hakkına sahip olmamız gerekmektedir. 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 gerekmektedir. O dosyanın içinde bulunduğu dizine "w" hakkına sahip olmamız gereklidir.

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 lmaz. 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 sayısıyla geri dönmektedir. Normalde write fonksiyonu talep edilen kadar bilgiyi yazma konusunda bir sorun çıkartmaz. Ancak disk doluya 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);
```

lseek fonksiyonu standart C'nin fseek fonksiyonu gibidir. Fonksiyonun birinci parametresi dosya betimleyicisini, ikinci parametresi konumlandırma offsetini, üçüncü parametresi ise konumlandırma orijinini belirtir. Fonksiyon başarı durumunda konumlanma 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 sisteme off_t signed long int biçiminde typedef edilmiştir. lseek 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 durumda -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 dosyaların 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 bulunurulmaz. Kopyalama işlemi bir dosyadan blok blok bilgilerin okunarak diğerine yazılmaya 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 bulunmaktadır. 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 çağrılmaktadır. 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 bulunmaktadır. 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 NULL geçilebilir. GetFileSize fonksiyonu başarısızlık durumunda INVALID_FILE_SIZE özel değerine geri döner. 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 işelman 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 dosayı 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 fonksiyonun prototipi <stdio.h> dosyasındadır:

```
#include <stdio.h>  
  
int remove(const char *pathname);
```

unlink fonksiyonunun prototipi ise <unistd.h> dosyasındaadır.

```
#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 betimleyicisindne 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 varsayılabılır. 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.

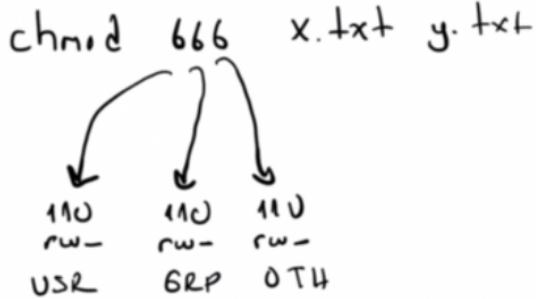
Komut satırında da chmod isimli POSIX kabuk komutu dosyanın erişim haklarını değiştirmek için kullanılmaktadır. chmod komutunun en basit kullanımı şöyledir:

```
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 ifallerinin 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'i 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 içinin boş olması gereklidir. mkdir yine proses umask değerinden etkilenmemektedir. Örneğin:

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

void exit_sys(const char *msg);

int main(void)
{
    if (umask(0) == -1)
        exit_sys("umsask");

    if (mkdir("test", S_IRWXU | S_IRWXG | S_IRWXO) == -1)
        exit_sys("mkdir");
}

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

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

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 Fonksiyonun 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. Istat ise stat fonksiyonun sembolik bağlantıları izlemeyen biçimidir. Yani bir sembolik bağlantı dosyasının kendisine stat uygulanırsa o bağlantı dosyasının referans ettiği dosyanın bilgileri elde edilmektedir. Sembolik bağlantı dosyaları I-Node tabanlı dosya sistemlerinin anlatıldığı bölümde ele alınmaktadır. Ancak Istat 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` işaretetsiz bir tamsayı türünü belirtmektedir. Bunun hangi bitlerinin hangi anlamda 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
<code>S_ISREG</code>	Dosya normal bir dosya mı?
<code>S_ISDIR(m)</code>	Dosya bir dizin mi belirtmektedir?
<code>S_ISCHR(m)</code>	Dosya karakter aygit süor?rucusu mü belirtiyor?
<code>S_ISBLK(m)</code>	Dosya blok aygit sürücüsü mü belirtiyor?
<code>S_ISFIFO(m)</code>	Dosya isimli bir boru mu belirtiyor?
<code>S_ISLNK(m)</code>	Dosya bir sembolik bağlantı mı belirtiyor?
<code>S_ISSOCK(m)</code>	Dosya bir soket 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_ISLNK
S_ISSOCK
```

Yani örneğin:

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

kontrolü ile,

```
if (m & S_IFDIR) {
    ...
}
```

kontrolü aynı işlev sahiptir.

Şimdi yol ifadesi ile verilen bir dosyayının 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';
    else
        str[0] = '?';

    str[1] = '-';

    for (i = 0; i < 9; i++)
        str[i + 1] = '0';

    str[10] = '\0';
```

```

        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 yazışal 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 bri dosyadır ve satırlardan oluşmuştur. Bu dosyanın içeriğinin bir bölümünü aşağıda görünsünüz:

```
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 içeriğe 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:Kaan,,,:/home/kaan:/bin/bash

```

↓ ↓ ↓
Kullanıcı id'si 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 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 gerekmektedir. Zaten bu işlemi yapan birkaç POSIX fonksiyonu bulunmaktadır. 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);

```

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);
```

Şimdi de ls -l komutundaki tüm satırı yazdırın yeni bir get_ls_info_line fonksiyonunu yazalım. Bu fonksiyon bizden dosyanın yol ifadesini alınsın ve bize tamamen ls -l formatında kodlanmış bir yazı versin:

```
const char *get_ls_info_line(const char *path);
```

Fonksiyon şöyle yazılabılır:

```
#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_info_line(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_info_line(argv[i]));  
    }  
  
    return 0;  
}  
  
const char *get_ls_info_line(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, " %lu", (unsigned 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, " %lu", (unsigned 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 verdiği 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 strftime standart C fonksiyonlarını kullanarak dosyanın son değiştirilme zamanını (st_mtime) ls -l komutunda verildiği biçimde dönüştürdük.

3. Dizin İçerisindeki Dosyaların Elde Edilmesi

Bir dizin içerisindeki dosyaların elde edilmesi isteğiyle uygulamalarda çok karşılaşmaktadır. 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 faydalınlırlar. .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 çağırmaktadır.

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 kullanılabılır. '*' 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 fonksiyonun 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 çağrıramayız. 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 maskelemek 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 sisteme 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 2^{64} byte olabilir.) Fakat bu 64 bitlik değer 32 bit sistemlerde iki ayrı 32 bitlik parçaya temsil edilmiştir. (32 bit sistemlerde 64 bit uzunlığında 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 içinden 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. (Eski DOS ve Windows bu dizinleri görüntüülüyordu. 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) belirtirmektedir.

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. Programının bu DIR yapısının içeriğini bilmesi gerekmektedir. (Típkı fopen fonksiyonunun geri döndürdüğü FILE yapısının içeriğinin bilinmesi gerekmeli 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 gereklidir.

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 içerisinde 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 öngörmektedir.

readdir dizin listesinin sonuna gelinmişse NULL adrese geri dönmemektedir.

Göründüğü 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 diizin girişinin sıradan bir dosya mı yoksa bir dizin mi olduğunu stat fonksiyonu uygulayarak anlayabiliyoruz.

3) İşlem bitince opendir fonksiyonuyla açılmış olan dizin closedir fonksiyonuyla kapatılır. Tabii dosyalarda olduğu gibi biz dizini kapatmamışsa 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 <dirent.h>

int main(void)
{
    DIR *dir;
    struct dirent *ent;

    if ((dir = opendir("/usr/include")) == NULL)  {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    while ((ent = readdir(dir)) != NULL)
        printf("%s\n", ent->d_name);

    closedir(dir);

    return 0;
}
```

UNIX/Linux sistemlerindeki işlemlerle Windows sistemlerindeki işlemler arasındaki iki önemli farklılığı dikkat ediniz:

- 1) Windows'ta dizin içerisindeki 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 <time.h>
#include <sys/stat.h>
#include <dirent.h>

void exit_sys(const char *msg);

int main(int argc, char *argv[])
{
    DIR *dir;
    struct dirent *ent;
    struct stat finfo;
    char path[4096];

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

    if ((dir = opendir(argv[1])) == NULL)
        exit_sys("opendir");

    while ((ent = readdir(dir)) != NULL) {
        sprintf(path, "%s/%s", argv[1], ent->d_name);
        if (stat(path, &finfo) == -1)
            exit_sys("stat");
        printf("-----\\n");
        printf("%s\\n", ent->d_name);
        printf("%ld\\n", (long)finfo.st_size);
        printf("%s", ctime(&finfo.st_mtime));
    }

    closedir(dir);

    return 0;
}

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

    exit(EXIT_FAILURE);
}
```

Şimdi de bir dizindeki dosyaların hepsini "ls -l" formatında yazdırma çalışalım. Bu programı daha önce yazmış olduğumuz get_ls_info_line fonksiyonu çağırarak kolay biçimde yazabiliriz:

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

```

#include <time.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>

void exit_sys(const char *msg);
const char *get_ls_info_line(const char *path);

int main(int argc, char *argv[])
{
    DIR *dir;
    struct dirent *ent;
    struct stat finfo;
    char path[4096];

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

    if ((dir = opendir(argv[1])) == NULL)
        exit_sys("opendir");

    while ((ent = readdir(dir)) != NULL) {
        sprintf(path, "%s/%s", argv[1], ent->d_name);
        printf("%s\\n", get_ls_info_line(path));
    }

    closedir(dir);

    return 0;
}

const char *get_ls_info_line(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, " %lu", (unsigned 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, " %lu", (unsigned 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:

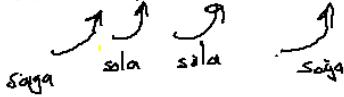
```

csd@csd-virtual-machine ~/Study/SysProg-2019 $ ./myls /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 myls
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 The ab 0 May 16 12:42 c name according to the current locale
-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 sampletime representation for the c
-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 as a 2-digit integer. (SU)
```



Anahtar Notlar: GCC derleyicisi ve ld bağlayıcısı default durumda matematik fonksiyonlarının 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 yazdırın 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 <dirent.h>
#include <pwd.h>
#include <grp.h>

void exit_sys(const char *msg);
void printls(const char *path);
char *get_ls_info_line(const char *path, int hlinkDigit, int unameDigit, int gnameDigit, int sizeDigit);

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

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

    printls(argv[1]);

    return 0;
}

void printls(const char *path)
{
    char *str;
    DIR *dir;
    struct dirent *dire;
    struct stat finfo;
    char buf[4096];
    int maxhDigit = 0, maxunameDigit = 0, maxgnameDigit = 0, maxsizeDigit = 0;
    int digit;
    struct passwd *pass;
    struct group *gr;
```

```

if ((dir = opendir(path)) == NULL)
    exit_sys("open_dir");

while ((dire = readdir(dir)) != NULL) {
    sprintf(buf, "%s/%s", path, dire->d_name);
    if (stat(buf, &finfo) == -1)
        exit_sys("stat");
    digit = (int)log10(finfo.st_nlink) + 1;
    if (digit > maxhDigit)
        maxhDigit = digit;
    if ((pass = getpwuid(finfo.st_uid)) == NULL)
        return;
    digit = strlen(pass->pw_name);
    if (digit > maxunameDigit)
        maxunameDigit = digit;
    if ((gr = getgrgid(finfo.st_gid)) == NULL)
        return;
    digit = strlen(gr->gr_name);
    if (digit > maxgnameDigit)
        maxgnameDigit = digit;
    digit = (int)log10(finfo.st_size) + 1;
    if (digit > maxsizeDigit)
        maxsizeDigit = digit;
}

rewinddir(dir);

while ((dire = readdir(dir)) != NULL) {
    sprintf(buf, "%s/%s", path, dire->d_name);
    str = get_ls_info_line(buf, maxhDigit, maxunameDigit, maxgnameDigit, maxsizeDigit);
    if (str != NULL)
        puts(str);
}

closedir(dir);
}

char *get_ls_info_line(const char *path, int hlinkDigit, int unameDigit, int gnameDigit, int sizeDigit)
{
    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, " %*lu", hlinkDigit, (unsigned long)finfo.st_nlink);

if ((pass = getpwuid(finfo.st_uid)) == NULL)
    return NULL;
index += sprintf(buf + index, " %-*s", unameDigit, pass->pw_name);

if ((gr = getgrgid(finfo.st_gid)) == NULL)
    return NULL;
index += sprintf(buf + index, " %-*s", gnameDigit, gr->gr_name);

index += sprintf(buf + index, " %*lu", sizeDigit, (unsigned 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 dizin listesinin elde edilmesi ve stat çağrıları birden fazla kez yapılmıştır. Bunu engellemek için önce dosyalar bulunarak dinamik bir diziye yerleştirilebilir. Sonra işlemlere bu diziden devam edilebilir. Ayrıca standart ls komutu dizini varsayılan durumda dosyaları isme göre sıralı göstermektedir. Oysa yukarıdaki kodda biz dizin listesini sıralamadık.

4. Fonksiyon Göstericileri (Pointers to Functions / Function Pointers)

Fonksiyonlar da aslında ardışıl makine komutlarından oluşmaktadır. Bir fonksiyonun çağrılması için makine dillerinde genellikle CALL isimli makine komutları kullanılır. Tipik olarak fonksiyon çağrımda kullanılan CALL makine komutu çağrılmak üzere fonksiyonun başlangıç adresini operand olarak almaktadır. CALL makine komutu sonraki komutun adresini stack'e atarak o adrese dallanır. Fonksiyondaki RET makine komutu da stack'ten geri dönüş adresini alarak akışın çağrılan yerden devam etmesini sağlar. Özette bir fonksiyonun çağrılabilmesi için aslında onun yalnızca başlangıç adresinin bilinmesi gereklidir. 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

```

Göründüğü 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 göstéricileri denilmektedir.

Fonksiyon göstérici bildirimlerinin genel biçimi şöyledir:

```
[geri dönüş değerinin türü]<(*<göstérici ismi>)(<parametre bildirimleri>);
```

Örneğin:

```
void (*pf1)(int);
int (*pf2)(long, long);
void (*pf3)(void);
```

Biz bir fonksiyon göstéricisine her türden fonksiyonun adresini yerleştirememiz. Ancak geri dönüş değerinin türü ve parametre türleri belirli biçimde olan fonksiyonların adreslerini yerleştirebiliriz. Yukarıdaki örnekte pf1 göstéricisine 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 göstérici 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 göstéricisi tanımlanmış */
void *pf2(int); /* fonksiyon prototip bildirimini yapılmış */
```

Fonksiyon göstérici bildiriminde parametre parantezi içerisinde parametre değişken isimleri de yazılabilir. Fakat programcılar genel olarak bunu pek tercih etmemektedir. Örneğin:

```
void (*pf2)(long a, long b); /* geçerli fakat pek tercih edilmıyor */
```

Tabii buradaki parametre isimleri tíkí prototípte olduğu gibi istenildiği gibi verilebilmektedir. Ancak yukarıda da belirttiğimiz gibi programcılar genellikle fonksiyon göstérici bildiriminde parametre isimlerini kullanmazlar.

C'de bir fonksiyonun yalnızca ismi (yani parantezler olmadan yalnızca ismi) onun başlangıç adresi anlamına gelmektedir. Fonksiyon çağrıırken kullandığımız parantezler operatör görevindedir. Örneğin:

```
#include <stdio.h>

void foo(void)
{
    printf("foo\n");
}
...
void (*pf)(void);

pf = foo;
```

Burada foo fonksiyonunun adresi pf göstéricisine 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 çağrılmamasından sonra elde edilen geri dönüş değeri anlamına gelmektedir.

`pf` bir fonksiyon göstericisi olmak üzere, bu göstericinin gösterdiği yerdeki fonksiyonu çağrılmak için iki eşdeğer ifade kullanılabilmektedir: `pf(...)` ya da `(*pf)(...)`. Örneğin:

```
#include <stdio.h>

void foo(void)
{
    printf("foo\n");
}

int main(void)
{
    void (*pf)(void);
    pf = foo;
    pf();          /* foo çağrıılır */
    (*pf)();      /* foo çağrıılır */

    return 0;
}
```

Aslında normal fonksiyonlar da `(*foo)(...)` ifadesiyle çağrılabılır. Ancak bazı programcılar okunabilirliği artırmak için gösterici yoluyla fonksiyonu çağrıırken `(*pf)(...)` sentaksını, normal fonksiyon çağrımlarında da `pf(..)` sentaksını tercih etmektedirler. Biz kursumuzda fonksiyon göstericileri ile fonksiyonları çağrıırken 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 çağrıduğumda yine parametre sayısı kadar argüman bulundurmak zorundayız. Çağrım işleminden yine çağrıdığımız 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 tutabilecegi 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 gereklidir. 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 türündedir. foo öyle bir fonksiyon adresine geri dönmektedir ki, onun geri dönüş değeri long, parametresi double türündedir. Bu bildirimin aşama aşama nasıl oluşturulduğunu gösterelim:

<code>foo(int a)</code>	Fonksiyonun ismi foo'dur ve parametresi int türdendir.
<code>(*foo(int a))</code>	Parantezlemeden dolayı fonksiyonun geri dönüş değeri bir fonksiyon adresidir.
<code>long (*foo(int a))</code>	Fonksiyonun geri dönüş değerine ilişkin fonksiyonun geri dönüş değeri long türdendir.
<code>long (*foo(int a))(double)</code>	Fonksiyonun geri dönüş değerine ilişkin fonksiyonun parametresi double türdendir.

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 çağrıma operatörrünün de soldan-sağda öncelikli grupta bulunduğuunu 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 çağrıları şöyle de yapılabilir:

```
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.)

Bir fonksiyon göstéricisine 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 uyuşmazlığı var */
pf = (void (*)(void)) foo; /* geçerli */
```

Burada fonksiyon adreslerinin tür ifadelerinde parantez içerisinde yalnızca * atomunun bulunduğuuna 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 bildirimi ile sadeleştirebiliriz. Örneğin:

```
int foo(int a)
{
    ...
}
...
typedef void (*PF)(void);

PF pf;
pf = (PF) foo; /* geçerli */
```

C'de void fonksiyon göstéricisi olmadığına göre bir fonksiyonun adresini geçici süre bir göstéricide saklayacaksak bunun için herhangi bir türden fonksiyon göstéricisini kullanılabılır. Örneğin:

```
int foo(int a)
{
    ...
}
...
typedef void (*PF)(void);

PF pfv;
pfv = (PF) foo;
...
int (*pfi)(int);
pfi = (int (*)(int)) pfv;
```

Gördüğü gibi typedef bildirimi ile karmaşık fonksiyon göstérici türlerini daha sade ifade edebilmektedir. Örneğin:

```
void (*PF)(void);
```

PF burada geri dönüş değeri void parametresi void olan bir fonksiyon göstéricisidir. Bu da C'de sembolik olarak " void (*)(void)" biçiminde temsil edilir. Şimdi bu bildirimin önüne typedef belirleyicisini getirelim:

```
typedef void (*PF)(void);
```

Burada PF artık void (*)(void) türünü temsil eder. Yani:

```
void (*pf)(void);
```

ile,

```
PF pf;
```

aynı anladadır. Örneğin:

```
double (*foo(int a))(long)
{
    ...
}
```

Bu bildirimi typedef bildirimi 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 gerekliliklerle kullanılabilir. Ancak en yaygın kullanım gereklisi "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 çağrıır. O fonksiyonun içini de biz yazacağımız için fonksiyona istediğimiz şeyi yaptırmış oluruz. Bir olay gerçekleştiğinde bizden alınan bir fonksiyonun çağrılması 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 çağrılmaktadır. `square` fonksiyonunun dizinin her elemanını onun karesiyle yer değiştirmek için, `disp` fonksiyonunun da dizi elemanlarını yazdırmağa dikkat ediniz.

Örneğin bir fonksiyon belli periyotlarla bizden aldığı bir fonksiyonu çağrıyor olabilir. Biz de periyodik bir işlem yapmak için böyle bir fonksiyondan faydalananabiliz. 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 çağrılmaktadır. Örneğimiz aksi bekletmek için Windows sistemlerindeki `Sleep` isimli API fonksiyonunu kullandık. `Sleep` fonksiyonu bekleme süresini bizden milisaniye olarak almaktadır. UNIX/Linux sistemlerinde de `sleep` isminde 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 saatı 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 verdigimiz bir fonksiyonu çağırın 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 "call back" mekanizmasının oluşturulması dışında başka amaçlarla da sıkılıkla kullanılmaktadır. Örneğin bir "komut yorumlayıcı (command interpreter)" yazarken belli bir komut yakalandığında belirlenen bir fonksiyonun çağrılması sağlanabilir. Böyle bir tasarım komut yorumlayıcısının daha pratik bir biçimde gerçekleştirilemesini 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 cmd[] = {
    {"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. Peki her türden diziyi sıraya dizebilen

tek bir sort fonksiyonu yazılabilir mi? 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 göstergelerini kullanarak türden bağımsız sıraya dizme işlemini yapan genel bir sort fonksiyonu yazılabilir. Bu fonksiyonda dizinin başlangıç adresi bir void göstergesi ile alınır. Dizinin bir elemanın uzunluğu ve dizi uzunluğu fonksiyona ayrı parametrelerle geçirilir. Algoritma ne olursa olsun her sort fonksiyonunda dizinin iki elemanın karşılaştırılıp yer değiştirilmesi gibi bir işlem yapılmaktadır. Gerçekten de dizinin iki elemanın yer değiştirilmesi aslında dizi elemanlarının uzunluğunu bilerek yapılabilmektedir. Ancak karşılaştırma işleminin fonksiyonu çağrıran programcı tarafından yapılması gereklidir. Böylece böyle bir sort fonksiyonunda bir fonksiyon göstergesi parametresi bulundurulur. Bu parametre için fonksiyonu çağrıran programcı kendi yazdığı bir karşılaştırma fonksiyonunu argüman olarak geçer. Fonksiyon da her karşılaştırmada bu fonksiyonu çağrıır. Böylece fonksiyon her türden diziyi sıraya dizebilir. İşlemi bu biçimde yapan sort fonksiyonunun prototipi aşağıdakine benzer 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 elemanın adresi argüman yapılarak çağrılcaktır. Karşılaştırma fonksiyonun 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 çağırıyalım:

```
#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 *pcbbase = (char *)base;
    char *pc1, *pc2;
    char temp;

    for (i = 0; i < count - 1; ++i)
        for (k = 0; k < count - 1 - i; ++k) {
            pc1 = pcbbase + k * width;
            pc2 = pcbbase + (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 kütüphanelerinde 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);
}

```

Pekiyi diziye küçükten büyüğe değil de büyükten küçüğe sıraya dizmesek istesek ne yapmamız gereklidir? 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 -strcmp(*ppc1, *ppc2);

    return 0;
}

```

Bu gösterici dizisinin her elemanı `char *` türünden olduğunu göre ve karşılaştırma fonksiyonu elemanların adresleriyle çağrılcagina göre karşılaştırma fonksiyonun parametrelerinin `char **` türüne dönüştürülmesi gereklidir. Büyük harf küçük harf duyarlılığı olmadan karşılaştırma yapan `strcmp` fonksiyonu standart bir C fonksiyonu değildir. Ancak hem Microsoft derleyicilerinde hem `gcc` ve `clang` derleyicilerinde ve diğer pek çok derleyicide bulunan bir eklenti (extension) fonksiyondur.

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 deklaratorde 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 kaapatıp 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. Balık bir deyişle biz bir gösterici dizisinin adresini aynı türden göstericiyi gösteren göstericiye atayabiliyoruz. 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 yazılıacaksa sabit ifadesi olması zorunludur. O halde aşağıdakibildirimin 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 bildirimi 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 atayabiliz. 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 içeirisne bir adres atayacaksız 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'luk 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;
```

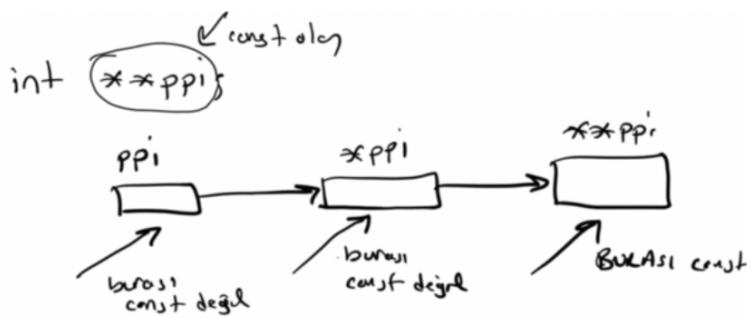
Tabii bu anahtar sözcükler birden fazla yere getirilebilirler. Örneğin:

```
const int * const * const ppi = ilkdeğer;
```

Aşağıdaki bildirime dikkat ediniz:

```
const int **ppi;
```

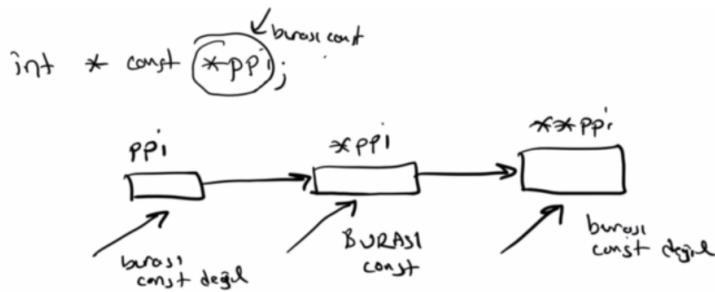
Burada const olan göstericiyi gösteren göstericinin gösterdiği göstericinin gösterdiği nesnedir. Gösterici gösteren göstericinin ya da göstericiyi gösteren göstericinin gösterdiği gösterici const değildir. Bunu şekilsel olarak şöyle ifade edebiliriz:



Şimdi aşağıdaki bildirimi inceleyiniz:

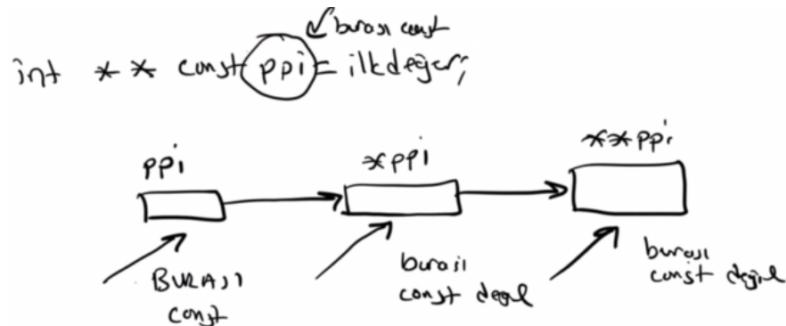
```
int * const *ppi;
```

Bu durumu da şekilsel olarak şöyle gösterebiliriz:



Şimdi de aşağıdaki bildirimi inceleyiniz:

```
int ** const ppi = ilkdeğer;
```



Göstericiyi gösteren göstericilerin `const`'luk `volatile`'lık durumunda anlaşılması zor bir tür uyuşmazlığı 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 uyuşmazlığı 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.

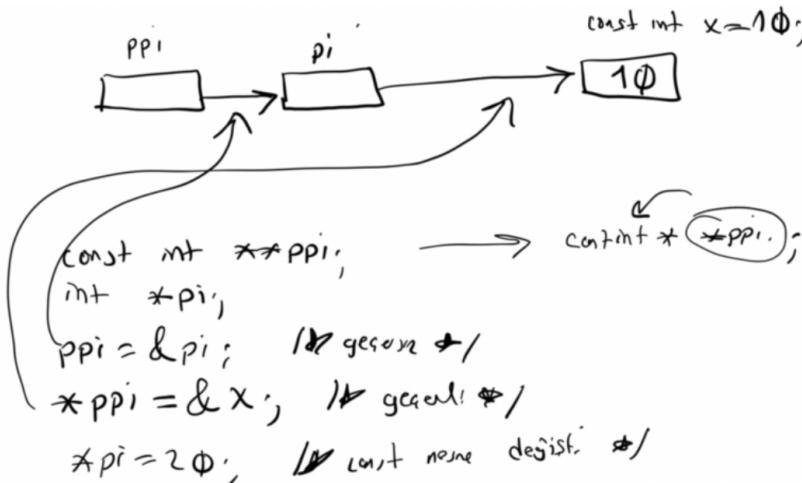
İşte bu durumun neden geçersiz olduğu basit bir örnekle açıklanabilir1:

- 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 varsyalım */`
- 3) `*ppi = &x; /* geçerli */`

Burada aslında biz `x`'in adresini çaktırmadan `pi`'ye atamış olduk. Bu durumda artık `pi` vasıtıyla biz `x`'i haksız yere değiştirebiliriz.



Pekiyi biz `T **` türünü `const T**`'a atayamıyorsak nasıl bire atayabilirim. 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.

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 ifadede kullandığımızda o dizi ismi otomatik olarak o dizinin ilk elemanın adresine dönüştürülür. Yani bir dizinin ismi aslında o dizinin tamamını belirtmekle birlikte bir ifadede kullanıldığında onun başlangıç adresini belirtiyor durumda olur. Örneğin yukarıdaki bildirimde `a` ifadesi `int[8]` türündedir. Ancak biz onu bir ifadede kullandığımızda o artık `int *` türü olarak işleme girer. Bildiğiniz gibi dizi isimleri bir ifadede 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 biçimini şöyledir:

```
<tür> (*<gösterici ismi>)[<sabit ifadesi>];
```

Örneğin:

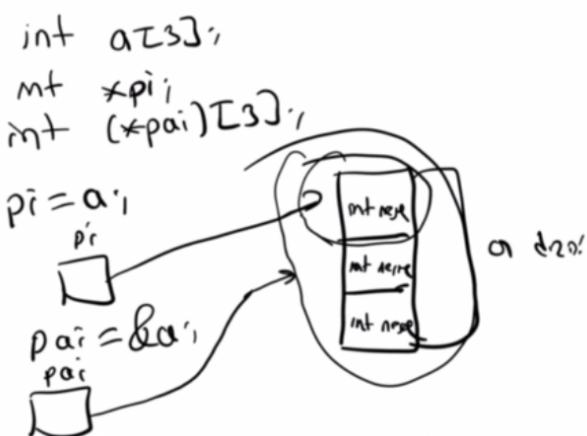
```
int (*pai)[3];
```

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 ifadede 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 gelmalıdır. 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 göstéricilerindeki köşeli parantez içerisindeki uzunluk tür bilgisine dahildir. Yani biz bir dizinin adresini uzunluk bilgisi aynı olan bir dizi göstéricisine atayabiliriz. Dizi göstéricilerindeki uzunluk ifadeleri sabit ifadeleri biçiminde olmak zorundadır. Örneğin:

```

int a[3];
int(*pai)[2];

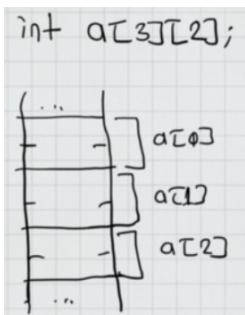
pai = a; /* geçersiz */

```

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:

*elementlerin dizi
asıl dizi;*

```
int a[3][2];
a 3 elemalı, her elemeni int[2]
turuñdan olan bir diziðir.
```

Peki C'de biz çok boyutlu bir dizinin adresini hangi türden bir göstériciye yerlestirebiliriz? Örneğin:

```
int a[3][2];
```

Burada `a`'nın adresini hangi türden bir göstériciye yerlestirebiliriz? 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 göstéricisine atanmalıdır. Örneğin:

```

int a[3][2] = { {1, 2}, {3, 4}, {5, 6} };
int (*pai)[2];

pai = a;      /* geçerli */

```

Göründüğü gibi bir matrisin adresi C'de bir göstericiye ya da göstericiyi gösteren göstericiye atanamamaktadır. Bir dizi göstericisine atanabilmektedir.

Çok boyutlu dizilerde her zaman ilk köşeli parantezin asıl diziyi sonraki köşeli parantezlerin eleman olan diziyi belirttiine dikkat ediniz. Örneğin:

```
int a[2][3][4];
```

Burada aslında bir dizi 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'inci 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];           /* eşdeğeri (*(pa + 1))[1] */
printf("%d\n", val);      /* 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 diziyi 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 diziyi 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 olmadığıma 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 aktarımında matrisin sütun uzunluğunun tür bilgisi içerisinde yer almadığını dikkat ediniz. Yani biz bu biçimle herhangi bir satır ve sütun uzunluğuna sahip matrisi fonksiyona aktarabiliriz.

Bir fonksiyon bir dizi nesnesinin ya da çok boyutlu bir dizinin adresi ile de geri dönebilir. Bu durumda yine parantezlerden faydalanyılır. 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 fonksiyonun geri dönüş değeri iki elemanlı bir dizi nesnesinin adresidir.

5. Yol İfadeleri (Path Names) ve Proseslerin Çalışma Dizinleri (Current Working Directory)

Bir dosyanın yerini belirten yazışal ifadeye yol ifadesi (path name) denilmektedir. Yol ifadeleri mutlak (absolute) ve görelî (relative) olmak üzere ikiye ayrırlar. 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 Bloğu"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. Görelî bir yol ifadesi bir sürücü de içerebilir. Örneğin prosesimizin çalışma dizini "d:\temp" olsun:

```
"c:a\b\c.dat"
"e:c.dat"
```

Buradaki görelî 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 (peki 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 sayısıyla (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 çağırmaktadır.)

Prosesin çalışma dizini Windows'ta SetCurrentDirectory API fonksiyonuyla değiştirilebilir:

```
BOOL SetCurrentDirectory(LPCTSTR lpPathName);
```

Fonksiyon yeni çalışma dizinin 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("chdir");

    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 prosesdir 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 çağrırlar. Kullanıcının girdiği yol ifadeleri mutlak ya da göreli 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;

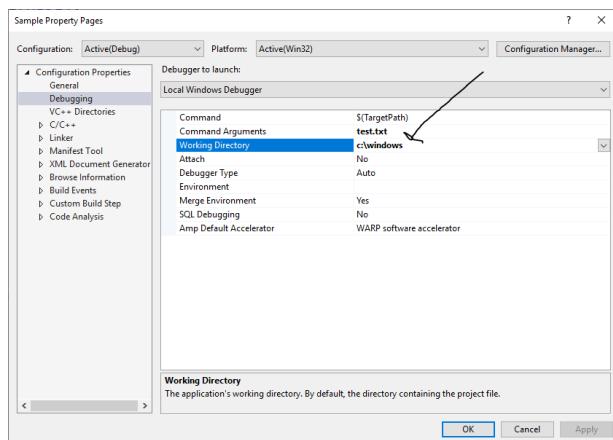
    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
                      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", msg, lpszErr);
        LocalFree(lpszErr);
    }
}

```

5.1. Proses Çalışmaya Başladığında Onun Çalışma Dizini Neresidir?

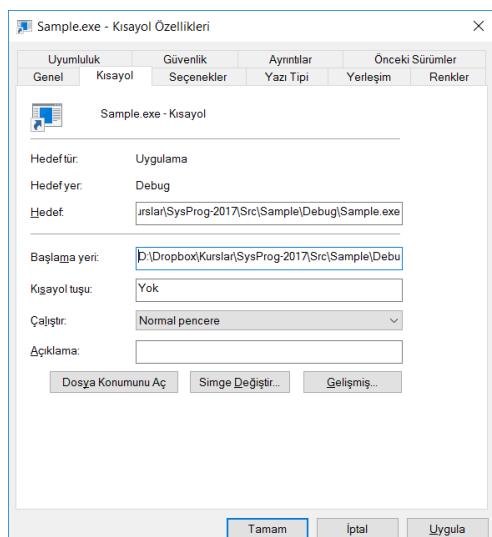
İşletim sistemlerinde her proses aslında bir sistem fonksiyonuyla başka bir proses tarafından yaratılmaktadır. Örneğin proses yaratmak için Windows'ta CreateProcess isimli API fonksiyonu, UNIX/Linux sistemlerinde ise fork isimli POSIX fonksiyonu kullanılmaktadır. Windows sistemlerinde prosesi yaratmak 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 prosesten (yani onu yaratan prosesten) 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 prosesi CreateProcess API fonksiyonunu uygulayarak çalıştırmaktadır. İşte Visual Studio IDE'si de C/C++ projelerinde çalıştırıldığı 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.



Düzenleme IDE'lerde de çalıştırılacak programın default çalışma dizini benzer biçimlerde belirlenebilir.

Pekiyi Windows'ta biz programı IDE'den değil de komut satırından ya da masaüstünden çalıştırduğumda 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 çalıştırduğumda cmd.exe prosesi her zaman çalıştırıldığı 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 prosedür. 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 çağırmaktadır.) 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 olmaktadır.

UNIX/Linux sistemlerinde prosesin çalışma dizini her zaman fork işlemi sırasında üst prosesinden 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 bulunduğuuz dizin) olmaktadır.

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 dizinlerinin solunda bir şey yoksa sanki orada prosesin çalışma dizini varmış gibiysem 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 olguya içermesi durumuna özyineleme (recursion) denilmektedir. Özyineleme hem doğada hem de bilgisayar bilimlerinde sıkılıkla karşılaştığımız bir olgudur. Örneğin dizin ağacını dolaşmak için dizin listesinde ilerlediğinizin 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 algoritmalar (recursive algorithms) denilmektedir. Yine dizin ağacını listeleme örneğine bakalım. Listeleme için kök dizinden girelim ve dosyaları buldukça yazdıralım. Pekiye 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 karşılaşıysak" muhtemelen bu algortima özyinelemeli bir algoritmadır.

Özyinelemeli algoritmalar tipik olarak kendi kendini çağırıp fonksiyonlarla (recursive functions) gerçekleştirilmektedir. Tabii özyinelemeli algoritmaların başka biçimlerde de çözümleri olabilmektedir. (Örneğin fonksiyonun kendisini çağırması yapay bir stack ile döngüsel biçimde sağlanabilmektedir.)

Yazılımda karşımıza çıkan tipik özyinelemeli algoritmaların 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.
- Matemetiksel bazı algoritmalar
- Bazı optimizasyon algoritmaları

Algoritmik problemleri özyineleme bakımından üç gruba ayıralım:

- 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 çağrılrken 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" bölmelerinde 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 çağrılığında yeni bir "a" nesnesi yaratılır. foo'nun çalışması sona erdiğinde akış bir önceki çağrımdan devam eder ve artık "a" bir önceki çağrımda yaratılan "a" olur.

Tabii fonksiyonun sürekli kendini çağırması sonsuz döngü oluşmasına yol açacaktır. O halde fonksiyon bir noktaya kadar kendini çağrımlı 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);
```

```

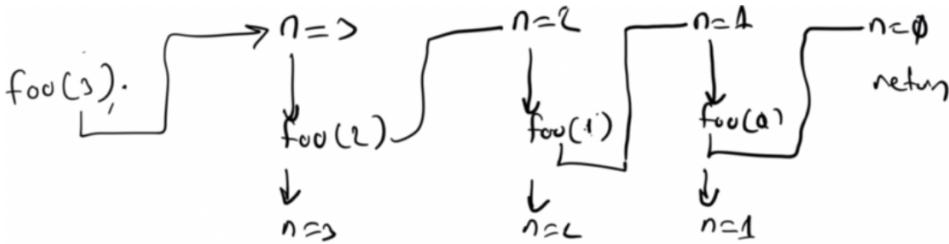
}

int main(void)
{
    foo(3);

    return 0;
}

```

Burada fonksiyon kendini hep bir eksik değerle çağrırmıştır. Parametre değişkeni olan n 0 değerine geldiğinde artık kendini çağrıma süreci sonlanmış ve çıkış süreci başlamıştır. Yukarıdaki çalışmayı şekilsel olarak aşağıdaki gibi gösterebiliriz:



Özyinelemeli çağrımayı şuna 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 bölüm 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 çokunda 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

Faktöriyel 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);

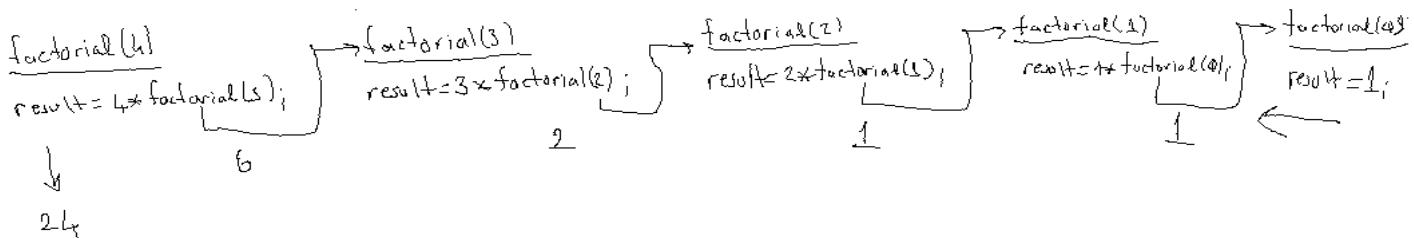
    return result;
}

int main(void)
{
    long result;

    result = factorial(4);
    printf("%lu\n", result);

    return 0;
}
```

Bu fonksiyon şöyle çalışmaktadır:



Fonksiyon daha basit olarak şöyle de yazılabılırdı:

```
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 işaretetsiz (örneğin `size_t`) bir türden olmasının soruna yol açacağına dikkat ediniz. Bu durumda kodu daha değişik düzenlemek gereklidir. 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 yazılabılırıldı:

```

void putsrev(const char *str)
{
    const char *temp = str;

    while (*str != '\0')
        ++str;
    do {
        --str;
        putchar(*str);
    } while (str != temp);
}

```

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 biçimini şöyle yazılabilir:

```

#include <stdio.h>

void mystrrev(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";

    mystrrev(s);
    puts(s);

    return 0;
}

```

Asıl işi yapan fonksiyonu çağrıran fonksiyonlara sarma fonksiyonları (wrapper functions) denilmektedir. Bazen özyineleme için özyinelenen fonksiyonların ilave parametrelere sahip olması gerekebilmektedir. İşte bu tür durumlarda sarma fonksiyonlarından faydalanyılır. Sarma fonksiyonlar parametreleri oluşturarak asıl özyinelemeli fonksiyonları çağırabilirler. Aşağıdaki örnekte mystrrev bir sarma fonksiyondur. Bu sarma fonksiyon mystrrev_recur isimli özyinelemeli fonksiyonu çağırmaktadır.

```

#include <stdio.h>

void mystrrev_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;

    mystrrev_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); /* mystrrev_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 Yazdırın Fonksiyon

Bu fonksiyonun özyineleme içermeyen klasik iteratifbiçimi şöyle yazılabılır:

```
#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 özyinelemeli biçimini 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 sayı 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 getirmemektedir. İstersek burada da bir sarma fonksiyondan faydalabiliriz:

```
#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 sayı 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 ayırtırıp 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 biçimini 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 yazılabildi:

```

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 çağrıının 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 yukarıındaki 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 sınırlarına gelindiği zaman ya da daha önce boyanan bir yere gelindiği 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 elemanıyla yer değiştirilmesi, sonra dizinin daraltılarak daraltılmış dizi için de aynı şeylerin yapılması biçiminde gerçekleştirilir. Seçerek sıralama algoritmasının normal (yani öz yinelemeli olmayan) biçimini 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 biribirlerini 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 çağırıkça ilk boş yere yine vezir yerlestirecektir. Böyle böyle 8 vezire gelindiğinde tahtanın durumu yazdırılır. Buradaki sorunlardan biri artık vezir yerlestiremeyeince 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 gereklidir. 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 tasarımlı şöyle olabilir: Fonksiyonun bir parametresi vardır, o da içi listelenenek 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 bırakabiliridik. 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 alamayabilirmiz. İşte eğer istersek listeleyemediğimiz dizinler için bir hata mesajını da aşağıdaki gibi yazdırılabiliriz:

```

#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 faydalanabilir:

```

#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 %* sentaksi kullanılır. Örneğin "%*d" format karakterleri için iki argüman girilmelidir. Birincisi * için uzunluk belirten argüman, ikincisi d için değerin bizzat kendisi. Örneğin:

```
char buf[] = "ankara";
```

```

int n;
scanf("%d", &n);
printf("%-*sxxx\n", n, buf);

```

Dizin ağacını tab'lارla 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 göstəriciləriyle genelleştirilebilir. Şöyledə 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 başqa 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 göstéricileri 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 yazılabılır:

```

#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 alarak 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;
}

```

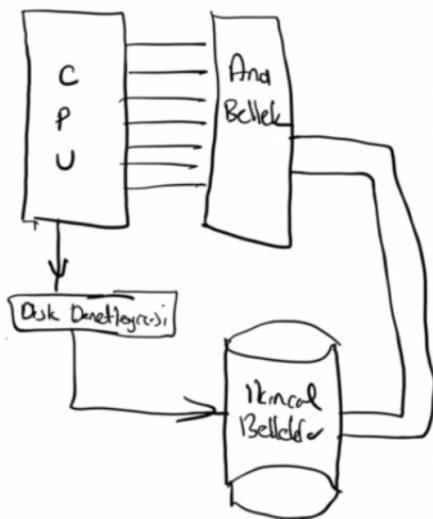
Yukarıdaki dizin ağacını dolaşan örneklerde dolaşımın "depth-first" biçimde yapıldığına dikkat ediniz. Yani biz bir dizinde ilerlerken o dizindeki ilk alt dizinin içerisinde dalmaktayız. Böyle bir dolaşımda dolaştığımız kökteki son dosya en son bulunacaktır. Fakat bazen arama işlemlerinde bu istenmez. Bu durumda yukarıdaki dolaşımda küçük bir değişiklik yapmak gerekebilir. Şöyle ki: Her dizinin içeriği toplamda iki kez elde edilebilir. İlk elde etmede bu dizin baştan sona dolaşılmış 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 Memory)" 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 yaygınlaşmıştır. Disketler ve teyp

bantları büyük ölçüde teknoloji dışı kaldılar. Artık bugünlerde SSD'lerin de hard disklerin yerini almaya başladığı söylenebilir.

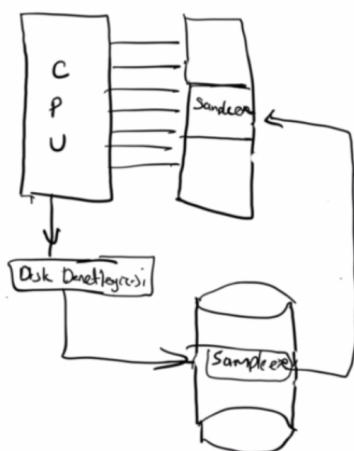
Tipik olarak bir bilgisayar sisteminin bellek mimarisini şöyle özetlenebilir:



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:

$$a = b + c;$$

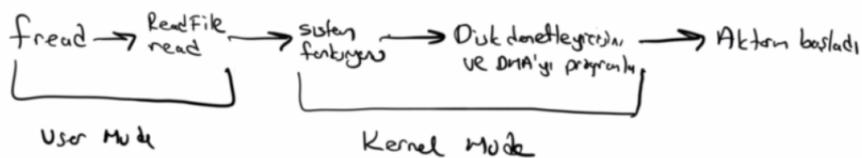
Burada a, b, ve c birincil bellektedir. CPU b ve c'yi birincil bellekten kendi içeresine çeker. Toplama işlemini kendi içerisindeki elektrik devreleriyle gerçekleştirir. Sonucu yeniden ana bellekteki a'ya aktarır. Programın makine komutları da yine ana belleklerde bulunmaktadır. Yani CPU aynı zamanda çalıştıracağı komutları da ana bellekten almaktadır. Yani program çalışırken programın kodu da değişkenler (nesneler) 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 çalışma sırasında aynı zamanda değişkenler (nesneler) de bellekten alınarak işleme sokulup güncellenmektedir.



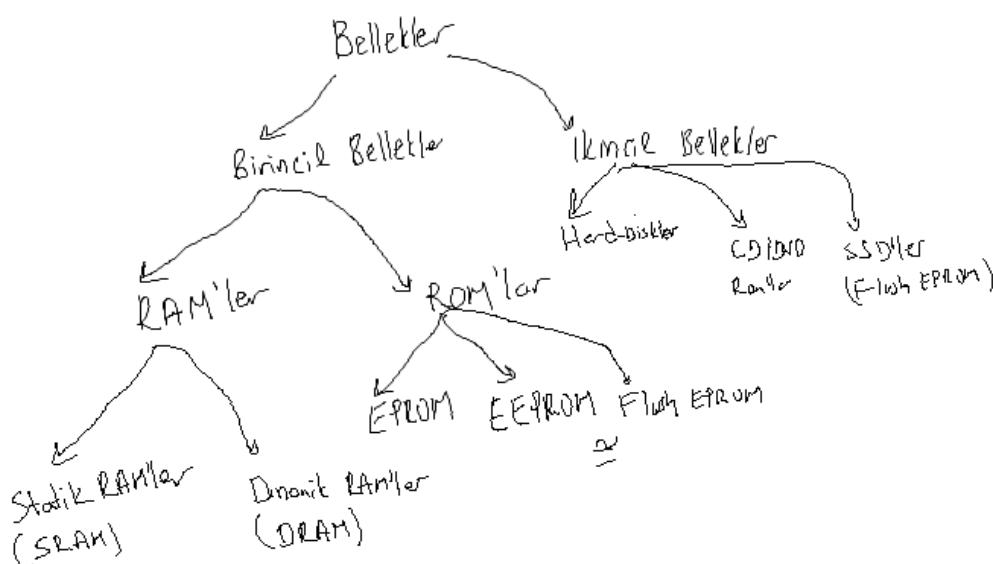
Çalıştırılabilen dosyaların içerisinde program kodlarının ve verilerinin nerede olduğu bilinmektedir. Genellikle bu dosyaların bir başlık kısımları (header) bulunmaktadır. Bu başlık kısımlarında dosya içerisindeki bölümler (sections) hakkında bilgiler bulunmaktadır. Tipik bir çalıştırılabilen dosyanın 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, disktten okumayı yapar. DMA da okunan bilgileri birincil belleğe belirlenen adreslen itibaren aktarır. İşlem bitince CPU bir donanım kesmesiyle haberدار 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 çalıştığı kodla yapmaktadır. Yani bütün işlemler aslında daha önce yazılmış olan kodların işletilmesi yoluyla yapılmaktadır. Örneğin aşağı seviyeli olarak biz fread standart C fonksiyonu ile disktten 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 çağırmaktadır. Bu fonksiyonlar da işletim sisteminin çekirdeği içerisinde bulunan sistem fonksiyonlarını çağrırlar. İşte disk denetleyicisini ve DMA'yı programlayan kodlar bu sistem fonksiyonlarının içerisindeindedir. 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 tazalenmesi 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 içerisinde 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 bilgiler tutabilmesidir. Halbuki RAM'ler güç kaynağı kesildiğinde içlerindeki bilgiyi kaybetmektedir. ROM'lardan sonra EPROM isimli (Erasable Programmable ROM) silinibilen ve programlanabilen ROM'lar geliştirildi. EPROM'lar EPROM silici denilen aygıtlı (ultraviyole ışınılarıyla silme yapılmıyordu) silinebiliyordu ve EPROM programlayıcı denilen aygıtlı da yeniden onlara bilgiler aktarılabiliyordu. Bugün artık EPROM'lar da teknoloji dışı kalmak üzeredir. Yeni teknolojide artık CPU tarafından doğrudan yazılabilen EEPROM (Electrically Erasable ROM) ve Flash EEPROM'lar kullanılmaktadır. Bunlar yine güç kaynağı kesildiğinde bilgiyi tutmaya devam ederler. Ancak hiç bilgisayardan çıkartılmadan bunların içlerine yeniden yeni bilgiler yazılabilmektedir.

Peki 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 adresden 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 adresen 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.



İkinci 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 hızlıdırlar. Flash EPROM ve EEPROM tarzı belleklerden okuma çok hızlıdır (nano saniye mertebesinde) fakat bunlara yazma göreli olarak yavaştır (mili saniyeler mertebesinde). Ayrıca bu teknolojilerde bir Flash EPROM bloğuna belli kez yazma yapılmaktadır. Gerçi bu sayı gitgide yükseltiliyor 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 sıkılıkla 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 görelî olarak yavaş olduğu için işlemci onun belli bölgelerini 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.
- Pek çok dilde dosya işlemi yapan fonksiyonlar ve sınıflar "user modda" dosyanın son okunan 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 olmadığını belirlenmesi gereklidir.

Önbellek için ayrılan cache line'ların tıka basa dolu olduğunu düşünelim. Yeni bir bilgi önbelleğe alınmak istendiğinde ne olacaktır? Bu durumda hangi cache line'i önbellekten çıkartmak gereklidir? Burada kullanılan algoritmalar önbellek yer değiştirme politikaları (cache replacement policy) denilmektedir. Tipik olarak üç tür yer değiştirme politikası vardır:

1) Least Frequently Used (LFU): Bu algoritmda her cache line için bir sayaç tutulur. O cache line'a erişildikçe sayaç artırılır. Sonra Önbellekten bir line çıkartılacağı zaman sayacı en az olan çıkarılır. Burada o zamana kadar az kullanılmış bir önbellek bloğunun ileride de az kullanılacağı varsayımlı yapılmaktadır.

2) Least Recently Used(LRU): Burada son zamanlarda en az erişilen cache line'lar önbellekten çıkartılma yoluna gidilir. Yani bu algoritmda "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 çıkarılır.

3) Most Frequently Used (MFU): Bazı sistemlerde line'lara toplam erişim sayısı önceden öngörülebilmektedir. Örneğin her bir bloğa toplamda 1000 civarında erişileceği biliniyor olabilir. Böyle bir sistemde tam tersine çok erişilmiş olan cache line'ların önbellekten atılması daha makuludur. 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 bir dosyanın "cache line" lar kullanılarak önbelleklenmesine ilişkin örnek bir arayüz verilmektedir. Bu örnek yalnızca fikir vermek için oluşturulmuş bir "pseudo code"dur:

```
typedef struct tagCACHE_LINE {
    char buf[LINE_SIZE];
    size_t offset;
    ...
} CACHE_LINE;

typedef struct tagCACHE_FILE {
    FILE *f;
    CACHE_LINE cacheLines[NCACHE_LINES];
    ...
} CACHE_FILE, *HCACHE_FILE;

HCACHE_FILE OpenCacheFile(const char *path, const char *mode);
size_t ReadCacheFile(HCACHE_FILE hFile, size_t count, void *buf);
size_t WriteCacheFile(HCACHE_FILE hFile, size_t count, const void *buf);
long LocateFilePointer(HCACHE_FILE hFile, long offset);
```

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 başlangıç offset numarasını tutar. Böylece sistem erişeceği yerin önbellekte olup olmadığını anlayabilecektir. Tüm cache line'ların CACHE_FILE içerisinde bir dizide saklandığına dikkat ediniz. Bu sistemi

kullanan programcı önce OpenCacheFile fonksiyonu ile dosyayı önbellekli biçimde açar. Buradan bir handle değeri elde eder. Bu handle değeri aslında önbellek bilgilerinin tutulduğu CACHE_FILE isimli yapı nesnesinin adresidir. Daha sonra programcı ReadCacheFile ve WriteCacheFile fonksiyonlarıyla önbellek sisteminden okuma ve yazma yapar. Bu fonksiyonlar önce okunacak ya da yazılacak yerin önbelleğin herhangi bir cache line'ında olup olmadığına bakacaktır. Eğer söz konusu yer önbellekteyse işlemi bu önbelleğe yaparlar. Eğer söz konusu yer önbellekte değilse bu kez gerçek dosyadan işlemi yaparlar. Bu nedenle gerçek dosyaya erişmek için FILE * biçimde bir handle değeri de handle alanında tutulmuştur. Aslında ilerde de görüleceği gibi normal dosyalar üzerinde programının 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 olmadığını hızlı biçimde belirlenmesi gereklidir. Örneğin yukarıdaki gibi bir sistemde toplam 1000 tane cache line bulunuyor 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.

Önbellek sistemlerinin Performansını Ne Etkiler?

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 algoritmalarla 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 mi olacağı da bazı unsurlara bakılarak belirlenmektedir.

Daha önceden de belirttiğimiz gibi read-write önbellek sistemlerinde yazma doğrudan önbelleğe yapılmaktadır. Peki yine 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];
    size_t offset;
    BOOL dirtyFlag;
    ...
} CACHE_LINE;
```

Cache line'a bir yazma yapıldığında bu dirtyFlag 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 kapatıldığında 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 tümden 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 çökertecik makine komutlarını kullanması da engellenmektedir. Buna da komut koruması denilmektedir.

İste modern büyük kapasiteli işlemciler koruma mekanizmasına sahip olarak tasarılanırlar (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 adreste bilgiye erişilmeye çalışılmıştır. İşte bu erişimin yapıldığı noktada işlemci prosesin kendi bellek alanı dışına erişim yaptığı belirleyecektir ve bunu işletim sistemine bildirecektir.

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     char *str = (char *)0xFC9999191919;
6
7     putchar(*str); X
8     printf("Son\n");
9
10    return 0;
11 }
12
13

```

Benzer biçimde örneğin:

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     __asm cli X
6
7     return 0;
8 }
9
10

```

Burada CLI isimli makine komutu sistemin çökmesine yol açabilmektedir. Bu makine komutuyla karşılaşan işlemci durumu işletim sistemine bildirmiş işletim sistemi de prosesi sonlandırmıştır.

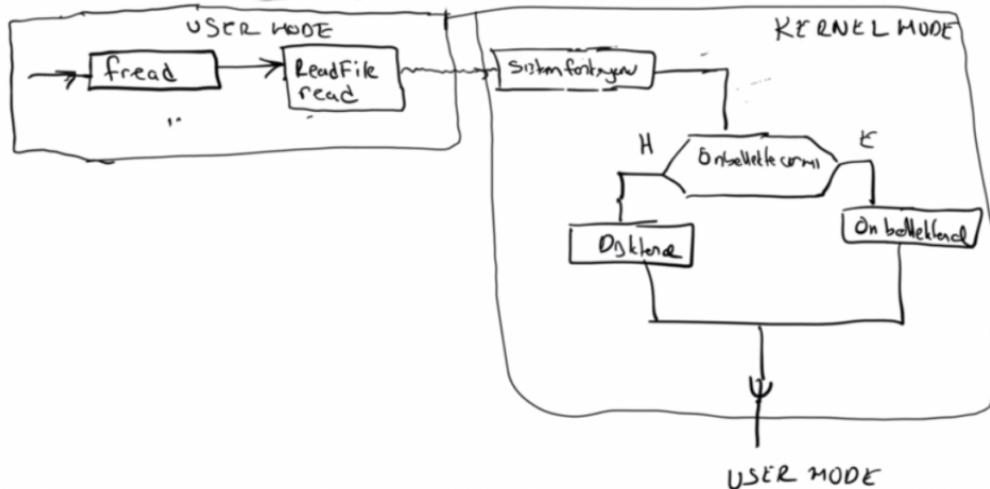
Pekiyi koruma mekanizması her prosese mi uygulanmaktadır? Bu mekanizma tarafından denetlenmeyen kodlar yok mudur? İşte aslında işletim sistemlerinin çekirdek kodları ve aygit sürücüler gereğinde tüm bellek alanına erişip tüm makine komutlarını kullanmaktadır. Aksi takdirde işletim sistemleri ve aygit sürücüler pek çok işi yapamazlardı. İşte koruma mekanizmasına sahip işlemcilerde çalışmakta olan kodun bir modu vardır: (Intel 4 mod kullanmasına karşın yalnızca iki mod işletim sistemi yazanlar tarafından kullanılmıştır. Diğer işlemcilerin çoğu iki moda sahiptir) "kernel mode" ve "user mode". Kernel kodları, kernel modülleri ve aygit sürücüler kernel modda çalışırlar. Kernel moddaki kodlar koruma mekanizmasına takılmazlar. Yani bu kodlar belleğin her yerine erişebilirler ve tüm makine komutlarını kullanabilirler. User mod kodları ise koruma mekanizması tarafından denetlenirler. Bizim ya da başkalarının Windows'ta ve Linux'ta yazmış olduğu normal programların hepsi user modda çalışmaktadır.

Pekiyi madem işletim sisteminin kodları kernel alanı içerisindeki verilere erişiyor ve özel komutları kullanabiliyor, o zaman biz bir sistem fonksiyonunu çağrıdığımızda ne olacaktır? Eğer bizim user moddaki akışımız o sistem fonksiyonunu çağrımişsa 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 çağrılığında otomatik olarak kapı meknizması sayesinde kernel moda geçiş yapar. Böylece işletim sisteminin sistem fonksiyonu kernel modda çalışırılmış olur. Sistem fonksiyonunun çalışması bittiğinde de proses yeniden user moda'a döner. Buna prosesin "user moddan kernel moda geçmesi (user mode to kernel mode transition)" denilmektedir. Tabii user moddan kernel moda geçisi sağlayan kapı mekanizması yalnızca kernel mod prosesler tarafından (tipik olarak işletim sistemi tarafından) yerleştirilebilmektedir. Böylece işletim sistemi yalnızca kendi sistem fonksiyonları çalıştırıldığında 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 gerçekleştirildikten sonra da birtakım işlemlerin yapılması gerekmektedir. Benzer biçimde kernel moddan user moda dönüşün de bir maliyeti vardır.

Pekiyi biz kernel modda çalışacak bir program yazamaz mıyız? Evet yazabiliriz. Bu tür programlara "kernel modülleri" ve "aygit sürücüler" denilmektedir. Kernel modülleri ve aygit sürücüler kernel alanına yüklenerek sanka kernel'in bir parçasıymış gibi çalışırlar. Kernel modüllerinin ve aygit sürücülerinin içerisindeki kodlar user mod programlar tarafından çağrılmaktadır. Bu durumda yine kapı mekanizması yoluyla kernel moda geçiş yapılır. Her işletim sisteminin bir aygit sürücü mimarisi vardır. Aygit sürücüler o işletim sistemine özgür (hatta o versiyona özgür) bir biçimde yazılırlar. Çünkü aygit sürücüler yalnızca kernel'daki fonksiyonları kullanırlar.

Şimdi örneğin C'deki fread gibi bir fonksiyonun arka planda aşama nasıl işletildiğine bakalım:



Bu şekilde görüldüğü gibi fread fonksiyonu Windows'ta ReadFile API fonksiyonunu UNIX/Linux sistemlerinde ise read POSIX fonksiyonunu çağrırmaktadır. Bu fonksiyonlar da gerçek dosya işlemleri içim kernel moda geçerek ilgili sistem fonksiyonlarını çağrırlar. Tabii aslında fread fonksiyonu user modda oluşturulmuş stdio önbelleğine de bakmaktadır. Bu konu izleyen bölümde ele alınmaktadır.

Bir de son olarak olası bir yanlış anlaşılmaya üzerinde duralım. Biz Windows sistemlerinde bir programı "Run as administrator" biçiminde çalıştırduğumda ya da UNIX/Linux sistemlerinde "sudo" ile çalıştırduğumda bu programlar kernel modda çalıştırılmamaktadır. Bu çalışma biçimlerinin kernel modda hiçbir ilgisi yoktur. Bu çalışma 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 çağrımak için 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 çağrırmaktadır, Windows'ta CreateFile API fonksiyonu da bir sistem fonksiyonunu çağrırmaktadır) 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.

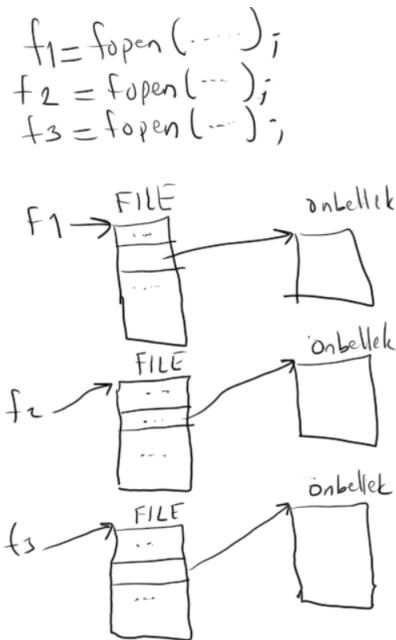
Anahtar Notlar: Standart C fonksiyonları için bu bağlamda önbellek yerine "tampon" sözcüğü kullanılmaktadır. Aslında tampon sözcüğü buradaki durumu iyi yansıtılmamaktadır. Çünkü tampon bilgilerin daha sonra işlenmek üzere saklandığı bellek bölgelerini belirtir. Halbuki önbellek hızlandırma amacıyla kullanılan 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 biçimdedir.
- Hemen her zaman tek önbellek kullanılır. Yani önbellekte dosyanın tek bir ardışılı bölüm 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 modda 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' görülene 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 söylemiştir. (Bu konu ileride ele alınacak). Fakat normal dosyaların default tamponlama modlarının ne olacağı konusunda birsey söylememiştir. 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 dahil edilmiştir. Ancak dosyanın tamponlama modu değiştirilecekle bu işlemin fopen fonksiyonundan hemen sonra (yani o dosya için başka bir dosya fonksiyonu henüz çağrılmadan) yapılması gereklidir. 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 değiştirememiz ve dosyayı satır tamponlamalı moda geçirememiz.

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

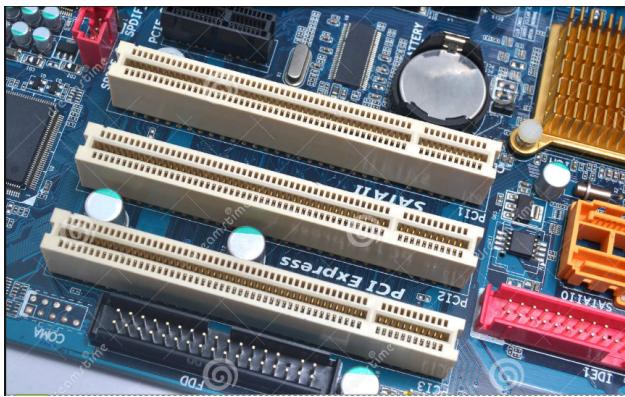
Bir dosyadan byte byte okuma yapmak istediğimi düşünelim. Bunun için fgetc ve getc fonksiyonlarının kullanıldığını biliyorsunuz. Pekiyi her ne kadar bir önbellek sistemi kullanlıyor olsa da byte byte okuma sırasında her defasında bir fonksiyon çağrılmış olması zaman kaybına yol açmaz mı?

```
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 yazılabiliceğidir. 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.

Aygıt Sürücü (Device Driver) Nedir ve Nasıl Kullanılır?

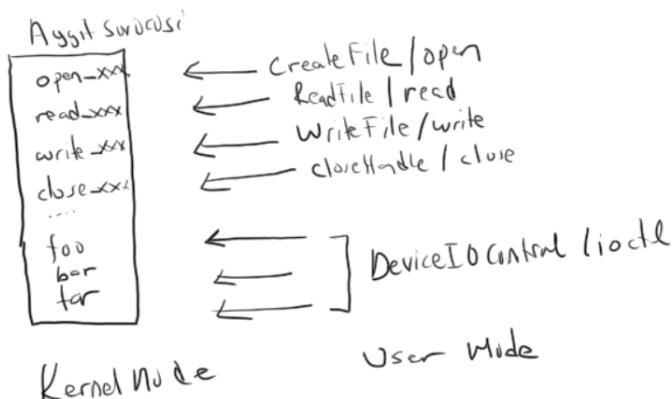
Bilindiği gibi masaüstü bilgisayarlar genişleme yuvalarına kartlar takılarak donanımsal olarak genişletilebilmektedir. Örneğin bugün PC'lerde kullandığımız kart yuvalarına PCI denilmektedir. Biz de bir donanım birimini bu PCI soketleri destekleyen bir kart halinde oluşturup bilgisayarımıza takabiliriz.



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 kullanılabilirler. İşte bu tür aygıtların programlanabilmesi için kernel modda çalışacak programlara gereksinim duyulur. Bunlara da genel olarak "aygit sürücüler (device drivers)" denilmektedir. Aygit sürücülerin aslında doğrudan bir donanım aygitini programlaması da gerekmektedir. Her ne kadar bu terimin ortaya çıkış noktası buyusa da aygit sürücülerini kernel modda çalışan ve işletim sisteminin kernel'ına eklenen kod parçaları olarak düşünebiliriz. Bazı sistemlerde (örneğin Linux) IO işlemi yapmayan aygit sürücülerine "kernel modülleri (kernel modules)" de denilmektedir. Yani bu sistemlerde yalnızca IO işlemi yapan kernel modüllerine aygit sürücüsü denilmektedir. Windows terminolojisinde IO işlemi yapın ya da yapmasın kernel modda çalışabilen kodlara aygit sürücüsü denir. Biz nasıl bilgisayarımıza kart takıp onu donanımın bir parçası haline getiriyorsak kernel'a da aygit sürücüsü ekleyip onu kernel'ın bir parçası haline getiririz.

Mikrokernel denilen kernel mimarisinde kernel kod olarak çok küçük tutulur. Bu tür mimarilerde aygit sürücüler de kernel modda değil user modda çalışmaktadır. Ancak Windows ve Linux gibi işletim sistemleri mikrokernel mimarisine sahip değildir. Dolayısıyla bu sistemlerde aygit sürücüler genel olarak kernel modda çalışmaktadır.

Genel olarak işletim sistemlerinde aygit sürücüler sanki birer dosyayı gibi kullanılmaktadır. Yani bir aygit sürücüsü Windows'ta dosya açan CreateFile API fonksiyonuyla, UNIX/Linux sistemlerinde open POSIX fonksiyonuyla açılır. Aygit sürücülerinden sanki bir dosyayı gibi okuma yazma işlemi yapılmaktadır. Ve en sonunda aygit sürücülerini yine bir dosyayı gibi kapatırlar. Bir aygit sürücüsünden okuma yaptığımızda o aygit sürücüsünün içerisindeki bizim tarafımızdan yazılmış olan okuma fonksiyonu çağrılır, yazma yaptığımızda da yine aygit sürücünün içerisindeki bizim tarafımızdan yazılmış olan yazma fonksiyonu çağrılmaktadır. Ayrıca biz aygit sürücüler okuma yazma fonksiyonları dışında istenildiği kadar çok fonksiyon da barındırabilemektedir. Biz aygit sürücüsü içerisinde bulunan bir fonksiyonu kernel moda geçerek çalıştırabiliriz. Bu işlem Windows'ta DeviceIOControl, UNIX/Linux sistemlerinde ioctl isimli fonksiyonla yapılmaktadır.



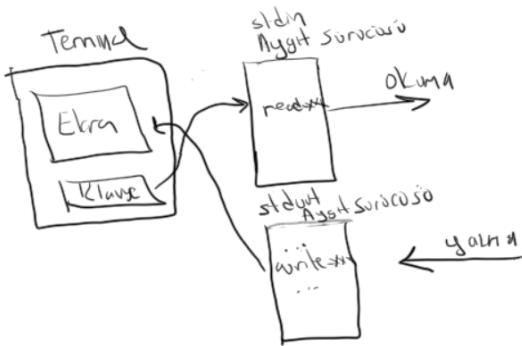
Şüphesiz bilgisayar sistemlerinde en sık karşılaştığımız aygit sürücüler terminal sürücüleridir. Ekran ve klavyeye terminal denilmektedir. Bu aygitlar dosya gibi kullanılan aygit sürücülerle yönetilirler. İşte C'de stdin dosyası (standard input) standart girdi için kullanılan (tipik olarak klavye), stdout (standard output) dosyası da çıktı için kullanılan (tipik olarak ekran) aygit sürücülerini temsil etmektedir. Yani örneğin biz stdout dosyasına dosya fonksiyonlarıyla yazma yaptığımızda

bu yazılacak bilgiler aygit sürücüsüne gider. Bu aygit sürücüsü de bizim yazmak istediklerimiz şeyleri ekran denilen donanım aygitına aktarır.

Pekiyi aygit sürücülerini nasıl yazılmaktadır? İşte her işletim sisteminin bir aygit sürücü mimarisi vardır. Aygit sürücüler işletim sistemine özgü hatta onların versiyonlarına özgü biçimde yazılmaktadır. Derneğimizde aygit 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 aygit sürücü dosyalarıdır. stdin ve stdout terminal aygit sürücüsüne yönlendirilmiştir. stdin dosyasının "read-only" modda stdout dosyasının ise "write-only" modda açıldığı kabul edilir. stdin dosyasından okuma yaptığımızda terminal aygit sürücüsü klavyeden alınanları bize verir. stdout dosyasına yazma yapmak istediğimizde de terminal aygit sürücüsü yazdırılmak istenen bilgileri ekrana yazdırmaktadır.



stderr programın error mesajlarının yazdırılacağı hedefi belirtmektedir. Default olarak sistemlerde stderr dosyası da "write-only" açılmıştır ve terminal aygit 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");
```

İleride ele alınacağı üzere bu bilgi önce diğer dosyalar gibi kütüphanenin oluşturduğu önbelleğe (tampona) yazılır. Buradan hedefe (yani aygit 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 yapılmaktadır.

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 '>' işaretini stdout dosyasının, '<' işaretini 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. '>' işaretini ile yönlendirmede hedef dosya zaten varsa sıfırlanıp yeniden açılmaktadır. Benzer biçimde stdin dosyası da şöyle yönlendirilebilir:

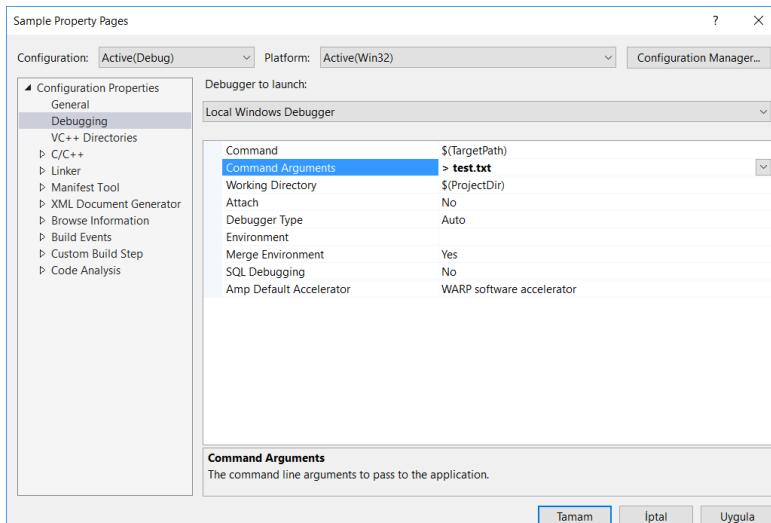
```
./sample < x.txt
```

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 yapılmaktadır. 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 yapılmaktadır. 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>' simbolü 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 aygit sürücü dosyasıdır. Bu aygit sürücünün yazma fonksiyonunun içi boştur. Böylece biz istersek aşağıdaki gibi programın hata mesajlarının kafa karıştırmasını 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 şansına sahip olacaklardır.

UNIX/Linux sistemlerindeki bash kabuğu &> 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 yapılacakı hedef dosayı belirtir. İkinci parametre bu dosyanın açış modunu belirtmektedir. Son parametre de yönlendirmenin kaynağını belirten dosya bilgi göstéricisidir. Fonksiyon başarı durumunda birinci parametreyle belirtilmiş olan hedef dosyanın bilgi göstéricisine geri döner. Başarısızlık durumunda NULL adrese geri dönmektedir. 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.

stdin Dosyalarında EOF Etkisinin Yaratılması

Yukarıda da açıklandığı gibi aslında biz stdin dosyasından okuma yapmak istediğimizde bu dosya bir aygit sürücüye yönlendirildiği için aslında biz terminal aygit sürücüsünden okuma yaparız. Terminal aygit sürücüsü de okuma sırasında klavyeden alınanları bize vermektedir. Peki bu klavye gerçek bir dosya olmadığını göre EOF etkisi nasıl oluşmaktadır? Yani örneğin aşağıdaki döngüde f dosyası stdin'e yönlendirilmişse biz döngüden nasıl çıkarız?

```

while ((ch = fgetc(f)) != 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 aygit 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 çalışıldığı ele alacağız.

Windows ve UNIX/Linux komut satırında '|' simbolü 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 oluşturuktan 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 dosayı 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 mekanizma olmasaydı biz bunu aşağıdaki gibi üç aşamada yapmak zorunda kalardık:

```
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 står stdout dosyasına (ekrana) yazdırmaktadır. grep ise belli bir kalıbı (regex kalibini) dosya içerisinde bulan ve bulduğu satırın tamamını yazdırın 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 programımızı 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 yazıklarını ise c stdin'den okur. Benzer biçimde UNIX/Linux sistemlerindeki cat programı da bazı ayrıntıları ihmal edilerek şöyle yazılabılır:

```
#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 aygit sürücüye aktarabilir. Benzer biçimde biz getchar ile bir karakter bile okuyacak olsak fonksiyon aygit sürücüden birden fazla karakteri okuyup bir tampona yerleştirir ve bize o tampondakiler bitene kadar oradan verir. Peki bu standart dosyaların default tamponlama modu nasıldır? Standartlar bu konuda şunları söylemektedir: Eğer stdin ve stdout karşılıklı etkileşimli (interaktif) bir aygıtla yönlendirilmişse (terminal karşılıklı etkileşimli bir aygıt) kesinlikle tam tamponlamalı olamaz. Fakat satır ya da sıfır tamponlamalı olabilir. Eğer stdout ve stdin karşılıklı etkileşimli bir aygıtla 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 gelecektir. 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 çalıştırılmış olalım:

```
./sample > x.txt
```

Burada '\n' karakteri görülmence değil tampon dolunca dosya yazma yapılacaktır.

3) stderr dosyası ister karşılıklı etkileşimli aygıta 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 aygıta transfer edilecektir.

Peki 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 aygit sürücüden 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 çağrırmaya devam edersek fonksiyon bize tampondakileri sırasıyla verecektir. En son getchar bize tampondaki '\n' karakterini verir. Artık tampon boşalmıştır. Bir daha getchar çağrırsak 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' yi alır. İkinci getchar tamponda bilgi olduğundan klavyeden giriş beklemez. O da '\n' yi alır. Bir daha getchar kullanısaydık yeniden giriş istenecekti. Peki 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 sondakı '\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 Dosyasını Kullanan Standart C Fonksiyonlarının Davranışı

Yukarıda zaten getchar fonksiyonu açıklanmıştır. 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 yapılaceği dizinin ne kadar uzunlukta olması gereği 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 yapamadan 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 çıkışlı olur.

Aslında gets fonksiyonu getchar kullanılarak yazılabılır (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 tasarımda 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 aygıtları sürücülerinin belliri bir okuma limiti vardır. Fakat hem bu limit belli değildir hem de dosyayı yönlendirme yapıldığında aynı sorun yine oluşabilir.) Bu nedenle id bağılayıcıları gets kullanıldığından 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 çağrılrken ikinci parametreye dizinin uzunluğunu geçmek gereklidir. Fonksiyonun diğer davranışları gets ile aynıdır. gets_s fonksiyonu da getchar kullanılarak şöyle yazılabılır:

```

#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.

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 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 kaldığını 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. Şöyledir bir sonuç elde edilir:

```
a = 100, b = -1, result = 1, s = "ali "
```

Gördüğü gibi a için başarılı yerleştirme yapılmıştır. b için scanf işlemi başlattığında önce "ali" nin solundaki boşluk karakterlerini atmıştır. Fakat 'a' yi beğenmemiş ve onu yeniden stdin tamponuna bırakmıştır. gets de geri kalanları almıştır.

scanf fonksiyonunda format karakterleri arasında 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. Şöyledir 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 karkateri (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 karkaterini 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 edirz:

```
buf1 = ali, buf2 =  veli selami
```

scanf eğer boşluk karakterlerini attıktan sonra (ya da atmadan önce) dosya sonuyla karşılaşırsa 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;
}
```

stdin Tamponunun Boşaltılması

Standart C fonksiyonlarında bazen tamponda kalanları değil kullanıcının yeni girişini okumak isteyebiliriz. Örneğin iki getchar fonksiyonunu peş peşe kullanacak olalım. İkinci getchar ile yeni bir giriş isteyebiliriz. Ya da gets ile scanf birlikte kullanılırken de benzer bir istek oluşabilir. Örneğin:

```
#include <stdio.h>

int main(void)
{
    int no;
    char name[64];

    printf("No:");
    scanf("%d", &no);

    printf("Name:");
    gets(name);

    printf("No = %d, Name = %s\n", no, name);

    return 0;
}
```

Maalesef C'de standart olarak stdin tamponunu boşaltan bir fonksiyon yoktur. Bazı derleyicilerde fflush(stdin)unu yapıyor olmakla birlikte, bu kullanım uygunsuzdur. Çünkü her şeyden önce fflush read-only dosyalarda kullanılamaz ve stdin de read-only bir dosya kabul edilmekteir. 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 no;
    char name[64];
```

```

printf("No:");
scanf("%d", &no);

empty_stdin();

printf("Name:");
gets(name);

printf("No = %d, Name = %s\n", no, name);

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)

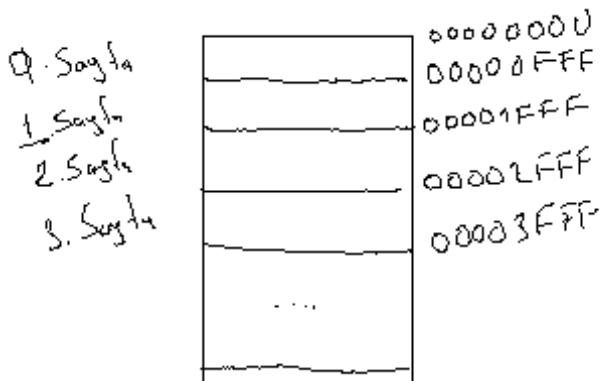
```

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 uzunlığında 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:

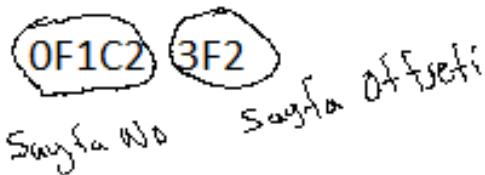
İşlemci	En Küçük Sayfa Uzunluğu	Desteklenen Diğer Sayfa Uzunlukları
32 Bit X86	4 KB	4 MB, 2 MB
64 Bit X86 (X64)	4 KB	2 MB, 1 GB
Itanium (IA-64)	4 KB	8 KB, 64 KB, 256 KB, 1 MB, 4 MB, 16 MB, 256 MB
Power PC	4 KB	64 KB, 16 MB, 16 GB
SPARC	4 KB	256 KB, 16 MB
Ultra SPARC	8 KB	64 KB, 512 KB, 4 MB, 32 MB, 256 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:"



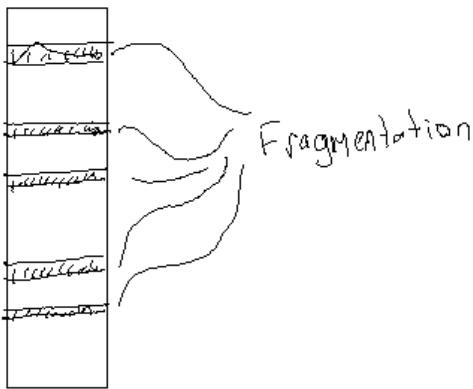
32 bit bir sistemde adresler 4 byte (8 hex digit) ile ifade edilir. Her fiziksel adres aslında bir sayfa içersindedir ve o sayfada belli bir offset'tedir. (Belli bir noktadan göreli uzaklığı "offset" denilmektedir.) Adresin hangi sayfada olduğunu bulmak için onu sayfa uzunluğuna bölmek gereklidir. (Tabii 2'lik sistemde bir sayıyı 4096'ya bölmek demek 12 kez sağa ötelemek, başka bir deyişle sağıdaki 12 biti atmak demektir.) Adresin o sayfanın hangi offset'inde olduğunu bulmak için ise adresin sayfa uzunluğuna bölümünden elde edilen kalana bakılır (sayının 4096'ya bölümünden elde edilen kalan düşük anlamlı 12 bitidir.) Örneğin:

0F1C23F2 fiziksel adres hangi sayfada ve o sayfanın hangi offset'tedir?



Sayfalama mekanizmasına sahip olan bir sistemde programlar fiziksel RAM'e ardışıl yüklenmek zorunda değildir. Ardışıl yüklenme zorunluluğunun 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 sonuctur. Eğer söz konusu belleğe (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 olduğu için pek bir işe yaramazlar. Fakat bellek kullanım oranını ciddi biçimde düşürürler. Bu olguya dışsal bölünme (external fragmentation) denilmektedir. Örneğin dışsal bölünmüş bir bellek şöyle görünümde olabilir (taranmış alanlar boş alanlar):



Örneğin 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 varlığı halde küçük olduklarından dolayı kullanılacak durumda değildir. Örneğin toplam her biri 20 byte'tan küçük yüzlerce boş alanın toplama 50K olabilir. Durum böyle olduğu halde biz 500 byte'lık bir alan bile tahsis edemeyiz.

Peki bir program fiziksel belleğe ardışıl yerleştirilmiyorsa nasıl çalışmaktadır? İşte program içerisindeki adreslere "doğrusal adresler (linear addresses), mantıksal adresler (logical addresses) ya da sanal adresler (virtual addresses)" denilmektedir. Biz burada "sanal adres" terimini kullanacağız. İşlemci bu sanal adresleri "sayfa tablosu (page table)" denilen bir tabloya bakarak fiziksel adreslere dönüştürmektedir. İşlemci sayfa tablosunu belli bir yazmacın (register) gösterdiği yerde arar. (Örneğin 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. Örneğin Intel'de iki kademeli sayfa tablosu kullanılır. Ancak biz burada kavramsal karmaşıklık oluşmasın diye tek kademeli bir sayfa tablosunu örnek vereceğiz. Sayfa tablosunun tipik biçimini şöyledir:

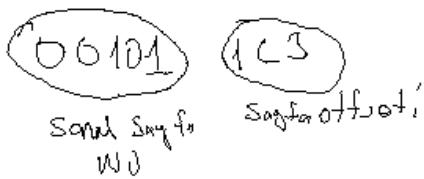
Sanal Sayfa No (Aslında Tabloda Yok)	Fiziksel Sayfa No	Sayfa Özellikleri
...
00100	01FC3	RW, P, User, D

00101	2FC40	RW, P, User, D
00102	1C167	RW, P, User, D
...

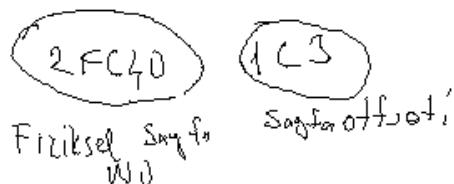
Tablonun ilk sütunu aslında bir indeks numarasıdır. Yani tabloda 00100 sanal sayfasına ilişkin giriş tabloda tablonun başından itibaren bir sayfa girişinin uzunluğu ile 0x100'ün çarpımı kadar uzaklıktadır. Tablodaki her bir satır "sayfa tablosu giriş'i (page table entry)" denilmektedir. İşlemci bir sanal adresle karşılaşlığında önce o sanal adresi sanal sayfa numarasına ve sayfa offsetine ayırtırır. 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:

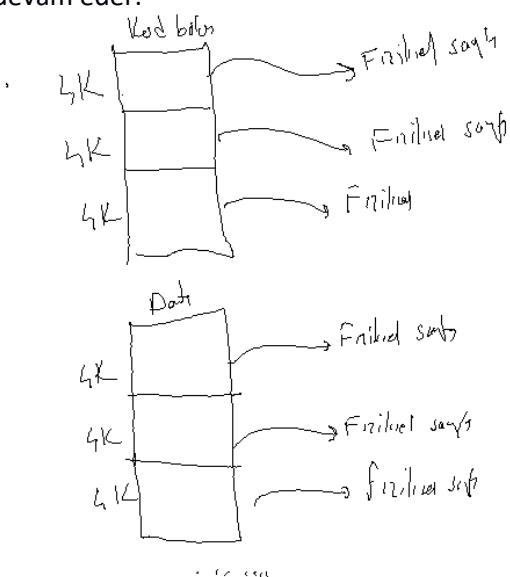


bundan sonra işlemci sayfa tablosuna bakarak 0x101 numaralı sanal sayfa numarasının hangi fiziksel sayfa ile eşleştirildiğine bakar. Yukarıdaki örnekte 0x101 numaralı sanal sayfa 2FC40 fiziksel sayfasıyla eşleştirilmiştir. Bu adresin 4096 ile çarpımına 1C3 eklenir ve şu değer bulunur:



Böylece erişimi fiziksel bellekteki 2FC401C3 adresinden yapar.

İşletim sistemi bir programı fiziksel belleğe sayfa sayfa (ardışılı olmayan sayfalar) yükler ve sayfa tablosundaki girişleri buna göre değiştirir. Böylece program işlemci tarafından sayfa tablosuna bakılarak çalıştırıldığı için kesiksiz çalışma devam eder.



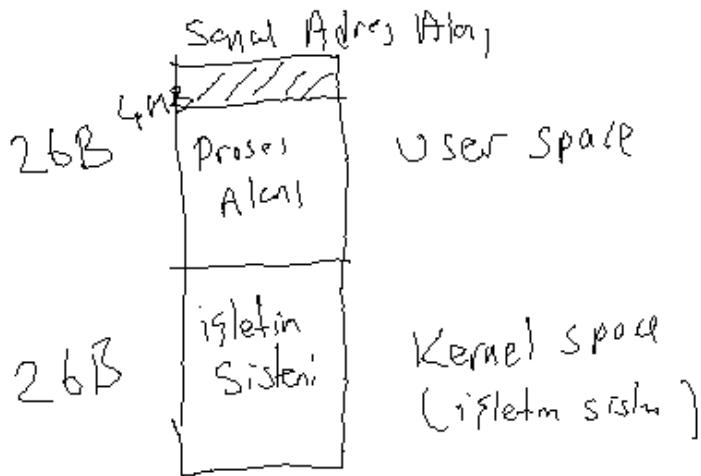
Sanal Bellek Mekanizması (Virtual Memory)

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 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 adresen itibaren fiziksel bellekte tek başlarına çalışacakmış gibi bir koda sahiptir.

Sanal bellek kullanan sistemlerde her prosesin ayrı bir sayfa tablosu vardır. İşletim sistemi programı yükleyeceği zaman onu fiziksel sayfalara dağıtır ve o prosesin sayfa tablosunu o fiziksel sayfları gösterecek biçimde düzenler. Böylece proseslerin sanal adres alanları aynı olmasına karşın onların parçaları farklı fiziksel adreslerde bulunurlar. Başka bir deyişle örneğin Windows'ta ve Linux'ta farklı proseslerdeki aynı sanal adresler aslında fiziksel bellekte aynı yere tekabül etmezler. Aslında bu tür sistemlerde fiziksel belleğin bir önemi yoktur. İşletim sistemi her prosesler arası geçiş (task switch) oluştuğunda otomatik olarak işlemcinin kullandığı sayfa tablosunu da o prosesin sayfa tablosunu gösterecek biçimde değiştirmektedir.

Sayfalama mekanizmasını kullanan işletim sistemlerinde proseslerin bellek alanları tam olarak izole edilmiştir. İşletim sistemi her prosesin sayfa tablosunda onların sanal sayfalarını farklı fiziksel sayflara 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 sayflara 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 olur. İşletim sistemi devreye girer ve prosesi cezalandırarak sonlandırır. Örneğin C ve C++'ta pek çok derleyici string'leri read-only sayflara yerleştirmektedir. Böylece bir string update edilmek istendiğinde exception olur.

```
#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.

Pekiyi 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 olacak 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

Sanal Sayfası NR.	Fiziksel Sayfa NR.
...	...
101A	38C7
101B	4714
101C	-
101D	4819
101F	-
...	...

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 karşılaşıldığından). 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 gerekiğinde yüklenebilir. Bu sisteme İngilizce "demand paging" denilmektedir.

Sanal bellek gerçekleştirmi adım adım şöyle yapılmaktadır:

1) Program kodunda bir sanal adrese erişilmişdir. 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 handlers) 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. Peki ya fiziksel bellek tıka basa doluya 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.

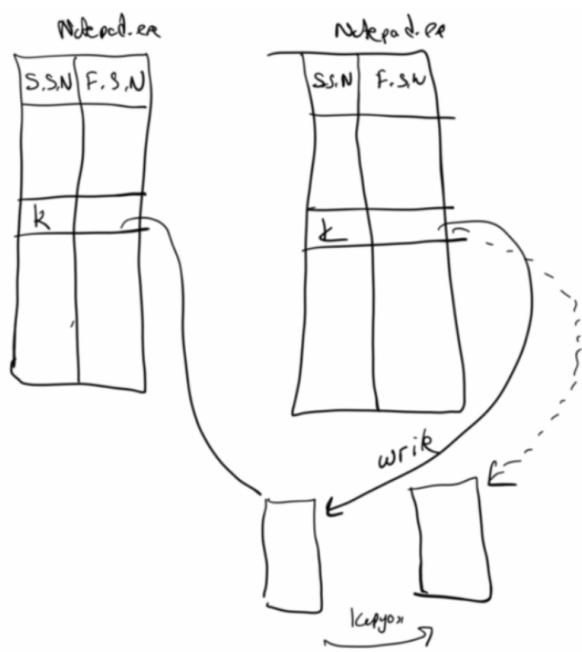
Yukarıdaki işlemlerin bazı ayrıntıları vardır. Örneğin işletim sistemi "swap-in" yapacağı zaman programın ilgili kısmını diskte nereden alınacaktır? Bu konuda işletim sistemlerinin değişik stratejileri vardır. Bazı işletim sistemleri her çalıştırılabilir (executable) dosyanın kendisini hem de bir "swap dosyasını" bu amaçla kullanır (örneğin Windows gibi). Bazı sistemlerde çalıştırılabilen (executable) dosya hiç kullanılmaz ve birden fazla swap dosyası kullanılır. Bazı sistemler swap dosyası yerine ayrı bir disk bölümünü (partition) kullanmaktadır. Peki neden çalıştırılabilen dosya bunun için yeterli olmamaktadır? Eğer bir fiziksel sayfa üzerinde değişiklik yapılmışsa o sayfa swap-out yapılırken onu çalıştırılabilen dosyaya yazmak mümkün değildir. Fakat read-only sayfalar her zaman çalıştırılabilen dosyadan alınabilmektedir. Peki bir sayfa fiziksel RAM'dan çıkartılacağı zaman her zaman o sayfanın diske geri yazılması gereklidir 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.

Peki 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 sonlandırır.

Peki 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.

Peki aynı program ikinci kez yüklenliğinde onun sayfaları yeniden fiziksel belleğe yüklenir mi? İşte işletim sistemleri bir program ikinci kez çalıştırıldığında yeni yaratılan prosesin sayfa tablosunda sayfa girişlerini mümkün olduğunda çalışmaka olan programın fiziksel sayfalarına yönlendirir. Yani ikinci kez çalıştırılan program onun ilk kopyasıyla aynı fiziksel sayfaları görür durumuna getirilir. Peki bu durumda proseslerden biri fiziksel sayfaya yazma yaptığından diğer bundan etkilenmeyecek midir? İşte aynı fiziksel sayfayı gösteren proseslerde proseslerden biri ortak sayfaya yazma yapmaya çalıştığından 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 çalıştırıldığından 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 aalında hiç değiştirilmezler. Bu sayfaların farklı prosesler tarafından kullanılmasında bir sorun oluşmaz.



Peki操作系统 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 değerin 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 kullanılabılır 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 özgürdür. Yani her prosesin ayrı bir çevre değişken listesi vardır. Genel olarak çevre değişkenleri pek çok işletim sisteminde üst prosesten 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.

Pekiyi prosesin çevre değişken listesi nerede tutulmaktadır? Bazı sistemlerde çevre değişkenleri Proses Kontrol Bloğunda tutulurken, modern sistemlerin çoğu (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 çağrıyor):

```
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 diziye yerleştirilen karakter sayısıyla (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 diziye 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 değişkenin 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 kullanılabilen 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 yeni 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 çağrıyor).

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 başarılı olması için önemli bir nedne yoktur. Geri dönüş değeri kontrol edilmeyebilir.

Örneğin:

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

int main(void)
{
    char *env;

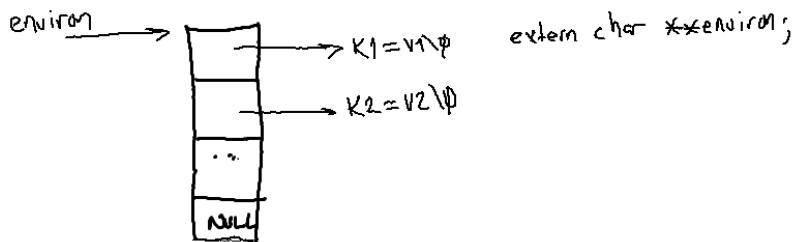
    env = GetEnvironmentStrings();

    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 bildirimi 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 adresi 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 oluşturulduğunda nasıl oluşturulmaktadır? Windows'ta prosesler CreateProcess isimli API fonksiyonuyla oluşturulur. 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 göreyorsunuz. 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 göreyorsunuz. 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 çalıştıracağı programın yeni bir çevre değişken listesi ile çalıştırılmasını sağlayabilir. Yani exec işlemi sırasında prosesin çevre değişken listesi atılarak yerine yeni 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 çalıştırduğumızda kabuğun çevre değişkenleri bizim programamıza aktarılır. Pekiyi 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" 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 çıkışınca yol olacaktır. Başka kabuk çalışmalarında 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 çalıştırırsak o değişken bizim programımıza da gelecektir.

Pekiyi UNIX/Linux ve MAC OS X sistemlerindeki kabuk programlarının sahip olduğu çevre değişkenleri nereden gelmektedir? Örneğin kursun yürütüldüğü nmakinede Linux terminalinde "env" komutu uygulandığında elde edilen kabuğun çevre değişken listesinin bir bölümü şöyledir:

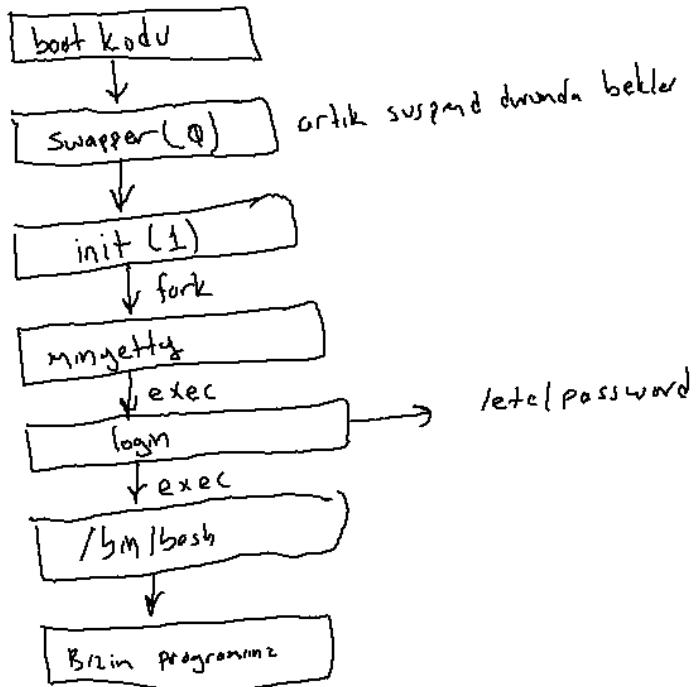
```

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 %
_=~/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:



İste 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 yönetici de bu dosyalara uygun şeyleri yazarak açılış sırasında sistemi konfigüre edebilir. Bu script dosyalarına aynı zamanda sistem yönetici birtakım çevre değişkenlerini de yazabilmektedir. Böylece bu çevre değişkenleri de ilgili prosese dolaylı biçimde eklenmiş olur.

Pekiyi UNIX/Linux ve MAC OS X sistemlerinde sistem boot edilirken hangi proses hangi script dosyalarına bakmaktadır? İşte bu da sistemde 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 çalıştırıldığını 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 prosesdir. 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
```

Gördüğü 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...

Pekiyi biz bir çevre değişkenlerinin kalıcılığını nasıl sağlayabiliriz? İşte UNIX/Linux ve MAC OS X sistemleri başlatılırken çalıştırılan çeşitli script dosyalarının içerisindene çevre değişken yaratım komutlarını eklersek sistem açıldığından otomatik olarak o çevre değişkenleri proses ağacına dahil edilmiş olur. Örneğin çevre değişkenleri için tipik olarak /bin/bash kabuğunun çalıştığı script dosyaları tercih edilmektedir. bash isimli kabuk programı "interaktif login" biçiminde, "interaktif login olmayan" biçimde ya da "interaktif olmayan" biçimde çalıştırılabilmektedir. (Buradaki "login" sözcüğü shell'in "user name" ve "password" sormasını belirtmektedir. Örneğin biz pencere yöneticisinden terminali açıyzorsak 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 çalıştmamaktadır. (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 masaüstü "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?

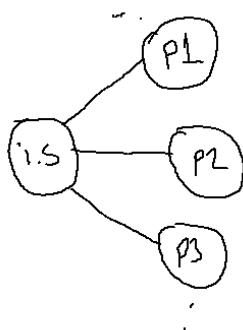
Çevre değişkenlerine işletim sistemi düzeyindeki global değişkenler gibi bakılabilir. Programlar birtakım dosyaları vs. bazı çevre değişkenlerinin belirttiği yerde arayabilirler. Örneğin bir veritabanı dosyası DATABASE isimli bir çevre değişkeninin belirttiği dizinde aranabilir. Bu durumda programcı 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 değişkenin 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 kullanılabilmektektir. Örneğin, üst proses belli bir çevre değişkenini set ederek alt prosesi çalıştırır. Alt proses böylece o çevre değişkeninden bilgi alabilir. Böylece 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 çalışabilemektedir? 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. Diğer yine bir süre çalıştırılır. Çalışma bu biçimde devam ettirilir. Prosesler kaldıkları yerden hep böyle çalışmaya devam ettirilirler. Dışarıdan bakıldığında sanki bunlar aynı anda çalışıyorum 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 biçimini uyumlu olması gerektiğinden genellikle sistem yönetici tarafından ayarlanamaz.

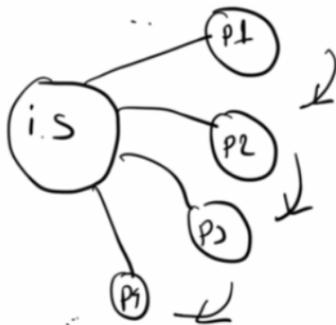
Bir prosesin quanta süresi dolduğunda çalışmasına ara verip diğer prosese geçilmesi sürecine "proseslerarası geçiş (process switch/task switch)" ya da daha genel olarak "bağlam geçiği (context switch)" denilmektedir. Şüphesiz proseslerarası geçiş sırasında bir zaman kaybı oluşmaktadır.

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 quata 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 böyledi. Bu sistemlerde proses kendi isteğiyle akışı bırakır ve proseslerarası geçiş oluşur. Bu tür sistemlerin gerçekleştirilemesi daha kolaydır fakat bunlarda bir proses sistemi tekeli altına alıbmaktadır.

Proseslerarası 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 geçiği 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 (schedulers) denilmektedir. Proseslerin hangilerinin ne zaman çalıştırılacağı konusunda kullanılan algoritmalar da "çizelgeleme algoritmaları (scheduling algorithms)" denir. En basit ve adil bir çizelgeleme algoritması "döngüsel çizelgeleme (round robin scheduling)" denilen algoritmadır. Bu algoritmda her proses sırasıyla belli bir süre çalıştırılır.



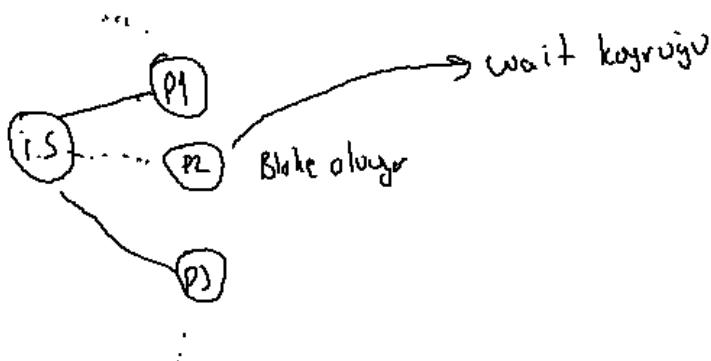
Proseslerin çizelgelenmesi birden fazla CPU ya da çekirdek olduğu durumda çok benzer yapılmaktadır. Bu durum birden fazla gişenin bulunduğu servis sistemlerine benzetilebilir. İşletim sistemleri birden fazla CPU ya da çekirdeğin bulunduğu durumlarda işlemciler ya da çekirdekler için genellikle ayrı kuyruklar oluştururlar. Yine her işlemci ya da çekirdek zaman paylaşımı 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 kuyruğuna aktarılması söz konusu olabilir. (Örneğin siz de bir markette kasada beklediğinizi 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 proses ona atanmaktadır. Tabii genel olarak kuyruktaki 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 faydalananabilmek için bir prosesin bir önceki quanta için

atanan 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 yila 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 takılabilen board'lar vardı. Çok çekirdekli sistemler önce "hyper threading" teknolojisi ile başlatılmıştır. Hyper threding'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.

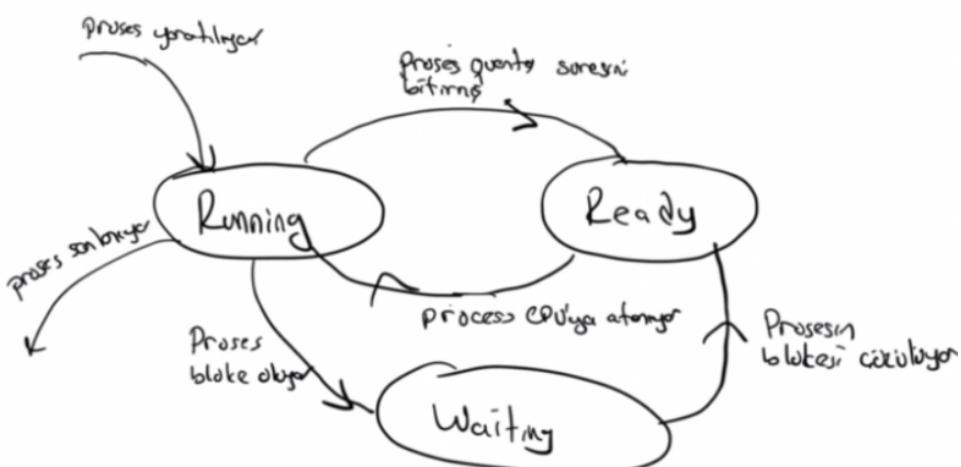
Bloke Kavramı

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) teknigiyle 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 doldurduguunda 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 kuruğu oluşturmaktadır.

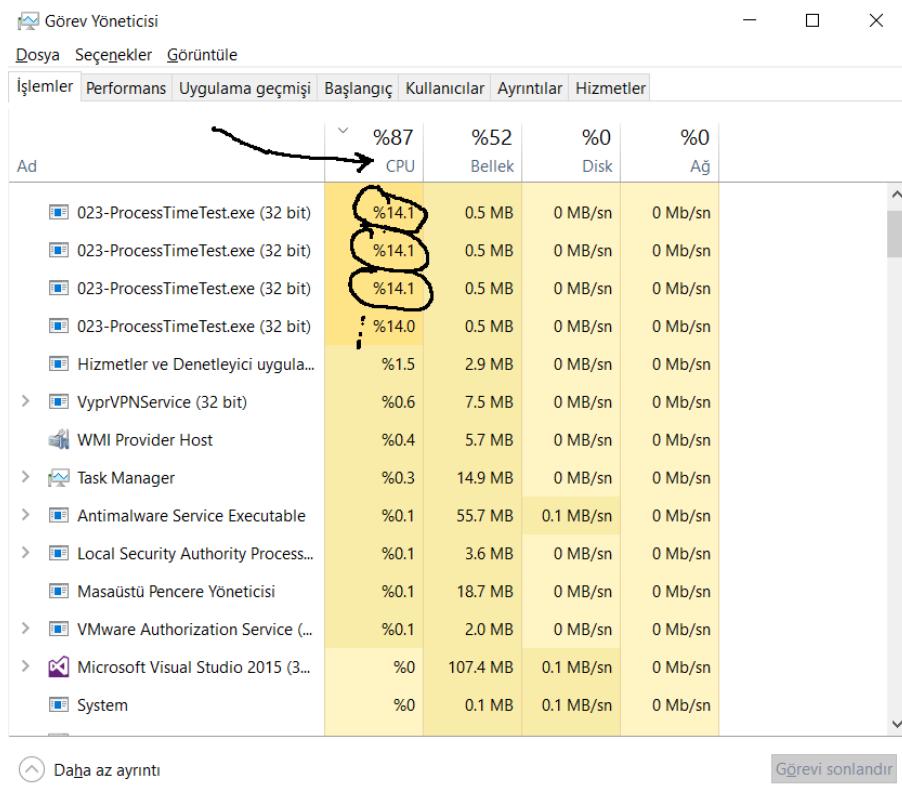
Pekiyi bir proses zmanının ne kadarını CPU'da ne kadarını wait kuyruğunda harcamaktadır? Tabii bu prosesinden prosesine değişir. Genel olarak çok CPU kullanan fakat az IO işlemi yapan proseslere "CPU yoğun (CPU bound)" prosesler, çok IO işlemi yapıp az CPU kullanan proseslere de "IO yoğun (IO bound)" 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. Pekiyi bu oran nasıl hesaplanmaktadır? Değişik hesaplama yöntemleri söz konusu olabilmektedir. Örneğin bir prosese 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:

CPU'da harcanan zaman

$$\frac{\text{CPU'da harcanan zaman}}{\text{CPU'da harcanan zaman} + \text{wait kuyruklarında harcanan zaman}}$$

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:



The screenshot shows the Windows Task Manager's Performance tab. A cursor points to the CPU column header, which is highlighted with a yellow arrow. The table lists various processes and their CPU usage percentages. The first process, '023-ProcessTimeTest.exe (32 bit)', has the highest CPU usage at 87%. Other processes like 'WMI Provider Host' and 'Task Manager' also have significant CPU usage, while others like 'System' and 'Local Security Authority Process...' have minimal usage.

Ad	CPU	Bellek	Disk	Ağ
023-ProcessTimeTest.exe (32 bit)	%87	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 boşta kaldığı (tüm proseslerin bloke olması dolayısıyla) zamana göre tespit edilebilir. Örneğin CPU zamanın 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 yüzdeslerini tüm CPU'lar'ın ortalaması olarak hesaplamış olabilir. Ancak

Windows'un Task Manager'daki bu hesaplamayı kesin olarak nasıl yaptığı herhangi bir dokümanda açıklamamıştır. Bu bilgiyi kaba bir fikir vermek amacıyla kullanıcıya göstermektedir. Aslında işletim sistemlerinde tüm prosesler bloke olabile işletim sisteminin bir prosesi (idle process) çalıştırılmaktadır. Bu da arka planda sistemi yormadan bazı küçük yardımcı işlemleri yapmaktadır.

Çok prosesli sistemlerde programın iki noktası arasında geçen zaman her çalıştırımda aynı olmayabilir. Prosesler zaman paylaşımı 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 dönüldüğü 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 < 10000000000; ++i)
        ;

    QueryPerformanceCounter(&li2);

    result = li2.QuadPart - li1.QuadPart;
    printf("%f\n", (double)result / freq.QuadPart);

    return 0;
}
```

Bu programdan çok sayıda çalıştırıldığında sürenin uzadığını gözlemlayabilirsiniz. Çünkü bu durumda sistemin yükü artmakta ve belli bir prosese daha geç çalışma sırası gelmektedir.

Peki de 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 ilgildir. 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 göreli 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 yazılı boşluklardan parse ederek argv dizisini oluşturmaktadır.) İlk komut satırı argümanın 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 kısmı 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 adreste bilgiyi saklayıp burayı tampon olarak kullanarak fonksiyon çıkışında yeniden orijinal yazıyı burada bırakır. Ama bu diziyi güncellemektedir. Halbuki string ifadeleri güncellenmez (C'de bir string ifadesinin güncellenmesinin tanımsız davranışa yol açtığını anımsayınız). Eğer birinci parametre NULL geçeilirse ve ikinci parametredeki dosya ismi '\' içermiyorsa Windows dosyayı sırasıyla şu dizinlerde arar:

- 1) CreateProcess fonksiyonunu uygulayan programın .exe dosyasının bulunduğu dizin
- 2) CreateProcess 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 içerisinde 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 çalıştmaktadır. Böylece komut satırından çalıştırılmak istenen program yukarıda belirtilen dizinlerde ve PATH çevre değişkeninde belirtilen dizinlerde sırasıyla aranmaktadır.

Fonksiyonun üçüncü ve dördüncü parametreleri proses ve prosesin ana thread'ine ilişkin güvenlik parametreleridir. Bu parametreler NULL olarak geçilebilir. Bu durumda default güvenlik durumu anlaşılır. Fonksiyonun beşinci parametresi kernel nesnelerinin alt proseslere geçirilebilmesine ilişkin ana şalter görevindedir. Bu parametre FALSE olarak geçilebilir. Altıncı parametre yaratılacak proses ile ilişkili belirlemeleri içermektedir. Bu parametre bazı bayrakların bit OR işlemeye 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 prosesinden alınır. Fonksiyonun sekizinci parametresi yaratılacak prosesin çalışma dizinini belirtir. Eğer bu parametre NULL geçilirse prosesin çalışma dizini üst prosesinden alınır. Fonksiyonun dokuzuncu parametresi yaratılacak proses ile ilişkili bazı ayırtıların 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 içini 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 geçirerek 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 biçimdedir. Yukarıdaki örnekte bunun ASCII yapılması gereklidir. Bunun için proje seçeneklerine gelinmeli "Character Set" seçeneği "Not Set" yapılmalıdır.

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 şurada 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\\"
                    "-I \"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 PROCESS_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 gördünüz 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 prosesin handle değeri yalnızca bu X prosesinde kullanılmak üzere oluşturulmuştur. Ancak prosesin Id değeri sistem genelinde tektir. Prosese özgü değildir. İşte biz Z prosesi Y prosesi üzerinde işlem yapacaksız Y prosesinin id değerini bilerek kendisine özgü bir handle değerini 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 yailacak 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 yapacaksız o prosesin Id değerini bilmeli ve bu Id değerinden OpenProcess 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:

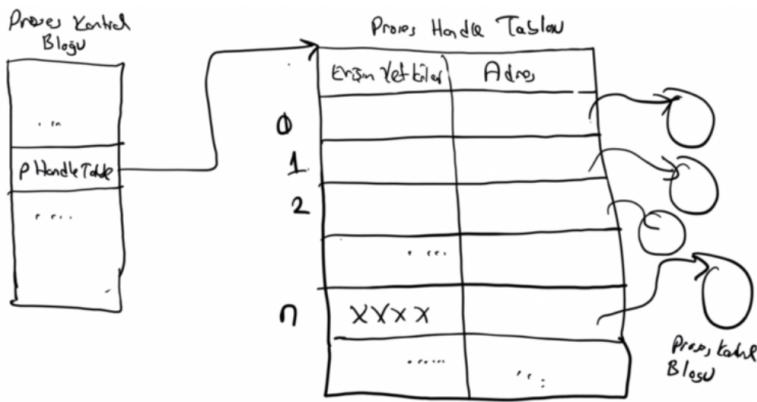
```
HANDLE GetCurrentProcess(void);
```

Benzer biçimde biz kendi prosesimiz id değerini de GetCurrentProcessId fonksiyonuyla elde edebiliriz:

```
DWORD GetCurrentProcessId();
```

Burada dikkat edilmesi gereken nokta şudur: Bizim kendi prosesimizn handle ve id değerini işin başında biz bilmemekteyiz. Bu değerleri bizi çalıştırın 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. Şöyled ki: Windows'ta bir grup kernel veri yapısına "kernel nesneleri" denilmektedir. Yalnızca prosesler değil, thread'ler, dosyalar, semaphore'lar vs. birer kernel nesnesidir. İşte tüm kernel nesnelerinin gerçek adresleri proses kontrol bloğunda "proses handle tablosu" denilen bir tabloda tutulmaktadır. Proses handle tablosu kernel nesnelerinin handle değerlerini, onların erişim özelliklerini ve gerçek adreslerini tutmaktadır. Genel yapısı aşağıdakine benzer biçimdedir:



İste aslında bir kernel nesnesinin handle değeri proses handle tablosunda bir indeks belirtmektedir. Asıl nesne adresi proses handle tablosunda bu indekstekten elde edilmektedir. Yukarıdaki şekilde bir prosesin handle değerinin n olduğunu varsayılmı. 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 gerçek 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 kernekli nesnesidir. Bu durumda biz CreateFile fonksiyonuyla bir dosyayı açtığımızda CreateFile fonksiyonunun bize verdiği verdiği handle değeri de proses handle tablosunda bir indeks belirtmektedir.

Windows sistemlerinde yalnızca proseslerin değil aynı zamanda tüm kernel nesnelerinin handle değerleri o prosese özgüdür. Çünkü bu handle değerleri o prosesin proses handle tablosunda bir indeks belirtmektedir. Bu indeks değerini biz başka prosese iletsek artık o indeks o prosesin handle tablosunda indeks belirteceği için bir anlamı olmayacaktır. Aşağıdaki örnekte proseste toplamda üç kernel nesnesi yaratılmıştır. (CreateProcess API fonksiyonunun ana thread için de bir kernel nesnesi yarattığını anımsayınız.) Ekrana yazdırılan bu değerlerin küçük olduğunu ve bir indeks belirrtiğine 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);
}

000000000000B0, 0000000000000AC 0000000000000B8
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 . . .
```

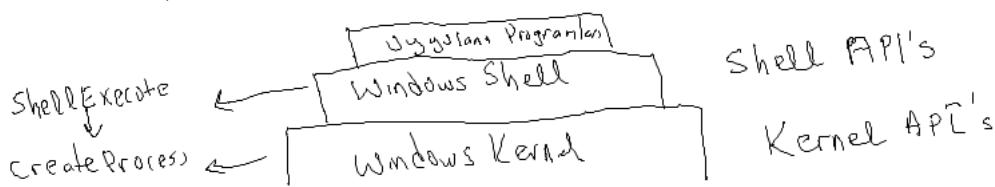
Windows'ta adresleri proses handle tablosunda saklanan tüm kernel nesneleri CloseHandle isimli API fonksiyonuyla kapatılmaktadır. Tabii programcı kapatımları yapmasa bile proses sonlandığında tüm kernel nesneleri sistem tarafından kapatılmaktadır. Bir kernel nesnesi CloseHandle fonksiyonuyla kapatıldığında proses handle tablosundaki ilgili satır boşaltılır. Artık bu satır (dolayısıyla bu indeks) yeni bir kernel nesnesi için kullanılabilecektir. Ayrıca bir prosesi ya da

thread'i CloseHandle fonksiyonuyla kapatmış olmamızın o prosesin ya da thread'in sonlanacağı anlamına gelmediğine dikkat ediniz. Bir prosesi ya da thread'i CloseHandle fonksiyonuyla kapatlığımızda o proses ya da thread sonlanmaz yalnızca bizim ona erişimimiz sonlandırılmış olur.

Anahtar Notlar: Windows'ta sistemde proseslerin (yani çalışmaktadır 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 yaratılabilmiştir. ShellExecute bir sistem fonksiyonu değildir. Zaten kendi içerisinde CreateProcess fonksiyonunu çağırmaktadır. ShellExecute bir kabuk (shell) fonksiyonudur. Windows'un kabuğu "Windows Explorer" denilmektedir. Aslında masaüstü olarak gördüğümüz bu kabuk da "explorer.exe" isimli bir prosesidir. 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 çalıştırılamayan dosyaları da biz ShellExecute fonksiyonuna verebiliriz. Bu durumda ShellExecute fonksiyonu "registry" denilen bir veritabanının kayıtlarına erişerek bu uzantılı dosyanın hangi çalıştırılabilen dosyaya ilişkilendirilmiş olduğunu belirler ve CreateProcess ile o çalıştırılabilen dosyayı çalıştırır. Sonra da bizim verdigimiz 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 çalıştmak 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 kayttır. Bu ilişkilendirme kayıtları Windows'un içerisindeki "registry" denilen dosyalarda tutulmaktadır. Dolayısıyla ShellExecute fonksiyonu da bu registry kayıtlarına bakmaktadır. Tabii regisry yalnızca dosya ilişkilendirmelerini tutan bir veritabanı değildir. Windows tüm ayaraları da bu registry veritabanında tutulmaktadır. Hatta isterse kendi program ayarlarını da bu registry veritabanında tutabilmektedir. Yani registry denilen bu veritabanına uygulama programcılar da erişebilmektedir. Ancak programcılar bu erişimi doğrudan yapmak yerine dolaylı olarak yapan ve ismine "registry API fonksiyonları" denilen bir grup API fonksiyonuyla yaparlar. Denergimizde "Windows Sistem Programlama" kurslarında bu registry veritabanı hakkında ayrıntılı bilgiler verilmektedir.

Örneğin biz masaüstünde bir .txt dosyasına çift tıklaşmış olalım. Sırasıyla şu olaylar gerçekleşecektir:

- 1) Masaüstü "explorer.exe" isimli prosesidir. Dolayısıyla masaüstündeki farenin çift tıklanması da bu proses tarafından ele alınır.
- 2) Masaüstü prosesi (explorer.exe) çift tıklandığı .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 paraametresi 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 belirmek 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 dönüştümeniz gereklidir.

Örnek bir ShellWxecute çağrıları 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 fonksiyounun 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 gereklidir. 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 başarısız 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 istiyoğanız programınızı (ya da Visual Studio IDE'sini) "Run as administrator" biçiminde çalıştırımalısınız. Kernel nesnelerinin güvenlik parametreleri "Windows Sistem Programlaması" kurslarında ele alınmaktadır.

Pekiye OpenProcess ile açılan bir prosesin bilgilerini nasıl elde edebiliriz. İşte artık handle değerini alarak 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 module 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 handledeğerlerini 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.

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 çağrımaktadır. Bu sistemlerde proses yaratmanın başka bir yolu yoktur. (fork fonksiyonunun vfork isimli bir versiyonu da vardır fakat bunun kullanım gereklisi 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 prosese kopyalanır. Yine alt prosesin sanal bellek alanı tamamen üst prosesten (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 idegeri 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 tamsayışal 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.

Pekiyi her iki proses de fork fonksiyonundan çıktığına göre ve aynı kodu çalıştırılacağına göre onları birbirlerinden nasıl ayıralımız? İşte fork fonksiyonundan üst proses 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 biçimini 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 üset prosesin kontrol bloğunun alt prosesin kontrol bloğuna kopyalandığını belirtmiştir. Bunun anlamı şudur: Alt 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 prosese 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 değerileyile alt prosesin ise 0 değeri ile çıktığını söylemiştim. Bu alt prosesin id değerinin 0 olduğu anlamına gelmemektedir. Yalnızca fork fonksiyonun geri dönüş değeri bu biçimdedir. Yoksa yeni yaratılmış olan alt prosesin de anlamlı bir id değeri vardır.

fork işlemi sırasında üset prosesin tüm bellek alanının da alt prosese kopyalandığını belirtmiştim. 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 işlemden sonra alt proses bir fiziksel sayfaya yazma yaptığından artık o sayfanın o anda bir kopyası çıkartılıp sayfalar birbirlerinden ayrılmaktadır. Bu mekanizmaya "copy on write" denildiğini anımsayıınız.

Eski thread'ler yoktu. Dolayısıyla bir işi birden fazla akışa yapabilmek 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 çalıştırılamaz. fork ancak bir prosesin özdeş kopyasını çalıştırır. Peki bir 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 çalıştığı kod bellekten atılır, onun yerine exec fonksiyonunda

belirtilen dosya beläge 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 execle(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 çağıracak 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 dönmez. 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... /* unreachable code */\n");

    return 0;
}
```

Fonksiyonun son parametresi için NULL argümanı girilirken tür dönüştürmesi yapılmalıdır. Yani aşağıdaki çağrıma biçim 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 sıfırılmış 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 gereklidir. (Burada NULL makrosunun dönüşümü yapılması gerekmemektedir.)

Şimdi komut satırı argümanını ile aldığı programı çalıştırın bir program yazmak isteyelim. Örneğin programımız sample olsun o da ls'yi çalıştırınsın:

```
./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 ':' karakteriyle 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 gereklidir. Aksi halde bu fonksiyonlar PATH çevre değişkenine bakmazlar ve tamamen p'siz versiyonlar gibi yol ifadesinin mutlak ya da görelî olması durumuna göre aramalarını yaparlar. Örneğin:

```
execlp("a/sample", "a/sample", (char *)NULL);
```

Burada execlp PATH çevre değişkenine bakmayıacaktır. 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:

```
execlp("sample", "a/sample", (char *)NULL);
```

Burada sample programı bulunulan dizinde olsa bile oraya bakılmayacaktır. Yalnızca PATH çevre değişkeni ile belirtilen dizinlere bakılacaktır. Örneğin:

```
execlp("ls", "ls", "-l", (char *)NULL);
```

Burada dosya isminde hiçbir '/' karakteri geçmediği için execlp dosyası yalnızca PATH çevre değişkeni ile belirtilen dizinlerde arar. Örneğin çalışma dizinizde "sample" isimli çalıştırılabilen bir dosya bulunuyor olsun. Aşağıdaki gibi bir exec çağrıSİ b dosyayı çalıştırabilir mi?

```
execlp("sample", "sample", (char *)NULL);
```

Yanıt hayır. Çünkü burada exec execlp dosya isminden '/' karakteri olmadığı için onu yalnızca PATH çevre değişkeni ile belirtilen dizinlerde arayacaktır. Prosesin çalışma dizinine bakmayıacaktır. Fakat exec işlemi şöyle yapılsaydı program çalıştırılabilirdi:

```
execlp("./sample", "./sample", (char *)NULL);
```

Ya da aşağıdaki bir çağrıda da sample programı çalıştırılabilir:

```
execl("./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 çalıştırılsrsa 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 içeresine bir '/' karakteri karıştırılmıştır ve bu '/' karakteri exec fonksiyonlarının p'li versiyonlarının artık PATH çevre değişkenine bilmemesine 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 && execle("app", "app", (char *)NULL, env) < 0) {
        perror("execle");
        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 çalıştığı kod eskisi ile aynı kod olmaktadır. Tabii biz üst prosesle alt prosesi fork çıkışında birbirinden ayırmamekteyiz. Yalnız başına exec fonksiyonları ise proses yaratmayıp mevcut prosesin başka bir kodla çalışmasına devam etmesini sağlar. Pekiyi hem bizim programımız çalışmaya devam ederken hem de başka bir programı nasıl çalıştırabilirmiz? İşte bunun için fork ve exec birlikte kullanılmalıdır. Önce bir kez fork yapılır, alt proseste exec uygulanır. Tipik kalıp şöyledir:

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

if (pid == 0)
    if (exec1(...) < 0) {
        perror("exec1");
```

```

        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 çalıştırıldığı için o akış artık bu koddan devam etmeyecektir. İkinci if deyimi içerisindeki if deyiminin else kısmına da gerek yoktur. Çünkü exec başarılı olursa zaten ekip bu koddan devam etmeyecektir. Ayrıca exec işleminin başarısızlığı durumunda exit ile yalnızca alt prosesin sonlandırıldığını dikkat ediniz. Yukarıdaki işlem daha kompakt yapılabildi:

```

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 sistemeinde 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 dosyasına yazılırsa bunlar hızlı biçimde çalıştırılabilmektektir. İçerisinde komutların bulunduğu bu kodlara "script" denilmektedir. Script genel bir kavardır. Kabuk programlarının çalıştırıldığı 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 kabul programına scrpt 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  
...
```

Peki exec fonksiyonları nasıl oluyor da bir text dosyayı çalıştırıyor? 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 scrpt dosyasını da bu programa komut satırı argümanı olarak verirler. Yani yukarıdaki gibi bir sample.sh dosyası tamamen exec fonksiyonlu 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 satır "shebang" denilmektedir.

Windows Sistemerinde 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 kullanılmıştır. 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 çağrırlar) fonksiyonu prosesi sonlandırmakta kullanılır. C'nin standart exit fonksiyonu ise dolaylı olarak bu fonksiyonları çağırmaktadır. 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);
```

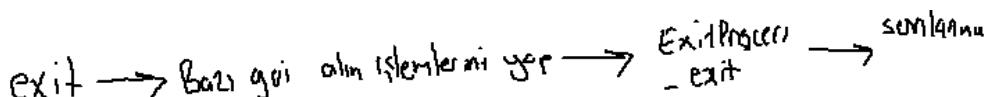
Yukarıda da belirttiğimiz gibi aslında C'nin standart exit fonksiyonu Windows sistemlerinde ExitProcess API fonksiyonunu, UNIX/Linux ve MAC OS X sistemlerinde ise _exit POSIX fonksiyonunu çağırmaktadır.

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 kullanılabilmektedir. Ancak ne olursa olsun bir prosesin başka bir bir prosesi ansızın bu biçimde sonlandırması iyi bir teknik değildir. Çünkü sonlandırılan proses önemli bir işlemin ortasında olabilir. Bu sonlandırma zaralı sonuçları oluşturabilir. TerminateProcess API fonksiyonunun prototipi şöyledir:

```
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);
```

Fonksiyonun birinci parametresi sonnadırılaç prosesin handle değerini ikinci parametresi ise onun exit kodunu alır.

C'nin standart exit fonksiyonu işletim sisteminin sistem fonksiyonlarını çağrımadan 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 yapmamışsa prosesi doğrudan işletim sisteminin API fonksiyonlarıyla ya da POSIX fonksiyonlarıyla da sonlandırabiliriz.



Bir proses sonlandığında işletim sisteme "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 dönülmüş 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.



Pekiyi exit kodu ne işe yaramaktadır? İşletim sistemi için exit kodunun kaç olduğunu bir önemi yoktur. İşletim sistemi bu kodu alır, saklar. Eğer üst proses isterse ona verir. Böylece üst proses alt prosesin hangi exit koduya (yani nasıl) sonlandığını bilmiş olur. Duruma göre birşeyler yapabilir. Geleneksel olarak başarılı sonlanmalarda sıfır değeri, başarız 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 gereklidir. Windows'ta GetExitCodeProcess fonksiyonu prosesin exit kodunu almaktadır:

```
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 başarılı olabilmektedir. (Örneğin beklenen 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 bulunmaktadır. 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("execl");

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 beklenenek 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 beklenenek prosesin id değerini belirtir. Fonksiyon yalnızca bu prosesi bekler.

-1: Bu durumda fonksiyon herhangi bir alt prosesi bekler. Bu seçenek fonksiyonun wait gibi davranışına yol açar.

0: Bu durumda proses grub id'si fonksiyonu çağrıran prosesin grub id'si ile aynı olan herhangi bir proses beklenir.

< -1: Bu durumda fonksiyon proses grub id'si burada belirtilen değerin 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 çağrısına bakınız:

```
waitpid(-1, &status, 0)
```

Bu çağrım aşağıdaki wait çağrımları 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 prosesinden daha önce sonlanmışsa bu durumda alt proseslere öksüz (orphan) prosesler denilmektedir. Sistem öksüz duruma düşen prosese init prosesini (id'si 1 olan prosesi) 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 prosesi alt prosesin exit kodunu alarak onun hortlak duruma düşmesini engeller. Pekiyi hortlaklılığın 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 prosese verilememektedir. wait fonksiyonlarının dışında otomatik hortlaklılığı engellemenin başka yolları da vardır. Fakat en normal durum fork işlemini 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.

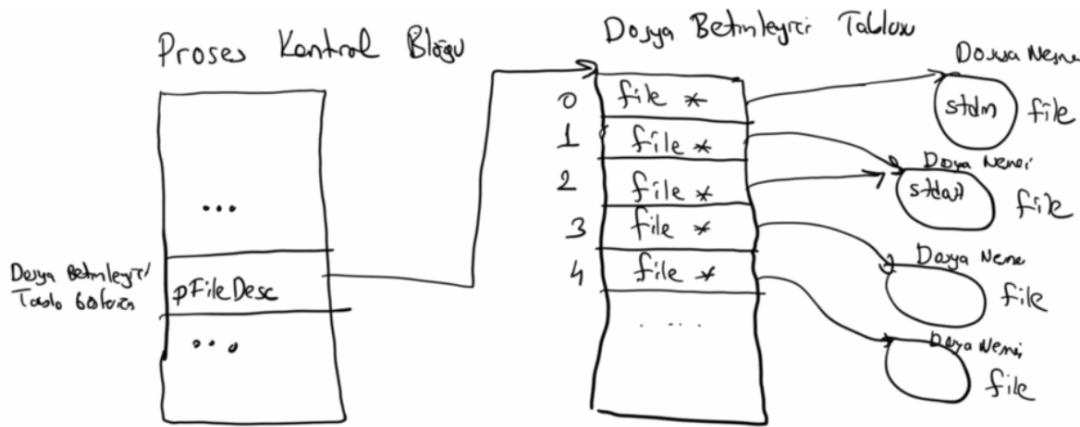
Pekiyi UNIX/Linux sistemlerinde üst proses alt prosesi wait ile beklemek istemiyorsa bu hortlaklık durumu yine de engellenebilir mi? İşte bunun için iki temel yöntem vardır. Birincisinde üst proses SIGCLD isimli sinyali ele alır. Alt proses sonanıp bu sinyal için set edilen fonksiyon çağrılığında bu fonksiyon da hiç bekleme yapmadan wait uygulayabilir. İkinci yöntemde daha alt proses yaratılmadan önce üst proses alt prosesin exit kodunu almayıcağı 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. Özette 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 kullanılabilmektedir. Bu yöntemler "UNIX/Linux Sistem Programlama" kurslarında ele alınmaktadır.

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 blogunda tutulmaktadır.

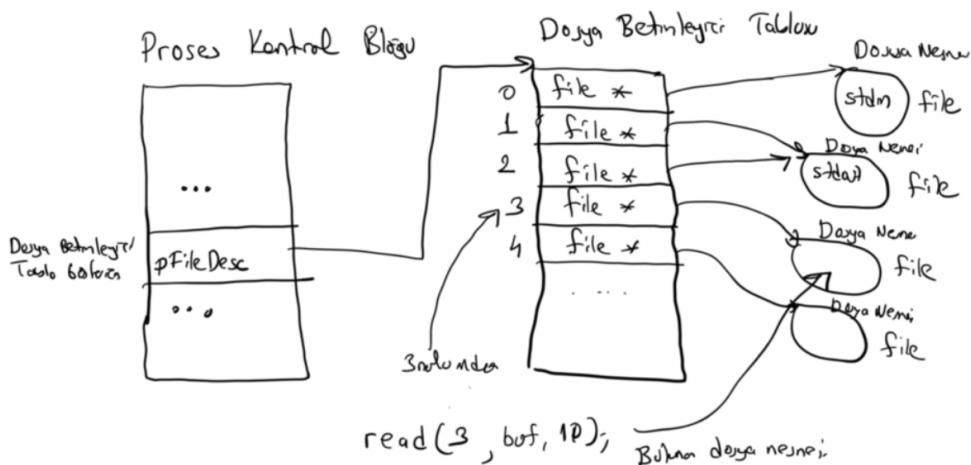
Proses kontrol bloğundaki bir gösterici ismine "Dosya Betimleyici Tablosu" denilen bir tabloyu göstermektedir. Dosya betimleyici tablosu dosya nesnelerinin adreslerini tutan bir gösterici dizisidir. (Örneğin Linux için bu dizinin her bir elemanı struct file * türündendir). Açılmış olan bir dosyanın tüm bilgileri bu dosya nesnesinin içerisindeindedir. 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.

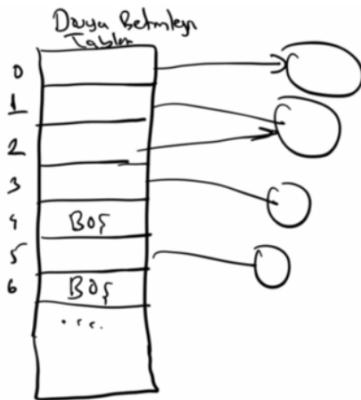


Dosya için gereken tüm bilgiler dosya nesnesinin içerisindeindedir.

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 filanca dosyayı belirtiyor olabilir.

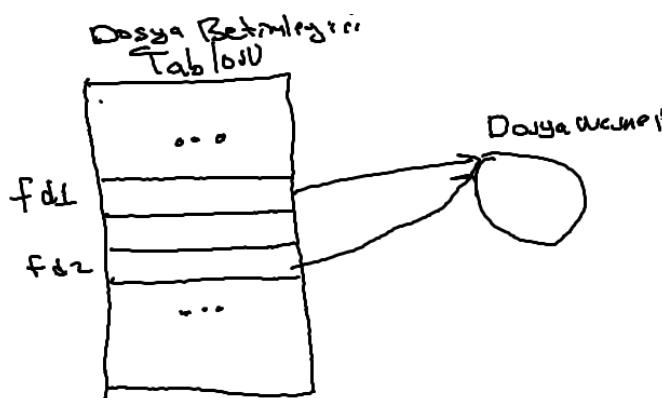
Bir dosya kapatıldığında eğer dosyayı gösteren başka bir betimleyici yoksa dosya nesnesi yok edilir. Dosya Betimleyici Tablosundaki ilgili giriş de boşaltılır. Yani belli bir anda Dosya Betimleyici Tablosu'nun bazı girişleri dolu bazıları boş (yani NULL değerinde) olabilmektedir.

POSIX standartlarında open fonksiyonunun tablodaki ilk boş betimleyiciyi vereceği garanti edilmiştir. Örneğin:



Burada open fonksiyonu uygulansa open başarı durumunda bize 4 numaralı betimleyiciyi verecektir.

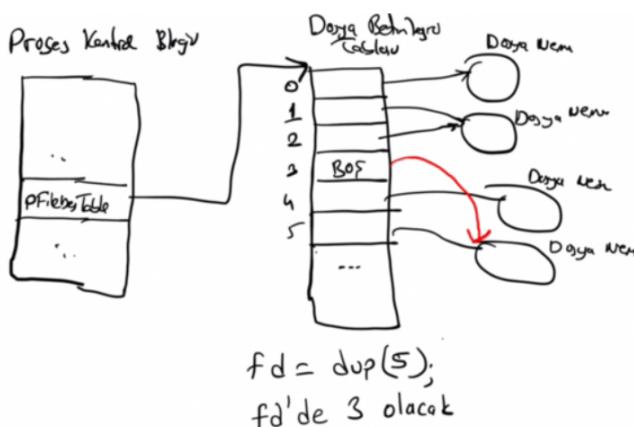
Bazen aynı dosya nesnesini gösteren birden fazla betimleyici olmasını isteyebiliriz. Örneğin:



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 programının fd1 ya da fd2 betimleyicisini kullanması arasında hiçbir farklılık yoktur. İki farklı betimleyicinin aynı dosya nesnesini gösterir hale getirilmesine "dosya betimleyicisinin çiftlenmesi" denilmektedir. Dosya betimleyicilerinin çiftlenmesiyle ilgili iki önemli POSIX fonksiyonu vardır. Bunlar dup ve dup2 fonksiyonlardır. 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.

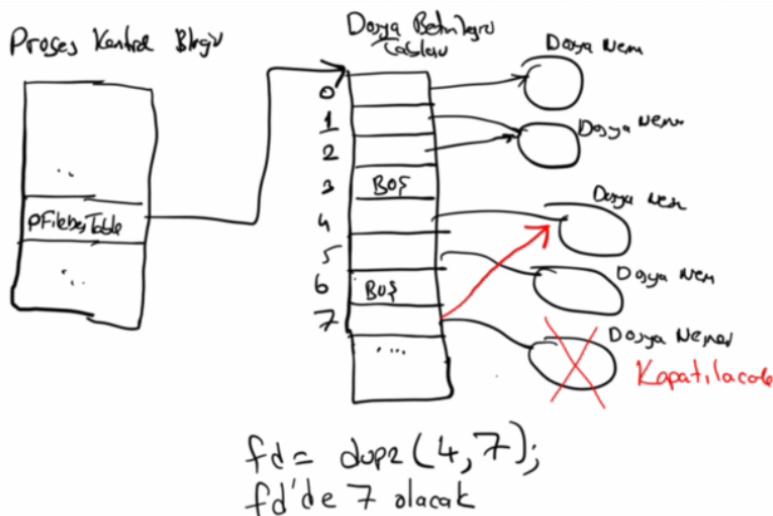
dup2 fonksiyonun prototipi ise şöyledir:

```
#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 parametrede belirtilen betimleyici açık bir dosya nesnesini gösteriyorsa önce onu kapatır. Sonra bu betimleyicinin ilk parametredeki 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.

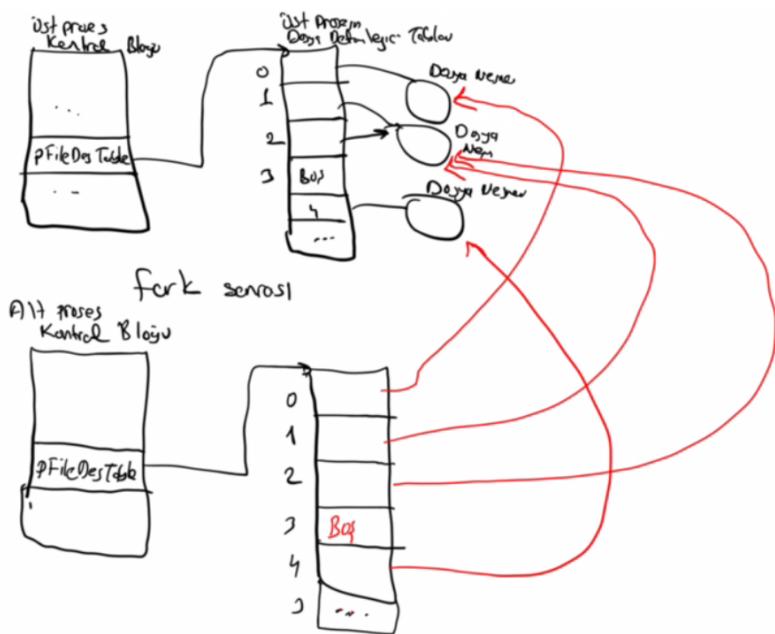
Dosya Betimleyici Tablosu'nun bir uzunluğu vardır. Örneğin Linux sistemlerinde default olarak her prosesin Dosya Betimleyici Tablosu 1024 eleman uzunluğundadır. Yani biz bu sistemlerde aynı anda en fazla 1024 dosyayı açık durumda tutabiliyoruz. Tabii root kullanıcısı isterse kendi ya da başka bir prosesin Dosya Betimleyici Tablosunu büyütебilir. 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 sayı 0'a düşmüştür 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;
    f_version;
    void                      *private_data;
    *f_iobuf;
    long                        f_iobuf_lock;
};
```

Yapının f_count elemanı nesne sayacını, f_pos elemanı dosya göstericisiin konumunu, f_uid, f_gid elemanları dosyanın kulalnıcı 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 prosese 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 proseseki aynı numaralı betimleyiciler aynı dosya nesnesini gösteriyor durumda olur. Tabii dosya nesnelerinin sayıları da birer artırılmış olacaktır. Örneğin:

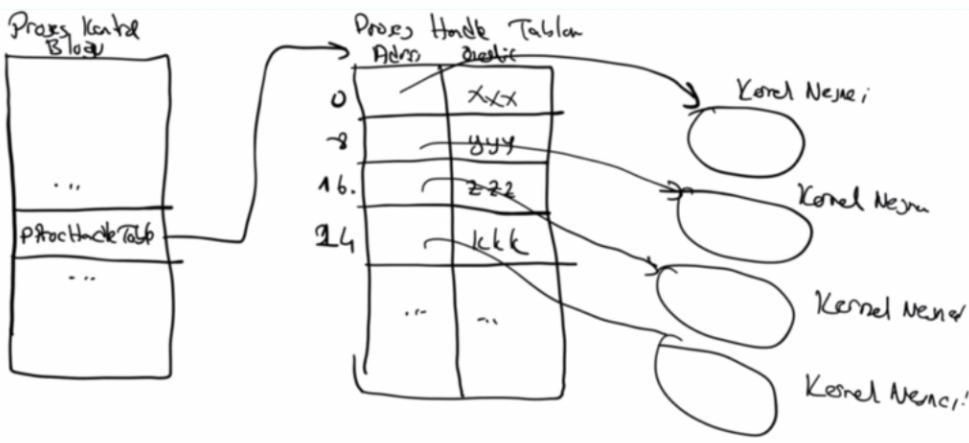


fork işleminden sonra exec işlemi yapıldığında artık yeni bir program çalışacağına göre çalışan program üst prosesin açmış olduğu dosyalarla ilgilenmeyecektir. (Tabii ilgilenecek biçimde de yazılmış olabilir.) İşte exec yapıldığında açık olan dosyaların otomatik kapatılması sağlanabilmektedir. İşletim sistemi açık olan her dosya için o prosese özgü bir "close on exec" bayrağı tutar. Bu bayrak default durumda "clear" edilmişdir. 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 kaapatılmaktadır.

Windows Sistemlerinde Dosya 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 Nesnsleri (Kernel Object)" denilen tüm nesneler benzer mekanizmaya sahiptir. Yani dosyalar için söz konusu olan mekazizma diğer kernel nesneleri için de aynı biçimde söz konusudur. Windows'ta kernel nesneleri denilen bir grup nesne aynı biçimde organize edilmektedir. Bu tasarım UNIX/Linux sistemlerinden biraz farklıdır. Ancak genel veri yapıyı 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 nesneleri 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 típki UNIX/Linux sistemlerinde olduğu bir tamsayısal indeks (tablonun başlangıcına göre offset) belirtir. Windows'taki sistem şekilsel olarak söyle 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 olabilirdik. 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 Yönlendirmesi

Anımsanacağı gibi UNIX/Linux ve Windows sistemlerinde aygit sürücüler de aslında birer dosya gibi organize edilmektedir. Kabuk üzerinden proses yaratıldığında üst prosten 0, 1 ve 2 numaralı betimleyiciler dolu olarak gelir. 0 numaralı betimleyicinin gösterdiği dosya nesnesi "stdin" (tipik olarak klavye) aygit sürücüsü ile, 1 numaralı betimleyicinin gösterdiği dosya nesnesi ise "stdout" (tipik olarak ekran) aygit 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" aygit 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 yazmak 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);
    close(2);
    close(3);
    if (fd = open("test.txt", O_CREAT | O_RDWR, 0644) < 0)
        exit_sys("open error");
    if (write(fd, "Hello, world!", 13) != 13)
        exit_sys("write error");
    if (lseek(fd, -13, SEEK_CUR) == -13)
        exit_sys("lseek error");
    if (read(fd, buffer, 13) != 13)
        exit_sys("read error");
    if (close(fd) < 0)
        exit_sys("close error");
}
```

```

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 std::cout dosya nesnesini değil, "test.txt" isimli dosyaya ilişkin dosya nesnesini gösteriyor durumda olur. Fakat yönlendirmede dup2 fonksiyonunu kullanmak daha güvenlidir. Çünkü çok thread'lu uygulamalarda bu tür işlemler senkronizasyon sorununa yol açabilirler. O halde yönlendirme daha biçimsel olarak aşağıdaki gibi yapılmalıdır:

```

#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);
}

```

Pekiyi kabuk ‘>’ ya da ‘<’ işaretleri 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 yapılacakı dosyayı ve çalıştıracağı programı komut satırı argümanlarıyla birlikte komut satırı argümanı olarak verir. Örneğin:

```
redirect test.txt ls -l
```

Böylece ls programı "-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 sistemlerinde sayfa tabloları yoluyla prosesler arasında tam bir izaolasyon sağlanmıştır. Bu sistemlerde her proses sanki fiziksel belleğe tek başına yüklenip çalışmış gibi bir illüzyon

oluşturulmaktadır. Bir prosesin 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 proses dış dünyadan elde ettiği verileri başka proseslere iletmek isteyebilir. Ya da bir proses başka bir proses tarafından yönetilmek isteyebilir. Örneğin bir chessboard.exe isimli satranç tahtası programını düşünelim. Bu program bir GUI ortamı oluşturarak taşların hareketlerini sağlaması. İşte bu programa hangi taşı nereye oynatacağını başka bir program proseslerarası haberleşme yöntemiyle söyleyebilir. Ya da bir araba oyunu arabanın o andaki konumunu proseslerarası haberleşme yöntemiyle başka bir programa iletебilir. O programda oyun için bir kabini yönetebilir. İstemci-Sunucu (Client-Server) mimarisinde arka planda hep proseslerarası haberleşme yöntemleri kullanılmaktadır. Bu mimaride işi asıl yapan programa "sunucu (server)" program denilmektedir. "İstemci (client)" program proseslerarası haberleşme yöntemleriyle isteğini sunucu programa iletir. Sunucu işlemini yapıp sonuçları yine proseslerarası haberleşme yöntemleriyle istemciye gönderir. İstemci-Sunucu mimarisi ilerde 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 girer. Bu amaçla oluşturulmuş değişik protokoller vardır. Bugün en yaygın kullanılan protokol Internet'in de kullandığı IP protokol ailesidir. Kursumuzda IP protokol ailesi ile haberleşme ele alınacaktır.

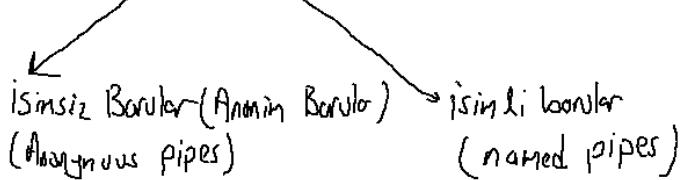
Anahtar Notlar: Raspberry Pi'nın 3 numaralı modeli oldukça hızlandırılmıştır (4 core). Microsoft da Windows 10'un Raspberry Pi versiyonunu oluşturmuştur. Bu tür çok sayıda embedded Linux kitleri bulunmaktadır. Ancak Raspberry Pi oldukça standartlaşmış durumdadır ve önemli ölçüde desteği sahiptir. Dokümantasyonu da oldukça genişdir. Raspberry Pi üzerine çok çeşitli işletim sistemi kurulabiliyorsa da şu an için en çok tercih edilen "Raspbian" isimli Linux dağıtımdır. Raspbian üzerine IDE'ler, editörler, server programlar kurulabilir. Böylece Raspberry Pi ucuz bir PC olarak kullanılabilmektedir. Bu tür embedded Linux kitleri özellikle IO amaçlı endüstriyel ortamlarda kullanılmaktadır. Yani bu kitlere kameralar, sensörler, step motorlar vs. bağlanarak bunlar başka amaçları gerçekleştirmek için kullanılabilir.

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 modelinde senkronizasyon da otomatik sağlanmaktadır. Bu bakımından kullanılması kolay bir yöntemdir. Boru haberleşmeleri her iki sistemde de "İsimli" ve "İsimsiz (anonim de denir)" olmak üzere ikiye ayrılmaktadır.

Boru Haberleşmeleri



Borular UNIX/Linux sistemlerinde tek kanallıdır. Yani bir taraf yazar, diğer taraf okur (half duplex). Fakat tek bir boruya 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 gereklidir.

İsimsiz borular yalnızca üst ve alt prosesler arasında haberleşmede kullanılmaktadır. İsimli borular ise aynı makine içerisindeki herhangi iki proses arasındaki haberleşmede kullanılabilmektedir.

Tipik bir boru haberleşmesi şöyle yapılır:

- 1) Önce boru yaratılır.
- 2) Proseslerden biri boruya bilgiyi yazar. Genellikle borular işletim sistemlerinde sanki birer dosyayı 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 geçirebilmesi) 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 olur. (Bloke (blocking) ileride başka konularda ele alınacaktır. Bloke olmak ilgili akışın donarak bekletilmesi anlamına gelmektedir.) Ta ki karşı taraf okuyup boruda yer açılmışa kadar. Benzer biçimde okuyan taraf borudakilerin hepsini okumuşa ve yeniden boradan okuma yapmak isterse yine bloke olur. 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 kaptığını 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) moda yapılmaktadır. Biz burada default durum olan blokeli moda boru haberleşmeleri üzerinde duracağız.

UNIX/Linux Sistemlerinde Isimsiz Boru Haberleşmeleri

Yukarıda da belirtildiği gibi isimsiz boru haberleşmeleri üst ve alt prosesler arasında yapılmaktadır.

Blokeli moda 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 içeresine 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 yazılabılır:

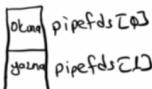
```
int pipe(int *pipefd);
```

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 içeresine 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 prosesi oluşturur. Böylece üst prosesin borusu betimleyicileri alt prosese 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 prosese aktarılmaktadır. Yani örneğin üst proses bir dosyayı açmışsa fork işleminden sonra alt proses de bu dosyayı açık görmektedir. İlgili betimleyici her iki proseste de aynı dosyaya yazmak ve okumak amacıyla kullanılabilmektedir. Bu konunun ayrıntıları "UNIX/Linux Sistem Programlama" kurslarında ele alınmaktadır. İsimli boru haberleşmesinde de üst proses borunu 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 yazılmak üzere 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 doluya 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 n byte'ın hepsini okuyana kadar blokede kalır. Yani örneğin biz borudan tek bir read fonksiyonuyla 100 byte okumak isteyelim. Ancak boruda 80 byte bulunuyor olsun. read fonksiyonu bu 80 byte'ı okuyup 80 değeri ile geri dönmez. 100 byte'ını hepsini okumak için bekler. Boruya bir 20 byte daha geldiğinde artık bu 100 byte'in tamamını okuyarak blokesi çözülür.

- Birden fazla kaynak aynı boruya yazma yaparken normalde iç içe geçme oluşmaz. Ancak borunun bir uzunluğu vardır (bu uzunluk <limts.h> içerisindeki PIPE_BUF sembolik sabiti ile programcıya verilmektedir). Eğer boruya write fonksiyonu ile boru uzunluğundan daha fazla miktarda byte yazılmasına çalışılırsa bu durumda iç içe geçme oluşabilemektedir. Yazma sırasında da tüm byte'lar yazılına kadar bloke çözülmeyecektir. Programcı borunun uzunluğunu değiştiremez ancak PIPE_BUF sembolik sabiti yoluyla bu uzunluğu elde edebilir.

Örnek bir program şöyle yazılabılır:

```
#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);
}

```

Pekiyi boruya yazmak ve borudan okuma yapmak için yalnızca birer betimleyicinin bırakılmasının anlamı nedir? Anımsanacağı gibi isimsiz boru haberleşmesinde boruya yanan 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 dosyayı gibi işleme sokulmakla birlikte aslında birincil bellekte tahsis edilmiş olan bölgelerdir.

Kabuk Boru İşlemi Nasıl Yapıyor?

Önceki konularda da görüldüğü gibi kabuk üzerinden,

a | b

gibi bir işlemde a'nın stdout dosyasına yazdıklarını b stdin dosyasından okumaktadır. Pekiyi 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 okuduğu 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 yazar.

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. Tabii aslında "b" "a"nın boruya yazdıklarını okur.

4) Kabuk her iki prosesin de bitmesini wait fonksiyonları ile bekler.

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 | simbolünü tırnak içerisinde almayı unutmayın. Eğer bu simbolü tırnak içerisinde almazsanız kabuk bunu komut satırı argümanı değil kendi boru simbolü 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 içerisinde almayı unutmayın.

```

#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 kullanılabilmektedir. 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 dizin girişlerinde 'p' dosya özelliğiyle görüntülenirler. Tabii boru dosyaları gerçek anlamda diskte yer kaplamazlar. Onların yalnızca bir dizin girişleri 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 dizin 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 gereklidir. 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, başarısız durumunda -1 değerine geri döner. Eğer birinci parametreyle belirtilen isimde dizin gibi 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 işlevlere 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 Ago 19 11:13 testpipe _
```

mkfifo isimli kabuk komutu mkfifo POSIX fonksiyonu kullanılarak şöyle yazılabılır:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.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 dosyamış gibi open POSIX fonksiyonuyla iki proses tarafından da açılır. (Ya da istenirse bunun için fopen standart C fonksiyonu da kullanılabilir. Ancak fopen kullanılaraksa dosya tamponlama stratejisinin "sıfır tamponlamalı moda" çekilmesi uygun olur.) open fonksiyonunda açış işlemini bir proses yalnızca okuma modunda diğer de yalnızca yazma modunda yapmalıdır. Yani isimlisiz 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 istediginde, digeri yazma modunda açana kadar open blokede kalmaktadir. Benzer bicimde bir proses boruyu yazma modunda açmak istediginde digeri okuma modunda açana kadar yine open blokede kalir.

3) Artik yazan taraf write fonksiyonuyla boruya yazma yapar. Okuyan taraf da read fonksiyonuyla borudan okur. Tabii yine boru doldugunda yazan taraf, boru bosaldiginda da okuyan taraf blokede kalir. Yani isimli borunun davranisi tamamen isimsiz boruda oldugu gibidir.

4) Yine once yazan tarafin boruyu kapatması gereklidir. Daha sonra karşı taraf read fonksiyonuyla sıfır byte okuduğunda o da boruyu kapatır.

Örnek bir program şöyle yazılabılır:

```

/* 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 kaldırınmak 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 Agu 19 11:45 process1
-rw-r--r-- 1 csd study 488 Agu 19 11:42 process1.c
-rwxr-xr-x 1 csd study 9104 Agu 19 11:45 process2
-rw-r--r-- 1 csd study 570 Agu 19 11:45 process2.c
prw-r--r-- 1 csd study 0 Agu 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 yapılmamaktadır. Blokesiz modda boru işlemleri için açış modunda O_NONBLOCK bayrağı kullanılmaktadır. Eğer boru isimsiz 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 farklılıklarını 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 İsimsiz Boru Haberleşmesi

Windows'ta isimsiz boru haberleşmeleri de UNIX/Linux sistemlerindekilerine çok benzer yapılmaktadır. İsimsiz borular Windows'ta CreatePipe API fonksiyonuyla yaratılır:

```
BOOL WINAPI CreatePipe(  
    PHANDLE hReadPipe,  
    PHANDLE hWritePipe,  
    LPSECURITY_ATTRIBUTES lpPipeAttributes,  
    DWORD nSize  
);
```

Fonksiyonun birinci ve ikinci parametreleri borudan okumak ve boruya yazmak için gereken handle nesnelerinin adreslerini alır. Fonksiyon buraya okuma ve yazma için gereken handle değerlerini yerleştirir. Üçüncü parametre kernel nesnelerinin güvenlik parametresidir. NULL geçilebilir. Son parametre ise yaratılacak borunun byte cinsinden uzunluğudur. 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 gerekmeyorsa diğer handle'in (örneğin üst proses yazma yapıyorsa bu yazma handle'inin) geçirilmesine gerek yoktur. CreateProcess'teki beşinci parametre "ana şalter" görevindedir. Ancak bu parametre TRUE yapılsa bile default olarak kernel nesnelerinin 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.

İsimsiz 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 gereklidir. 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. Bu sistemlerde CreateProcess adeta UNIX/Linux sistemlerindeki fork/exec işlemine karşılık gelir. Bu durumda alt proses üst proseden aktarılan boru handle değerini bilemez. O halde bu handle değerinin de alt prosese aktarılması gereklidir. Bu aktarım komut satırı argümanları yoluyla ya da çevre değişkenleri yoluyla yapılabilir.

Windows'ta da üst prosesin handle değerlerinin alt prosese aktarılması benzer bir 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 prosese geçirilmektedir.

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, TRUE))
        ExitSys("SetHandleInformation");

    sprintf(cmdLine, "ChildProcess.exe %lu", 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], "%lu", &hReadPipe);

    while (ReadFile(hReadPipe, &val, sizeof(int), &dwRead, NULL) && dwRead != 0)
        printf("%d ", val), fflush(stdout);
    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);
}

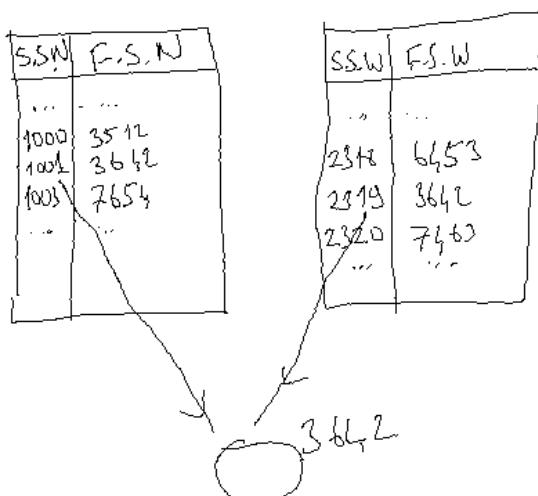
```

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 içerisindeindedir.

Paylaşılan Bellek Alanları (Shared Memory) Yöntemi ile Proseslerarası Haberleşme

Anımsanacağı gibi modern sistemlerde işlemcinin sağladığı bir sayafalama 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 yazmalığında diğer proses diğer sanal sayfayı içeren bir adresinden bunları okuyabilir. Başka bir deyişle iki proses sanki farklı sanal adreslerle aynı fiziksel adres'e 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 paylaşan bellek alanına yazdığında diğerinin onun yazdığını ne zaman alacaktır? Diğerini yeni birşey yazarsa eskisi ezilmez mi? İşte senkronizasyon için üretici-tüketiciliği (producer-consumer problem) gibi bir yöntemin kullanılması gereklidir.

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 paylaşan bellek alanları şu adımlardan geçirilerek oluşturulur:

1) Öncelikle her iki proseste de bir "file mapping" nesnesinin oluşturulması gereklidir. 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 paylaşan 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. Fonksiyon 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 değerin sayfa katlarında olması anlamlıdır. Son parametre proseslerarası paylaşım için gereken isimdir. Bu isim herhangi bir biçimde verilebilir. İşletim sistemi birden fazla processin 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 proseste de çağrılmalıdır. İlk çağrıran proses nesneyi yaratmış olur. Artık diğer çağrırandaki parametreler etkili olmaz. İkinci çağrıran proses yalnızca nesneyi açmış olur. Dolayısıyla fonksiyonu sonradan çağrıran kişi için üçüncü, dördüncü, beşinci parametrelerin bir önemi yoktur. Tabii iki proseste de aynı değerler ve isimler kullanılırsa bu proseslerin hangisinin önce çalıştığını bir önemi kalmayacaktır. Eğer istenirse proseslerden biri CreateFileMapping fonksiyonuyla nesneyi yaratırken diğer 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 paylaşan 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 yapabileceğini hem de oradan okuma yapabileceğini 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. Fonksiyon 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 geçmemeliyiz. Son parametre paylaşılacak bellek alanının büyülüklüğüdür. Mapping nesnesinde belirtilen büyülüklük maksimum büyülüktür. Buradaki değer ondan küçük olabilir ya da ona eşit olabilir. Ancak ondan büyük olamaz. Fonksiyon başarı durumunda paylaşılan bellek alanına erişmeye çalışan adresi, başarısızlık durumda NULL adrese geri döner.

3) Kullanım bitince paylaşılan bellek alanının sisteme iade edilmesi gereklidir. 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 yapılmaktadır. Yazma işlemi aslında doğrudan belleğe yapılır. Ancak işletim sistemi yapılan değişiklikleri dosyaya kendisi yansımaktadır. 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 açılabilmektektir. 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 arasındaki belleğe çekileceğini belirtiriz.
- 4) Dosya işlemleri bittiğinde UnMapViewOfFile fonksiyonu ile dosya için bellekte tahsis edilen alan serbest bırakılır. Sonra sırasıyla "file mapping" ve "file" nesneleri 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 güncellemeye dosyanın başına "Ankara" yazısı yazdırılmaktadır.

Pekiyi 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 yazılımsal yükü vardır. (Örneğin okunacak yer için dosya göstericisinin konumlandırılması ve her okumanın başarısının kontrol edilmesi gereklidir.) 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 gerçekleştirilebilmesini sağlamaktadır. Örneğin bir bmp dosyasının üzerinde değişikler yapmak isteyelim. Dosayı 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 çıkış 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 geçirilerek 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 benzeturdir:

```
#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 alanın proseslerarası kullanımı için gereken ismidir. Bu ismin kök dizinde sanki bir dosya ismiymiş gibi verilmesi gerekmektedir. (Tabii aslında böyle bir dosya yoktur. Dizin girişinde de görülmeyecektir.) 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 yaratırken 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 gereklidir. `ftruncate` fonksiyonu asıl olarak dosyayı küçütmek ya da büyütmek 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 alanın uzunluğunu belirlemek için de kullanılmaktadır. Siz `shm_open` ile yeni yaratılan paylaşılan bellek alanı dosyasının başlangıçta 0 uzunlukta olduğunu düşünübilirsiniz. İşte `ftruncate` fonksiyon 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 prosese 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 benzeturdir. 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 uygunsa 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 truncate 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 bayrağı 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	Paylaşilan 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 değerile 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 bölgelerini okuduğumuzda o uçlardaki gerilim değerini okumuş olabiliriz. Bu teknigue "bellek tabanlı IO teknigi (mempry mapped IO)" denilmektedir. Örneğin Raspberry PI bu teknigue 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 çağrıması 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 blokede 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 <sys/types.h>
#include <fcntl.h>
#include <unistd.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 Kuyrukları (MessageQueue) Yöntemi

Bu proseslerarası haberleşme yönteminde ismine mesaj kuyruğu (message queue) denilen bir kuyruk sistemi yaratılır. Bir proses bu kuyruğa yazma yaparken diğer okuma yapar. Kuyruk sistemi FIFO biçimdedir. Bu nedenle haberleşme biçim bakımından boru haberleşmesine benzemektedir. Ancak mesaj kuyruklarında paket tarzı bir aktarım vardır. Bir taraf mesaj adı altında bir paket bilgi gönderir. Diğer 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 kuyrukları paket tabanlı (datagram) bir haberleşme sunmaktadır. Yani boru haberleşmesi bu bakımdan IP ailesindeki "tcp" protokolüne benzetilirse, mesaj kuyrukları "udp" protokolüne benzetilebilir.

Mesaj kuyrukları Windows işletim sisteminde GUI alt sistemindeki mesajlar yoluyla kullanılabilmektedir. Zaten Windows'un GUI alt sistemi tamamen mesaj tabanlı bir sistemdir. Mesaj kuyrukları sistemi daha çok Windows'tan ziyade UNIX/Linux sistemlerinde daha popülerdir.

UNIX/Linux Sistemlerinde Mesaj Kuyruklarının Kullanımı

Mesaj kuyrukları için de tıpkı paylaşılan bellek alanlarında olduğu gibi iki farklı fonksiyon grubu kullanılabilmektedir. Bunlardan biri klasik Sistem 5 mesaj kuyruklarıdır. Diğerisi de daha modern POSIX mesaj kuyruklarıdır. (Yukarıda da belirtildiği gibi her ne kadar isimleri böyle anılıyorsa da aslında her iki fonksiyon grubu da POSIX standartlarında tanımlanmışlardır.) POSIX mesaj kuyruklarının 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 kuyruklarını kullanmaktadır. Kursumuzda modern POSIX mesaj kuyruklarını göreceğiz. Klasik Sistem 5 mesaj kuyrukları "UNIX/Linux Sistem Programlama" kursunda ele alınmaktadır. POSIX mesaj kuyruklarını 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 çağrılığında 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_getsetattr fonksiyonuyla elde edilmiş mq_setsetattr fonksiyonuyla yeniden değiştirilebilmektedir.

mq_open fonksiyonu eğer mesaj kuyruğu nesni sıfırdan yaratılmayacak var olan açılacaksa iki parametrelidir 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 <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...);
```

`mq_open` fonksiyonu başarı durumunda kuyruğa ilişkin bir handle değeri ile başarısızlık durumunda -1 değerileyile 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`, kuyrukta mesajı almak için `mq_receive` fonksiyonları kullanılır. `mq_send` fonksiyonun prototipi şöyledir:

```
#include <mqueue.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 dönmemektedir. Fonksiyonun ikinci parametresinin `const char *` türünden olması size yanlış bir izlenmim 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 gereklidir. 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 dönmemektedir.

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 nesneleri silinmemişlerse onları yeniden `mq_open` fonksiyonu ile açıp kullanabiliriz.

4) Tíkpi paylaþılan bellek alanı nesnelerinde olduğu gibi mesaj kuyrukları da sistem kapatılana kadar yaratılmış biçimde kalmaktadır. Sistem reboot edildiðinde bu nesneler silinmiş durumda olur. Ancak biz reboot işleminden önce de mesaj kuyruklarını `mq_unlink` fonksiyonuyla yok edebiliriz. Fonksiyonun prototipi şöyledir:

```
#include <mqueue.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 <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 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(msqid, 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(msqid);

if (mq_unlink("/ThisIsMyTestMessageQueue") == -1)
    exit_sys("mq_unlink");

return 0;
}

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

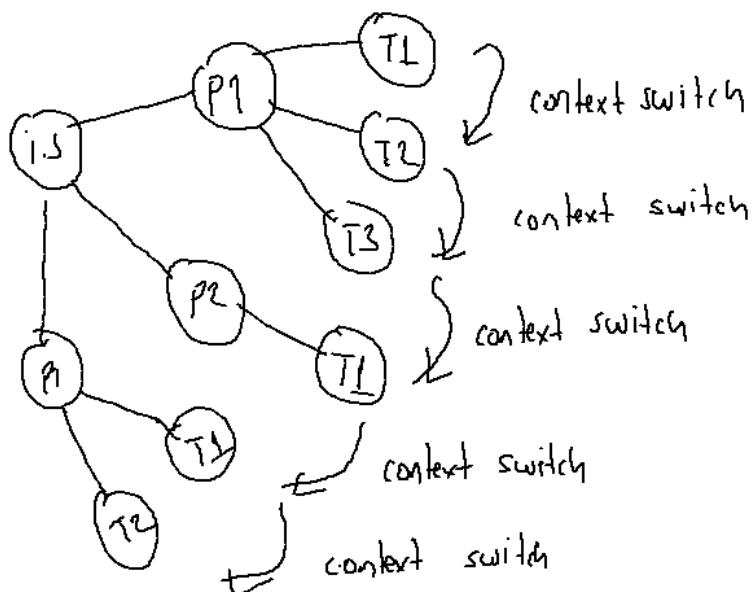
    exit(EXIT_FAILURE);
}

```

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'lı 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'lı işletim ssistemlerinde artık çizelgeleyici için prosesler değil, thread'ler çizelgelenmektedir. Bu durumda thread'sız sistemler tek thread'lı sistemler gibi düşünülebilir.

Çok thread'lı 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 akış çağrıstan birkaç terim sıkılıkla birbirlerine karıştırılabilir. "Concurrency" terimi genel bir terimdir. Aynı anda birden fazla akışın bulunduğu durumları anlatmaktadır. Bu terim genelleşmiş olarak "concurrent computing" biçiminde de kullanılabilmektedir. Buradaki birden fazla akışın iç içe geçmesi ve birekliği 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 içersindedir. "Parallel Programming" işlerin thread'lere ayrılarak aynı makinede farklı CPU ya da çekirdeğe atanması yoluyla eş zamanlı çalışma gereği için kullanılmaktadır. "Distributed Computing" ya da "Distributed Programming" terimi ise bir işin network içerisinde farklı makinelere 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 saatı basmak isteyelim. Saati ne zaman basacağımız. Her işlemin arasında saatı basmamız gereklidir. 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) izleyeceğimiz olalı. 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 olayları izlenmeye devam edecektir.
- 2) Thread'ler bir programı hızlandırmak için kullanılabilirmektedir. 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 etkilendir.
- 4) Thread'ler paralel programlama için mecburen kullanılmaktadır. Paralel programlama bir işi parçalara ayırarak onu aynı anda birden fazla işlemci ya da çekirdeğe atayarak gerçekleştirmeye 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 çağırılan 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 sisteminin yukarıda belirttiğimiz temel thread fonksiyonlarını çağırmaktadır. 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 yatailmaktadır:

```

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 geçirilirse 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);
```

Göründüğü gibi thread fonksiyonunun geri dönüş değeri DWORD parametresi LPVOID (yani void *) türünden olmak zorundadır. Ayrıca thread fonksiyonları __stdcall çağrıma biçimine sahip zorundadır.

Yani örneğin aşağıdaki fonksiyon bir thread fonksiyonu olabilir:

```

DWORD __stdcall ThreadProc(void *param)
{
    ...
}

```

CreateThread fonksiyonunun dördüncü parametresi thread yaratıldığından thread'e geçirilecek argümanı belirtir. Beşinci parametre yaratıma ilişkin bazı özellikleri belirtmektedir. Bu parametre sıfır geçilebilir ya da CREATE_SUSPENDED biçiminde geçirilebilir. Bu durumda thread yaratıldığından henüz çalışmıyor durumda olur. Onu çalıştırmak için ResumeThread API fonksiyonunun çağrılması gereklidir. Fakat bu parametre sıfır geçirilirse 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 setderr dosyasına yazdırıramayız. Biz therad fonksiyonlarının veridiği 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 kütüphanedir. Linker default durumda yalnızca libc kütüphanesine baktığı için thread fonksiyonlarını bulamaz. Bu nedenle derlemeyi yaparken linker'in phread kütüphanesine bakmasını sağlamak gereklidir. Bu işlem -lpthread seçeneğiyle yapılmalıdır. 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-üslük ilişkisi yoktur. Yani bir thread herhangi bir thread akışı tarafından yaratılabilir. Thread'in hangi thread akışı tarafından yaratıldığıının 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 çizgelenirler.

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 tipki 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 türyle, UNIX/Linux sistemlerinde de void * türyle 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 çağrıları yapan pek çok POSIX fonksiyonunun birer sonlandırma noktası belirttiğini söyleyebiliriz. TerminateThread API fonksiyonun prototipi şöyledir:

```

BOOL WINAPI 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 hiçbirini exit fonksiyonu çağrılmamışsa 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 sonlandırılıyor */

    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 artırmış olurdu. Çünkü global ve static yerel nesneler "data veya bss" bölmelerinde yaratılmaktadır. Bu bölgeler prosese özgüdür. Tüm thread'ler aynı data ve bss bölümünü ortak kullanırlar. Benzer biçimde prosesin heap alanı da toplamda bir taneidir. 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 yaptığından diğer thread'ler onu hemen değişmiş olarak görürler.

Aşağıdaki örnekte iki thread (ana thread ve yeni yaratılan thread) aynı Foo fonksiyonunu çağrırmıştır. 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 olmadığından 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 nesneler ya da heap yoluyla yapılabilir. Global nesneler 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 Sonlanması 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 dolduguunda 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 çağırılan thread'i blokede bekletir. Fonksiyonun geri dönüş değeri ilerde 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);
}

```

pthread_join 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 <pthread.h>

void *thread_proc(void *param);
```

```

int main(void)
{
    pthread_t tid;
    int result;
    void *exitVal;

    if ((result = pthread_create(&tid, NULL, thread_proc, NULL)) != 0) {
        printf("pthread_create: %s\n", strerror(result));
        exit(EXIT_FAILURE);
    }

    pthread_join(tid, &exitVal);

    printf("Ok: %ld\n", (long)(intptr_t)exitVal);

    return 0;
}

void *thread_proc(void *param)
{
    int i;
    intptr_t exitVal = 123;

    for (i = 0; i < 10; ++i) {
        printf("New thread: %d\n", i);
        sleep(1);
    }

    return (void *)exitVal;
}

```

Windows sistemlerindeki WaitForSingleObject, UNIX/Linux sistemlerindeki pthread_join fonksiyonun bekleme işlemini ilgili thread'i bloke ederek CPU zamanı harcamadına yaptığına dikkat ediniz. Bu işlem global bir değişken yoluyla meşgul bir döngüyle de yapılabildirdi. 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'in 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 açılmaktedir. Çünkü bu sistemlerde bir thread'in başka bir thread'in stack'ine erişmesi problemler bir konusudur. 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 gerekmektedir. 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 bitiremeden kesilebilir. Diğer bir thread yarımla kalan iş ile ilgili birşeyler yaptığında tüm program biribirine girebilir, dolayısıyla çökebilir.

Kritik Kod (CriticalSection) 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 olabileceğ gibi dış dünyadaki donanımsal bir aygit 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 kaynağını kullanmak istediği sorun çıkar. Örneğin bir thread'in ortak kullanılan bir bağlı listeye ekleme yaptığıni 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 ++ operatöryle 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 değildirler. Örneğin bu işlemi aşağıdaki gibi üç makine komutuyla yapabilirler:

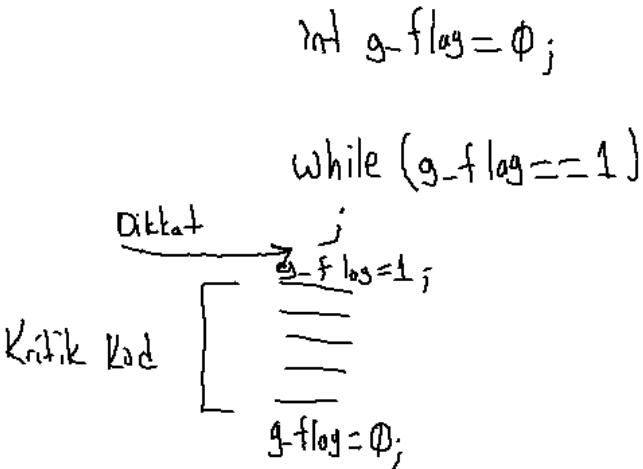
```

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 oluşursa birden fazla thread kritik koda girebilir.

Peki bu işlem güvenli bir biçimde nasıl yapılabilir? Aslında yine bir bayrak tutulabilir. Ancak kontrol ve set işlemi yapılrıken thread'ler arası geçiş kapatılabilir. Böylece bayrak güvenli olarak set edilebilir. Fakat thread'ler arası geçisin 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 nesneleri" 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 yapılrıken kernel moda geçiş engellenebilmektedir. Fakat ne olursa olsun bu işi yapan fonksiyonların özel bir biçimde yazılması gereklidir. İşte işletim sistemleri ya da temel kütüphaneler bunları yapan hazır fonksiyonlar bulundurmaktadır.

Windows'ta Kritik Kodların Oluşturulması

Windows'ta kritik kod oluşturmak için `CriticalSection` nesnelerinden, `mutex` ya da `semaphore` nesnelerinden faydalaniılmaktadır. Burada öncelikle nesnesinin kullanımını göreceğiz. `CriticalSection` nesnesi işlemlerini user modda yapmaktadır. `CriticalSection` nesnesinin kullanımı şöyledir:

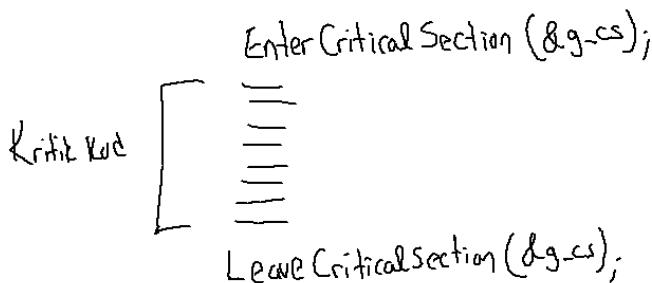
- 1) `CRITICAL_SECTION` isimli yapı türünden global bir nesne yaratılır.

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 çalıştırılmış 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 açıldığından 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 geçersek 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);
}

```

Örneğin C++'ta iki thread dinamik büyüyen bir diziye (vector) ekleme yapacak olsun. Ekleme işleminin yanında kesilmemesi gereklidir. 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'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 kapalısa 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 sonlanmaşısa bu nesneler 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 aşamından dolayı sonlanmışsa WAIT_TIMEOUT özel değerine geri döner. (Ancak "mutex" nesnelerinde mutex'e sahip thread sonlanmışsa fonksiyon WAIT_ABANDONED özel değeriyle geri dönmemektedir.) Eğer nesne açık duruma geldiğinden dolayı fonksiyon geri dönmişse WAIT_OBJECT_0 değeri elde edilir. Fonksiyon başarısız olabilir. Başarısızlık durumunda WAIT_FAILED değerine geri döner.

WaitForMultipleObjects fonksiyonu birden fazla nesneyi beklemek 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_ABONDENT_0 + n biçimindedir.) Buradaki n yine dizide indeks belirtmektedir. Fonksiyon başarısız olursa yine WAIT_FAILED değerine geri döner.

Şimdiye kadar biz yalnızca iki tane kernel senkronizasyon nesnesi gördük. Onlar proses ve thread nesneleriydi. 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 nesneleridir.

Windows Sistemlerinde Kritik Kodların Mutex Nesneleriyle Oluşturulması

Windows'ta mutex nesneleri kritik kod bloklarını oluşturmak için kullanılan kernel senkronizasyon nesneleridir. Burada "kernel senkronizasyon nesnesi" terimi "kernel moda geçiş yaparak çalışma" anlamına gelmektedir. Aslında Windows'ta mutex nesneleri CriticalSection nesneleriyle aynı amaçla kullanılırlar. Ancak CriticalSection nesneleri kernel moda geçmeden tamamen user modda çalışmaktadır. Bu nedenle CriticalSection nesnesi mutex nesnesinden daha hızlı çalışma eğilimindedir. Pekiyi Windows'ta CriticalSection nesnesi varken biz neden mutex nesnelerini tercih edebiliriz? İşte Winmdows'ta kernel senkronizasyon nesneleri isimli nesnelerdir. Biz onları onları istersek farklı proseslerin threadleri arasında da senkronizasyon amacıyla kullanabiliriz. Halbuki CriticalSection nesneleri yalnızca aynı prosesin thread'leri arasında kullanılabilmektektir. Ayrıca mutex nesneleri ilerde de ele aldığı 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 nesneleriyle oluşturulması 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 nesnelerinin thread temelinde bir sahipliği vardır. Bir mutex nesnesinin sahipliği bir thread tarafından alınır. Sahiplik ancak onu alan thread tarafından bırakılabilir. Mutex nesnelerinin eğer sahipliği bir thread tarafından alınmışsa nesne kapalı durumdadır. Eğer bunların sahipliği bir thread tarafından alınmamışsa nesne açık durumdadır.

Mutex nesneleri şöyle kullanılmaktadır:

- 1) Mutex nesnesi CreateMutex API fonksiyonuyla yaratılır. Proses içi kullanımında mutex'in handle değeri global bir değişkene 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 belirlemekte kullanılır. Eğer bu parametre sıfır dışı (TRUE) geçilirse CreateMutex'i çağrıran thread nesnenin sahipliğini de işin başında almış olur. Sıfır (FALSE) geçilirse nesne sahipliği alınmadan yaratılır. Son parametre proseslerarası kullanımındaki 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
Kd [====]
ReleaseMutex(g_hMutex);

```

Mutex nesnesinin sahipliği bir thread tarafından alınmışsa nesne kapalı (nonsignaled) durumdadır. Nesne kapalıken WaitForSingleObject fonksiyonu blokede bekler. Nesnenin sahipliği onu almış olan thread tarafından bırakıldığından 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 nesneleri 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 terkedilmiş mutex'ler (abondoned 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 çağırılmış olalım. Bar fonksiyonu başka yerden de çağrılabileceği 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 sistemlerine mutex nesneleri proseslerarası kullanılmayacaksızın kernel moda geçmez. Yani UNIX/Linux sistemlerindeki mutex nesneleri performans bakımından Windows sistemlerindekiCriticalSection 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 fonksiyonuyla 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:

Kritik kod [*pthread_mutex_lock(&g_mutex);*
 [
 [
 [
 pthread_mutex_unlock(&g_mutex);

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 nesneleri özyinelemeli değildir. Yani bir thread mutex nesnesinin sahipliğini aldıktan sonra yeniden almaya çalışırsa kendi kendini kilitler. (Windows'ta mutex nesnelerinin özyinelemeli olduğunu anımsayınız.) Ancak biz istersek UNIX/Linux sistemlerinde mutex nesnesini özyinelemeli hale getirebiliriz. Bunun için pthread_mutex_init fonksiyonunun ikinci parametresi olan mutex özniteliklerinin uygun biçimde set edilmesi gereklidir. Bu set işlemi için önce pthread_mutexattr_t türünden bir nesne alınıp bu nesne pthread_mutexattr_init fonksiyonuyla ilkdeğerlenir. Sonra da özyinelemeli mutex bu öznitelik 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 nesneleri için ne kadar sayıda pthread_mutex_lock fonksiyonu uygulanmışsa nesne açmak için o sayıda pthread_mutex_unlock fonksiyonu uygulanmalıdır.

Semaphore Nesneleri

Trenlerdeki dur geç lambalarına semaphore denilmektedir. Semaphore'lar sayaçlı senkronizasyon nesneleridir. Bir kritik koda en fazla n tane akışın girebilmesi için kullanılırlar. Semaforun bir sayacı vardır. Bu sayaç sıfırdan büyüğe 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 çıkışında 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 gereklidir. 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 paylaşımak amacıyla kullanılmaktadır.

Semafor sayacı başlangıçta 1 ise böyle semaphore'lara ikili (binary) semaphore denilmektedir. İkili semaphore'lar mutex nesnelerine 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 nesneleri özellikle üretici-tüketicileri (producer-consumer) probleminin çözümü için kullanılmaktadır.

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 diğerini 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 simülle 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 bazlarını ise birden fazla kez aldığına dikkat ediniz.

İşte üretici-tüketici problemleri tipik olarak semaphore nesneleri kullanılarak gerçekleştirilir.

Windows Sistemlerinde Semaphore Nesnelerinin Kullanılması

Windows sistemlerinde semaphore nesneleri şöyle kullanılır:

1) CreateSemaphore fonksiyonuyla semaphore yaratılır ve handle değeri global bir değişkenin içerisinde 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 değeridir. Programcı bu değeri kritik kodda paylaşılacak kaynak sayısını referans alarak girer. Üçüncü parametre semaphore sayacının çıkacağı maksimum değeri belirtir. Genellikle ikinci ve üçüncü parametreler aynı değer girilerek oluşturulmaktadır. Son parametre proseslerarası kullanım için gereken ismi belirtir. Bu parametre proses içi kullanımında NULL geçilebilir. Fonksiyon başarı durumunda semaphore nesnesinin handle değerine, başarısızlık durumunda NULL adrese geri döner.

2) Kritik kod şöyle oluşturulur:

Kritik Kod {
 WaitForSingleObject(g_hSem, INFINITE);
 ReleaseSemaphore(g_hSem, 1, NULL);
}

Bir semaphore nesnesi sayı 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 sayacını 1 eksiltmektedir. Başka bir deyişle akış WaitForSingleObject fonksiyonunu her geçtiğinde semaphore sayacı 1 eksiltilmiş olur. ReleaseSemaphore fonksiyonu ise semaphore sayacını artırmaktadır. ReleaseSemaphore fonksiyonun prototipi şöyleledir:

```
BOOL WINAPI ReleaseSemaphore(  
    HANDLE hSemaphore,  
    LONG LReleaseCount,  
    LPVOID LpPreviousCount  
) ;
```

Fonksiyonun birinci parametresi semaphore'un handle değerini, ikinci parametresi sayacın artırılacak değerini 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 başlangıçtakş 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)
{
    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);
}

```

Örnek çıktıının bir bölümü şöyledir:

```

0 0 0 0 0 0 0 0 1 2 0 1 2 0 1 2 1 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 1 0 0 1 0 0 1 0 0
0 1 1 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0
0 0 1 2 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 2
0 1 2 1 2 0 1 1 2 0 1 1 2 1 0 1 0 0 0 0 1 0 0 0 ....

```

UNIX/Linux Sistemlerinde Semaphore Nesnelerinin Kullanımı

UNIX/Linux sistemlerinde semaphore işlemleri iki grub POSIX fonksiyonuyla yapılmaktadır. Bunlardan biri eskiden beri kullanılan klasik System 5 semaphore nesneleridir. Bunların parametrik yapıları oldukça karışık ve kullanımı da oldukça zordur. 90'lı yıllarda thread'lerle birlikte yeni modern semaphore fonksiyonları da oluşturulmuştur. Bunlara POSIX semaphore'ları denilmektedir. Halbuki her iki fonksiyon grubu da POSIX standartlarında yer almaktadır. Biz burada modern POSIX semaphore nesnelerinin kullanımını göreceğiz:

1) Önce sem_t türünden global bir nesne tanımlanır. Sonra sem_init fonksiyonuyla bu semaphore nesnesi initialize edilir.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Fonksiyonun birinci parametresi semaphore nesnesinin adresini alır. İkinci parametre proseslerarası paylaşım ile ilgilidir. Nesne prosesler arasında paylaşılacaksa bu parametre sıfır dışı bir değer olarak, paylaşılmayacaksa sıfır olarak geçilir. Üçüncü parametre başlangıçtaki semaphore sayacını belirtir. sem_init fonksiyonu ile semaphore nesnesini proseslerarası paylaşmak işlem yükü gerektirmektedir. Çünkü bunun için semaphore nesnesinin (sem_t nesnesinin) paylaşılan bellek alanı içerisinde oluşturulması gerekmektedir. sem_init fonksiyonu başarı durumunda 0 değerine başarısızlık durumunda ise -1 değerine geri döner. Bu fonksiyon errno değişkenini set etmektedir.

POSIX semaphore'larında tipki diğer nesnelerde olduğu gibi proseslerarası paylaşım için semaphore nesnelerine isim de verilebilmektedir. sem_open isimli POSIX fonksiyonu proseslerarası kullanım için isimli bir semaphore nesnesi yaratmaktadır:

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

Fonsiyon iki parametreli olarak ya da dört parametreli olarak kullanılabilmektedir. Yani aslında fonksiyonun gerçek prototipi şöyledir:

```
sem_t *sem_open(const char *name, int oflag, ...);
```

Fonksiyonun birinci parametresi semaphore nesnesinin ismini belirtir. Bu isim yine kök dizinde bir dosya ismi gibi oluşturulmalıdır. İkinci parametre açış modunu belirtmektedir. Bu parametre 0 olarak girilirse read/write işlem anlaşılır. O_CREAT bayrağı açış moduna eklenebilir. Bu durumda semahore yoksa yaratılmakta varsa olan açılmaktadır. Bu 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 girilirse 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 değişkenini set etmektedir.

2) Kritik kod şöyle oluşturulur:

```

    sem_wait(&g_sem);
    [ === ]
    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 değişkenini 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 aşımılı 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 semahore 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 Tüketicisi Probleminin Semaphore Nesneleriyle Çözümü

Semaphore'lar tipik olarak üretici-tüketicisi problemlerini çözmek için kullanılabilirler. 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;           // Başlangıç sayıci = 1
DATA_TYPE g_shared;             // Başlangıç sayıci = P
g_hSemProducer = CreateSemaphore(NULL, 1, 1, NULL); // Başlangıç sayıci = P
g_hSemConsumer = CreateSemaphore(NULL, 0, 1, NULL); // Başlangıç sayıci = 0

ÜRETİCİ
for(;;) {
    <bilgiyi et>
    WaitForSingleObject(g_hSemProducer, INFINITE);
    <bilgiyi paylaşılırken yerleştirir>
    ReleaseSemaphore(g_hSemConsumer, 1, NULL);
    3
}

TÜKETİCİ
for(;;) {
    <bilgiyi paylaşılırken alır>
    WaitForSingleObject(g_hSemConsumer, INFINITE);
    <bilgiyi tüketir>
    ReleaseSemaphore(g_hSemProducer, 1, NULL);
    3
}

```

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 yerlestirdikten sonra tüketicinin semaphore sayacını bir artırır. Böylece tüketici blokeden 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 blokeden 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 yazılabılır:

```

#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 Tamponlu Versiyonu

Üretici Tüketici probleminde ortadaki paylaşılan alanın tek bir elemandan oluşması yerine onun birden fazla elemandan oluşan bir tampon bölge olması sistemi hızlandırır. Çünkü bu durumda bekleme olasılığı azalacaktır. Şöyle ki: Bu

durumda üretici tampon tamamen doluyken, tüketici de tampon tamamen boşken bekleme yapar. Bu ortadaki paylaşan tampon kuyruk veri yapısıyla gerçekleştirilmelidir. 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 yazılabılır:

```

#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;
}

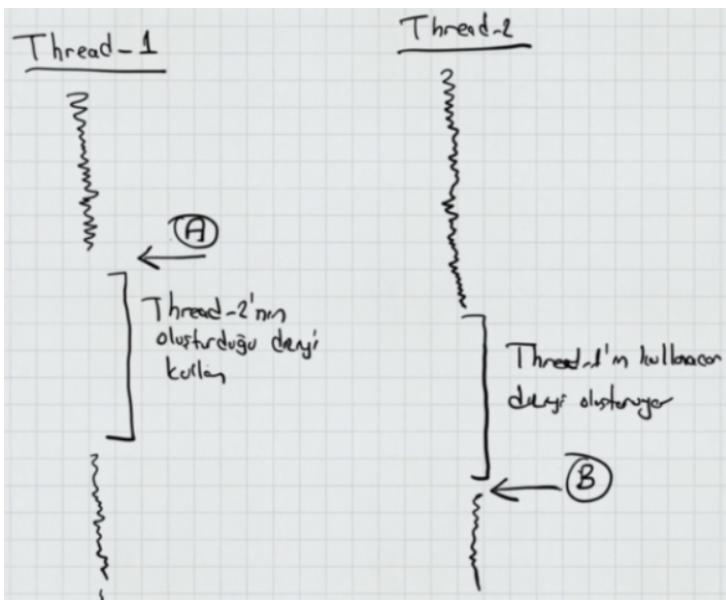
```

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. Ancak iletim sonlandığında bunların bazıları blokede bekler durumda kalabilir. Bunların o durumdan kurtarılması gereklidir. 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 isminden 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 gereklidir. Ta ki 2 Numaralı thread B noktasından geçene kadar. İşte event nesneleri bu tür amaçlarla kullanılmaktadır. Event nesnelerinin benzeri UNIX/Linux sistemlerinde "koşul değişkenleri (conditional variables)" ismiyle bulunmaktadır.

Event nesnelerinin kullanımı 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" mı yoksa "manuel" mı olacağını belirtmektedir. Bu parametre FALSE girilirse event nesnesi otomatik, TRUE girilirse manuel modda olur. Nesne açık duruma geçtiğinde WaitForXXX fonksiyonlarından geçiş yapıldığında nesne otomatik olarak yeniden kapalı duruma geçiyorsa bu moda otomatik mod denilmektedir. Manuel modda 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 prosesteki 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 manuellese 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 amacıyla erişliğini düşünelim. Burada şüphesiz kritik kod oluşturmak gereklidir. 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 bitirine kadar. Benzer biçimde bir thread bağlı listede arama yaparken diğer thread'ler bağlı listeye eleman eklemeye çalışmamalıdır. Taki arama işlemini yapan thread bu işlemi bitirene 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 istediğiinde bunu aynı anda yapamayacaklardır. İşye okuma yazma kilitleri bu sorunu şözmektedir. 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 ilklenir.

```
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 erişeceğe 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 erişeceğe bu durumda kilidi AcquireSRWLockExclusive fonksiyonuyla elde etmeye çalışmalıdır. Bu durumda diğer bir thread eğer kilidi AcquireSRWLockShared fonksiyonuyla ya da AcquireSRWLockExclusive fonksiyonuyla elde etmek istediğiinde 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 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 lere 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);
```

Göründüğü 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;
}

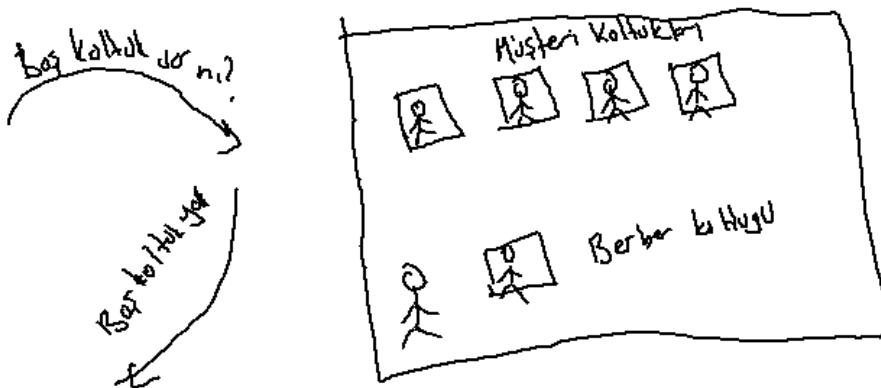
```

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 söylememez. Burada bunlardan kısaca bahsedeceğiz.

Uyuyan Berber Problemi (Sleeping Barber Problem)

Bu problemde bir berber dükkanı vardır. Dükkanın 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 istediginde koltuklara bakar. Koltuklar doluya 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 olmadığını anlaması gerekmektedir. Bunun için bir sayaç tutulur. Tabii sayaç da kritik kod bloğu ile azaltılıp yükseltilir. Peki de aynı 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 nesneleri 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("CreateEvent", EXIT_FAILURE);

if ((g_hEventShaving = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
    ExitSys("CreateEvent", 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("(%) Berber uyandi musteriyi koltuga davet etti\n", name);
SetEvent(g_hEventCustomer);

printf("Berber musteriyi tiras ediyor\n");
Sleep(rand() % 300);

printf("Berber tirasi 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("(%) musteri geldi ve dukkanı bakiyor\n", name);
Sleep(rand() % 300);

WaitForSingleObject(g_hMutexSeat, INFINITE);

if (g_emptySeatCount > 0) {
printf("(%) bos koltuga oturdu ve uyudu\n", name);
--g_emptySeatCount;
ReleaseMutex(g_hMutexSeat);

ReleaseSemaphore(g_hSemaphoreBarber, 1, NULL);

WaitForSingleObject(g_hEventCustomer, INFINITE);
printf("(%) musteri uyandi berber koltuguna oturdu\n", name);
WaitForSingleObject(g_hEventShaving, INFINITE);
printf("(%) musteri tirasini oldu, dukkanidan cikiyor\n", name);
}
else {
printf("(%) musteri dukkanı giremeden ayrılmıyor\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. Kuyruk içerisinde koltukta bekleyen müşterilere ilişkin event nesneleri 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 uyandi musteriyi koltuga davet etti\n", name);
        SetEvent(g_hEventCustomer);

        Sleep(rand() % 300);

        printf("Berber tirasi 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 musteri geldi ve dukkana bakiyor\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 musteri uyandi berber koltuguna oturdu ve tıraş oluyor...\n", name);
        WaitForSingleObject(g_hEventShaving, INFINITE);
        printf("(%)s musteri tirasini oldu, dukkandan cikiyor\n", name);
    }
    else {
        printf("(%)s musteri dukkana giremeden ayriliyor\n", name);
        ReleaseMutex(g_hMutexSeat);
    }

    return 0;
}

```

```

}

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

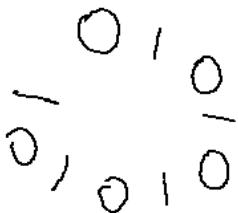
    if (FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM, NULL, dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&lpszErr, 0, NULL)) {
        fprintf(stderr, "%s: %s", lpszMsg, lpszErr);
        LocalFree(lpszErr);
    }

    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 yiyp 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 bazlarının aç kalma riski de vardır.

Problemin çözümü burada yapılmayacaktır. Ancak klasik çözümde kişiler iki senkronizasyon nesnesiyle (örneğin semaphore) önce soldaki, sonra sağdaki çatalları 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 gereklidir. 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 tübüne sahip kişi alırsa sigarayı içer. Fakat farklı kişiler alırsa yürütütünün 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 olmadan) 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)" işlemleridir. Atomik işlemlerin ayrıca senkronize edilmeleri gerekmektedir.

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 sonuca global değişkenin değişkenin 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 gereklidir. Halbuki muhtemelen böyle çıkmayacaktır. Bu tür basit işlemlerin bile kritik kod içerisinde yapılması gereklidir. 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);
}

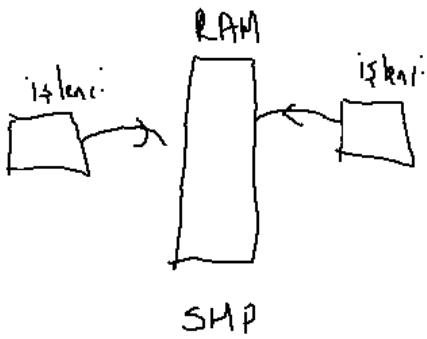
```

Tabii bu tür basit işlemler için bile kritik kod oluşturulması biraz sıkıcı olabilmektedir.

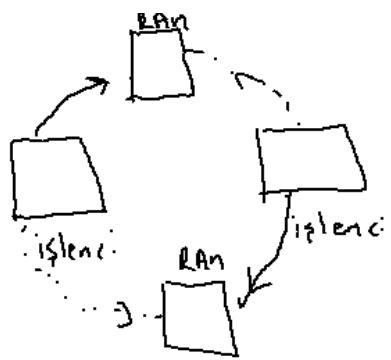
Ayrıca çok işlemcili ya da çok çekirdekli sistemlerde tek bir makine komutunun atomikliği de tartışıılır hale gelebilmektedir. Örneğin iki çekirdek de aşağıdaki makine komutunu tesadüfen aynı anda çalıştırılmış olsun:

```
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 diğerini 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.)

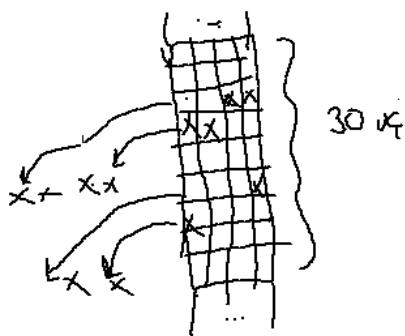


NUMA'da (Non Unified Memory Access) ise her işlemcinin ya da çekirdeğin RAM bloğu (bank) ayrıdır. Fakat bunlar birbirlerinin RAM'lerine (bank'lerine) de erişebilmektedir. Tabi bu durumda erişim daha yavaş olur (Zaten Non Unified Memory Access ismi buradan gelmektedir.) Böylece bu mimaride işlemcilerin ya da çekirdeklerin RAM'e erişirken birbirlerini durdurmasıyla daha az karşılaşmaktadır.



Bizim kullandığımız ana kartların çok büyük çoğu SMP mimarisine sahiptir. (Ancak örneğin AMD'nin Opteron işlemcilerinin kullandığı 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 çoğu 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. Özette tek bir makine komutu bile çok işlemcili ya da çok çekirdekli sistemlerde aynı bellek bölgesine erişirken soruna yol açabilmektedir. Bu konunun o sistem için araştırılması gereklidir.

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 gerekmektedir.

İşte tek makina komutuyla yapılabilecek artırma, eksiltme gibi basit işlem için kritik kod oluşturmaya gerek kalmasın diye Microsoft bir grup InterlockedXXX isimli API fonksiyonu bulundurmuştur. 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
);
```

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);
}

```

UNIX/Linux sistemlerinde Windows'taki InterlockedXXX fonksiyonlarının bir eşdeğeri var mıdır? Belli bir sürümden sonra gcc derleyicileri içsel (intrinsic) fonksiyon biçiminde atomik işlem yapan fonksiyonlar bulundurmaya başlamıştır. İçsel fonksiyonlar (intrinsic functions) derleyicinin kendisinin derleme sırasında hiç kütüphaneye başvurmadan açtığı 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. 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 nesnesin işlem görmeden öncekiş değerini geri dönüş değerini 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).

Ayrıca Microsoft derleyicileri belli bir sürümden sonra InterlockedXXX API fonksiyonlarının içsel (intrinsic) biçimlerini de bulundurmaya başlamıştır. Bunların isimlerinin başında bir _ karakteri bulunmaktadır. Örneğin: _InterlockedIncrement, _InterlockedDecrement gibi. 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 nitelikci (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;
```

bildirimini ile:

```
_Atomic int a;
```

bildirimini 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 metodları 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 Güvenliği (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 kılıtsız erişen fonksiyonlar thread güvenli degillerdir. 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'lerde 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 yazıldığında thread güvenli olurdu:

```

char *myitoa(int val, char *str)
{
    sprintf(str, "%d", val);

    return str;
}

```

Burada artık fonksiyon programcının aldığı adrese yerleştirme yapmaktadır.

Thread güvenli olmayan fonksiyonların çok thread'li uygulamalarda kritik kod içerisinde çağrılmaması gereklidir. Başka bir deyişle thread güvenli olmayan fonksiyonları çağrıran kişi senkronizasyon nesneleriyle onun aynı anda tek bir thread tarafından çağrılmmasını garanti altına almalıdır. İşte sonuç olarak programcının çok thread'li uygulamalarda fonksiyonların thread güvenli olup olmadığına dikkat etmesi gereklidir. 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 strttok, localtime, asctime, rand, tmpnam gibi fonksiyonlar statik data kullanmaktadır.

C standartlarında C11'e kadar hiç thread lafi edilmemiştir. C11'de thread'ler isteğe bağlı öğeler olarak dile dahil edilmiştir. Standartlarda thread lafinın edilmemesi derleyici yazarların bu fonksiyonları thread güvenli ya da thread güvensiz yazabilecegi anlamına gelmektedir. Peki örneğin biz localtime fonksiyonunu iki thread'ten de kullanıversak senkronize etmeli miyiz? Yanıt: Derleyiciyi tanımadığımız localtime fonksiyonunun o derleyici de thread güvenli olup olmadığını bilmemiz gereklidir.

İşte Microsoft Visual Studio 2005'e kadar standart C kütüphanesinin "single threaded" ve "multi threaded" olmak üzere iki versiyonunu da bulunduruyordu. Programcı da hangi kütüphaneyle 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 oluşturursa da artık Microsoft bu kayının 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 isimdeki 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 davranış gösteren 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 oluşturdu? 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. Peki bu örnekte her thread'in ayrı bir g_x global değişkeni olsaydı bir sorun olusur muydu? Yanıt hayır, olusmazdı. İşte thread'e özgü static alanlar aslında buna benzer bir işlemi gerçekleştirmektedir.

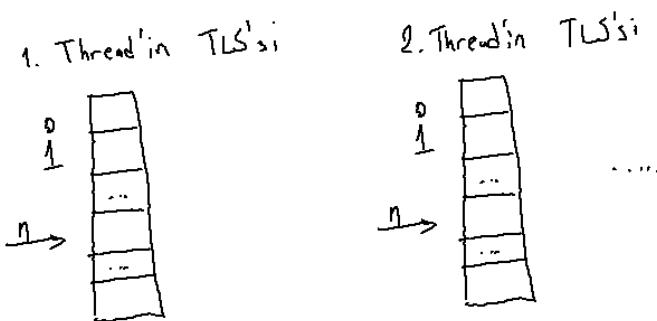
Thread'e özgü statik alanlara Windows dünyasında "Thread Local Storage (TLS)", UNIX/Linux dünyasında "Thread Specific Data (TSD)" denilmektedir. Thread'e özgü statik alanlar adeta thread'e özgü global değişken etkisi yaratmaktadır.

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 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 slot'a erişirsek o thread'in TLS'sindeki 5 numaralı slot'a erişmiş oluruz. Böylece aynı kod farklı thread'lerde farklı nesneleri 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 çağrılmak üzere fonksiyonu belirtir. 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 soket uygulamalarında) çok sayıda kısa ömürlü thread'in oluşturulması gerekebilmektedir. Örneğin server programa bir client'tan istek gelir. Server bir thread açarak o thread'inunu yapmasını sağlar. Çünkü o sırada server başka client'ların isteklerini de bekletmek istemez. Thread'lerin oluşturulma ve yok edilme zamanları göreli olarak maliyetlidir. İşte bu tür uygulamalarda thread'lerin oluşturulması ve yok edilmesini hızlandırmak için thread havuzları düşünülmüştür.

Thread havuzları aslında thread mekanizmasını kullanan bir organizasyondur. Thread havuzlarında belli miktarda thread oluşturulmuş ama çalışmıyor biçimde (suspend biçimde) bekletilir. Programcı havuzdan bir thread istediği zaman havuzda zaten oluşturulmuş olan bir thread ona verilir. Programcının thread ile işi bittiğinde thread yok edilmez, yeniden havuzda bekletilir. Tabii havuz yetersiz kalırsa ihtiyaca göre büyütülebilir. Havuzda çok fazla sayıda thread atılıp biçimde bekliyorsa yine havuz dinamik olarak küçültülebilmektedir.

Thread huvuzları Windows sistemlerinde API fonksiyonları tarafından desteklenmektedir. Yani thread havuzundan thread alan, thread'i bırakın API fonksiyonları vardır. Tabii bu API fonksiyonları aslında taban fonksiyonları değildir. Bunlar thread API fonksiyonlarını kendi içerisinde çağırmaktadır. UNIX/Linux sistemlerinde thread havuzu işlemlerini yapan standart POSIX fonksiyonları yoktur. Dolayısıyla thread havuz organizasyonunu programcılar 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 sisteminde 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 diğer 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 çalıştığımız 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 değer oluşturmaktadır. Bu taban değere thread'in görelî önceliği (+ ya da - olarak) toplanır. Prosesin öncelik sınıfı üst prosten alt prosese 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 uygulanmıştır. 2.4 çekirdeğiyle birlikte "O(1) çizelgelemesi" denilen yeni bir algoritma geçilmiştir. Ancak 2.6 ile bundan vaz geçilmiş ve eski sistemin bir benzerine dönülmüştür. 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 politikaaları 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 boşta 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 paylaşımı 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 varsayıyalım. Arik bu thread'in CPU'dan alınması için ya bloke olması ya da daha yüksek statik öncelikli bir thread'in çalışmaya hazır duruma gelmesi gereklidir. 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 bırakıldığı anda yeniden bu thread CPU'ya atanacaktır. Eğer SCHED_FIFO bir thread bloke olursa bloke çözüldüğünde kuyruğun sonuna aynır.
- SCHED_RR politikasına sahip thread'ler eğer aynı statik önceliğe sahipse döngüsel (round robin) çizelgelenirler. 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 paylaşımı 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 paylaşımı 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 paylaşımı çalışmala 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 düşüğünde 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 işleminin yapılması için prosesin etkin kullanıcı id'sinin 0 olması gereklidir. Yani yetkili olmayan prosesler kendi çizelgeleme politikalarını değiştiremezler.

```
#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 yapımasına 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 çizelgeme 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 değiştirmiştir 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 ayınmasını (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 sahipti. 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 çağrıramazlar). Yetkisiz prosesler yalnızca dinamik önceliklerini düşürebilirler (yani nice fonksiyonunu pozitif değerle çağrılabılırler.)

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 değiştirir. Linux'ta herhangi bir thread'in dinamik önceliğini alıp değiştirebilmek için getpriority ve setpriority POSIX fonksiyonları kullanılmaktadır:

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

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

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 götürüren 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 tasarılmış 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.

Pekiyi 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 özyineleme içerecek biçimde tasarılmaktadır.

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 gereklidir. Algoritmaları kıyaslamak için iki önemli ölçüt vardır: Hız ve kaynak kullanımı. Ancak baskın ölçüt hızdır. Bu nedenle algoritmaları karşılaştırmak denildiğinde default olarak hız ölçüne göre karşılaştırma anlaşılr. 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 denemeinin 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 kullanılabilecek sağlam yöntemlerden biri algoritmadaki çözüm için toplamda kaç işlemin gerektigine bakmaktır. (Tabii tüm işlemlerin makine zamanları aynı değildir fakat bu durum ihmali 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.

Algoritmada 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 işlemden 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 faydalılmaktadır. Bunların en yaygın kullanılanı Big O notasyonudur.

Big O Notasyonu

Big O notasyonu algoritmaları karmaşıklıklarına göre kategorilere (sınıflara) ayırır. Kategoriler arasında da iyilik kötülik ilişkisi vardır. Yani Big O notasyonunda karmaşıklıktaki algoritmaları aynı kategoriye (sınıfa) içerisinde yerleştirilir. Bu kategoriler da 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 davranışılır. Eğer algoritma birden fazla kategoriye giriysorsa en kötü kategori onun gerçek kategorisini belirtir. İyiden kötüye doğru Big O kategorileri şöyledir:

O(1): Bu kategoriye "sabit karmaşıklık" da denilmektedir. Eğer algoritmda hiç döngü yoksa her şey tekil işlemlerle yapılyorsa algoritma $O(1)$ karmaşıklıktadır. (Örneğin üçgenin alanının bulunması gibi). Tekil işlemlerin sayısı önemli değildir.

O(log N): Bu karmaşıklıkta N uzunlığında bir girdi kümlesi (örneğin bir dizi) vardır. Algoritmda açık ya da özyinelemeli 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 algoritmda bulunabilir.) Bu karmaşıklık kategorisine "logaritmik karmaşıklık" da denilmektedir.

O(N): Bu kategoriye "doğrusal karmaşıklık" da denilmektedir. Bu tür algoritmarda 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 algoritmalarla 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 algoritmalarla 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 karmaşıklıktaki problemlerin bugünkü bilgisayarlarla makul zamanda çözülmesi mümkündür.

O(K^N): Bu kategoriye üstel karmaşıklık (exponential complexity) de denilmektedir. Bu kategorideki algoritmaların 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 algoritmalarla 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 (dügünden) çıkararak tüm şehirleri (düğümleri) dolaşıp tekrar başlangıç yerine geri gelir. Amaç en kısa turu atacak çözümün bulunmasıdır. Bu problemde N tane düğüm varsa $(N - 1)! / 2$ tane rotayı hesaplamak gereklidir. 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 iyise 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 çalışabil almaktadır. 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 dizimlerde "bubble sort" algoritması "quick sort" algoritmasından hızlı çalışabil almaktadır. 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 değerlendirdip 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 ölçtlere göre sınıflandırılabilir. Tipik kullanılan sınıflandırma ölçütleri şunlardır:

Gerçekleştirim (Implementation) Biçimlerine Göre Sınıflandırma: Bu sınıflandırmada algoritmalar onların kodlarının nasıl yazılığına 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 Yönerek 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 İle 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 sınıflanırmada algoritmanın karmaşıklığına bakılır. Zaten bu sınıflandırma yukarıda ele alınmıştır.

- $O(1)$ Karmaşıklıktaki Algoritmalar
- $O(\log N)$ Karmaşıklıktaki Algoritmalar
- $O(N)$ Karmaşıklıktaki Algoritmalar
- $O(N \log N)$ Karmaşıklıktaki Algoritmalar
- $O(N^2)$ Karmaşıklıktaki Algoritmalar
- $O(N^3)$ Karmaşıklıktaki Algoritmalar
- $O(N^k)$ Karmaşıklıktaki Algoritmalar
- Üstel Karmaşıklıktaki Algoritmalar ($O(a^N)$)
- Faktöryel Karmaşıklıktaki Algoritmalar ($O(N!)$)

Donald Knuth "The Art Of Computing Programming" kitap serisinde algoritmaları ayrı kitaplar biçiminde söyle sınıflandırmıştır:

- Temel Algoritmalar (Fundamental Algorithms): Bunlar günlük programlamada karşılaşılan temel algoritmalarıdır.
- Yarı Nümerik Algoritmalar (Seminumerical Algorithms): Bunlar nümerik tarafı da olan ama programlama dünyasında karşılaşılabilir 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 denklem 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 sentaksi tarafından zaten doğrudan desteklenmektedir. Bazlarını 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 yaşılarına 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 aşı yukarı aynı hızda erişim" anlamındadır. Örneğin disk erişimleri de bu anlamda rastgeledir. Dizi elemanlarının ardışıl olmasının diğer bir faydası da onların başlangıç adresleri yoluyla fonksiyonlara aktarılabilirliğidir. Biz bir diziyi fonksiyona parametre yoluyla aktaracaksak dizinin her elemanını ayrı ayrı aktarmayız. Dizini 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 gereklidir. 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. Peki 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 gereklidir. Elemanlara birer vermek ve onlara bu isimler yoluyla erişmek daha okunabilir bir kullanım sunmaktadır. Yapı elemanları da bellekte ardışıl bir biçimde bulunduğu için yapı nesneleri fonksiyonlara başlangıç adresi yoluyla aktarılabilirler.

Yapı elemanlarının aralarında "hızalama (alignment)" amaçlı boşluklar (padding) bırakıldığını biliyorsunuz. Bu boşluklar kurallı bir biçimde bırakıldığı için bu durum yapı elemanlarının ardışılığını bozmamaktadır. Burada C açısından önemli bir durum vardır. Biz bir yapıyı kullanan fonksiyonları derleyip kütüphaneye yerleştirmiş olalım. Onu kullanırken yapı bildirimindeki elemanların yerlerini değiştirememiz. Eğer elemanların yerlerini değiştirirsek derlenmiş kodla onu kullanan kod için elemanların yerleri farlı olacaktır. Benzer biçimde derleyerek kütüphaneye yerleştirdiğimiz yapı kullanan bir kodun hızlama gereksinimi ile o kütüphaneyi kullanan kodun hızlama gereksinimlerinin aynı olması gereklidir.

Birlikler elemanları aynı adresten itibaren çıkışık yerleştirilen veri yapılarıdır. Bu durumde 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 sınıfı biçimde gerçekleştirilmiştir. Örneğin C++'ın, Java'nın, .NET'in standart kütüphanelerinde pek çok veri yapısı hazır biçimde zaten bulunmaktadır. O ortamlarda çalışan programcılar bunları yeniden yazması gerekmektedir. 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 teknikle birlikte kullanılmaya başlanmış bir terimdir.

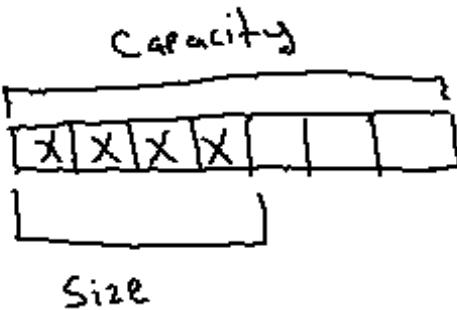
Temel Veri Yapıları

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 noktaların bir dizide tutulduğunu düşünelim. Başlangıçta bu dizinin hangi uzunlukta olacağını bilemeyez. Bu dizi duruma göre büyüyebilir. Ya da bir dizin içerisindeki dosyaları bir diziye yerleştirmek isteyelim. Dizindeki dosyaları bulurken önceden bunların kaç tane olduğunu bilemeyez. İşte bu tür durumlarda bir dizi oluşturarak duruma göre bunu büyütmek gerekir. Bu işlemi yapan veri yapısı ve fonksiyonlara "Dinamik Dizi" dizi denilmektedir.

Dinamik dizi gerçekleştiriminde 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 dizeye eleman eklenirken eleman "size" ile belirtilen indekse eklenir, "size" bir artırılır. "size" değeri "capacity" değerine geldiğinde yeniden tahsisat yapılarak (reallocation) "capacity" değeri artırılır. Peki容量 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" artırımı 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 varsayıbiliriz). Bu durumda sona eleman ekleme işlemi aslında doğrusal karmaşıklıktadır. Yani biz her eklemeyi $1/5$ kadar dönen bir döngüyle yaptığımızı düşünebiliriz. Halbuki büyütme eskitisinin iki katı kadar yapılrsa 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 gereklidir ("expand" işlemi). Aradan eleman silerken de o noktadan dizinin sıkıştırılması gereklidir ("shrink" işlemi).

Dinamik dizileri handle sistemi kullanarak aşağıdaki gibi oluşturabiliriz:

```
/* DynamicArray.h */

#ifndef DYNAMICARRAY_H_
#define DYNAMICARRAY_H_

#include <stddef.h>

/* Symbolic Constants */

#define FALSE          0
#define TRUE           1

#define DEF_CAPACITY   5
#define DARRAY_FAILED ((size_t)-1)

/* Type Declarations */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagDARRAY {
    size_t size;
    size_t capacity;
    DATATYPE *pArray;
} DARRAY, *HDARRAY;

/* Function Prototypes */

HDARRAY CreateDArray(void);
HDARRAY CreateDArrayWithCapacity(size_t capacity);
HDARRAY CreateDArrayWithArray(const DATATYPE *pArray, size_t size);
BOOL SetCapacity(HDARRAY hDArray, size_t newCapacity);
size_t AddItem(HDARRAY hDArray, DATATYPE val);
BOOL GetItemRef(HDARRAY hDArray, size_t index, DATATYPE *val);
size_t FindItem(HDARRAY hDArray, DATATYPE val);
BOOL InsertItem(HDARRAY hDArray, size_t index, DATATYPE val);
BOOL DeleteItem(HDARRAY hDArray, size_t index);
BOOL RemoveItem(HDARRAY hDArray, DATATYPE val);
void TrimToSize(HDARRAY hDArray);
void CloseDArray(HDARRAY hDArray);

/* Macros */

#define GetSize(hDArray)      ((hDArray)->size)
#define GetCapacity(hDArray)   ((hDArray)->capacity)
#define GetItem(hDArray, index) ((hDArray)->pArray[index])
#define Clear(hDArray)        ((hDArray)->size = 0)

#endif
/* DynamicArray.c */
```

```

/* DynamicArray.c */

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

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->size = 0;
    hDArray->capacity = capacity;

    return hDArray;
}

HDARRAY CreateDArrayWithArray(const DATATYPE *pArray, size_t size)
{
    HDARRAY hDArray;

    if ((hDArray = (HDARRAY)malloc(sizeof(DARRAY))) == NULL)
        return NULL;

    if ((hDArray->pArray = (DATATYPE *)malloc(sizeof(DATATYPE) * size)) == NULL) {
        free(hDArray);
        return NULL;
    }

    memmove(hDArray->pArray, pArray, sizeof(DATATYPE) * size);

    hDArray->size = size;
    hDArray->capacity = size;

    return hDArray;
}

BOOL SetCapacity(HDARRAY hDArray, size_t newCapacity)
{
    DATATYPE *newArray;

    if (newCapacity < hDArray->size)
        return FALSE;

    if ((newArray = (DATATYPE *)realloc(hDArray->pArray, sizeof(DATATYPE) * newCapacity)) == NULL)
        return DARRAY_FAILED;

    hDArray->pArray = newArray;
    hDArray->capacity = newCapacity;

    return TRUE;
}

```

```

size_t AddItem(HDARRAY hDArray, DATATYPE val)
{
    DATATYPE *newArray;

    if (hDArray->size == hDArray->capacity && !SetCapacity(hDArray, hDArray->capacity * 2))
        return FALSE;

    hDArray->pArray[hDArray->size++] = val;

    return hDArray->size - 1;
}

BOOL GetItemRef(HDARRAY hDArray, size_t index, DATATYPE *val)
{
    if (index >= hDArray->size)
        return FALSE;

    *val = hDArray->pArray[index];

    return TRUE;
}

size_t FindItem(HDARRAY hDArray, DATATYPE val)
{
    size_t i;

    for (i = 0; i < hDArray->size; ++i)
        if (hDArray->pArray[i] == val)
            return i;

    return DARRAY_FAILED;
}

BOOL InsertItem(HDARRAY hDArray, size_t index, DATATYPE val)
{
    if (index > hDArray->size)
        return FALSE;

    if (hDArray->size == hDArray->capacity && !SetCapacity(hDArray, hDArray->capacity * 2))
        return FALSE;

    memmove(hDArray->pArray + index + 1, hDArray->pArray + index, (hDArray->size - index) *
sizeof(DATATYPE));
    hDArray->pArray[index] = val;
    ++hDArray->size;

    return TRUE;
}

BOOL DeleteItem(HDARRAY hDArray, size_t index)
{
    if (index >= hDArray->size)
        return FALSE;

    memmove(hDArray->pArray + index, hDArray->pArray + index + 1, (hDArray->size - index - 1) *
sizeof(DATATYPE));
    --hDArray->size;

    return TRUE;
}

BOOL RemoveItem(HDARRAY hDArray, DATATYPE val)
{
    size_t index;

```

```

if ((index = FindItem(hDArray, val)) == DARRAY_FAILED)
    return FALSE;

return DeleteItem(hDArray, index);
}

void TrimToSize(HDARRAY hDArray)
{
    hDArray->pArray = (DATATYPE *)realloc(hDArray->pArray, sizeof(DATATYPE) * 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;
    size_t i;
    DATATYPE val;
    DATATYPE vals[10] = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };

    if ((hDArray = CreateDArrayWithArray(vals, 10)) == NULL) {
        fprintf(stderr, "cannot create dynamic array!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < GetSize(hDArray); ++i) {
        val = GetItem(hDArray, i);
        printf("%d ", val);
    }
    printf("\n");

    InsertItem(hDArray, 5, 1000);

    for (i = 0; i < GetSize(hDArray); ++i) {
        val = GetItem(hDArray, i);
        printf("%d ", val);
    }
    printf("\n");

    DeleteItem(hDArray, 11);

    for (i = 0; i < GetSize(hDArray); ++i) {
        val = GetItem(hDArray, i);
        printf("%d ", val);
    }
    printf("\n");

    RemoveItem(hDArray, 70);

    for (i = 0; i < GetSize(hDArray); ++i) {
        val = GetItem(hDArray, i);
        printf("%d ", val);
    }
    printf("\n");
}

```

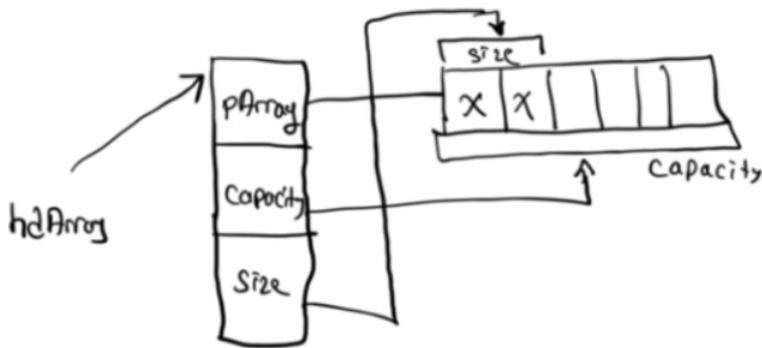
```

        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 büzer. 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 büyümeye 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;
}

```

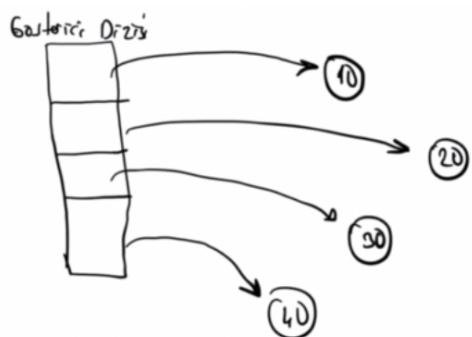
Bağlı Listeler

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.

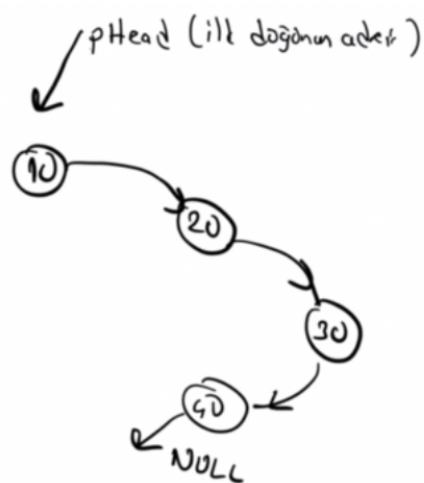
1) Elemanların adreslerini bir gösterici dizisinde tutmak

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:

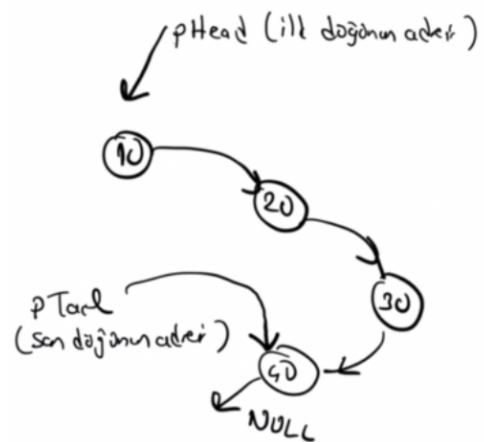


Önceki elemanın sonraki elemanın göstermesi biçiminde bir organizasyon çok daha uygundur. Örneğin:



İşte veri yapıları dünyasında önceki elemanın sonraki elemani 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.

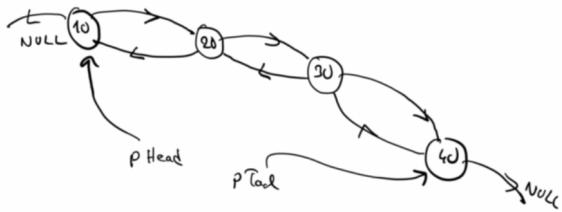
Bağlı listelerdeki her elemana düğüm (node) denilmektedir. Bağlı listenin ilk düğümünün yeri bir göstericide tutulur. (Genellikle bu göstericiye İngilizce "head pointer" denilmektedir.) Her ne kadar zorunlu olmasa da bağlı listelerde genellikle son düğümünün yeri de tutulmaktadır. (Genellikle son düğümün 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;
};
```

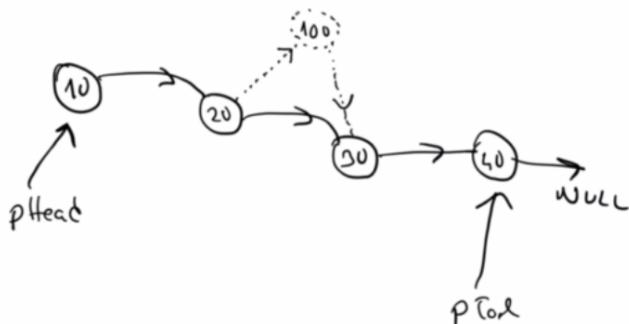
Eğer bir bağlı listede yukarıdaki gibi her düğüm yalnızca kendisinden bir sonraki düğümün adresini tutuyorsa böyle bağlı listelere "tek bağlı listeler (single linked list)" denilmektedir. Fakat eğer bir bağlı listede her düğüm hem kendinden sonraki düğümün hem de kentin önceki düğümün adresini tutuyorsa böyle bağlı listelere "çift bağlı listeler (doubly linked list)" denilmektedir.



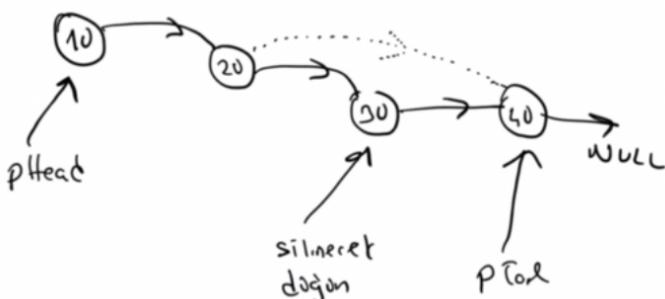
Tek bağlı listelerde bir düğümden yalnızca ileriye gidilebileceğine çift bağlı listelerde ise hem ileriye hem de geriye gidilebileceğine dikkat ediniz. Çift bağlı listelerin düğümleri C'de aşağıdaki gibi bir yapıyla temsil edilebilir:

```
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 yapılabilmektedir:



Bağılı listelerin başına ve sonuna eleman ekleme de yine sabit karmaşıklıkta bir işlemidir.

Pekiyi 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 kullanılma gereklilikleri 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 silemeyez. 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.

Bağılı Listelere Neden Gereksinim Duyulmaktadır?

- 1) Bağılı 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'lı) 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 edileceklərdir. İşte bunların bağlı listelerden silinmesi dinamik dizilere göre çok daha kolaydır. İşletim sistemi gerçekleştiriminde benzer pek çok veri yapısı bağlı listeler biçiminde organize edilmektedir.

Bağılı 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. (Tabi 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. Bağılı listeler ardışıl alana gereksinim duymazlar.
- 6) Bağılı listelerin bellekte toplam kapladığı alan dizilerden fazladır. (Ancak bölünme bundan çok daha büyük bir belleğin kullanım dışı kalmasına yol açan bir etkendir.)

Bağılı 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 gerçekleştirmi 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;

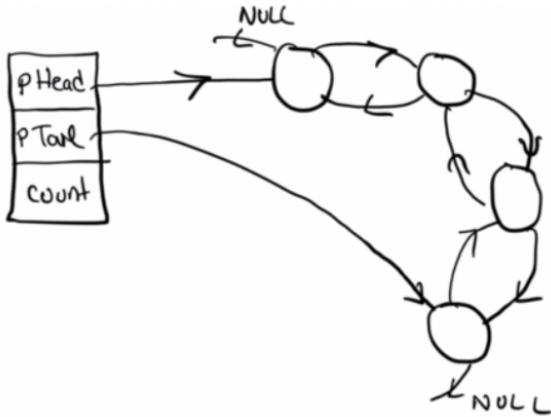
```

Bağlı listeyi kontrol eden handle alanı şöyle olabilir:

```

typedef struct tagLLIST {
    NODE *pHead;
    NODE *pTail;
    size_t count;
} LLIST, *HLLIST;

```



Çift bağlı listenin örnek gerçekleştirmesi 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 hLLList, NODE *pNode);
BOOL DeleteItemIndex(HLLIST hLLList, size_t index);
DATATYPE *FindItem(HLLIST hLLList, BOOL(*Compare)(const DATATYPE *));
NODE *FindItemNode(HLLIST hLLList, BOOL(*Compare)(const DATATYPE *));
BOOL WalkList(HLLIST hLLList, BOOL(*Proc)(DATATYPE *));
BOOL WalkListRev(HLLIST hLLList, BOOL(*Proc)(DATATYPE *));
void Clear(HLLIST hLLList);
void CloseList(HLLIST hLLList);

/* Macros */

#define GetCount(hLLList) ((hLLList)->count)
#define IsEmpty(hLLList) ((hLLList)->count == 0)

#endif

/* LinkedList.c */

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

HLLIST CreateLList(void)
{
    HLLIST hLLList;

    if ((hLLList = (HLLIST)malloc(sizeof(LLIST))) == NULL)
        return NULL;

    hLLList->pHead = hLLList->pTail = NULL;
    hLLList->count = 0;

    return hLLList;
}

NODE *AddItemTail(HLLIST hLLList, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (hLLList->pHead != NULL)
        hLLList->pTail->pNext = pNewNode;
    else
        hLLList->pHead = pNewNode;

    pNewNode->pNext = NULL;
    pNewNode->pPrev = hLLList->pTail;
    hLLList->pTail = pNewNode;

    ++hLLList->count;

    return pNewNode;
}

NODE *AddItemHead(HLLIST hLLList, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (hLLList->pHead != NULL)

```

```

    hLLList->pHead->pPrev = pNewNode;
else
    hLLList->pTail = pNewNode;

pNewNode->pNext = hLLList->pHead;
pNewNode->pPrev = NULL;
hLLList->pHead = pNewNode;

++hLLList->count;

return pNewNode;
}

NODE *InsertItemPrev(HLLIST hLLList, NODE *pNode, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (pNode == hLLList->pHead)
        hLLList->pHead = pNewNode;
    else
        pNode->pPrev->pNext = pNewNode;

    pNewNode->pPrev = pNode->pPrev;
    pNewNode->pNext = pNode;
    pNode->pPrev = pNewNode;

    ++hLLList->count;

    return pNewNode;
}

NODE *InsertItemNext(HLLIST hLLList, NODE *pNode, DATATYPE val)
{
    NODE *pNewNode;

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return NULL;
    pNewNode->val = val;

    if (hLLList->pTail == pNode)
        hLLList->pTail = pNewNode;
    else
        pNode->pNext->pPrev = pNewNode;

    pNewNode->pPrev = pNode;
    pNewNode->pNext = pNode->pNext;
    pNode->pNext = pNewNode;

    ++hLLList->count;

    return pNewNode;
}

NODE *InsertItemIndex(HLLIST hLLList, size_t index, DATATYPE val)
{
    size_t i;
    NODE *pNode;

    if (index > hLLList->count)
        return NULL;

    if (index == hLLList->count)

```

```

    return AddItemTail(hLLList, val);

pNode = hLLList->pHead;
for (i = 0; i < index; ++i)
    pNode = pNode->pNext;

return InsertItemPrev(hLLList, pNode, val);
}

void DeleteItem(HLLIST hLLList, NODE *pNode)
{
    if (pNode == hLLList->pHead)
        hLLList->pHead = pNode->pNext;
    else
        pNode->pPrev->pNext = pNode->pNext;

    if (pNode == hLLList->pTail)
        hLLList->pTail = pNode->pPrev;
    else
        pNode->pNext->pPrev = pNode->pPrev;

    --hLLList->count;

    free(pNode);
}

BOOL DeleteItemIndex(HLLIST hLLList, size_t index)
{
    size_t i;
    NODE *pNode;

    if (index >= hLLList->count)
        return FALSE;

    pNode = hLLList->pHead;
    for (i = 0; i < index; ++i)
        pNode = pNode->pNext;

    DeleteItem(hLLList, pNode);

    return TRUE;
}

DATATYPE *FindItem(HLLIST hLLList, BOOL(*Compare)(const DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLLList->pHead; pNode != NULL; pNode = pNode->pNext)
        if (Compare(&pNode->val))
            return &pNode->val;

    return NULL;
}

NODE *FindItemNode(HLLIST hLLList, BOOL(*Compare)(const DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLLList->pHead; pNode != NULL; pNode = pNode->pNext)
        if (Compare(&pNode->val))
            return pNode;

    return NULL;
}

BOOL WalkList(HLLIST hLLList, BOOL(*Proc)(DATATYPE *))

```

```

{
    NODE *pNode;

    for (pNode = hLLList->pHead; pNode != NULL; pNode = pNode->pNext)
        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

BOOL WalkListRev(HLLIST hLLList, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLLList->pTail; pNode != NULL; pNode = pNode->pPrev)
        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

void Clear(HLLIST hLLList)
{
    NODE *pNode, *pTempNode;

    pNode = hLLList->pHead;
    while (pNode != NULL) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    hLLList->pHead = hLLList->pTail = NULL;
    hLLList->count = 0;
}

void CloseList(HLLIST hLLList)
{
    NODE *pNode, *pTempNode;

    pNode = hLLList->pHead;
    while (pNode != NULL) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    free(hLLList);
}

/* 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 hLLList;
    int i;
    NODE *pNode;
    DATATYPE *pVal;

    if ((hLLList = CreateLList()) == NULL) {
        fprintf(stderr, "cannot create linked list\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i) {
        if ((pNode = AddItemTail(hLLList, i)) == NULL) {
            fprintf(stderr, "cannot add node!..\n");
            exit(EXIT_FAILURE);
        }
    }

    WalkList(hLLList, DispItem);
    printf("\n");

    DeleteItemIndex(hLLList, 7);

    WalkList(hLLList, DispItem);
    printf("\n");

    if ((pVal = FindItem(hLLList, MyComparer)) == NULL) {
        fprintf(stderr, "cannot find item!");
        exit(EXIT_FAILURE);
    }

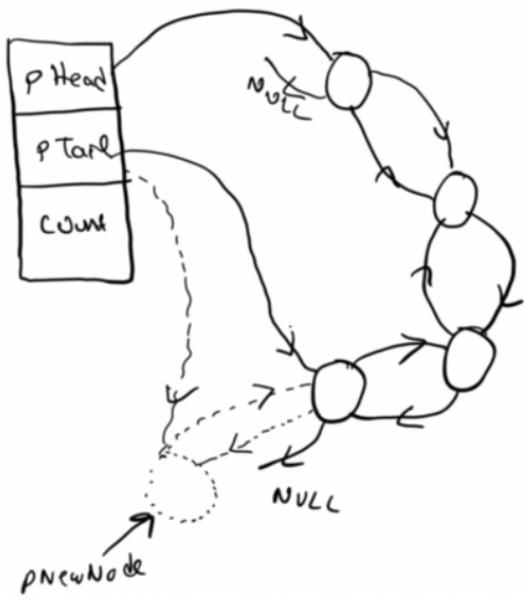
    printf("Found: %d\n", *pVal);
    printf("Total Item: %u\n", GetCount(hLLList));

    CloseList(hLLList);

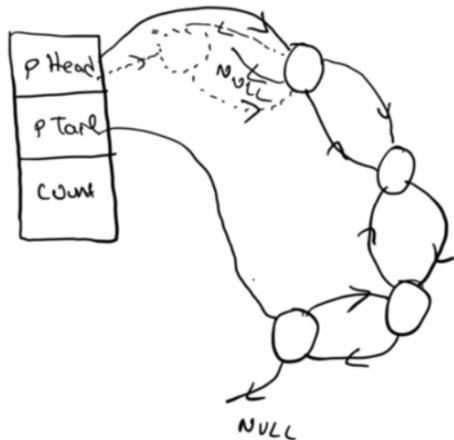
    return 0;
}

```

Sona eleman ekleme işleminde handle alanındaki pTail göstericisi eklenen elemanı gösterecek biçimde getirilmektedir. Listenin tamamen boş olması durumu da ayrıca kontrol edilmiştir:



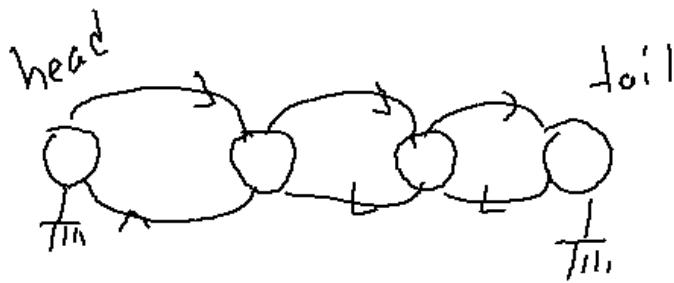
Listenin başına eleman eklenmesi de benzer bir çaba gerektirmektedir:



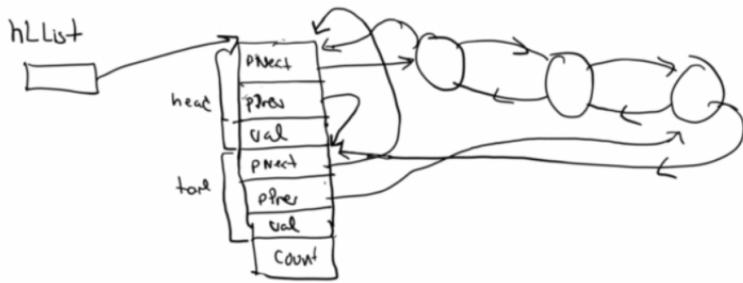
Çift bağlı liste gerçekleştiriminde handle alanında ilk ve son düğümlerin adresleri değil de düğümün kendisi tutulursa bazı özel durumlar özel olmaktan çıkartılabilir. Dolayısıyla tasarım daha sadeleşebilir. Tabii handle alanında tutulan ilk ve son düğümler sözde düğümler olur. Bu tasarım düğümdeki veri 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 eklimiş 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 düğümden sonra 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 CreateLLList(void);
NODE *AddItemTail(HLLIST hLLList, DATATYPE val);

```

```

NODE *AddItemHead(HLLIST hLLList, DATATYPE val);
NODE *InsertItemPrev(HLLIST hLLList, NODE *pNode, DATATYPE val);
NODE *InsertItemNext(HLLIST hLLList, NODE *pNode, DATATYPE val);
NODE *InsertItemIndex(HLLIST hLLList, size_t index, DATATYPE val);
void DeleteItem(HLLIST hLLList, NODE *pNode);
BOOL DeleteItemIndex(HLLIST hLLList, size_t index);
DATATYPE *FindItem(HLLIST hLLList, BOOL(*Compare)(const DATATYPE *));
NODE *FindItemNode(HLLIST hLLList, BOOL(*Compare)(const DATATYPE *));
BOOL WalkList(HLLIST hLLList, BOOL(*Proc)(DATATYPE *));
BOOL WalkListRev(HLLIST hLLList, BOOL(*Proc)(DATATYPE *));
void Clear(HLLIST hLLList);
void Closelist(HLLIST hLLList);

/* Macros */

#define GetCount(hLLList) ((hLLList)->count)
#define IsEmpty(hLLList) ((hLLList)->count == 0)

#endif

/* LinkedList.c */

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

HLLIST CreateLList(void)
{
    HLLIST hLLList;

    if ((hLLList = (HLLIST)malloc(sizeof(LLIST))) == NULL)
        return NULL;

    hLLList->head.pNext = &hLLList->tail;
    hLLList->head.pPrev= &hLLList->tail;
    hLLList->tail.pNext = &hLLList->head;
    hLLList->tail.pPrev = &hLLList->head;

    hLLList->count = 0;

    return hLLList;
}

NODE *AddItemTail(HLLIST hLLList, DATATYPE val)
{
    return InsertItemPrev(hLLList, &hLLList->tail, val);
}

NODE *AddItemHead(HLLIST hLLList, DATATYPE val)
{
    return InsertItemNext(hLLList, &hLLList->head, val);
}

NODE *InsertItemPrev(HLLIST hLLList, 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;
}

```

```

++hLLList->count;

return pNewNode;
}

NODE *InsertItemNext(HLLIST hLLList, 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;

    ++hLLList->count;

    return pNewNode;
}

NODE *InsertItemIndex(HLLIST hLLList, size_t index, DATATYPE val)
{
    size_t i;
    NODE *pNode;

    if (index > hLLList->count)
        return NULL;

    if (index == hLLList->count)
        return AddItemTail(hLLList, val);

    pNode = hLLList->head.pNext;
    for (i = 0; i < index; ++i)
        pNode = pNode->pNext;

    return InsertItemPrev(hLLList, pNode, val);
}

void DeleteItem(HLLIST hLLList, NODE *pNode)
{
    pNode->pPrev->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNode->pPrev;

    --hLLList->count;

    free(pNode);
}

BOOL DeleteItemIndex(HLLIST hLLList, size_t index)
{
    size_t i;
    NODE *pNode;

    if (index >= hLLList->count)
        return FALSE;

    pNode = hLLList->head.pNext;
    for (i = 0; i < index; ++i)
        pNode = pNode->pNext;

    DeleteItem(hLLList, pNode);

    return TRUE;
}

```

```

}

DATATYPE *FindItem(HLLIST hLLList, BOOL(*Compare)(const DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLLList->head.pNext; pNode != &hLLList->tail; pNode = pNode->pNext)
        if (Compare(&pNode->val))
            return &pNode->val;

    return NULL;
}

NODE *FindItemNode(HLLIST hLLList, BOOL(*Compare)(const DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLLList->head.pNext; pNode != &hLLList->tail; pNode = pNode->pNext)
        if (Compare(&pNode->val))
            return pNode;

    return NULL;
}

BOOL WalkList(HLLIST hLLList, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLLList->head.pNext; pNode != &hLLList->tail; pNode = pNode->pNext)
        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

BOOL WalkListRev(HLLIST hLLList, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;

    for (pNode = hLLList->tail.pPrev; pNode != &hLLList->head; pNode = pNode->pPrev)
        if (!Proc(&pNode->val))
            return FALSE;

    return TRUE;
}

void Clear(HLLIST hLLList)
{
    NODE *pNode, *pTempNode;

    pNode = hLLList->head.pNext;
    while (pNode != &hLLList->tail) {
        pTempNode = pNode;
        pNode = pNode->pNext;
        free(pTempNode);
    }

    hLLList->head.pNext = &hLLList->tail;
    hLLList->tail.pPrev = &hLLList->head;

    hLLList->count = 0;
}

void CloseList(HLLIST hLLList)
{
    NODE *pNode, *pTempNode;
}

```

```

pNode = hLLList->head.pNext;
while (pNode != &hLLList->tail) {
    pTempNode = pNode;
    pNode = pNode->pNext;
    free(pTempNode);
}
free(hLLList);
}

/* 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 hLLList;
    int i;
    NODE *pNode;
    NODE *pInsNode;
    DATATYPE *pVal;

    if ((hLLList = CreateLList()) == NULL) {
        fprintf(stderr, "cannot create linked list\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i) {
        if ((pNode = AddItemTail(hLLList, i)) == NULL) {
            fprintf(stderr, "cannot add node!..\n");
            exit(EXIT_FAILURE);
        }
        if (i == 6)
            pInsNode = pNode;
    }

    WalkList(hLLList, DispItem);
    printf("\n");

    InsertItemPrev(hLLList, pInsNode, 1000);

    WalkList(hLLList, DispItem);
    printf("\n");

    Clear(hLLList);

    WalkList(hLLList, DispItem);
    printf("\n");
}

```

```

    CloseList(hLLList);

    return 0;
}

```

Bağlı listelere ilişkin verdığımız iki örnekte de bağlı listelerin içerisinde tıpkı dinamik dizi örneğinde olduğu gibi DATATYPE türünden nesneler olduğunu varsayıdık. 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 hLLList, DATATYPE val);
```

Eğer DATATYPE bir yapıysa bu fonksiyonun aşağıdaki gibi bir prototipe sahip olması daha uygun olacaktır:

```
NODE *AddItemTail(HLLIST hLLList, const DATATYPE *val);
```

Cünkü 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 oluşturulabilir miyiz? Yanıt maalesef hayır. Çünkü DATATYPE typedef edildiğine göre programda bir daha değiştirilemez. Böylece biz aynı projede aynı türden birden fazla bağlı liste oluşturulabiliriz 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. Kuyruklarla 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.

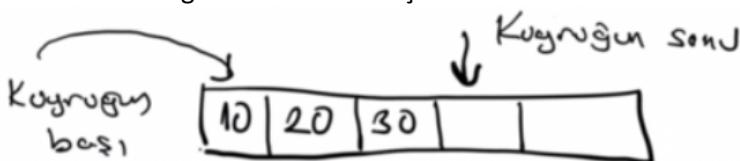
Kuyruk Veri Yapısına Neden Gereksinim Duyulmaktadır?

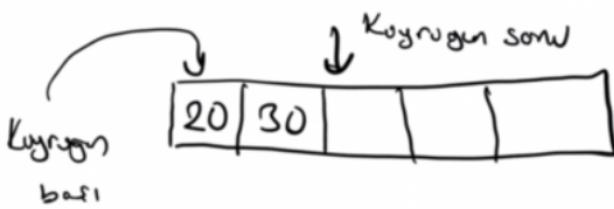
Kuyruklar genellikle bilgilerin geçici olarak sırası bozulmadan bekletilmesi için kullanılırlar. (Yani tipik olarak kuyruklar tampon alanların gerçekleştiririminde kullanılmaktadır.) Kuyruk veri yapısıyla pek çok yerde karşılaşılabilir. Örneğin arabalı vapura binme ve inme kuyruğu, yemek kuyrukları gerçek 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:

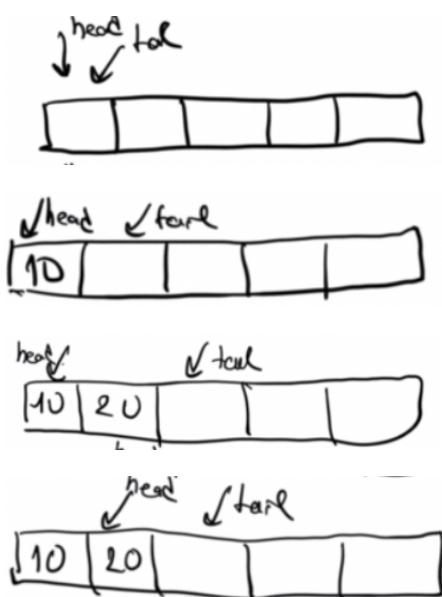
En basit yöntem dizi yöntemidir. Bu yöntemde kuyruk için bir dizi yaratılır. Bir index kuyruğun sonunu tutar. Eleman kuyruğa yerleştirilmek istendiğinde bu indeksin gösterdiği yere yerleştirilir ve indeks bir artırılır. Kuyruktan eleman alınmak istendiğinde de dizinin başındaki eleman alınır. Dizi ve indeks bir kaydırılır.



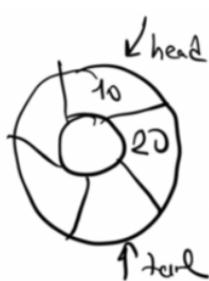


Bu yöntemde kuyruğa eleman yerleştirmek $O(1)$ karmaşıklıkta, kuyruktan eleman almak ise $O(N)$ karmaşıklıkta yapılmaktadır. Uygulamada diğer iki yöntem daha iyi olduğu için bu yöntem tercih edilmemektedir.

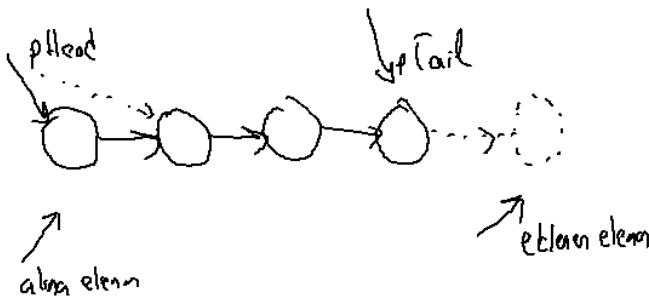
İkinci yöntemde (buna döngüsel kuyruk gerçekleştirmi de denilmektedir) yine kuyruk için bir dizi yaratılır. Kuyruğun başı ve sonu iki gösterici ya da indeksle tutulur. Geleneksel olarak bunlardan birine "head" göstericisi, diğerine "tail" göstericisi denilmektedir. Eleman "tail" göstericisinin gösterdiği 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 büyütülp 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 gerçekleştirminde 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");
}

```

Pekiyi döngüsel kuyruk sistemi ile bağlı listeli kuyruk sistemi arasındaki farklar nelerdir? Döngüsel kuyruk sisteminde tüm tahsisat işin 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 $O(N)$ karmaşıklıktadır. Kuyruktan eleman alma işlemi her iki sistemde de $O(1)$ karmaşıklıkta yapılabilmektedir.

Stack Veri Yapısı

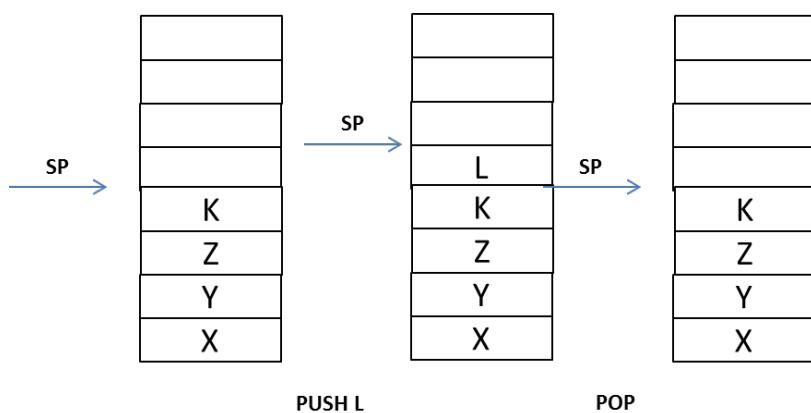
Stack LIFO pensibiyle çalışan bir kuyruk sistemidir. Yani stack'e son yerleştirilen bilgi ilk olarak alınır. Geleneksel olarak stack'e eleman ekleyen fonksyonlar İngilizce "push", stack'ten eleman alan fonksyonlar "pop" biçiminde isimlendirilmektedir. Push ve pop işlemleri $O(1)$ karmaşıklıkta gerçekleştirilmektedir.

Stack Veri Yapısına Neden Gereksinim Duyulmaktadır?

Stack sistemleriyle gerçek hayatı da karşılaşılmaktadır. Örneğin asansöre son binenler (genellikle) ilk girerler. Tabakları üst üste koyduğumuzda en üsttekini önce alırız. Alış veriş 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 çağrısızdır. Bazı algoritmaların gerçekleştiririmde de stack sistemleri kullanmaktadır. Örneğin aslında özyineleme içeren algoritmalar özyineleme olmadan yapay bir stack'le de gerçekleştirilebilirler. Parsing algoritmaları stack kullanan algoritmalar için tipik örneklerdir. Stack sistemleri mikroişlemcilerin çalışabilmesi için mutlak bulunması gereken bir mekanizmadır.

Stack Veri Yapısının Gerçekleştirilmesi

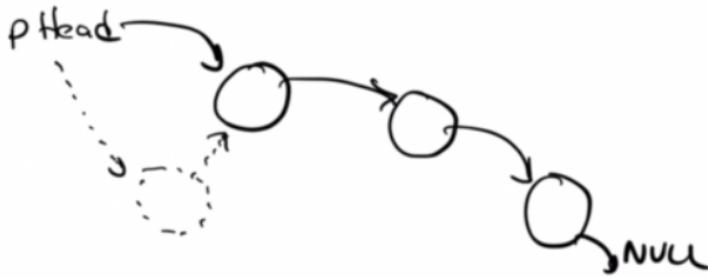
Stack sistemleri de dizilerle ve bağlı listelerle gerçekleştirilebilir. Dizilerle gerçekleştiririmde stack bir dizi olarak oluşturulur. Stack'in aktif noktası bir göstericiyle gösterilir. Bu göstericiye geleneksel olarak "stack gösterici (stack pointer)" denilmektedir. Push işlemi sırasında stack göstericisi 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 çalışılsa (yani push edilmemiş elemanlar pop edilmeye çalışılsa) bu kez stack aşağıdan taşacaktır (stack underflow).

Bağlı liste teknlığında (list stack) push işlemi sırasında bağlı listenin önüne eleman eklenir. Pop işleminde de bağlı listenin önündeki eleman alınır.



Tıpkı kuyruk sistemlerinde olduğu gibi satck sistemlerinde de dizi teknigi ile bağlı liste teknigi aynı avantaj ve dezavantajlara sahiptir. Yani dizi teknigiden push ve pop işlemleri O(1) karmaşıklıktadır. Bağlı liste teknigiden 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 gerçekleştirmi şöyleden 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 elemanın ek maliyetli sabit zamanlı olmasıdır. Anımsanacağı gibi normal dinamik dizilerde sona eleman ekleme ek maliyetli sabit karmaşıklıktayken başa (ya da araya) eleman eklemek doğrusal karmaşıklıktadır. Çift yönlü dinamik dizilerde elemana erişmek ek maliyetli sabit zamanlı bir işlemidir.

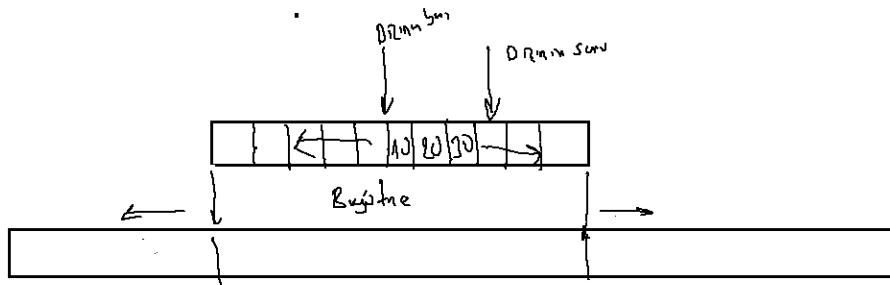
Çift Yönlü Dinamik Dizilere Neden Gereksinim Duyulmaktadır?

Bazı sistemlerde hem başa hem de sona eleman eklemek mümkün olabilmektedir. Örneğin gelen tek sayıları dizinin

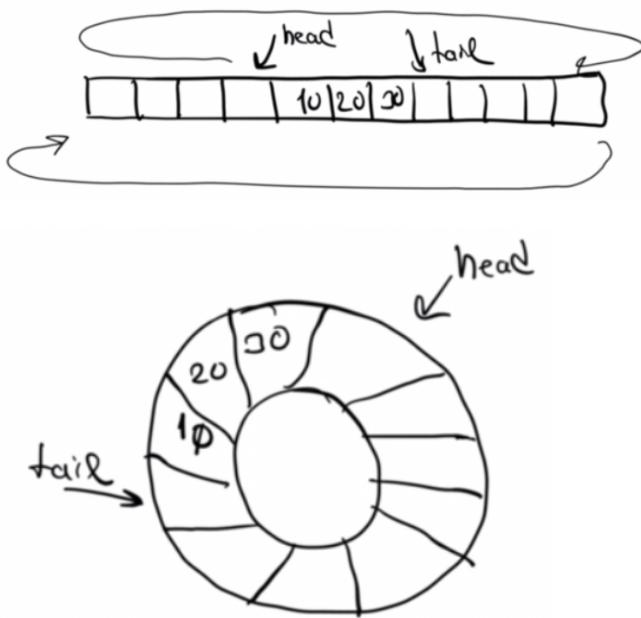
başına çift sayıları dizinin sonuna yerleştirmek istediğimizi düşünelim. Bunu normal dinamik dizile yapmak istersek eleman ekleme işlemi için doğrusal karmaşıklık gerekir. Fakat çift yönlü dinamik dizilerde hem başa hem de sona eleman ekleme sabit karmaşıklıkta yapılabilmektedir. Çift yönlü dinamik diziler genel bir veri yapısı olarak da kullanılabilir. Yani örneğin bunlar hem stack hem kuyruk de gerçekleştiririmde kullanılabilmektedir.

Çift Yönlü Dinamik Dizilerin Gerçekleştirilmesi

Çift yönlü dinamik dizilerin gerçekleştirimi tipik olarak iki biçimde yapılabilmektedir. Birinci biçimde bir dizi alınır. Fakat dizinin indeks olarak başı ve sonu ayrı bir göstericiyle tutturulur. Böylece başa ve sona eklenme sabit zamanlı yapılabilir. Örneğin:

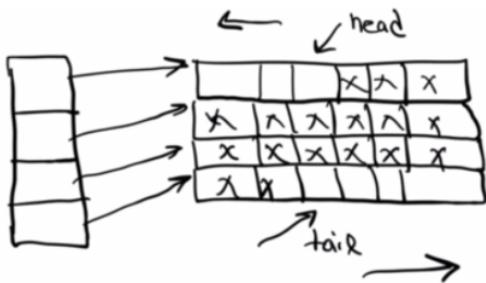


Tabii istenirse bu dizi sanki döngüsel kuyruk sisteminde olduğu gibi de ele alınabilir. Yani dizinin başını ve sonunu gösteren göstergeler iki uca geldiklerinde diğer uçtan çıkabilirler.

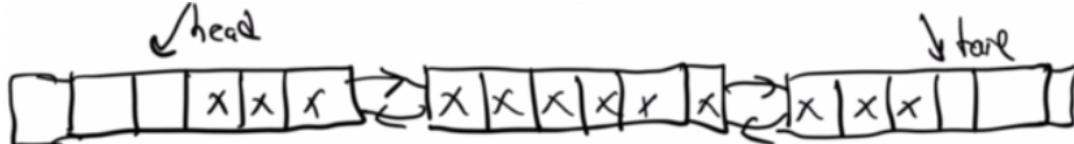


Bu biçimdeki gerçekleştirimde genellikle dizi de dinamik olarak büyütülür. Büyütme yine eskisinin iki katı olacak biçimde yapılır.

İkinci yöntemde bloklar bir listeye birbirine bağlanır. Bu bağlama işlemi sabit uzunluktaki blokların adreslerinin bir gösterici dizisinde tutulmasıyla yapılabilmektedir. 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.



Ya da her blok diğerini yine bir bağlı liste ile gösteriyor olabilir:



Bu yöntemde dizinin başında ve sonunda daha az boş yer kalır. Ayrıca araya ekleme ve aradan eleman silme durumunda bundan tüm bloklar etkilenmez. Fakat bu gerçekleştirimelemana erişmek daha zordur Eleman erişim O(N) karmaşıklığa doğru gitmektedir. İki gerçekleştirimin de iyi ve kötü tarafları vardır.

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>

/* Symbolic Constants */
#define FALSE

#define DEF_CAPACITY      10
#define DEQUE_FAILED      ((size_t)-1)

/* Type Declarations */

typedef int DATATYPE;

typedef struct tagDEQUE {
    DATATYPE *pArray;
    size_t head;
    size_t tail;
    size_t capacity;
} DEQUE, *HDEQUE;

/* Function Prototype */

HDEQUE CreateDeque(void);
HDEQUE CreateDequeWithCapacity(size_t capacity);
size_t AddItem(HDEQUE hDeque, DATATYPE val);
size_t AddItemFront(HDEQUE hDeque, DATATYPE val);

/* Macros */

#define GetItem(hDeque, index)      ((hDeque)->pArray[(hDeque)->head + (index)])
#define GetSize(hDeque)            ((hDeque)->tail - (hDeque)->head)

#endif

/* Deque.c */
```

```

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

#define RESIZE_TAIL    0
#define RESIZE_HEAD    1

static size_t resizeDeque(HDEQUE hDeque, int direction);

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->pArray = (DATATYPE *)malloc(sizeof(DATATYPE) * capacity)) == NULL) {
        free(hDeque);
        return NULL;
    }
    hDeque->capacity = capacity;
    hDeque->head = hDeque->tail = capacity / 2;

    return hDeque;
}

size_t AddItem(HDEQUE hDeque, DATATYPE val)
{
    if (hDeque->tail == hDeque->capacity)
        if (resizeDeque(hDeque, RESIZE_TAIL) == DEQUE_FAILED)
            return DEQUE_FAILED;

    hDeque->pArray[hDeque->tail++] = val;

    return hDeque->tail - hDeque->head - 1;
}

size_t AddItemFront(HDEQUE hDeque, DATATYPE val)
{
    if (hDeque->head == 0)
        if (resizeDeque(hDeque, RESIZE_HEAD) == DEQUE_FAILED)
            return DEQUE_FAILED;

    hDeque->pArray[--hDeque->head] = val;

    return 0;
}

static size_t resizeDeque(HDEQUE hDeque, int direction)
{
    DATATYPE *pNewArray;
    size_t amount;

    if ((pNewArray = (DATATYPE *)malloc(sizeof(DATATYPE) * hDeque->capacity * 2)) == NULL)
        return DEQUE_FAILED;

    amount = (hDeque->tail - hDeque->head) * sizeof(DATATYPE);
    if (direction == RESIZE_TAIL) {
        memcpy(pNewArray + hDeque->head, hDeque->pArray + hDeque->head, amount);
    }
}

```

```

    }
} else {
    memcpy(pNewArray + hDeque->capacity, hDeque->pArray, hDeque->tail * sizeof(DATATYPE));
    hDeque->head = hDeque->capacity;
    hDeque->tail = hDeque->capacity + hDeque->tail;
}
hDeque->capacity *= 2;
free(hDeque->pArray);
hDeque->pArray = pNewArray;

return hDeque->capacity;
}

/* Test.c */

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

int main(void)
{
    HDEQUE hDeque;
    int i;

    if ((hDeque = CreateDequeWithCapacity(5)) == NULL) {
        fprintf(stderr, "cannot create deque!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        AddItem(hDeque, i);

    AddItemFront(hDeque, 100);
    AddItemFront(hDeque, 200);
    AddItemFront(hDeque, 300);
    AddItemFront(hDeque, 400);
    AddItem(hDeque, 500);

    for (i = 0; i < GetSize(hDeque); ++i) {
        DATATYPE val;

        val = GetItem(hDeque, i);
        printf("%d ", val);
    }

    printf("\n");
    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) denir. Ancak yalnızca arama denildiğinde default olarak içsel arama anlaşılmaktadır.

Arama işleminin mümkün olduğu kadar çabuk yapılma arzu edilir. Peki arama işlemleri nasıl hızlandırılabilir? Örneğin bir dizi içerisinde bir değeri aramak isteyelim, hangi yöntemleri kullanabilirmiz? 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 diziye indeks yapılarak eleman dizinin o indeksine yerleştirilir. Sonra arama yapılrken doğrudan anahtara ilişkin indekse bakılır. Örneğin 100 kişinin bilgilerini bir diziye 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. Araken 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 diziye yerleştirmek isteyelim. Bizim çok çok büyük bir dizi açmamız gereklidir. Bu dizinin çok az kısmı dolu olacaktır.) Bu yöntemin diğer bir problemi de anahtarın bire bir indekse dönüştürülmesi gerekliliğidir. Örneğin anahtar TC kimlik numarası değil de ad soya biçiminde olsa biz belli bir kişiyi dizinin kaçinci indeksine yerlestireceğiz? İşte bizim yazımı indeks numarasına dönüştüren bir hash fonksiyonuna ihtiyacımız olur. Bu hash fonksiyonun da bire bir olması gereklidir. Fakat pratikte bire bir hash fonksiyonu mümkün değildir. (Yani örneğin iki farklı ad soyad aynı indeks değerini üretebilmelemdir.) Fakat yine de indexli arama özel durumlarda ideal biçimde kullanılabilmelemdir. Örneğin bir 1000 kişiye birer numara verip onları $O(1)$ karmaşıklıkta numaralarına göre arayabiliriz.

Peki yukarıdaki nedenlerden dolayı indexli arama mümkün değilse ne yapabiliriz? Örneğin 100 civarında kişiyi TC kimlik numaralarına göre bir diziye yerleştirip aramak isteyelim. Nasıl bir arama yöntemi kullanabiliriz? Eğer bir dizimde 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.

$$\frac{1+2+3+\dots+N}{N} = \frac{N(N+1)/2}{N} = (N+1)/2$$

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 gereklidir.) Sıralı arama 20'den küçük sayıda diziler için belki de en iyi arama yöntemidir. Çünkü ilerde 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 dizimdeki elemanlar sıraya dizilmişse ikili arama yöntemi (binary search) en iyi yöntemdir. İkili aramada iki çubuk (yani indeks) tutulur. Ortadaki elemanı bakılır. Sol ya da sağ çubuk oraya çekilir. Yani her defasında dizim yarısı kadar küçültülmektedir. İkili aramanın en kötü durum karmaşıklılığı $\log_2 N$ 'dır. 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>

typedef int DATATYPE;

DATATYPE *BinarySearch(const DATATYPE *array, int size, DATATYPE val)
{
    int left = 0, right = size - 1, mid;
    DATATYPE *pVal = NULL;

    while (left <= right) {
        mid = (left + right) / 2;
        if (array[mid] < val)
            left = mid + 1;
        else if (array[mid] > val)
            right = mid - 1;
        else {
            pVal = (DATATYPE *)&array[mid];
            break;
        }
    }
    return pVal;
}

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);

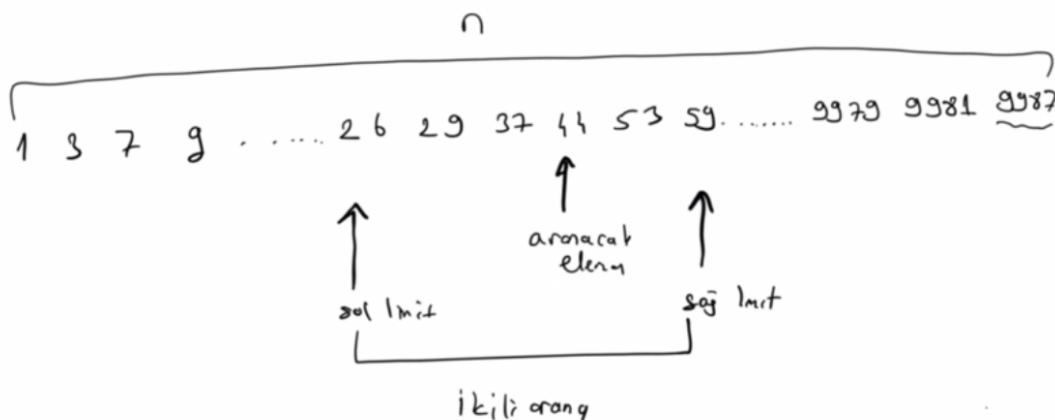
return 0;
}

```

Pekiyi sıralı olmayan diziyi önce sıraya dizip sonra ikili arama yapmak nasıl olur? Diziyi sıraya dizmenin en iyi karmaşıklığı $N \log N$ 'dir. Bu durumda sıraya dizmenin maliyeti zaten sıralı aramanın maliyetinden yüksek olur. Şüphesiz diziye hiç eleman eklenmeyecekse fakat çok sayıda arama yapılacaksa bu yöntem uygun hale gelebilir. Hatta diziye 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 diziye 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 $N \log (N)$ 'dır. Daha sonra N tane kişiyi de $\log (N)$ karmaşıklıkta arayacağımız düşünelim. İşlemenin toplam karmaşıklığı $N \log (N) + N \log (N)$ olur. Bu da $2N \log (N)$ anlamına gelir. Big O notasyonuna göre karmaşıklık $N \log (N)$ 'dır. 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 diziye sıraya dizip ikili arama yapmak daha avantajlidir. Şimdi yarışa katılan herkesin değil de yalnızca 10 kişinin durumunu soracağına düşünelim. Bu durumda doğrusal arama $O(N)$ karmaşıklıkta kalacaktır ve diğerinden daha iyi olacaktır.

Pekiyi sıralı diziler için ikili aramadan daha iyi bir yöntem olabilir mi? Eğer biz dizilimin ortasına değil de daha uygun bir yerine bakabilsek 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 karışımda bulunabilmek için dizilimin dağılımı hakkında bilgi sahibi olmak gereklidir. Bu yönteme "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 iyileştirilmiş biçimine de "üstel arama (exponential search)" denilmektedir. Bu yöntemde önce sıralı dizi iki taraftan yarıya bölmeye yöntemiyle daraltılır. Sonra daraltılmış olan alanda ikili arama yöntemi uygulanır. Bu yöntem özellikle değerler düzgün dağılmadığı durumlarda hız kazancı sağlamaktadır.



Üstel arama aşağıdaki gibi yapılabilir:

```

/* sample.c */

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

typedef int DATATYPE;

int BinarySearch(const DATATYPE *array, int size, DATATYPE val)
{

```

```

int left, right, mid;

left = 0;
right = size - 1;

while (left <= right) {
    mid = (left + right) / 2;
    if (array[mid] > val)
        right = mid - 1;
    else if (array[mid] < val)
        left = mid + 1;
    else
        return mid;
}

return -1;
}

int ExponentialSearch(const DATATYPE *array, int size, DATATYPE val)
{
    int right, left;
    int result;

    right = size - 1;
    while (array[right] > val)
        right /= 2;
    if (array[right] < val)
        right *= 2;

    left = 1;
    while (array[left] <= val)
        left *= 2;
    left /= 2;

    result = BinarySearch(array + left, right - left + 1, val);
    if (result == -1)
        return result;

    return left + result;
}

int main(void)
{
    int a[10] = { 3, 6, 9, 12, 14, 17, 20, 23, 26, 30 };
    int pos;

    if ((pos= ExponentialSearch(a, 10, 3)) == -1) {
        fprintf(stderr, "cannot find item!..\n");
        exit(EXIT_FAILURE);
    }
    printf("Item found: %d\n", pos);

    return 0;
}

```

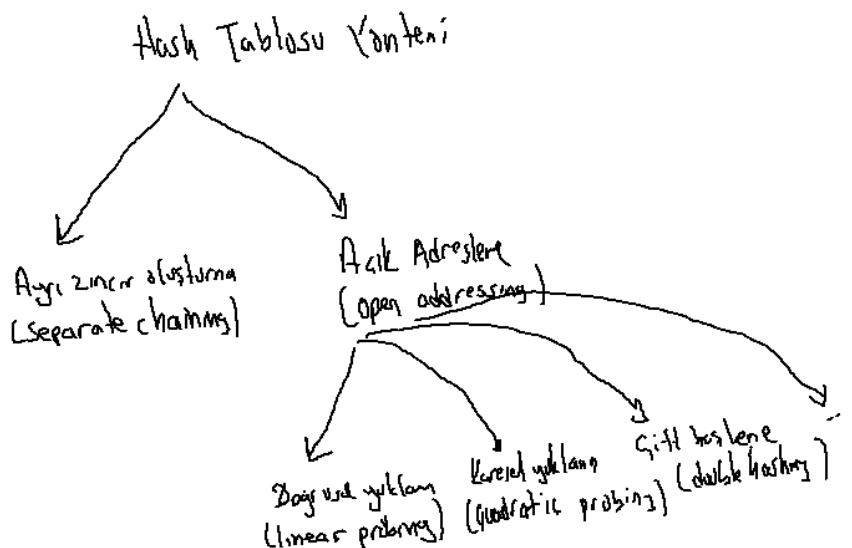
Pekiyi dizinin sıralı olmadığını düşünelim. Daha etkin bir arama nasıl yapılabilir? İşte bunun için başka algoritmik arama yöntemleri kullanılmaktadır. Bu yöntemlerde aslında eleman daha veri yapısına yerleştirilirken onun aranabileceği fikriyle bazı notlar işin başında alınmaktadır. Algoritmik arama 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.

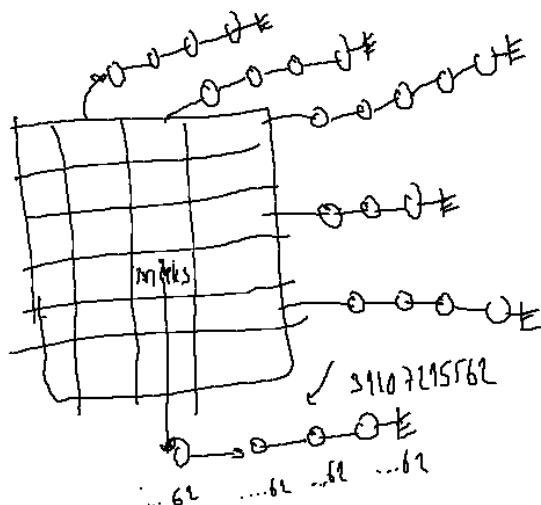
Şüphesiz yukarıda açıklanan yöntemde her zaman bir çakışma (collision) olasılığı vardır. Örneğin 23445623267 TC numaralı kişi için de aynı indeks ele edilecektir. İşte hash tablosu yöntemi çakışma durumundaki stratejiye göre iki temel alt gruba ayrılmaktadır:



Açık adresleme yöntemi de kendi aralarında pek çok alt yönteme 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 yapılrken 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 işlemidir. Arama ise doğrusal karmaşıklıkta gözükmekle 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 doğusal aramaya yaklaşmaya başlar. O halde tablo uzunluğunu baştan iyi öngörmek gereklidir. 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 yapılrken 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 parçasının 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 anahtarları dizi indeksine dönüştürmektedir. İyi bir hash fonksiyonunun şu özelliklere sahip olması beklenir:

- Tabloya yaydırmayı iyi yapmalıdır. Yani örneğin anahtar değerleri yanlış 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, yazıldan 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;
}

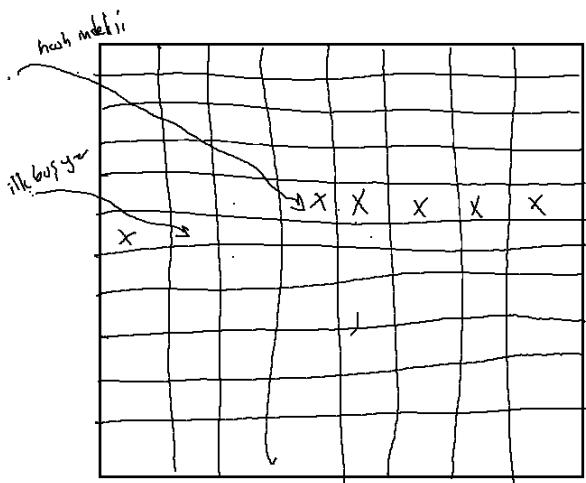
void getRandomItem(PERSON *pper)
{
    size_t k;

    for (k = 0; k < 31; ++k)
        pper->name[k] = rand() % 26 + 'A';
    pper->name[k] = '\0';
    pper->no = rand() % 100000;
}

```

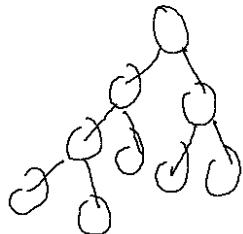
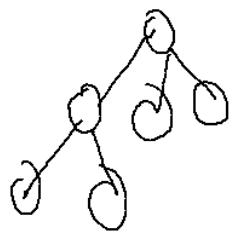
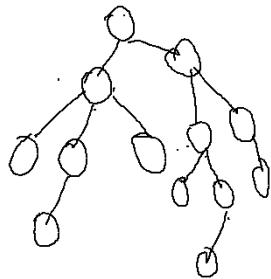
Hash Tablolarında Açık Adresleme (Open Addressing)

Yukarıda da belirtildiği gibi açık adresleme yöntemi pek çok alt yönteme 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. Doluya sırasıyla elemanlar üzerinde ilerlenerek ilk boş yer bulunur. Eleman ilk boş yere eklenir. Arama yapılrken 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 kaybettiştir.

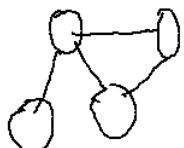


Ağaç Veri Yapıları (Tree Data Structure)

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:



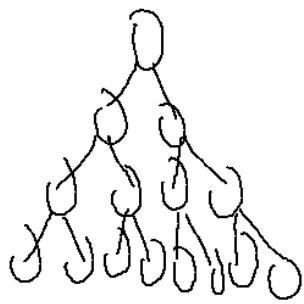
Ancak aşağıdaki veri yapısı bir ağaç değildir, graf belirtir:



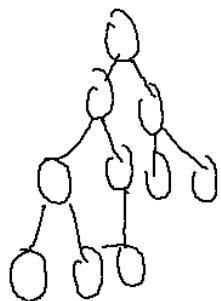
Ağaç Terminolojisi

Ağaçlarda kendisine hiçbir yerden gelinmeyen 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 geçen 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:

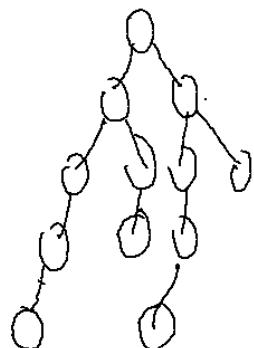


Eğer bir ağaçta sondan bir önceki kademe dışındaki tüm düğümlerin tam olarak n tane alt düğümü varsa böyle ağaçlara tam n 'li ağaçlar (complete n 'ary tree) denilmektedir. Örneğin aşağıda tam bir ikili ağaç görülmektedir:

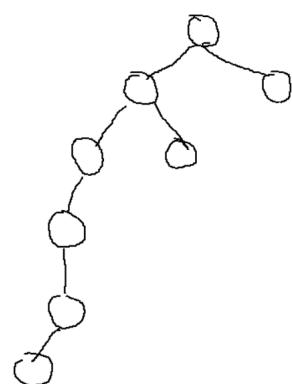


Tam ağaçlarda son kademe kaldırılırsa ağaçın tam dolu olması gerektiğine dikkat ediniz. Tabii tam dolu ağaçlar da tam ağaçlardır.

Eğer kökten itibaren her yaprağa giden yolun yüksekliği arasındaki fark en fazla belli sınırlarda kalıyorsa bu tür ağaçlara dengelenmiş ağaçlar (balanced tree) denir. Örneğin:



Burada her yaprağa giden yolun yükseklikleri arasında en fazla iki fark vardır. Böyle bir ağaç dengeli değildir. Bu bir koşulu altında aşağıdaki ağaç dengeli değildir:



Uygulamada dengeleme için en fazla iki yükseklik farkı kullanılmaktadır. Fakat görüldüğü gibi dengelenmiş (balanced) terimindeki denge ağaçın her tarafındaki yüksekliğinin belli bir sınır içerisinde olmasını gerektirmektedir.

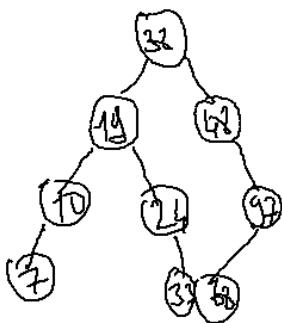
Arama Ağaçları (Search Trees)

Ağaçlar pek çok şeyi modellemek için oluşturulabilir. Örneğin bir dizin yapısı da aslında bir ağaçla temsil edebilir. Bir işletmedeki hiyerarşik yapı da bir ağaçla temsil edilebilir. Eğer ağaçlar arama amaçlı oluşturulmuşlarsa böyle ağaçlara ağaçlara "arama ağaçları (search trees)" denilmektedir. En çok kullanılan arama ağaçları ikili arama ağaçlarıdır.

Bir arama ağacında düğümler kayıtları temsil eder ve her düğümde bir anahtar bulunmaktadır. (Örneğin her düğümde bir kişinin bilgileri saklanabilir. Anahtar da kişinin TC kimlik numarası olabilir.) Arama ağaçları şekilsel olarak betimlendiğinde genellikle düğüm üzerinde yalnızca anahtarlar gösterilir.

İkili arama ağaçlarında aramanın gerçekleştirilebilmesi için anahtarlar "küçük olanlar sola, büyük olanlar sağa" ya da "küçük olanlar sağa, büyük olanlar sola" biçiminde belli bir kurala göre yerleştirilmektedir. Örneğin aşağıdaki sayılar sırasıyla sisteme gelmiş olsun ve biz bundan ikili arama ağacı oluşturmak isteyelim:

38, 19, 48, 10, 97, 24, 62, 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 düğümle temsil edilebilir:

```
struct tagNODE {  
    DATATYPE val;  
    struct tagNODE *pLeft;  
    struct tagNODE *pRight;  
} NODE;
```

Tabii ağaçın 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 düşüme büyük değerler sağ alt düşüme yerleştirilir. Arama yapılrken 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 struct tagNODE {
```

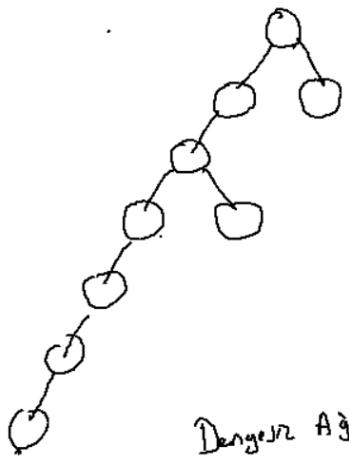
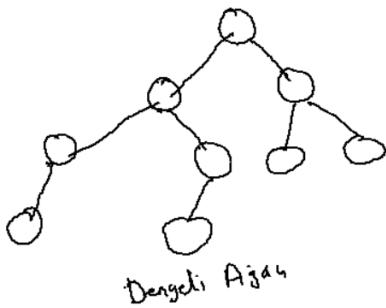
```

DATATYPE val;
struct tagNODE *pLeft;
struct tagNODE *pRight;
} NODE;

typedef struct tagBINARYTREE {
    NODE *pRoot;
    size_t count;
} BINARYTREE, *HBINARYTREE;

```

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 yapılmaktadır. 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. Bu kaydırma da doğrusal karmaşıklıkta yapılmaktadır. Halbuki dengelenmiş ikili ağaçlarda eleman eklenmesi de logaritmik karmaşıklıktadır.

İkili Ağaca Eleman Eklenmesi

İkili arama ağacına eleman eklerken önce ekleme yerinin bulunması gereklidir. Bunun için yine kök düğümden girilir, duruma göre sola ve sağa gidilerek eklenecek düğümün yeri tespit edilir ve ekleme yapılır. Örneğin:

```

BOOL InsertItem(HBINARYTREE hBinaryTree, const DATATYPE *pVal)
{
    NODE *pNewNode, *pNode, *pparentNode;

    pNode = hBinaryTree->pRoot;
    while (pNode != NULL) {
        pparentNode = pNode;

```

```

    if (pVal->no > pNode->val.no)
        pNode = pNode->pRight;
    else if (pVal->no < pNode->val.no)
        pNode = pNode->pLeft;
    else
        return FALSE;
}

if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
    return FALSE;
pNewNode->pLeft = pNewNode->pRight = NULL;
pNewNode->val = *pVal;

if (hBinaryTree->pRoot == NULL)
    hBinaryTree->pRoot = pNewNode;
else if (pVal->no > pParentNode->val.no)
    pParentNode->pRight = pNewNode;
else
    pParentNode->pLeft = pNewNode;

++hBinaryTree->count;

return TRUE;
}

```

Ekleme işlemi özyinelemeli olarak da yapılabilir. Bunun için kök düğümden 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 ağaçlarında arama yapılrken kök düğümden girilir. Duruma göre sola ve sağa sapılarak eleman bulunmaya çalışılır. Örnek bir arama işlemi şöyle yapılabilir:

```

DATATYPE *FindItem(HBINARYTREE hBinaryTree, KEY no)
{
    NODE *pNode;

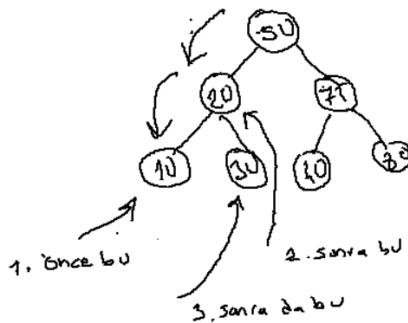
    pNode = hBinaryTree->pRoot;
    while (pNode != NULL) {
        if (no > pNode->val.no)
            pNode = pNode->pRight;
        else if (no < pNode->val.no)
            pNode = pNode->pLeft;
        else
            return &pNode->val;
    }

    return NULL;
}

```

İkili Ağacın Dolaşılması

İkili ağacın dolaşılması (traverse / walk) özyinelemeli bir algoritmayla üç farklı yolla yapılabilmektedir: Inorder, Preorder ve Postorder dolaşımları. 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 çıktılığında 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-büyükçe, eğer önce sağ alt düğüm ziyaret edilirse büyükten-küçüğe elde edilir.

Inorder dolaşımın özyinelemeli temel algoritması şöyledir:

```

void InOrderWalk(NODE *pNode)
{
    if (pNode->pLeft != NULL)
        InOrderWalk(pNode->pLeft);

    printf("%d\n", pNode->val.no);

    if (pNode->pRight != NULL)
        InOrderWalk(pNode->pRight);
}

```

Postorder dolaşımında düğümler alt-alt-üst biçiminde ziyaret edilir. Düğümlerin silinmesi tipik olarak bu tarz bir dolaşımla sağlanmaktadır. Postorder dolaşımın da özyinelemeli temel algoritması şöyledir:

```

void PostOrderWalk(NODE *pNode)
{
    if (pNode->pLeft != NULL)
        PostOrderWalk(pNode->pLeft);

    if (pNode->pRight != NULL)
        PostOrderWalk(pNode->pRight);

    printf("%d\n", pNode->val.no);
}

```

Preorder dolaşımında düğümler üst-alt-alt biçiminde ziyaret edilir. Preorder dolaşım tipik olarak ağaçları görüntülemek amacıyla uygulanmaktadır.

Yukarıdaki handle sistemiyle oluşturmuş olduğumuz ağaçları dolaşırken ziyaret sırasında programcının ne yapacağını bilmediğimizden dolayı dolaşım işleminde bir "call back" fonksiyondan faydalananabiliriz. Bu callback fonksiyon FALSE ile geri dönerse dolaşım sonlandırılabilir. Budurumda örneğin InOrder dolaşım şöyle olacaktır:

```

BOOL WalkInOrder(HBINARYTREE hBinaryTree, BOOL(*Proc)(const DATATYPE *))
{
    return walkInOrder(hBinaryTree->pRoot, Proc);
}

static BOOL walkInOrder(NODE *pNode, BOOL(*Proc)(const DATATYPE *))
{
    if (pNode->pLeft != NULL && !walkInOrder(pNode->pLeft, Proc))
        return FALSE;

    if (!Proc(&pNode->val))
        return FALSE;

    if (pNode->pRight != NULL && !walkInOrder(pNode->pRight, Proc))
        return FALSE;
}

```

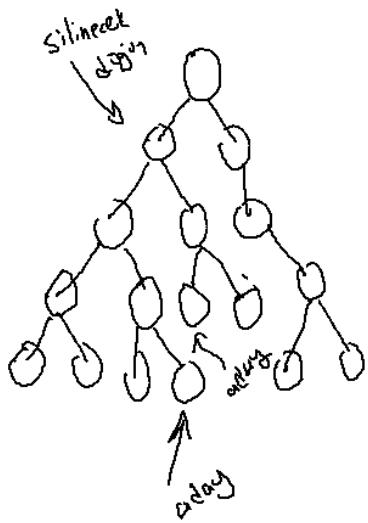
```

        return FALSE;
    return TRUE;
}

```

İkili Ağaçtan Düğüm Silinmesi

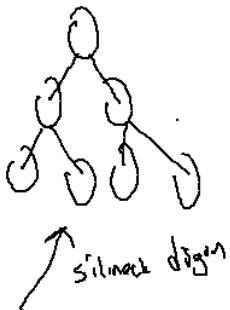
İkili ağaçtan eleman silmek biraz daha zahmetlidir. Çünkü silinecek elemanın yerine ağaçtaki başka elemanın taşınması gereklidir. Basit bir fikir şöyle olabilir: Silinecek düğüm onun solundakilerden büyük sağındakilerden küçük olduğuna göre onun yerine gelecek düğüm onun solundakilerin en büyüğü ya da sağındakilerin en küçüğü olabilir. Örneğin:



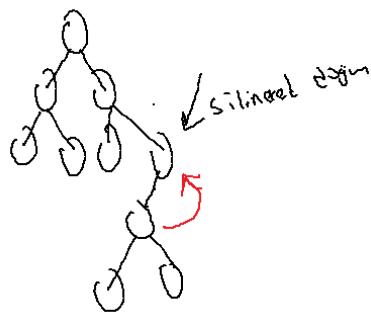
Silme işleminde 4 özel durum söz konusudur:

- 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.

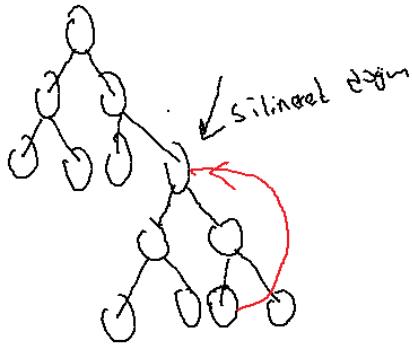
Silinecek düğümün hiç alt düğümü yoksa düğüm doğrudan silinir. Tabii onun üst düğümünün uygun linki NULL yapılır.



Silinecek düğümün yalnızca bir alt düğümü varsa bu alt düğüm silinecek düğüm yerine geçer.



Eğer silinecek düğümün iki alt düğümü de varsa bu durumda onun sağindakinin en küçüğü ya da solundakinin en büyüğü onun yerine geçer.



Eğer silinecek düğüm kökse handle alanında kökü güncellemek gereklidir. 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.

```
BOOL DeleteItem(HBINARYTREE hBinaryTree, KEY key)
{
    NODE *pRemoveNode, *pRemoveParentNode, *pReplaceNode;

    pRemoveNode = hBinaryTree->pRoot;
    while (pRemoveNode != NULL) {
        if (pRemoveNode->val.no == key)
            break;
        pRemoveParentNode = pRemoveNode;
        if (pRemoveNode->val.no > key)
            pRemoveNode = pRemoveNode->pLeft;
        else if (pRemoveNode->val.no < key)
            pRemoveNode = pRemoveNode->pRight;
    }

    if (pRemoveNode == NULL)
        return FALSE;

    pReplaceNode = findReplaceNode(pRemoveNode);

    if (pRemoveParentNode->pLeft == pRemoveNode)
        pRemoveParentNode->pLeft = pReplaceNode;
    else if (pRemoveParentNode->pRight == pRemoveNode)
        pRemoveParentNode->pRight = pReplaceNode;
    else {
        pReplaceNode->pLeft = pRemoveNode->pLeft;
        pReplaceNode->pRight = pRemoveNode->pRight;
    }

    if (pRemoveNode == hBinaryTree->pRoot)
        hBinaryTree->pRoot = pReplaceNode;

    free(pRemoveNode);
```

```

--hBinaryTree->count;

return TRUE;
}

static NODE *findReplaceNode(NODE *pRemoveNode)
{
    NODE *pNode, *pparentNode;

    if (pRemoveNode->pRight == NULL)
        return pRemoveNode->pLeft;

    if (pRemoveNode->pLeft == NULL)
        return pRemoveNode->pRight;

    pParentNode = pRemoveNode;
    pNode = pRemoveNode->pRight;
    while (pNode->pLeft != NULL) {
        pParentNode = pNode;
        pNode = pNode->pLeft;
    }
    if (pParentNode->pLeft == pNode)
        pParentNode->pLeft = pNode->pRight;
    else
        pParentNode->pRight = pNode->pRight;

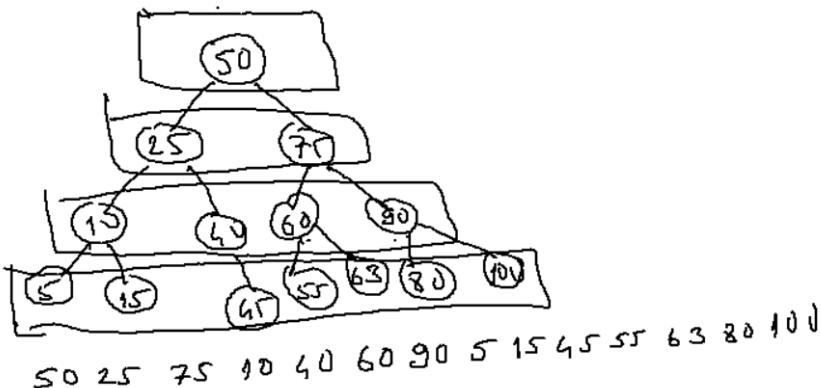
    return pNode;
}

```

Burada önce silinecek düğüm ve onun üst düğümü bulunmuştur. Sonra silinecek düğümün yerine gelecek olan düğüm findReplaceNode fonksiyonuyla tespit edilmiştir. Bundan sonra da gerekli bağlantılar yapılmıştır. findReplaceNode fnksiyonunda silinecek düğümün tek bir alt bir alt düğümü varsa silinecek düğümün yerine gelecek düğüm o alt düğüm olarak belirlenmiştir. Eğer silinecek düğümün iki alt düğümü varsa silinecek düğümden büyük olan en küçük alt düğüm bulunmuştur. Tabii silinecek düğümün hiç alt düğümü de olmayabilir. (Yani silinecek düğüm bir yaprak olabilir.) Bu durumda findReplaceNode NULL adresle geri dönmektedir.

İkili Ağaçlarda Depth-First ve Breadth-First Dolaşımalar

Ağaçlarda arama yaparken eğer ağaç bir arama ağacı değilse her düğüme bakmak gereklidir. Aslında bu durum genel olarak graflar için de söylenebilir. Böylece ağaçları dolaşırken iki strateji izlenebilmektedir. Depth-First dolaşımında her zaman dibe dalınarak özyinelemeli 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 BINARYSEARCHTREE_H_
#define BINARYSEARCHTREE_H_


/* Symbolic Constants */

#define FALSE      0
#define TRUE       1

/* Type Declarations */

typedef int BOOL;

typedef struct tagPERSON {
    char name[32];
    int no;
} PERSON;

typedef PERSON DATATYPE;
typedef int KEY;

typedef struct tagNODE {
    DATATYPE val;
    struct tagNODE *pLeft;
    struct tagNODE *pRight;
} NODE;

typedef struct tagBINARYTREE {
    NODE *pRoot;
    size_t count;
} BINARYTREE, *HBINARYTREE;

typedef struct tagQNODE {
    NODE *pNode;
    struct tagQNODE *pNext;
} QNODE;

/* Function Prototypes */

HBINARYTREE CreateBinaryTree(void);
BOOL InsertItem(HBINARYTREE hBinaryTree, const DATATYPE *pVal);
NODE *InsertItemRecur(HBINARYTREE hBinaryTree, const DATATYPE *pVal);
DATATYPE *FindItem(HBINARYTREE hBinaryTree, KEY no);
BOOL WalkInOrder(HBINARYTREE hBinaryTree, BOOL (*Proc)(DATATYPE *));
BOOL WalkPreOrder(HBINARYTREE hBinaryTree, BOOL(*Proc)(DATATYPE *));
BOOL WalkPostOrder(HBINARYTREE hBinaryTree, BOOL(*Proc)(DATATYPE *));
BOOL WalkBreadthFirst(HBINARYTREE hBinaryTree, BOOL(*Proc)(DATATYPE *));
BOOL DeleteItem(HBINARYTREE hBinaryTree, KEY key);
void DispAsTree(HBINARYTREE hBinaryTree);
void Clear(HBINARYTREE hBinaryTree);
void CloseBinaryTree(HBINARYTREE hBinaryTree);

/* Macros */

#define GetCount(hBinaryTree) ((hBinaryTree)->count)

#endif

/* BinarySearchTree.c */

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

```

#include "BinarySearchTree.h"

static NODE *insertItemRecur(NODE *pNode, NODE *pNewNode);
static BOOL walkInOrder(NODE *pNode, BOOL(*Proc)(DATATYPE *));
static BOOL walkPreOrder(NODE *pNode, BOOL(*Proc)(DATATYPE *));
static BOOL walkPostOrder(NODE *pNode, BOOL(*Proc)(DATATYPE *));
static NODE *findReplaceNode(NODE *pRemoveNode);
static void dispAsTree(NODE *pNode);
static void clear(NODE *pNode);

static BOOL putQueue(NODE *pNode);
static NODE *getQueue(void);
static void clearQueue(void);

static QNODE *gs_pHead;
static QNODE *gs_pTail;

static int gs_level = 1;

HBINARYTREE CreateBinaryTree(void)
{
    HBINARYTREE hBinaryTree;

    if ((hBinaryTree = (HBINARYTREE)malloc(sizeof(BINARYTREE))) == NULL)
        return NULL;

    hBinaryTree->count = 0;
    hBinaryTree->pRoot = NULL;

    return hBinaryTree;
}

BOOL InsertItem(HBINARYTREE hBinaryTree, const DATATYPE *pVal)
{
    NODE *pNewNode, *pNode, *pparentNode;

    pNode = hBinaryTree->pRoot;
    while (pNode != NULL) {
        pparentNode = pNode;
        if (pVal->no > pNode->val.no)
            pNode = pNode->pRight;
        else if (pVal->no < pNode->val.no)
            pNode = pNode->pLeft;
        else
            return FALSE;
    }

    if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
        return FALSE;
    pNewNode->pLeft = pNewNode->pRight = NULL;
    pNewNode->val = *pVal;

    if (hBinaryTree->pRoot == NULL)
        hBinaryTree->pRoot = pNewNode;
    else if (pVal->no > pparentNode->val.no)
        pparentNode->pRight = pNewNode;
    else
        pparentNode->pLeft = pNewNode;

    ++hBinaryTree->count;

    return TRUE;
}

NODE *InsertItemRecur(HBINARYTREE hBinaryTree, const DATATYPE *pVal)
{

```

```

NODE *pNewNode, *pResultNode;

if ((pNewNode = (NODE *)malloc(sizeof(NODE))) == NULL)
    return NULL;
pNewNode->val = *pVal;
pNewNode->pLeft = pNewNode->pRight = NULL;

if (hBinaryTree->pRoot == NULL) {
    hBinaryTree->pRoot = pNewNode;
    ++hBinaryTree->count;
    return pNewNode;
}

if ((pResultNode = insertItemRecur(hBinaryTree->pRoot, pNewNode)) != NULL)
    ++hBinaryTree->count;

return pResultNode;
}

NODE *insertItemRecur(NODE * pNode, NODE * pNewNode)
{
    if (pNewNode->val.no > pNode->val.no) {
        if (pNode->pRight == NULL) {
            pNode->pRight = pNewNode;
            return pNewNode;
        }
        else
            return insertItemRecur(pNode->pRight, pNewNode);
    }
    else if (pNewNode->val.no < pNode->val.no) {
        if (pNode->pLeft == NULL) {
            pNode->pLeft = pNewNode;
            return pNewNode;
        }
        else
            return insertItemRecur(pNode->pLeft, pNewNode);
    }
    else
        return NULL;
}

DATATYPE *FindItem(HBINARYTREE hBinaryTree, KEY no)
{
    NODE *pNode;

    pNode = hBinaryTree->pRoot;
    while (pNode != NULL) {
        if (no > pNode->val.no)
            pNode = pNode->pRight;
        else if (no < pNode->val.no)
            pNode = pNode->pLeft;
        else
            return &pNode->val;
    }

    return NULL;
}

BOOL WalkInOrder(HBINARYTREE hBinaryTree, BOOL(*Proc)(DATATYPE *))
{
    return walkInOrder(hBinaryTree->pRoot, Proc);
}

static BOOL walkInOrder(NODE *pNode, BOOL(*Proc)(DATATYPE *))
{
    if (pNode->pLeft != NULL && !walkInOrder(pNode->pLeft, Proc))

```

```

    return FALSE;

if (!Proc(&pNode->val))
    return FALSE;

if (pNode->pRight != NULL && !walkInOrder(pNode->pRight, Proc))
    return FALSE;

return TRUE;
}

BOOL WalkPreOrder(HBINARYTREE hBinaryTree, BOOL(*Proc)(DATATYPE *))
{
    return walkPreOrder(hBinaryTree->pRoot, Proc);
}

static BOOL walkPreOrder(NODE *pNode, BOOL(*Proc)(DATATYPE *))
{
    if (!Proc(&pNode->val))
        return FALSE;

    if (pNode->pLeft != NULL && !walkPreOrder(pNode->pLeft, Proc))
        return FALSE;

    if (pNode->pRight != NULL && !walkPreOrder(pNode->pRight, Proc))
        return FALSE;

    return TRUE;
}

BOOL WalkPostOrder(HBINARYTREE hBinaryTree, BOOL(*Proc)(DATATYPE *))
{
    return walkPostOrder(hBinaryTree->pRoot, Proc);
}

static BOOL walkPostOrder(NODE *pNode, BOOL(*Proc)(DATATYPE *))
{
    if (pNode->pLeft != NULL && !walkPostOrder(pNode->pLeft, Proc))
        return FALSE;

    if (pNode->pRight != NULL && !walkPostOrder(pNode->pRight, Proc))
        return FALSE;

    if (!Proc(&pNode->val))
        return FALSE;

    return TRUE;
}

static BOOL putQueue(NODE *pNode)
{
    QNODE *pNewNode;
    if ((pNewNode = (QNODE *)malloc(sizeof(QNODE))) == NULL)
        return FALSE;

    pNewNode->pNode = pNode;
    pNewNode->pNext = NULL;

    if (gs_pHead == NULL)
        gs_pHead = pNewNode;
    else
        gs_pTail->pNext = pNewNode;

    gs_pTail = pNewNode;

    return TRUE;
}

```

```

}

static NODE *getQueue(void)
{
    NODE *pNode;
    QNODE *pQTempNode;

    if (gs_pHead == NULL)
        return NULL;

    pNode = gs_pHead->pNode;
    pQTempNode = gs_pHead;
    gs_pHead = gs_pHead->pNext;

    if (gs_pHead == NULL)
        gs_pTail = NULL;

    free(pQTempNode);

    return pNode;
}

static void clearQueue(void) {
    QNODE *pQNode;
    QNODE *pQTempNode;

    pQNode = gs_pHead;
    while (pQNode != NULL)
    {
        pQTempNode = pQNode;
        pQNode = pQNode->pNext;
        free(pQTempNode);
    }
    gs_pTail = NULL;
}

BOOL isEmptyQueue(void)
{
    return gs_pHead == NULL;
}

BOOL WalkBreadthFirst(HBINARYTREE hBinaryTree, BOOL(*Proc)(DATATYPE *))
{
    NODE *pNode;
    BOOL retVal = TRUE;

    if (hBinaryTree->pRoot == NULL)
        return TRUE;

    if (!putQueue(hBinaryTree->pRoot))
        return FALSE;

    while (!isEmptyQueue()) {
        pNode = getQueue();

        if (pNode->pLeft != NULL && !putQueue(pNode->pLeft)) {
            retVal = FALSE;
            break;
        }

        if (pNode->pRight != NULL && !putQueue(pNode->pRight)) {
            retVal = FALSE;
            break;
        }

        if (!Proc(&pNode->val)) {

```

```

        retVal = FALSE;
        break;
    }
}

clearQueue();

return retVal;
}

void DispAsTree(HBINARYTREE hBinaryTree)
{
    gs_level = 0;
    dispAsTree(hBinaryTree->pRoot);
}

static void dispAsTree(NODE *pNode)
{
    printf("%*s%d\n", gs_level * 6, "", pNode->val.no);
    ++gs_level;

    if (pNode->pLeft != NULL)
        dispAsTree(pNode->pLeft);
    else {
        printf("%*s-\n", gs_level * 6, "");
        if (pNode->pRight != NULL)
            printf("%*s\n", gs_level * 6, "");
    }
    if (pNode->pRight != NULL)
        dispAsTree(pNode->pRight);
    else {
        printf("%*s-\n", gs_level * 6, "");
        if (pNode->pLeft != NULL)
            printf("%*s\n", gs_level * 6, "");
    }
    --gs_level;
}

BOOL DeleteItem(HBINARYTREE hBinaryTree, KEY key)
{
    NODE *pRemoveNode, *pRemoveParentNode, *pReplaceNode;

    pRemoveNode = hBinaryTree->pRoot;
    while (pRemoveNode != NULL) {
        if (pRemoveNode->val.no == key)
            break;
        pRemoveParentNode = pRemoveNode;
        if (pRemoveNode->val.no > key)
            pRemoveNode = pRemoveNode->pLeft;
        else if (pRemoveNode->val.no < key)
            pRemoveNode = pRemoveNode->pRight;
    }

    if (pRemoveNode == NULL)
        return FALSE;

    pReplaceNode = findReplaceNode(pRemoveNode);

    if (pRemoveParentNode->pLeft == pRemoveNode)
        pRemoveParentNode->pLeft = pReplaceNode;
    else if (pRemoveParentNode->pRight == pRemoveNode)
        pRemoveParentNode->pRight = pReplaceNode;
    else {
        pReplaceNode->pLeft = pRemoveNode->pLeft;
        pReplaceNode->pRight = pRemoveNode->pRight;
    }
}

```

```

}

if (pRemoveNode == hBinaryTree->pRoot)
    hBinaryTree->pRoot = pReplaceNode;

free(pRemoveNode);

--hBinaryTree->count;

return TRUE;
}

static NODE *findReplaceNode(NODE *pRemoveNode)
{
    NODE *pNode, *pparentNode;

    if (pRemoveNode->pRight == NULL)
        return pRemoveNode->pLeft;

    if (pRemoveNode->pLeft == NULL)
        return pRemoveNode->pRight;

    pParentNode = pRemoveNode;
    pNode = pRemoveNode->pRight;
    while (pNode->pLeft != NULL) {
        pParentNode = pNode;
        pNode = pNode->pLeft;
    }
    if (pParentNode->pLeft == pNode)
        pParentNode->pLeft = pNode->pRight;
    else
        pParentNode->pRight = pNode->pRight;

    return pNode;
}

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 "BinarySearchTree.h"

```

```

BOOL Proc(DATATYPE *pVal);

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;
    PERSON *pper;

    if ((hBinaryTree = CreateBinaryTree()) == 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 node!\n");
            exit(EXIT_FAILURE);
        }

    if ((pper = FindItem(hBinaryTree, 68)) == NULL) {
        fprintf(stderr, "cannot find item!..\n");
        exit(EXIT_FAILURE);
    }

    printf("%d node(s) in binary tree... \n", GetCount(hBinaryTree));
    printf("Found: %s, %d\n", pper->name, pper->no);

    DeleteItem(hBinaryTree, 27);

    WalkInOrder(hBinaryTree, Proc);
    printf("\n\n");

    WalkBreadthFirst(hBinaryTree, Proc);
    printf("\n\n");

    WalkBreadthFirst(hBinaryTree, Proc);

    printf("\n\n");
    DispAsTree(hBinaryTree);
    Clear(hBinaryTree);

    return 0;
}

BOOL Proc(DATATYPE *pVal)
{
    printf("%d %s\n", pVal->no, pVal->name);

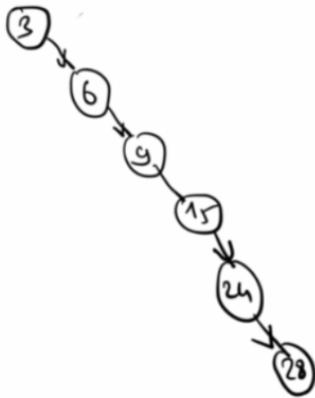
    return TRUE;
}

```

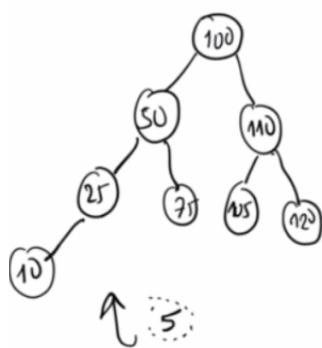
İkili Arama Ağaçlarının Dengelenmesi

İkili arama ağaçlarında yaprakların yükseklikleri birbirlerindne 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:

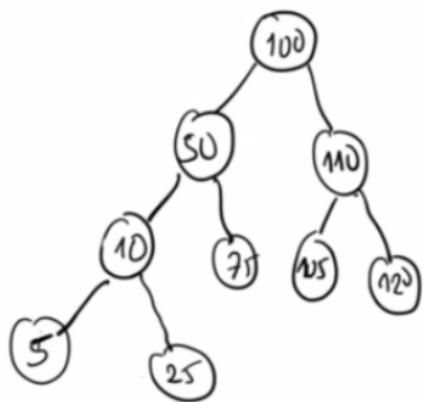
3 6 10 15 24 28



İşte ağaçın her yaprağının birbirlerine yakın bir yükseklikte olmasını sağlamak için "dengelenmiş (balanced)" ağaçlar kullanılmaktadır. Denegeli eleman eklenirken ya da silinirken dengenin bozulduğunu anlayıp $O(1)$ karmaşıklıkta ya da $O(\log N)$ karmaşıklıkta işlemlerle dengenin bozulmasını engellemeye faaliyetine denilmektedir. Örneğin aşağıdaki ikili arama ağacına 5 anahtarını ekleyeceğiz:



Bu ekleme işlemi yapıldığında ağaçın dengesi bozulacaktır. Biz bunu anlayıp bir düzeltme yapabiliriz:

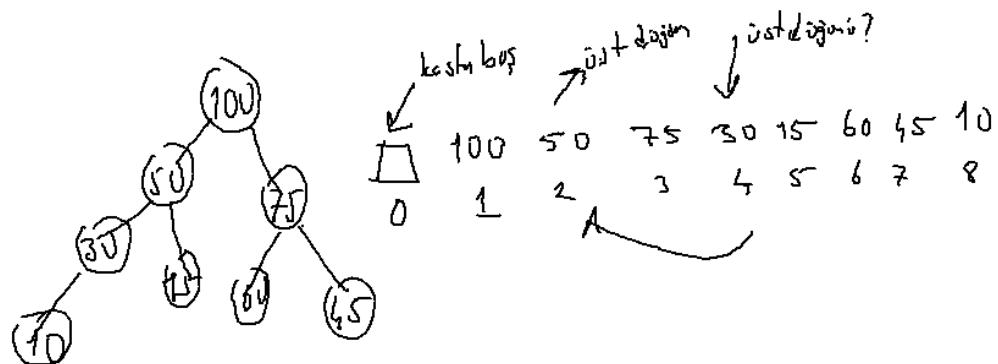


Tabii eklenecek elemana göre dengeleme işleminde yapılması gereken şeyler farklılaşabilmektedir. Genel olarak ikili arama ağaçlarında en çok kullanılan dengeleme algoritmaları AVL (Adelson-Velsky-Landis) ve "Red Black Tree" isimli algoritmalarıdır. AVL dengelemesi pek çok yerde daha fazla tercih edilmektedir. İki dengeleme algoritmasının birbirlerine göre avantajları ve dezavantajları vardır. İkili arama ağaclarının dengelenemesi "Sistem Programlama ve İleri C Uygulamaları-II" kursunda ele alınmaktadır.

Heap Veri Yapısı (Heap Ağaçları)

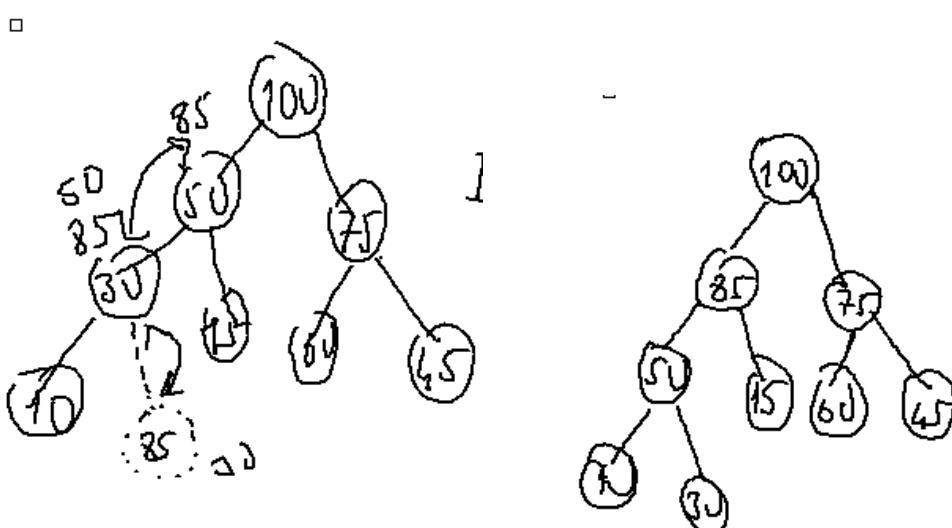
Heap özel bir ağaç türüdür. Öyle ki heap ağacında her zaman üst düğümün anahtarı alt düğümlerinin anahtarlarından ya daha büyütür (max heap) ya da daha küçütür (min heap). Heap ağaçları her zaman tam (complete) ağaçlardır. Bunun dışında heap ağacının başka bir koşulu yoktur. Heap ağaçları da n'li olabilir. Ancak en çok kullanılan heap ağaçları ikili heap ağaçlarıdır (binary heap).

Heap ağaçları aslında ağaç olarak gerçekleştirmek için tasarılmamıştır dizi olarak gerçekleştirilmek için tasarlanmıştır. Şöyle ki, biz bir heap ağını breadth-first bir dolaşımıla bir dizeye çevirebiliriz (buna terminoloji de "heapify" denilmektedir. Örneğin:

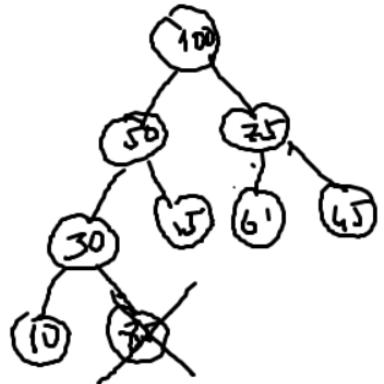
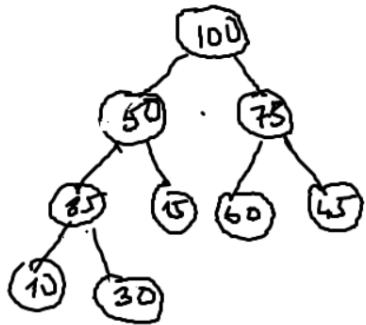


Uygulamada dizinin ilk elemanı hesaplamalar kolay yapılışın diye boş bırakılır (boş bırakılmasa da olur ancak hesaplamaların daha kolay yapılması için bu tercih edilmektedir.) Görüldüğü gibi heap ağıacı dizeye dönüştürüldüğünde n 'inci indeksteki bir düğümün üst düğümü $n/2$ 'inci indekste bulunur. Benzer biçimde n 'inci indeksteki düğümün alt düğümleri de $2*n$ ve $2 * n + 1$ numaralı indekslerde bulunacaktır.

Peki y kuralı bozmadan heap ağını temsil eden bir dizeye 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 ağaça biz 85 ekleyelim:



Heap ağıcından eleman silmek için bunun tam tersi yapılır. Silinecek eleman alınır. Onun yerine gelecek eleman onun alt düğümlerinden biridir. Bunlardan hangisi büyükse o yukarı alınır. Böyle devam edilir. Ta ki ağaçın yapraklarına gelene kadar. Örneğin yukarıdaki ağacta 85'i silmek isteyelim. İşlemler şu adımlardan 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.

İkili Arama Ağaçları İle Hash Tablolarının Karşılaştırılması

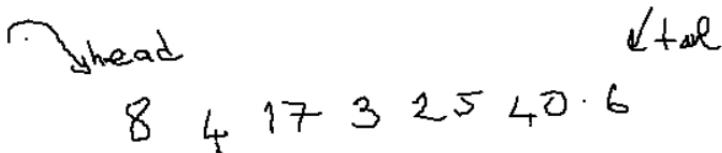
Hash tablolarında elemanlar hiç döngü olmadan $O(1)$ karmaşıklıkta eklenebilmektedir. Halbuki dengelenmiş ikili arama ağaçlarına elemanlar $O(\log N)$ karmaşıklıkta eklenir. Hash tablolarında arama tablonun uzunluğu makul tutulmuşsa ortalama 10 adımda gerçekleştirilebilmektedir. Halbuki dengelenmiş ikili 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 elemanarama 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 çok büyütmek gerekir. Bu da tablonun fazlaca bellek kullanmasına yol açar. Halbuki dengelelenmiş ikili ağaçların büyütülmesi daha az bellek kullanılarak yapılmaktadır. Hash tabloları sıralı dolaşma izin vermemektedir. Halbuki ikili ağaçlar sıralı bir biçimde dolaşılabilirler.

Heap Ağacına Neden Gereksinim Duyulmaktadır?

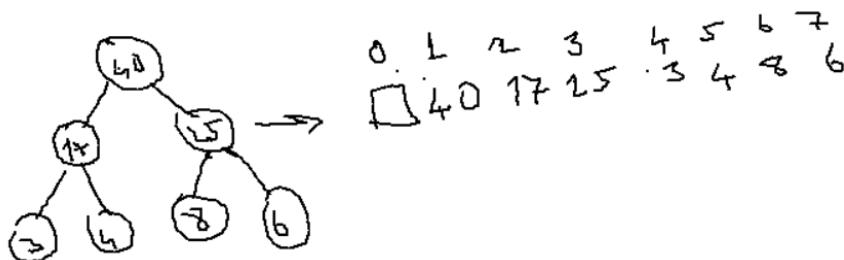
Heap ağacının en önemli uygulama alanı öncelik kuyruklarıdır (priority queues). Öncelik kuyruklarında kuyrukta birtakım elemanlar vardır. Kuyruktan eleman alınacağı zaman en yüksek önceliğe sahip olan eleman alınır. Böyle bir kuyruk sistemi düz bir diziyle gerçekleştirilmeye çalışılırsa her defasında en büyük elemanın bulunması için dizinin dolaşılması gereklidir. 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)

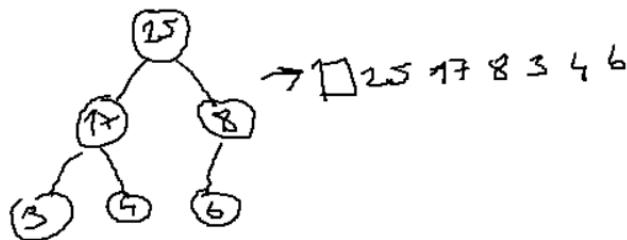
Öncelik kuyrukları kuyruğa yerleştirilen elemanların bir öncelik derecesinin olduğu kuyruk sistemleridir. Dolayısıyla kuyruktan eleman alınırken ilk sıradaki eleman değil öncelik derecesi en yüksek olan eleman alınmaktadır. Örneğin öncelik dereceleri birer sayı ile belirtilmiş olsun. Aşağıdaki gibi bir kuyruk sisteminin bulunduğu düşünelim:



Burada normal bir kuyruk sistemi söz konusu olduğunda eleman alınacakken kuyruğun önündeki 8 alınırdı. Ancak öncelik kuyruklarında kuyruğun başındaki eleman değil önceliği en yüksek olan eleman alınmaktadır. Böyle bir kuyruk sistemini daha önce yaptığımız gibi döngüsel biçimde tasarlarsak eleman alma işlemi doğrusal karmaşıklıkta yapılabilir. 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 gereklidir. İşte bu veri yapısını gerçekleştirmek için en etkin yöntem heap ağaçlarıdır. Çünkü heap ağaçlarına eleman eklemek $O(\log N)$ karmaşıklıkta bir işlemidir. Heap ağaçlarının tepesinden eleman alma da yine $O(\log N)$ karmaşıklıkta yapılabilmektedir. Örneğin yukarıdaki değerlerden bir heap ağıacı oluşturulmuş olsun:



Burada biz öncelik kuruğundan 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 yapılmıştır.



Heap Veri Yapısının Gerçekleştirilmesi

Heap veri yapısı için bir dizi kullanılır. Tabi dizinin dinamik bir dizi olması tercih edilir. Heap'a eleman insert edilirken insert edilecek yer en aşağıdan hareketle yukarı doğru çıkışlarak yer değiştirmelerle belirlenir. Heap'in kökündeki elemanın alınması benzer biçimde yapılır. Kök elemanın yerine geçecek eleman aşağıya doğru inilerek belirlenir. Örnek bir heap gerçekleştirmişi 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(HEAP))) == 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;
}

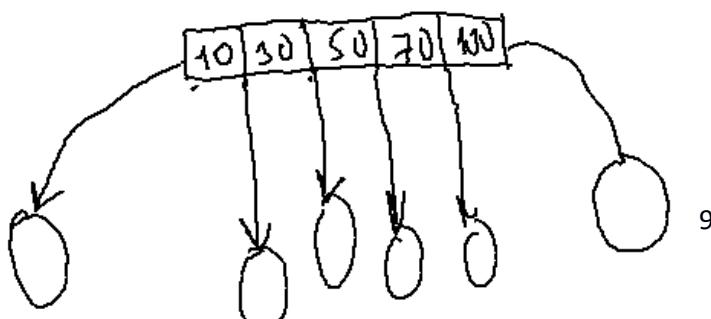
```

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 gereklidir. 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" olduğunu 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 varsayıyalım. 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 geçiş oluşur. Bu da disk okulamalarının 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.

Pekiyi 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 çukurde 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 yapılacak 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 gereklidir.

Veri yapılarını genelleştirerek biz onların kodlarına dokunmadan istediğimiz her türlü tutmasını sağlayabiliriz. Veri yapılarının genelleştirilmesi için üç yol izlenebilir:

1) Her tür için o veri yapısını yeniden kodlamak. Örneğin IntLinkedList, PersonLinkedList gibi. Şüphesiz bu yöntem kodu büyütme eğilimindedir. Ancak böyle oluşturulmuş veri yapılarının kullanımı kolaydır ve böyle oluşturulmuş veri yapıları görelî olarka 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;
```

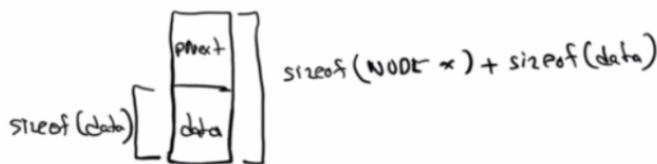
Burada list template bir sınıfıtır. Kendisi şablon bir bildirimdir. Derleyici her tür için onu yeniden yazar.

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) Bu üçüncü yöntemde veri yapısı türden bağımsız olarak yalnızca metadata kısımlarla ilişkili biçimde oluşturulur. Örneğin Linux, BSD ve CSD işletim sistemlerindeki genelleştirme için bu yöntem kullanılmıştır. Bu yöntemin de kullanımı zordur. Ancak kod etkinliği yüksektir.

Veri Yapılarının Türden Bağımsız Olarak Genel Fonksiyonlarla Genelleştirilmesi (2. Yöntem)

Teorik olarak yukarıda sözünü ettigimiz veri yapılarını oluşturmak için aslında biz DATATYPE türünün ne olduğunu derleme zamanında bilmek zorunda değiliz. O türün byte uzunluğunu bilsek bu bize yeter. Şöyled 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 kullanılarak 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 memcpy 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;
}

```

Türden bağımsız veri yapısı oluşturmanın diğer bir yolu da veri yapısının metadata kısımlarını asıl veri yapısının içerisinde 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:

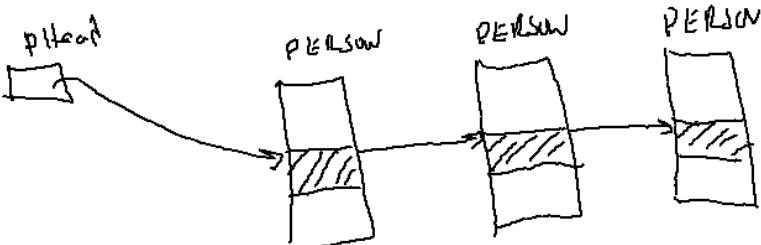
```

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 yazılabılır:

```
#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 listeörneğinde bizim node adresinden offsetof kadar eksiltme yapıp sonucu ilgili türü dönüştürmemiz gereklidir. 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 gerçekleştirmi şöyleden 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 CreateLLList(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 CreateLLList(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)

```

```

{
    pNode->pNext = pNewNode;
    pNode->pPrev = pNewNode->pPrev;
    pNewNode->pPrev->pNext = pNode;
    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 hLLList, NODE *pNewNode)
{
    return InsertItemNext(hLLList, &hLLList->head, pNewNode);
}

NODE *AddItemTail(HLLIST hLLList, NODE *pNewNode)
{
    return InsertItemPrev(hLLList, &hLLList->head, pNewNode);
}

void DeleteItem(HLLIST hLLList, NODE *pNode)
{
    --hLLList->count;

    deleteItem(pNode);
}

void deleteItem(NODE *pNode)
{
    pNode->pPrev->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNode->pPrev;
}

BOOL WalkList(HLLIST hLLList, BOOL(*Proc)(NODE *))
{
    NODE *pNode = hLLList->head.pNext;

    while (pNode != &hLLList->head) {
        if (!Proc(pNode))
            return FALSE;
        pNode = pNode->pNext;
    }
    return TRUE;
}

BOOL WalkListRev(HLLIST hLLList, BOOL(*Proc)(NODE *))
{
    NODE *pNode = hLLList->head.pPrev;

    while (pNode != &hLLList->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 hLLList;
    NODE *pNode;

    if ((hLLList = CreateLList()) == NULL) {
        fprintf(stderr, "cannot create linked list!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 6; ++i) {
        if (i == 3)
            pNode = AddItemTail(hLLList, &persons[i].node);
        else
            AddItemTail(hLLList, &persons[i].node);
    }

    WalkList(hLLList, WalkProc);
    DeleteItem(hLLList, pNode);
    printf("-----\n");
    WalkList(hLLList, WalkProc);
    printf("-----\n");
    WalkListRev(hLLList, WalkProc);

    CloseList(hLLList);

    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 hLLList)
{
    hLLList->count = 0;

    hLLList->head.pNext = &hLLList->head;
    hLLList->head.pPrev = &hLLList->head;
}

```

```

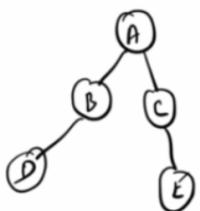
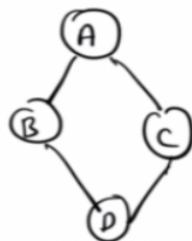
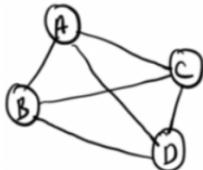
}

void CloseList(HLLIST hLLList)
{
    free(hLLList);
}

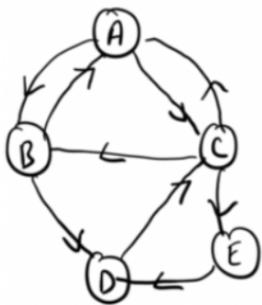
```

Graf Veri Yapısı

Graflar bir düğüme birden fazla düğümden gelinebilen 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 göründürsünüz:



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:

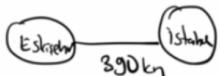


Burada örneğin B'den A'ya, B'den D'ye bir yol olduğu halde B'den C'ye bir yol yoktur. Aslında yönsüz grafları çift yolu olarak da düşünebiliriz. Gerçek hayatı her iki tür grafla da sıkılıkla karşılaşılmaktadır. Örneğin kara yollarının beli bir bölümü yönlü grafla modellenebilir. Ancak örneğin sosyal ağlardaki arkadaşlık bağlantıları yönsüz bir graf biçiminde modellenebilir. (Yani örneğin Facebook'ta X Y'nin arkadaşıysa Y de X'in arkadaşıdır. Ancak takip sistemi tek yönlü grafla modellenebilir.)

Bir grafi düğümler ve kenarlar (yolla) 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 kata yolu bağlantısı) ve aralarındaki bu bağlantıının 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 hayatı en fazla karşılaşılan veri yapılarından biridir. Dolayısıyla gerçek hayat sorunlarını bilgisayar ortamında çözmek için grafların bir yapısı biçiminde modellenmesi gereklidir. Örneğin gerçek hayatı karşılaşılan bazı graf veri yapıları şunlardır:

- Bir navigasyon programında herhangi ik yol arasındaki en kısa mesafenin bulunması istensiz. Aslında ollar grafin kenarlarını gidilecek merkezler de düğümlerini oluşturmaktadır. Dolayısıyla bu en kısa yol problemi için önce bir graf veri yapısının oluşturulması gereklidir.
- Bir dağıtıçı 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 gereklidir. Bu problem de ancak bir graf veri yapısı kurularak çözülür.
- Bir sosyal ağıda belli bir kişiye kaç kişiden geçilerek ulaşılabilceği hesaplanması istenmektedir. Sosyal ağlardaki ilişkiler de graf veri yapısıyla modellenebilir.
- Bir elektrik devresinde devre elemanları tellerle birbirlerine bağlanmıştır. Bu devre şeması da yine bir graf veri yapısı ile modellenebilir.
- Akrablık ilişkileri bir graf veri yapısı ile modellenebilir.

- Kablolarla oluşturulmuş bilgisayar ağları da graflarla modellenebilir.

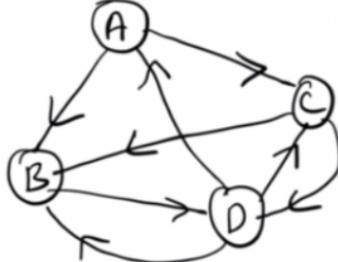
- Ansiklopedik pek çok kavram graflarla modellenmektedir. Çünkü bir kavram pek çok konu ile ilgilidir. Örneğin Wikipedia'da yazılar kategorilerin içerisinde bulunur. Kategoriler de başka kategorilerin içerisinde bulunabilmektedir. Örneğin "dil" bir bir kavram pek çok kategorinin konusudur. Yani "dil" 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

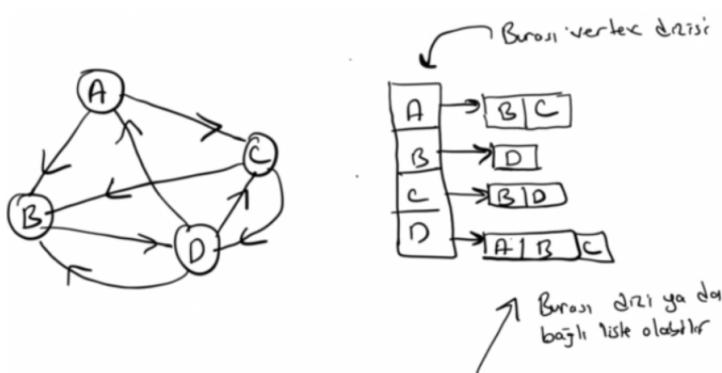
Komşuluk matrisi graftaki hangi düğümlerin hangi düğümlerle bağlantılı olduğunu temsil eden bir matristir. Eğer graf yönüzse bu matris simetrik olur. Yönlü ise simetrik olmayıpabilir. Örneğin:



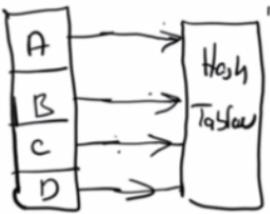
$$\begin{array}{c} \begin{matrix} & A & B & C & D \\ A & 1 & 1 & 1 & 0 \\ B & 0 & 1 & 0 & 1 \\ C & 0 & 1 & 1 & 1 \\ D & 1 & 1 & 1 & 1 \end{matrix} \end{array}$$

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 düğüme bağlantılı olduğu bir dizi ya da bağlı liste ile değil bir hash tablosuyla da tutulabilir. Bu durumda belli bir düğümün belli bir düğüme bağlantılı olup olmadığı hash tablosu sayesinde çok daha hızlı tespit edilebilmektedir. Örneğin:



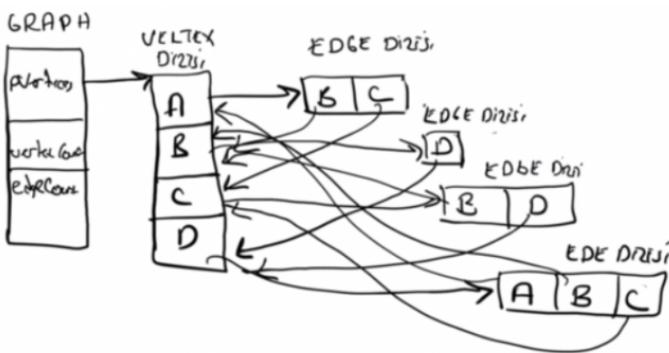
Burada biz komşuluk listesi yönteminin gerçekleştirimini yapacağız. Komşuluk listesi yönteminin gerçekleştirmi çeşitli biçimlerde yapılabilir. Genel bir gerçekleştirim için düğümlerin ve kenarların ayrı yapılar biçiminde oluşturulması uygun olur. Çünkü genel olarak düğümler ve kenarlar başka birtakım bilgiler içerebilmektedir. Graf yaratılırken grafta kaç düğüm olacağı ve hangi düğümlerde hangi düğümlere bağlantı olacağı biliniyorsa gerçekleştirim daha kolay yapılabilir. Ancak bu işin başında bilinmiyorsa graf veri yapısına düğüm ve kenar ekleme fonksiyonlarının yazılması gereklidir. Bu da 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 düğümünün hedefini bize verir. İşte bu tür aramaların daha etkin olması için (örneğin isme dayalı arama yapılıyor 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 uygunudur. Çünkü graf yaratıldığından genellikle düğüm sayısı bilinir ancak kenar sayısı baştan bilinmeyebilir. Bu durumda EDGE dizilerinin bağlı liste ya da dinamik dizi biçiminde olması daha uygunudur. Eğer işin başında düğüm sayıları da bilinmiyorsa

VERTEX dizisinin de benzer biçimde binamik 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 yazılabılır:

```
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 ilişirelim. 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 içerdigini 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;
}

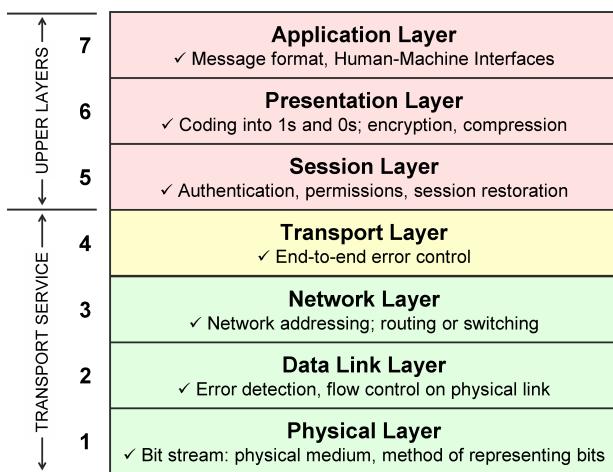
```

Farklı Makinelerin Prosesleri Arasında Haberleşme

Farklı makineler birbirlerine ağ içerisinde bağlanmış olabilir. Biz bir makinede çalışan bir programın 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 kablolama sisteminden kullanılan hub'a kadar bazı donanım birimleri işin içine karışmaktadır. Üstelik bu tür haberleşmelerde işletim sistemleri bile birbirlerinden farklı olabilmektedir. İşte hetorejen böyle ortamlarda haberleşmenin sağlıklı yürütülmesi için önceden belirlenmiş birtakım kuralların bulunması gereklidir. Örneğin kablo standartları ve konnektörler nelerdir? Network kartının özellikleri nasıl olacaktır? Bilgiler nasıl paketlere ayrılp gönderilecektir? Makineler nasıl birbirlerinden ayrılacaktır vs. gibi... İşte tüm bu belirlemelere protokol denilmektedir.

Tıpkı fonksiyonlarının birbirlerini çağırarak daha yüksek seviyeli işlemleri yapar hale gelmesi gibi protokoller de üst üste yiğilerek ayrı ayrı oluşturulmaktadır. Her üst protokol aşağıdaki zaten hazır olduğu fikriyle yalnızca kendi gereksinimlerini tanımlamaktadır. Böyle katmanlı tasarımin pek çok faydası vardır. Örneğin bu sayede üst seviye protokoller detay barındırmazlar ve aşağı düzeydeki protokollerin değişmesinden fazlaca etkilenmezler. İşte farklı makinelerin haberleşmesi için bu biçimde oluşturulmuş pek çok protokol ailesi vardır. Örneğin AppleTalk, NETBIOS vs. gibi...

Network altında bilgisayar haberleşmesi için protokol katmanlarının nasıl oluşturulması gerekiğine yönelik IEEE, ismine OSI (Open System Interconnection) denilen bir doküman yayımlanmıştır. Buna OSI model denilmektedir. OSI model bir protokol ailesi değildir. Protokol ailesi oluşturacaklar için bir kılavuz niteliğindedir. OSI'nin toplam 7 katmanı vardır:



OSI'nin en aşağı katmanına "Fiziksel Katman (Physical Layer)" deilmektedir. Fiziksel katmanda iletişimini yapılacak ortam tanımlanmaktadır. Örneğin kullanılacak kablolar, konnektörler, gerilim seviyeleri gibi. Bunun üzerinde "Veri Bağlantı Katmanı (Data Link Layer)" bulunmaktadır. Bu katmanda network kartlarına ilişkin belirlemeler, fiziksel adresleme belirlemeleri vs. bulunmaktadır. Örneğin Ethernet kartlarının protokolü olan Ethernet Protokolü bir Veri Bağlantı Katmanı Protokolüdür. Network katmanı (Network Layer) mantıksal adreslemenin tanımlandığı, bilginin nasıl paketlere ayrılp gönderileceğinin tanımlandığı en önemli katmanlardan biridir. Örneğin IP protokol ailesinin IP Protokolü (Internet Protocol) OSI'ye göre Network katmanına ilişkindir. Network katmanında ayrıca "internetworking" için rotalama belirlemeleri de bulunmaktadır. Network üzerinde "İleti Katmanı (Transport Layer)" bulunmaktadır. Burada paketlerin numaralandırılması, mantıksal port adreslerinin tanımlanması, hata durumunda bunun telafi edilmesi gibi belirlemeler bulundurulmaktadır. Örneğin IP protokol ailesindeki TCP ve UDP protokollerini ileti Katmanına ilişkin protokollerdir. "Oturgum Katmanı (Session Layer)" pek çok ailede bulunmamaktadır. Burada haberleşme için gereken oturum açmaya yönelik belirlemeler bulunur. Örneğin izinler, kimlik doğulama gibi. Bunun yukarıısında da "Sunum Katmanı (Presentation Layer)" bulunmaktadır. Sunum katmanında gönderiliip alınına bilgilerin sıkıştırılmasına, açılmasına, şifrelenmesine vs. yönelik belirlemeler bulunmaktadır. IP protokol ailesi Sunum Katmanına da sahip değildir. Nihayet en tepede "Uygulama Katmanı (Application Layer)" bulunmaktadır. 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 Protolüdür.

Internet'in Kısa Tarihi

Bilgisayarları birbirlerine bağlamak ilk kez 60'lı 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 hayatı geçirilmiş oldu. ARPANET'te 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 arık ağ Internet ismiyle yayılmaya devam etmiştir. Internet 80'lı yıllarda Avrupa'ya ve Türkiye'ye de geldi. Ancak tabi kişisel bilgisayarlar daha yeniydi ve Internet'e ancak Üniversitelerden ve bazı devlet kurumlarından, özel sektörden bağlanılabiliriyordu. 1990-91 yıllarında HTTP protokü tasarlandı ve ilk Web sayfaları oluşturulmaya başlandı. 90'lı yılların ortalarına doğru tüm dünyada kişisel bilgisayarlarla servis sağlayıcılar sayesinde Internet'e girmek mümkün hale gelmiştir. Daha sonraları modern modem/router'larla yüksek hızlı evden erişimler sağlanmıştır.

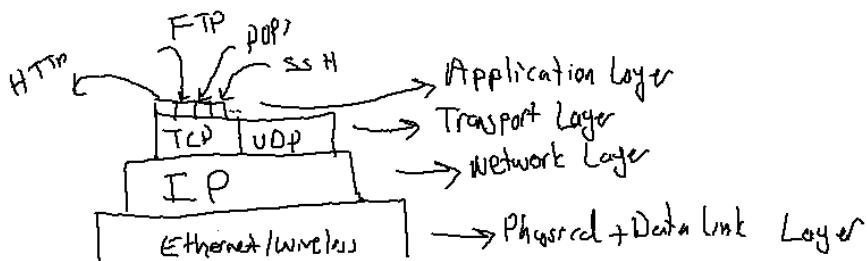
Internet ismi "internetworking" sözcüğünden gelmektedir. Internetworking "yerel ağların birbirlerine router isimli cihazlarla bağlanmalarıyla oluşturulmaktadır. Internetworking temel bir terimdir ve IP protokol ailesinin ismi buradan gelmektedir. Bugün Internet denildiğinde herkesin bağlandığı ARPANET'ten evrimleşen dev ağ aklıma gelir. (Internet yazarken I'yı büyük yazarsak bu ağ anlaşılır.) Şüphesiz mevcut protokoller sayesinde herkes kendi internetini kurabilir. Örneğin biz de birkaç arkadaşımızla ayrı bir Internet dünyası oluşturabiliriz. Hatta bazı ülkelerin bu biçimde kendilerine özgü Internet'leri vardır.

IP Protokol Ailesi

IP açık bir protokol ailesidir. Burada açık demekle hiçbir şirketin malının olmadığı bağımsız konsorsiyumlar tarafından yönetildiği anlamına gelmektedir. Ayrıca dokümanlar herkes tarafından paylaşılmakta ve isteyen kişiler önerilerde bulunabilmektedir.

IP protokolü Vint Cerf ve Bob Kahn tarafından 1974 yılında önce TCP sonra IP biçiminde tasarlanmıştır. Sonra aileye diğer üyeleri katılmıştır. İlk ciddi gerçekleştirmi BSD sistemlerinde yapılmıştır. 1983 yılında ARPANET'in IP ailesine geçmesiyle popüleritesi çok artmıştır.

IP protokol ailesinin temel protokollerini dört katmandan oluşmaktadır.

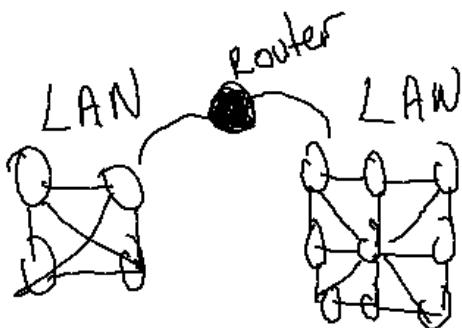


IP protokol ailesi aslında geniş bir ailedir. Ailede pek çok yardımcı protokol vardır. Yukarıdaki şekil yalnızca kursumuzda söz konusu edilen konuları kapsayacak biçimde oluşturulmuştur.

Ailenin en önemli taban protokolü IP (Internetworking Protocol) protokolüdür. Zaten aileye ismini bu protokol vermiştir. IP protokolü paket anahtarlama (packet switching) bir protokoldür. Yani bilgiler paket denilen öbeklere ayrılarak gönderilip alınır. IP protokolünde adresleme artık 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 adres bunun donanımsal olarak belirlenmediği yazılımsal olarak atandığı anlamına gelmektedir. Fakat örneğin Ethernet protokolünün kullandığı MAC adresi fiziksel bir adresdir. Fiziksel adres bunun donanımsal olarak kartın üzerine çakılı olduğu ya da donanımın kendisinin bunu tespit edip işlem yaptığı adres demektir. Dolayısıyla mantıksal adresler dinamiktir, fiziksel adresler statiktir. Mantıksal adresler biz ağa dahil olduğumuzda bize atanmaktadır. Tabi biz de istediğimiz adresin atanması konusunda israrçı olabiliriz.

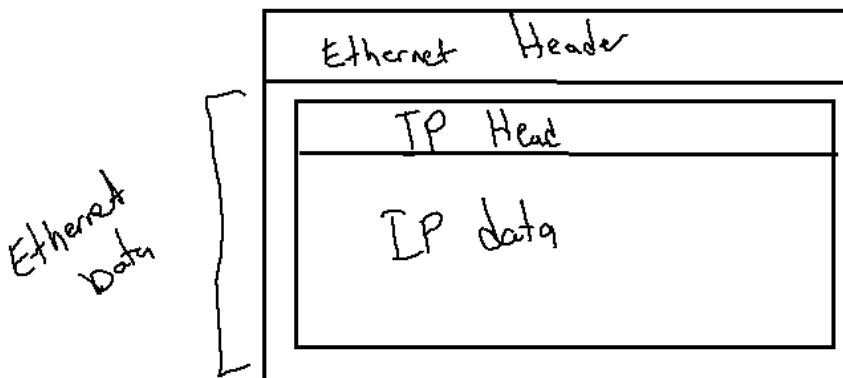
IP protokolünün de versiyonları vardır. Şu anda hala ağırlıklı kullanılan versiyon IPV4'tür. Ancak IPV6 yavaş yavaş daha yaygın kullanılır hale gelmiştir. IPV4'te IP adresleri 4 byte uzunluktadır. Ancak IPV6'da IP adresleri 16 byte'tır. 4 byte'lık IP adresleri şu an için artık çok yetersiz kalmaktadır.

Bugün bilgisayarlarımıza fiziksel ve data link katmanı olarak Ethernet ve Wireless Protokollerini kullanılmaktadır. Ethernet protokolü ethernet kartına gereksinim duyar. Bu kart fiziksel olarak bilgileri bilgisayarımızdan dışarı gönderip almakta kullanılır. Ethernet protokolü de paket anahtarlamalı bir protokoldür. Yani bilgiler paket paket gönderilip alınır. Paket anahtarlama hattın etkin kullanımını sağlar. Biz Ethernet kartlarını bir hub'la biribirine 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 bulunduğu 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 standıyla 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 oluşurumamız gereklidir. Zaten TCP protokolü buna benzer bir protokoldür.

TCP protokolü güvenli (reliable) bir protokoldür. Burada güvenlik demek alış verisin 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 tefafisini sağlayabilmektedir. TCP stream tabanlı bir protokoldür. Stream tabanlı demekle byte byte okumaya kaldığı yerden devam edebilmek anlaşılır. TCP ile biz daha büyük bilgileri gönderip alabiliyoruz. 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. Tabii bu özelliğinden dolayı UDP daha hızlıdır. UDP özellikle periyodik data gönderimlerinde, televizyon yayını gibi işlemlerde tercih edilmektedir.

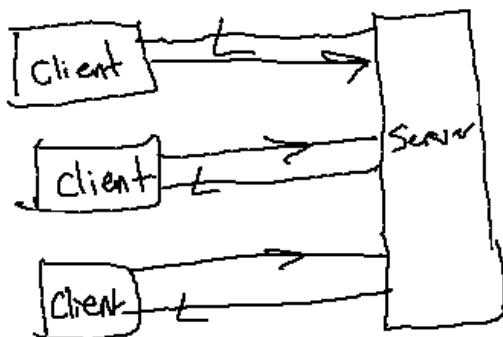
TCP bağlantılı (connection oriented) bir protokoldür, UDP bağlantısızdır (connectionless). Bağlantılı protokol demek iki taraf haberleşmeden önce birbirlerine bağlanıp karşılıklı konuşma için birbirlerini tanımları demektir. TCP tipik olarak client-server tarzda bir çalışmayı akla getirmektedir. Client-server 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 uygulamanının hangi port ile ilgilendiğinin de bilinmesi gereklidir. 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 ayrılr). IPv6'da ise port numaraları 4 byte uzunluğundadır. IPv4'te ilk 1024 port numarası Internet'in kendi uygulama protokelleri 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 belirleyeceğsek 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 çağrıstırıysa da aslında bu bir haberleşme mimarisidir. Yani aslında client-server çalışma için IP ailesinin

kullanılması gerekmektedir. Bu çalışma örneğin 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 olabiliz. 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 iletilir. 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 davranışmaktadır.
- 3) Server program client'lar arasında işbirliği sağlayabilir. Onlar arasındaki iletişime aracılık edebilir. Örneğin bir char programında client'lar birbirini tanıtmamaktadır. Herkes yalnızca server'i tanır. Her client server'a bağlanır. Server client arasında haberleşmeye aracılık eder. Ağ üzerinde çalışan oyun programları bu biçimde bir server'in işbirliği ile gerçekleştirilmektedir.
- 4) Client-Server çalışma dağıtık uygulamalarda da karşıımıza çıkabilecektir. 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 gerçekleştirılmıştır. 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 protokoller de kapsayan genel bir arayıldır. Bu nedenle fonksiyonların parametrik yapıları biraz daha karmaşık olma eğilimindedir.

TCP/IP İskelet Client-Server Programlarının Yazımı

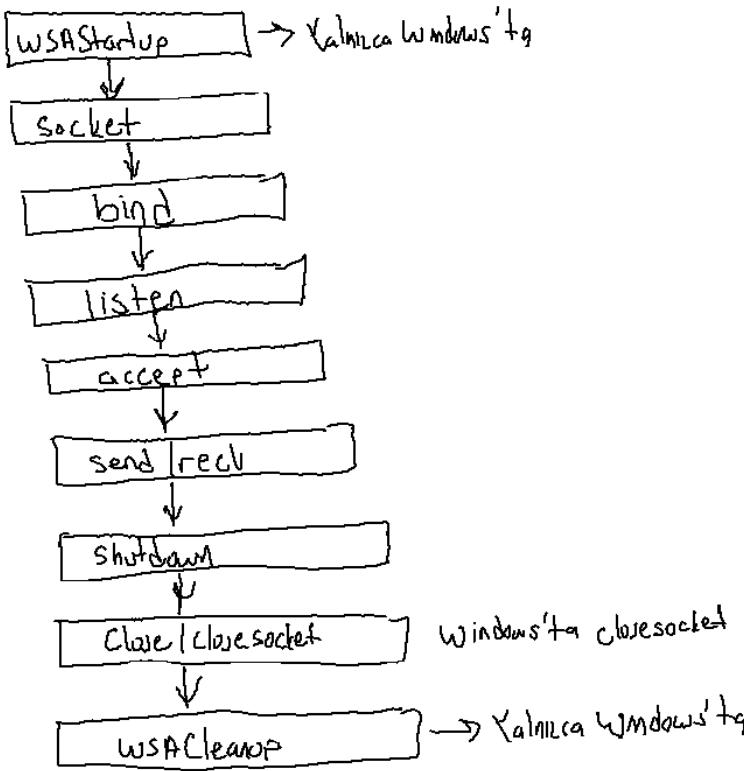
Burada iskelet bir TCP/IP client-server programın yazımı ele alınacaktır. Uygulamada klasik BSD soket fonksiyonları kullanılacaktır. Windows'a özgü farklılıklara konu içerisinde değinilmektedir.

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 dosyası içerisinde yer almaktadır. Soket uygulamaalarında 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 gereklidir. Ayrıca Windows'ta son soket fonksiyonu hatayla geri dönmişse 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 prosese bir kez yapılmak zorundadır. WSAStartup fonksiyonun prototipi şöyledir:

```

int WSAStartup(
    _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 içini 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 anlaşılr. 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 fonksiyonun 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 ikinci parametresi genel bir tür olarak struct sockaddr * türündendir. (Eski void göstericiler yoktu bu parametre bu nedenle genel bir tür 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];
};
```

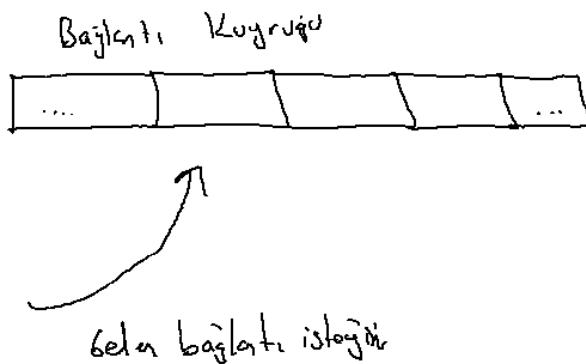
Yapının sin_family elemanı kullanılan protokol ailesini belirtir. IP ailesi için bu elemana AF_INET girilmelidir. Yapının sin_port elemanına server'in dinlemek istediği port numarası girilmelidir. Yapının sin_addr elemanına da server'in dinlemek istediği network arayüzünün (kartının) IP numarası girilmelidir. 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 gereklidir. 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 artrık server aktif dineme 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 isteyi geldiğinde o isteği bir bağlantı kuyruğuna ekler. Artık server program da bağlantı isteklerini kuyrukta alacaktır.



Listen fonksiyonunun bağlantıyi 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 SOKET_ERROR değerine geri döner. Örneğin:

```
if (listen(serverSock, 8) == SOCKET_ERROR)  
    ExitSys("listen", EXIT_FAILURE, WSAGetLastError());
```

Şimdi sıra artık accept işlemeye gelmiştir. Accept fonksiyonu kuyrukta sıradaki bağlantı isteğini alır ve bağlantıyı sağlar. Eğer kuyukta hiçbir bağlantı yoksa blokeli modda bağlantı isteyi 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 soketinin handle değerini alır. İkinci parametre bağlanılan client'in 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 konuşmaka kullanılacak soketin handle değerine geri döner. Başarısızlık durumunda da INVALID_SOCKET (UNUX/Linux sistemlerinde -1) değerine geri dönmektedir.

Göründüğü gibi her accept işlemi bize bağlanılan client ile konuşmaka kullanılabilecek yeni bir soket vermektedir. Yani server'ın bir tane dinleme soketi vardır. Bununla listen ve accept yapar. Ancak her accept işleminde yeni bir soket 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'in IP numarasını ve port numarasını sockaddr_in yapısından alabilir. Örneğin:

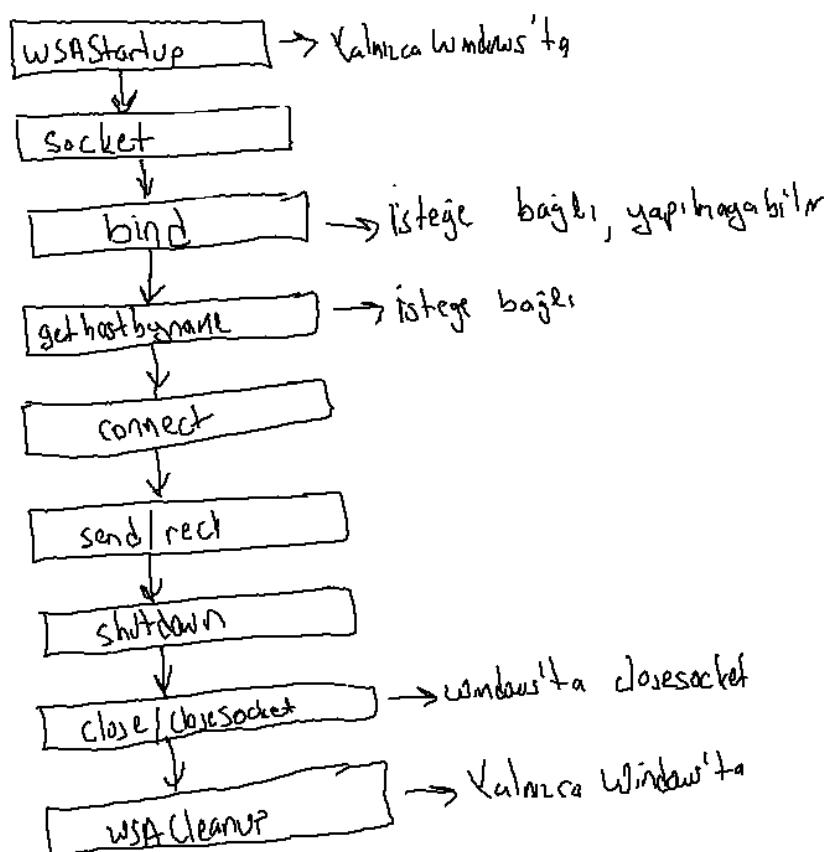
```
printf("Client connected: %s:%d\n", inet_ntoa(sinClient.sin_addr), ntohs(sinClient.sin_port));
```

Burada inet_ntoa, IP adresini noktalı formatta yazıya dönüştürmektedir. ntohs ise htons fonksiyonunun ters işlemini yapar.

Bağlantı sağlandıktan sonra artık send/recv 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:



Gördüğü gibi client da işin başında bir soket yaratır. Soket'in bind edilmesi server'da zorunludur. Ancak client soketi bind etmeyebilir. Bu durumda işletim sistemi kaynak port olarak rastgele bir port numarası atar. Bağlantıda kaynak soketin de hedef soketin 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. Internet'te host isimleri ile IP adreslerini eşleştiren veritabanları bulunmaktadır. Bunlara "Doman 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 gereklidir. Bunu gethostbyname isimli soket API fonksiyonu yapar. Bu fonksiyonun ters işlemini yapan gethostbyaddress isimli bir fonksiyon da vardır:

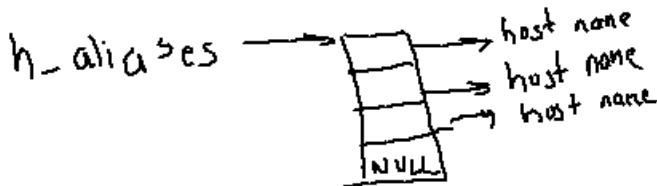
```
struct hostent *gethostbyname(
    const char *name
);

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;
```

Yapınının h_name elemanı domain host ismini vermektedir. h_aliases elemanı diğer host isimlerini verir.



Yapınının 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 yazılı 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 yazılı önce bu fonksiyona sokup eğer yazılı 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 soketin 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 içerisinde 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];
};
```

Göründüğü gibi IP adresi yapının sin_addr elemanındadır. 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;
```

Göründüğü 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 = gethostname(SERVER_NAME)) == NULL)
        ExitSys("gethostname", 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 soket verir.

Client İle Server Arasında Bilgi Alış Verisi 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 belirlemekte kullanılır. Bu parametre sıfır geçilebilir. Fonksiyon blokeli modda network tamponuna aktarılan byte sayısı ile geri döner. Network tamponu doluya fonksiyon tampona yazıldığı 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 doluya send tüm bilginin tampona yazılmasını beklemeyebilir. Yazıldığı kadarını yazıp o değere geri dönebilir. (Bazı sistemlerde send tüm bilgi yazılsa kadar blokede kalabilmektedir. O sistemlerde bile çok büyük bilgiler tam olarak tek seferde tampona yazılamayabilir.) Fakat send eğer netowork tamponu tıka basa doluya 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'in 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'i tek bir recv ile okuyamayabiliz. 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 karşık 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 netowork 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 İşlemleri

Soket bir handle sistemidir. Dolayısıyla tüm handle sistemlerinde olduğu gibi işimiz bitince soketleri de kapatmalıyız. Tabii biz soketi kapatmamışsa 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 çağrılmak tavsiye edilir. Önce shutdown sonra close işlemine "graceful close" denilmektedir. shutdown fonksiyonu ile biz soketin kapatılması için hazırlık yaparız. Böylece TCP düzeyinde iletişim kesilmesi için bir el sıkışma gerçekleşir. shutdown fonksiyonunun prototipi şöyledir:

```
int shutdown(
    __in      SOCKET s,
    __in      int how
);
```

Fonksiyonun birinci parametresi soketin 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 yaptığından sanki soketi kapatılmış gibi görür. Fakat bu sırada SD_SEND uygulamış taraf soketten okuma yapabilir. SD_RECEIVE ise "ben artık soketten okuma yapmayacağım, fakat gönderme yapabilirim" anlamına gelir. SD_BOTH her iki durumu da kapsamaktadır. İşte soket önce el sıkışmalı biçimde shutdown ile sonra da close uygulanarak kapatılmalıdır. Server'ın dinleme soketi için shutdown uygulamasına gerek yoktur.

Windows sistemlerinde iskelet bir TCP/IP Client server program söyle yazılabılır:

```
/* 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 = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0)
        ExitSys("WSAStartup", 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("WSASStartup", EXIT_FAILURE, result);

if ((clientSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == SOCKET_ERROR)
    ExitSys("socket", EXIT_FAILURE, WSAGetLastError());

#ifndef _WIN32_WCE
#endif

#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("WSASStartup", 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 programın Linux sistemlerinde eşdeğeri de şöyle yazılabılır:

Çok Client'lı Uygulamalar

Normal olarak server'lar aynı anda birden fazla client'a hizmet verebilecek biçimde tasarılanırlar. Bu durum server programlarının yazılmasını biraz karmaşık hale getirmektedir. Öncelikle çok client'lı server programlar yazılrken yalnızca

bir tane accept uygulanmaması gereklidir. 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'in 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 göndermemiştir olsun. Server akışı blokede kalacağından dolayı server artık diğer client'larla konuşamaz hale gelir.) Peki server'in 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 edecek 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("CreateThread", EXIT_FAILURE, GetLastError());
}
}
```

Göründüğü gibi Thread fonksiyonuna soket 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 soket geçirilmiştir. Eğer thread fonksiyonuna soketin 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("CreateThread", 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'in 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 kullanılmaktadır. 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 soket 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 soket işlemlerinde kullanılmaktadır.)

select (ve poll) kullanımı şöyledir: Biz soketleri bir dizide toplayıp select fonksiyonuna veririz. select bu soketleri kendisi izler. Eğer onların hiçbirinde bir IO olayı yoksa bizi blokede bekletir. Eğer soketlerin bir ya da birden fazlasında bir IO olayı gerçekleşmişse select blokeyi çözer. Biz de hangi soketlerde hangi IO olayının gerçekleştiğini sorgularız. Yalnızca onlar üzerinde blokeye yol açmadan işlem yaparız. Örneğin biz select'e 100 client'in soketini vermiş olalım. O sırada bir client'tan bilgi gelmiş olsun. select blokeyi çözükcektir. Biz de o soketten okuma yaparsak hiç bloke oluşmayacaktır. Tabii select bir kez çağrılabilecek 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 olabiliz. 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. Tabi çağrıma 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 serve 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 gönderilicek 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"
```

Gördüğü 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'in server'a fiziksel olarak bir porttan bağlanmış olması server'in onu kabul edeceği anlamına gelmemektedir. Bazen bu iki kavram "fiziksel bağlantı" ve "mantıksal bağlantı" biçiminde de ifade edilebilmektedir.

Komutlar ve yanıtlar yazışal olarak gönderilip alınırken alan taraf yazının sonunu nasıl bileyecktir? 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 yerlestirmesi ve o tamponda sonlandırıcı karakteri araması daha uygun olur. İkinci yöntemde her yazının uzunluğunun yazıldan önce gönderilmesidir. Yazı uzunluğu binary olarak gönderilebilir.

IP protokol ailesinin uygulama katmanındaki FTP, POP3, TELNET, HTTP gibi protokoller text tabanlı mesajlaşma yöntemini kullanmaktadır. Bu yöntemde client server'a isteğini yukarıda belirtildiği gibi yazı biçiminde gönderir. Yazının sonu CR/LF karakterleriyle sonlandırılmaktadır. Soketten CR/LF karakterlerini görene kadar tamponlu okuma yapan bir fonksiyon şöyle yazılabılır:

```
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 by byte almaz. Paketi tümden 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 iş 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 Programın Organizasyonu

Tipik bir UDP server program şöyle oluşturulmaktadır:



Server hangi port'tan gelen bilgileri alacağını yine bind işlemi ile belirler. recvfrom her türlü client'in 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 soketin handle değerini, ikinci parametresi UDP paketinin yerleştirileceği adresi alır. Üçüncü parametre bu dizinin uzunluğunu belirtir. Eğer paket daha büyüğe böylece dizi taşmaz, kirpilerak diziye yerleştirilir. Dördüncü parametre 0 geçilebilir. Fonksiyonun beşinci parametresine sockaddr_in yapısının adresi girilir.

Böylece paketin kimden geldiği anlaşılmaktadır. Fonksiyonun son parametresi bizim ve karşı tarafın sockaddr yapısının byte uzunluğunu almaktadır. Biz fonksiyonu çağrımdan önce buraya kendi sockadd_in yapımızın sizeof değerini girmemiz gereklidir. 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 soketinin yaratılırken protokolün UDP olarak belirlenmiş olması gereklidir. 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 = WSAStartup(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);

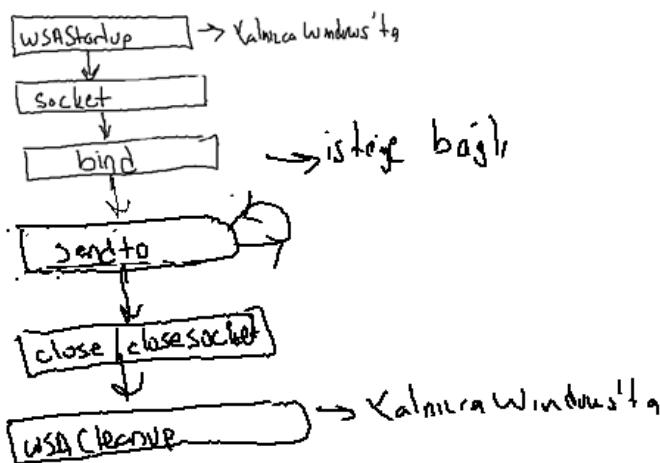
    WSACleanup();

    return 0;
}
```

}

UDP Client Programın Organizasyonu

UDP/IP haberleşmede geleneksel olarak paketi gönderen tarafa client denilmektedir. İskelet bir client program şöyle oluşturulur:



sendto fonksiyonun prototipi şöyledir:

```
int sendto(
    SOCKET s,
    const char *buf,
    int len,
    int flags,
    const struct sockaddr *to,
    int tolen
);
```

Fonksiyonun birinci parametresi soketin 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 sockaddr_in yapısının adresini alır. Son parametre yine bu sockaddr_in yapısının bye 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 = WSASocket(AF_INET, SOCK_DGRAM, IPPROTO_UDP, &wsaData)) != 0) {
    fprintf(stderr, "WSASocket 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, "gethostbyname 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;
}

```

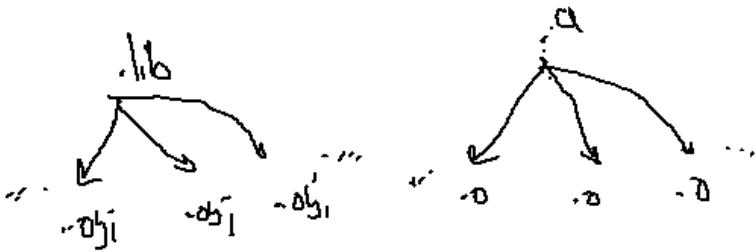
Kütüphanelerin Oluşturulması ve Kullanılması

İçerisinde derlenmiş bir biçimde fonksiyonların bulunduğu dosyalara kütüphane (library) denilmektedir. Kütüphaneler statik ve dinamik olmak üzere ikiye ayrılırlar. Windows'ta statik kütüphane dosyalarının uzantıları .lib (library), UNIX/Linux sistemlerinde .a (archive) biçimindedir. Benzer biçimde dinamik kütüphanelerin de uzantıları Windows sistemlerinde .dll (dynamic link library), UNIX/Linux sistemlerinde .so (shared object) biçimindedir.

Kütüphane dosyası herhangi bir dilde yazılıp derlenerek oluşturulabilir. Ancak en çok karşılaşılan durum bunların C/C++ gibi dillerde oluşturulmuş olmasıdır. .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 dll .NET dünyası için oluşturmuş bir dll olabilir ya da doğal kod içeren bir dll olabilir. Biz kursumuzda doğal kod içeren kütüphaneleri ele alacağız.)

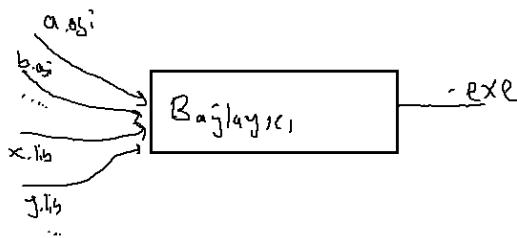
Statik Kütüphaneler

Statik kütüphane dosyası Windows'ta ve UNIX/Linux sistemlerinde aslında object modül dosyalarını tutan bir dizin gibidir.



Object modüller içerisinde de derlenmiş fonksiyonlar bulunmaktadır. Biz C'de bir grup fonksiyonu bir kütüphane içerisinde yerleştirmek istersek önce onları bir kaynak dosya içerisine yazıp derleriz ve object modül elde ederiz. Sonra bu object modülü kütüphane dosyası içerisine yerleştiririz.

Normalde bağlama işlemine girdi olarak birden fazla object modül ve statik kütüphane dosyası sokulabilmektedir. Örneğin:



Biz C'de bağımsız olarak derlediğimiz object modüllerini ve onların kullandığı fonksiyonların bulunduğu kütüphane dosyalarını linker programına girdi olarak verebiliriz. Linker bizim verdığımız object modüllerini birleştirir. Bu object modüllerden çağrılmış olan ve kütüphaneler içerisinde bulunan fonksiyonları tespit eder. O fonksiyonların içinde bulunduğu object modülleri de kütüphane dosyalarının içinden alarak onu da .exe dosyaya ekler. Böylece biz bir statik kütüphaneden tek bir fonksiyon çağrılmış olsak bile o fonksiyonun içinde bulunduğu object modülün tamamı çalıştırılabilen dosyaya eklenecektir. Görüldüğü gibi statik kütüphanelerden bir çağrı yapıldığında çağrılan fonksiyon oradan alınarak çalıştırılabilen dosyanın içerisine yerleştirilmektedir. Böylece program çalışırken artık statik kütüphaneye gereksinim duyulmamaktadır.

Statik kütüphaneler tek parça bir çalıştırılabilen program oluşturduğu için kullanım ve konușlandırma (deployment) bakımından bazı durumlarda avantaj sağlamaktadır. Ancak statik kütüphaneler çalıştırılabilen dosyaları büyütme eğiliminde olduğu için toplamda disk alanı bakımından bir dezavantaja sahiptir. Ayrıca ileride de ele alınacağı gibi dinamik kütüphaneler kodların ortak kullanımını mümkün hale getirebildiğiinden bir optimizasyona da yol açabilmektedir.

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 sağlayabiliyoruz.

Anahtar Notlar: Microsoft'un C ve C++ derleyicisinin ismi cl.exe'dir. Linux'taki ağırlıklı kullanılan C derleyicisi ise gcc'dir. 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'nun ayrı bir C/C++ derleyicisi yoktur. Visual Studio derleme sırasında cl.exe derleyicisini kullanmaktadır. cl.exe default durumda derlemeyi yaptıktan sonra Microsoft'un bağlayıcı programı olan "link.exe" programını ürettiği object kodları ve standart kütüphaneleri girdi yaparak çalıştırmaktadır. Ancak biz istersek /c ya da -c seçeneği (only compile) ile cl.exe'nin link.exe'yi çalıştırmasını engelleyebiliyoruz. Benzer biçimde aynı çalışma sistemi Linux'ta da böyledir. Linux'un C derleyicisi gcc'dir. gcc default durumda Linux'un bağlayıcı programı olan "ld" programını çalıştırır. Eğer biz bunu istemiyorsak yine gcc'yi -c seçeneği ile (only compile) çalıştırılmalıdır.

Soru: Windows'ta Microsoft'un C derleyici ile komut satırında sample.c programı nasıl derlenir ve link edilir?

Yanıt: Aşağıdaki gibi:

```
c1 sample.c
```

Tabii biz istesek cl.exe'ye birden fazla kaynak dosya verebiliriz. Bu durumda cl.exe onları bağımsız olarak derler. Tüm object dosyaları link.exe'ye girdi olarak verir. Örneğin:

```
c1 a.c b.c c.c
```

Soru: Windows'ta cl.exe "link.exe" bağlayıcısını çalıştırırken girdi olarak ona ne verir?

Yanıt: cl.exe'den üretilen object modülleri ve standart C kütüphanesini ve Windows'un bazı API'lerinin bulunduğu temel kütüphaneleri

Soru: Biz gerek Windows'ta gerekse Linux'ta derleme ve bağlantı işlemlerini ayrı ayrı yapabilir miyiz?

Yanıt: Evet. Bunun için derleyiciyi -c seçeneğiyle (only compile) çalıştırıp object dosyası elde etmek ve sonra yine "link.exe" ya da "ld" bağlayıcısını komut satırından çalıştmak gereklidir.

Soru: Visual Studio IDE'sinin cl.exe'yi ve link.exe'yi hangi seçeneklerle çalıştırıldığını anlayabilir miyiz?

Yanıt: Evet bunun için "Tools/Options/Projects and Solutions/Build And Run/Ms Build project output verbosity" menüsü kullanılabilir.

Statik Kütüphanelerin Oluşturulması

Statik kütüphane dosyaları Windows'ta Microsoft'un "lib.exe" isimli yardımcı programıyla oluşturulur. lib.exe ile sıfırdan bir static kütüphane dosyası yaratıp içerisinde eklemeler şöyle yapabiliriz:

```
lib /OUT: MyStaticLib.lib A.obj B.obj C.Obj
```

Statik kütüphanenin içerisindeki object modülleri /LIST seçeneği ile görüntüleyebiliriz:

```
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 Visusal 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 bizim projeye eklediğimiz .c dosyaları derlenir, bunlar object modül haline dönüştürülür. Sonradan lib.exe programı çalıştırılarak .lib dosyası oluşturulur.

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 düşününebiliriz.) ar programı ile bir statik kütüphane dosyası yaratıp (c seçeneği) içerisinde object modül ekleme (r seçeneği) işlemi şöyle yapılmaktadır:

```
ar cr mystaticlib.a a.o b.o c.o
```

"c (create)" seçeneği eğer kütüphane dosyası yoksa yaaratmak 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 kullanılabilir:

```
ar t mystaticlib.a
```

Kütüphane içerisindeki bir object modülü silmek için ise "d (delete)" seçeneği kullanılmaktadır. Örneğin:

```
ar d mystaticlib.a o.o
```

ar programının diğer pek çok seçenekleri vardır. Bunlar ilgili dokümanlardan öğrenilebilir.

Statik Kütüphanaların Kullanılması

Yukarıda da belirtildiği gibi statik kütüphane dosyalarına "bağlayıcı (linker)" bakmaktadır. Bağlayıcı object modülü inceleyerek derleyicinin bulamadığı fonksiyonları kütüphane dosyalarında arar. Eğer onları kütüphane dosyalarında bulursa oradan çekerek çalıştırılabilen (executable) dosyaya yazmaktadır. Fakat bağlayıcı programlar yalnızca standart C fonksiyonlarının bulunduğu kütüphane dosyalarına otomatik biçimde bakmaktadır. Programcı bağlayıcının kendi statik kütüphane dosyalarına da bakmasını istiyorsa bunu açıkça belirtmelidir.

Microsoft'un "cl.exe" derleyicisinde komut satırında .lib dosyaları belirtilirse cl.exe derleyicisi bunları "bağlayıcı (link.exe)" programı çalıştırırken ona komut satırı argümanı yapar. Böylece biz cl.exe ile komut satırında kendi kütüphanemize bakılacak biçimde derlemeyi aşağıdaki yapabiliriz:

```
cl App.c MyStaticLib.lib
```

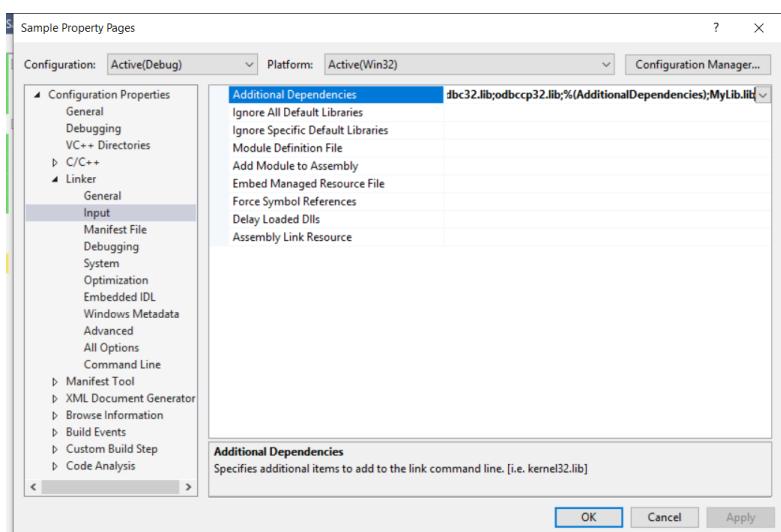
Tabii biz bu işlemi iki aşamada da aşağıdaki gibi yapabiliyoruz:

```
cl /c App.c  
link App.obj MyStaticLib.lib
```

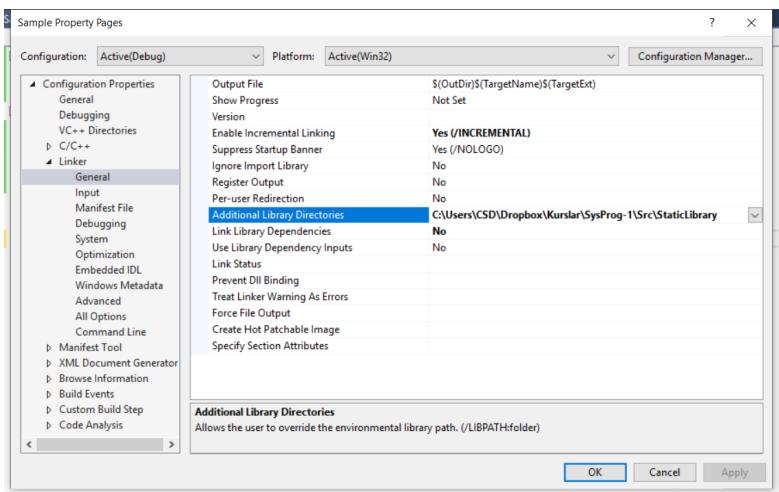
Microsoft'un link.exe isimli bağlayıcı programı otomatik olarak standart C fonksiyonlarının ve API fonksiyonlarının bulunduğu kütüphaneleri de link aşamasına dahil etmektedir. Fakat programcı isterse bazı seçeneklerle bu default özelliği kaldırabilir. Microsoft'un "link.exe" isimli bağlayıcısının daha pek çok komut satırı argümanı seçeneği vardır. Bu seçenekleri dokümanlardan inceleyebilirsiniz.

Windows'ta Visual Studio IDE'sinde bir kütüphane dosyasını kullanabilmek için şu işlemlerin yapılması gereklidir:

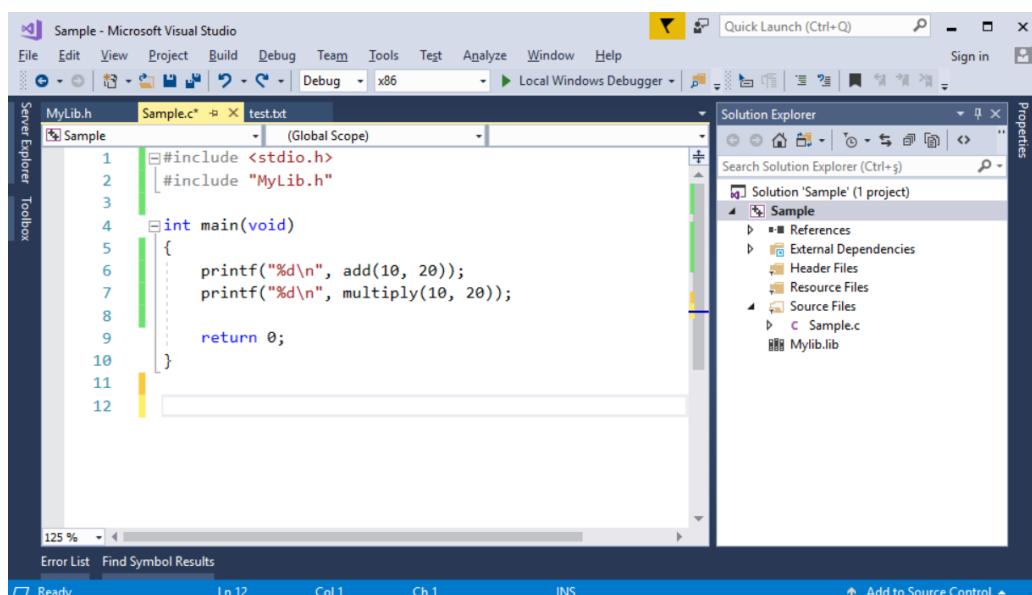
- 1) Önce kullanılacak kütüphane dosyasının ismi yol ifadesi olmadan proje özelliklerinden Linker/Input/Additional Dependencies sekmesine eklenir:



- 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:



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
```

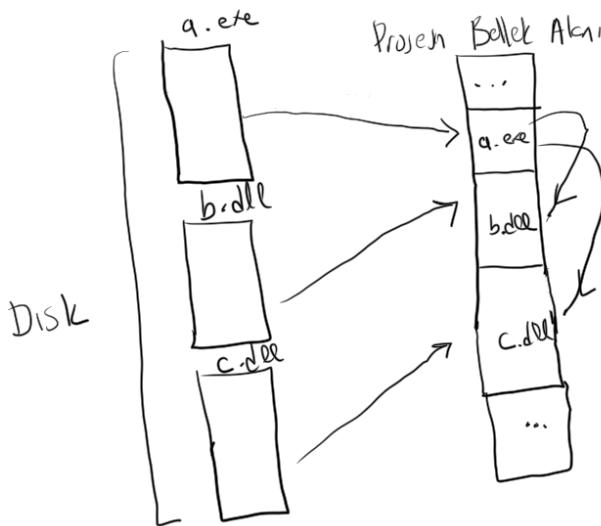
UNIX/Linux sistemlerinde statik kütüphane dosyaları kaynak dosya listesinin sonunda belirtilmelidir. Aksi takdirde onun aşağıdakiler için bu kütüphaneler link işlemine sokulmamaktadır.

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 çağrılığımızda bağlayıcı onu oradan alarak çalıştırılabilen dosyanın içerisinde yazmaz. Bağlayıcı yalnızca çalıştırılabilen dosyanın içerisinde 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 istediğiinde yalnızca o dosyayı değil b.dll ve c.dll dosyalarını da prosesin sanal bellek alanına yüklemektedir.



Artık program çalışırken dll çağrıması yapıldığında akış bellekteki dll'in içerişine 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 taşıırken o dinamik kütüphanelerin de uygun yerlere (ileride açıklanacak) kopyalanması gereklidir. 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 içerişine yazılmasız. 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 çağırırsak 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 tekrar tekrar yüklemez. Dinamik kütüphanenin yalnızca bir kopyasını fiziksel belleğe yükler. (Tabii aslında bu kopyanın tamamı da iziksel 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 çağrılığında 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 içerisindeindedir. 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" bağlayıcını /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 yapabilirdik:

```
cl /c mydll.c  
link /DLL mydll.obj
```

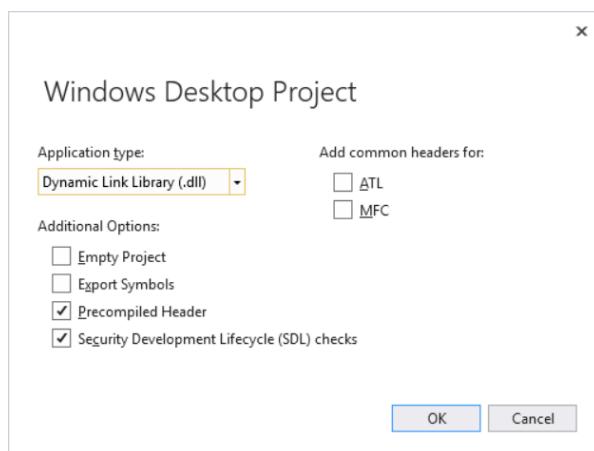
Tabii oluşturulacak hedef dosyanın isipi /Fe seçeneği ile değiştirebilir. Örneğin:

```
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 ortamındaki 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 gereklidir. __declspec Microsoft derleyicilerine özgü bir eklenedir. __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 bilirtilmesi gereklidir. 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 bölümdeki 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 içerisindeındır. 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 herhangi bir yerine yüklenmesi gereklidir. Çünkü o anda sanal bellek alanının neredelerinin boş olduğu önceden bilinmemektedir. İşte Windows sistemlerinde dinamik kütüphanelerin herhangi bir yere yüklenmesi için PE dosya formatında "relocation" bilgilerinin bulundurulması gerekmektedir. Böylece Windows'ta işletim sisteminin yükleyicisi dinamik kütüphaneyi uygun yere yükledikten sonra bu "relocation" tablosundan faydalananarak 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 gereklidir. 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 gereklidir. Ayrıca derleme işleminden sonra "`ld`" bağlayıcısı da `-shared` seçeneği ile çalıştırılmalıdır (Bu Microsoft'taki `/DLL` seçeneğine benzemektedir). Bu durumda bir dinamik kütüktane 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 yapılmaktadır:

```
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 çağrıma yapan programı link ederken link aşamasında "o dinamik kütüphanenin import kütüphanesi" denilen bir `.lib` dosyasını kullanmamız gereklidir. 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 içerisindeindedir. 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 işlemeye dahil etmesi gereklidir. Bu dahil etme işlemi tamamen statik kütüphanelerdeki gibidir. Yani statik kütüphaneler konusunda da belirtildiği gibi bu işlem iki biçimde yapılmaktadır:

1) Import kütüphane dosyası projeye "Add Existing Item" seçenekleri 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(dllexport)` bildirimi yerleştirilmelidir. Örneğin:

```
__declspec(dllexport) int Add(int a, int b);
```

Aslında `__declspec(dllexport)` belirleyicisi DLL'deki fonksiyonları kullanırken mutlak anlamda gereklidir. 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(dllexport)` işin başında derleyicinin daha etkin kod üretmesini sağlar.

Sonuç olarak DLL içerisindeki fonksiyonları dışarıdan kullanabilmek için `__declspec(dllexport)`, dışardan kullanırken `__declspec(dllexport)` belirleyicisinin kullanılması gereklidir. Bu işlemi kolaylaştırmak için söyle 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_

#ifndef DLLEXPORT
#define DLSPEC __declspec(dllexport)
#else
#define DLSPEC __declspec(dllexport)
#endif

/* Function Prototypes */

DLSPEC int Add(int a, int b);
DLSPEC int Multiply(int a, int b);

/* extern data Declarations */

DLSPEC extern int g_a;

#endif
```

Burada görüldüğü gibi `DLLEXPORT` isimli bizim uydurdugumuz makro eğer daha önce define edilmişse yine bizim uydurdugumuz `DLSPEC` yerine önişlemci `__declspec(dllexport)` belirleyicisini yerleştirecektir. Eğer bu makro daha önce define edilmemişse bu sefer `DLLEXPORT` yerine önişlemci `__declspec(dllexport)` belirleyicisini yerleştirir. Bu durumda bu başlık dosyası hem DLL oluşturulurken hem de kullanılırken include edilir. Örneğin bu durumda DLL'i oluşturan kod söyle olacaktır:

```
#define DLLEXPORT

#include <stdio.h>
#include "MyDll.h"

DLSPEC int g_a;

DLSPEC 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 gereklidir.

Göründüğü gibi bir DLL oluştururken 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(dllexport) ile yapıldığı bir başlık dosyasına. (örneğimizdeki MyDll.h)

Dinamik kütüphane kullanan bir program çalıştırılırken kullandığı DLL'in de o makinede bulunuyor olması gereklidir. Fakat kullanılan DLL hangi dizinde olmalıdır? İşte Windows sırasıyla DLL'leri bazı dizinlerde aramaktadır. Ancak bu konuda Windows'un versyonları 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) CreateProcess fonksiyonunu uygulayan prosesin (yani exe'yi çalıştırılan programın. Bu program Windows'un komut satırı ya da masaüstü (explorer.exe) de olabilir.) çalışma dizini.
- 3) Windows'un altındaki System32 dizini (bu dizin GetSystemDirectory API fonksiyonuyla elde edilebilir.)
- 4) Windows'un altındaki System dizini.
- 5) Windows'un dizinini kendisi (bu fizin GetWindowsDirectory API fonksiyonuyla elde edilebilir)
- 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
```

Pekiyi 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 işlemini 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 içerisinde 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
```

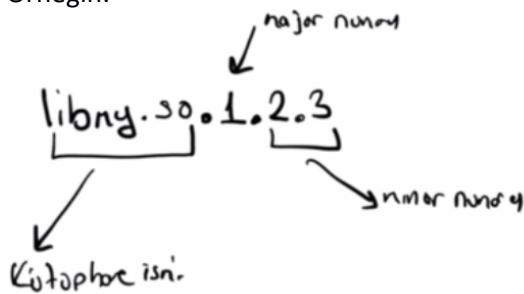
UNIX/Linux Sistemlerinde Dinamik Kütüphanelerin Kullanımına İlişkin Ayrıntılar

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 versiyondaki böceklerden kaçınmak için yazılmış olan (workaround) kodlar düzeltilmiş yeni versiyonda uygun biçimde çalışmaz.) 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 numalarla 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 geçmişte 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:

```
libmy.so.1.0.0
libmy.so.1.0.1
libmy.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 biçimini söylemek:

`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`

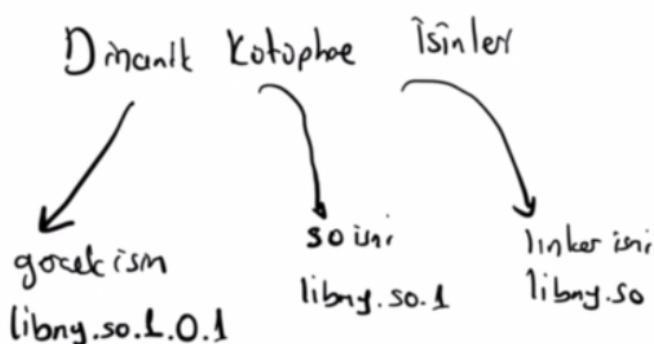
Bunların ortak so ismi söylemek:

`libmy.so.1`

Dinamik kütüphanenin "bağlayıcı ismi (linker name)" ise hiçbir majör ve minör numara içermeyen ismidir. Genel biçimini söylemek:

`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:

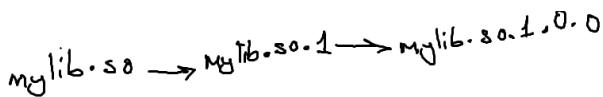
```
gcc -o libmy.so.1.0.1 -fPIC -shared foo.c bar.c
```

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 ismimize bir sembolik link oluşturalım:

```
ln -s libmy.so.1 libmy.so
```



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
```

Pekiye 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 sağlamaktadır. Yani amaç dinamik kütüphaneyi kullanan programda hiçbir değişiklik yapmadan onun istenilen son minör versiyonu kullanmasını sağlamaktır. Bağlayıcı isminden amaç ise bağlama aşamasında bağlama işlemini yapanın kafasının karışmaması için sade 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 görelî olduğu 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
```

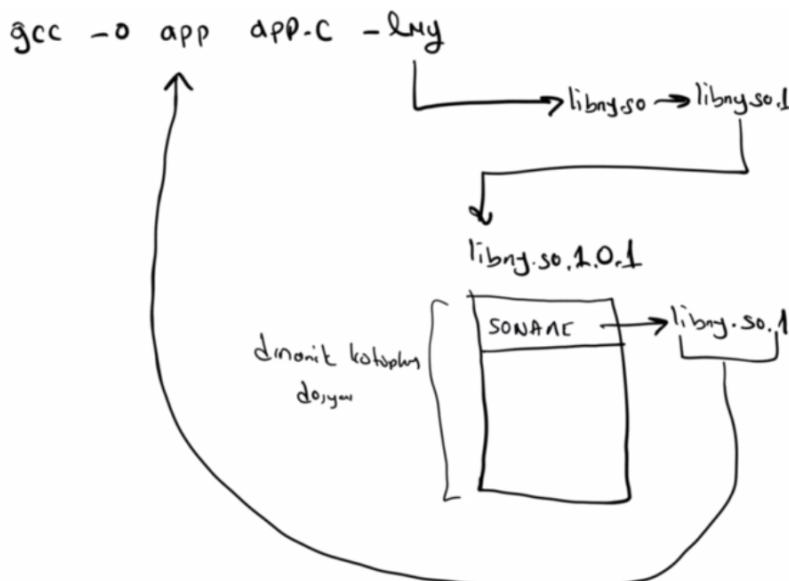
Burada derleyici sistemi ilgili dizinde libmy.so dosyasını arar. Bunu bir sembolik link dosyası biçiminde bulur. Bu dosya da so ismine link vermiştir. Linki izler. so ismi de gerçek isme link vermiştir. Bu durumda aslında yukarıdaki basit biçim sonuçta gerçek dinamik kütüphaneye varılmasını sağlayacaktır.

Bağlayıcı ismi link aşamasında kolay bir isim kullanmaya olanak sağlamaktadır. Pekiye 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 hedeflemiştir 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.

Pekiyi çalıştırılabilen programın içerisinde so ismi yerleştirilmektedir? Çünkü program aşağıdaki gibi derlenip link edildiğine göre bu programın içerisinde (yani ELF başlığında) gerçek ismin bulunması gerekmek 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 oluştururken buna bir so ismi ilişitirebilir. 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 gömülü olan so ismini çalıştırılabilen dosyaya yerleştirir. Böylece her ne kadar çalıştırılabilen program gerçek dosyaya referans edilerek oluşturulmuşsa da aslında onun içerisinde referans edilen dinamik kütüphane so isimli kütüphane olur.



Pekiyi so ismini gerçek kütüphane dosyasının ELF başlığına nasıl yazdırırız? İşte bunun için soname isimli bağlayıcı seçeneği kullanılmaktadır. Animşanacağı 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 libmyso.1 dosya ismi bir so ismi olarak gömülüdür. 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 gerekliliklerle 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/Havelsan-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 iki sorunun yanıtını vermektedir?

1) app programı hangi dinamik kütüphaneleri kullanmaktadır? Bu bilgi ldd tarafından ELF formatının ilgili bölümünden alınmaktadır.

2) Yükleyici bu dinamik kütüpheneyi 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 faydalananmış 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 içerisindeki fonksiyonlar kullanılmıştır.

Bir dinamik kütüphanenin diğerini kullanması durumunda işletim sistemi yükleme sırasında söz konusu tüm dinamik kütüphaneleri özyinelemeli olarak yükler. Böylece bizim de programı konuşladırırken tüm bu dinamik kütüphaneleri çalıştırılabilen dosyaya birlikte hedef makineye taşımamız gereklidir.

Dinamik Kütüphanelerin Dinamik Yüklenmesi

Hem Windows sistemlerinde hem de UNIX/Linux sistemlerinde dinamik kütüphaneler programın çalışma zamanı sırasında dinamik biçimde yüklenemektedir. Daha önce gördüğümüz dinamik kütüphane kullanma biçimlerinde dinamik kütüphaneler işin başında program dosyası belleğe yüklenirken işletim sisteminin yükleyicisi tarafından yükleniyordu. Peki 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 yükleniği 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("MyD11.D11")) == NULL)  
    ExitSys("LoadLibrary", EXIT_FAILURE);
```

Görüldüğü gibi dinamik kütüphanelerin dinamik yüklenmesi için "import kütüphanesi"ne gereksinim duyulmamaktadır.

2) Dinamik kütüphane yüklenikten 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 yerinde 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 MyD11.dll
```

GetProcAddress fonksiyonu geri dönüş değeri olarak bize DLL içerisindeki fonksiyonun adresini vermektedir. Geri dönüş değerindendeki FARPROC bir fonksiyon typedef türüdür. Programının tür dönüştürmesi ile bu adresi uygun fonksiyon türüne dönüştürmesi gereklidir. (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 içerisinde kullanılabılır. Örneğin:

```
result = padd(10, 20);  
printf("%d\n", result);
```

4) Kullanım bittiğinden 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 istediğimz zaman FreeLibrary API fonksiyonuyla bellekten boşaltabiliriz.

Fonksiyon parametre olarak DLL'in bellekteki yüklenme adresini almaktadır.

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 yüklediği adresi belirtir. Dinamik yükleme fonksiyonları libdl kütüphanesinin içerisinde olduğu için link işlemine -ldl seçeneğinin eklenmesi gereklidir. 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. Isimlerin dekaore 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 çağrılabılır. 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);
    }

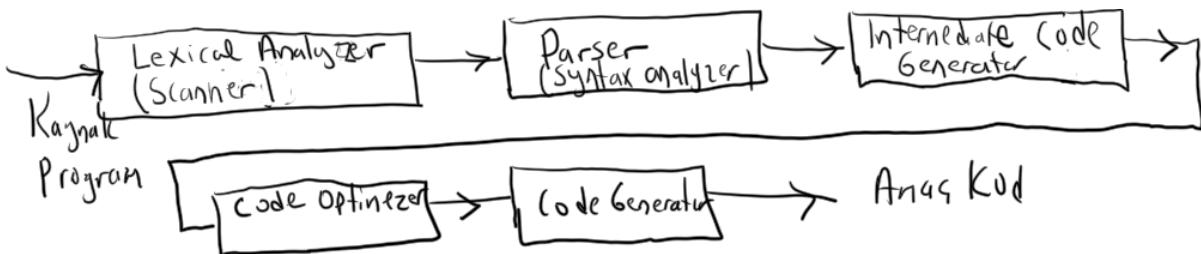
    result = padd(10, 20);
    printf("%d\n", result);

    dlclose(dl);

    return 0;
}
```

Derleyicilerin Çalışma Mekanizması

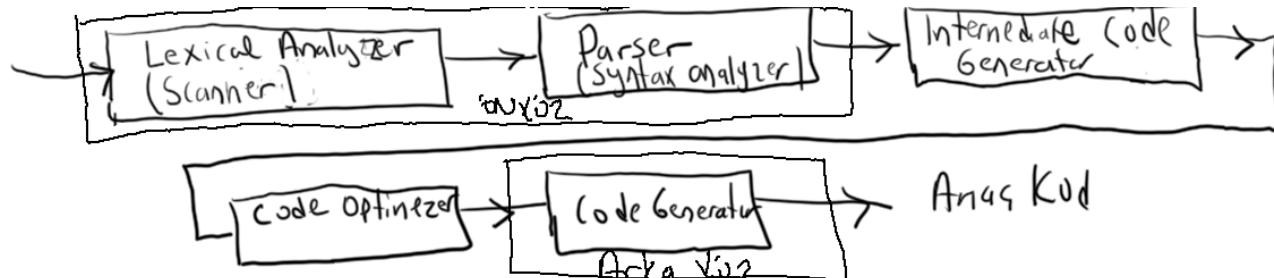
Bir derleyici tipik şu aşamalardan geçerek derleme işlemini yapar:



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 ver yapısına dönüsürür. Parser modül tarafından oluşturulan bu ağaç biçimindeki veri yapısına "parse tree" denilmektedir. "Parse tree" programın işlenebilir halidir. Yani artık program bir yazı olmaktan çıkışmış 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 module "intermediate code generator" denilmektedir. Bu aşamada gerçek makine komutlarının değil de arakodların

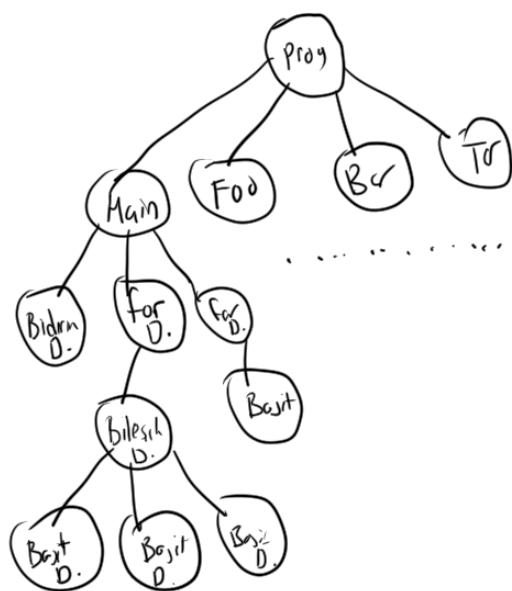
üretilmesinin iki önemli nedeni vardır: Üretilen arakodlar gerçek makine kodlarına göre daha kolay işlenebilir biçimdedir. Dolayısıyla optimizasyon gibi işlemler için daha iyi bir girdi oluşturmaktadır. Ara kodlar aynı zamanda derleyiciyi yazanlar için standart bir birleştirme noktası oluşturmaktadır. 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 module 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 ıstesin. Normalde bunun için kaç derleyici yazması gereklidir? 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 faydalananmak 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 modüllerini yapabilir. Bu faaliyete gcc için "frontend" yazma faaliyeti denilmektedir. Şimdi biz yeni bir işlemci için C derleyicisi yazmak isteyelim. Ve bunun için gcc derleyicisinden faydalananmak isteyelim. Bu durumda gcc'nin C frontend'i doğrudan kullanılabilir. Bunun için bizim gcc için bir "backend" yazmamız gereklidir.

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:



Göründüğü 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 çalıştmak için yeniden düzenleme süreci"dir.

Anahtar Notlar: Aslında buradaki optimizasyon terimi yanlış uydurulmuştur. Optimizasyon eniyleme demektir. Halbuki derleyicilerin böyle bir iddiası yoktur. Derleyiciler kodları onların işlevleri değiştirmek 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 gereklidir. Yani optimize edilmemiş kod ile edilmiş kod tamamen aynı davranış göstermelidir.
- Optimizasyon genel olarak "hız" ya da "kod büyülüğu" temel alınarak yapılabilir. Fakat baskın ölçüt ve default durum "hız" optimizasyonudur. Hatta derleyiciler kodun daha hızlı çalışmasını sağlarken kodu büyütübilirler.
- 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'lı 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'lı uygulamalarda programcı bu durumu göz önüne almalıdır. (Tabii çok thread'lı 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 maaliyetlerle 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 etkileyerek biçimde yapılır. Döngü optimizasyonları döngü içlerini optimiza etmeyi hedefler. Burada optimizasyonlar karışık sıradır ele alınacaktır.

1) Ortak Alt İfadelerin Elimine Edilmesi (Common Subexpression Elimination): Bir gurp işlemde ortak olan bazı alt ifadelerin her defasında yeniden yapılması yerine derleyici bunların sonuçlarını bir yazmaca 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 yazmaca 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 kapatıldığından 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 çalışmasıyle 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 çıktıgı görülmüştür.

Fonksiyon çağrıları (hele ki derleyici fonksiyonun içini göremiyorsa) döngü dışına çıkartılamaz. Örneğin:

```
for (i = 0; i < strlen(str); ++i) {  
    ...  
}
```

Burada programcı derleyicinin optimazasyon yaparak strlen fonksiyonunu bir kez çağrımasını beklememelidir. Derleyiciler standart C fonksiyonlarının ne yaptığını bilmediği için onlara normal bir fonksiyon muamelesi yaparlar. Bu nedenle derleyici strlen fonksiyonunu her defasında çağrımak zorundadır. Döngü içerisinde yapılan çağrıların döngü dışına çıkarılamayabileceğine dikkat ediniz. Yukarıdaki kodu programcı şöyle düzenlemeliydi:

```
len = strlen(str);  
for (i = 0; i < len; ++i) {  
    ...  
}
```

Pekiyi 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 "intirinsic" fonksiyonlar için prototip bildirimi de gerekmez. Tabii "intrinsic" fonksiyon kavramı C standartlarında yoktur. Bu nedenle taşınabilirlik bakımından bunların "intirinsic" 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 üretilenbilir. 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 kodda 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 değişkenin de result değişkenin de koda bir etkisi yoktur. Yani başka bir deyişle program aşağıdaki gibi 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 akliniza 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 gereklisiyle 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 farklı 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 yapılıyor 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 değercek bir kazanç sağluyorsa uygulanmalıdır. Örneğin burada optimiza edilmiş biçim kodun daha hızla çalışmasını sağlamasını karşın hem kodu daha karmaşık gibi göstermeye hem de kodu büyütmemektedir. Buradaki döngü eğer çok az dönüyorsa derleyicinin böyle bir düzenleme yapması toplamda önemli bir hız kazancı sağlamayacaktır.

6) Döngü Açıımı (Loop Unrolling): Döngü açımı döngü içerisindeki deyimleri çöklayarak 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 çağrılmıştır. 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 biçimde 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
j1    SHORT $LL2@main

```

Jump işlemi pek çok işlemci için *pipeline* mekanizması üzerinde olumsuz etkileri olan bir işlemidir. İşlemci *Jump* işlemini gördüğünde sonraki komutların çalışıp çalışmayaacağını bilmediğinden onlar için hazırlık işlemleri yapmayı bilir. Bu konuda modern işlemciler arasında çeşitli farklılıklar bulunduğu belirtelim.

8) Döngü Birleştirme (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 döngülerin birleştirilmek yerine tam tersine ayırtırmak (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 $a[i][j][k]$ elemanı tek boyutlu dizinin $i * \text{colSize} + k$ 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 döngülerini 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 yazmaca 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 yazmaca da olabilir */
a = temp + 1;
b = temp + 2;
c = temp + 3
```

Tabii derleyicinin bu optimizasyonu yapabilmesi için kesinlikle o sırada bu göstericinin gösterdiği yerdeki değerin değişimeyeceğini garanti altına alması gereklidir. 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 incelemeden 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 *mymemcpy(void *dest, void *source, size_t size)
{
    char *cdest = (char *) dest;
    char *csource = (char *) source;

    while (size-- > 0)
        *cdest++ = *csource++;

    return dest;
}
```

Eğer bloklar çakışıksa derleyici bu makine komutundan faydalananamaz. İşte mymemcpy'yi çağrıran 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 eriştiği garantisini alır. Şimdi yukarıdaki memcpy fonksiyonundaki göstericileri restrict yapalım:

```
void *mymemcpy(void * restrict dest, void * restrict source, size_t size)
{
    char * restrict cdest = (char *) dest;
    char * restrict csouce = (char *) source;

    while (size-- > 0)
        *cdest++ = *csouce++;

    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 örnektekteki 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 kalamasına çalışılır. Böylece derleyici aynı değişkeni tekrar bellekten yazmaya 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'lı akışa göre yapılmaktadır. Yani eğer çok thread'lı uygulama yapılyorsa derleyicinin bu optimizasyonları bizim kodumuzun anlamını değiştirebilir. Çok thread'lı 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 varsayıyalım. Derleyici burada başka bir thread'in x ve y'yi değiştirebileceğini dikkate almaz. Ve optimizasyonunu aşağıdaki gibi yapabilir:

```
temp = x + y;  
----> Bu noktada x, y değişirse ne olur?  
b = temp + 20;  
c = temp + 30;
```

Göründüğü 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 sıfır çektığında 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 yapmayı yukarıda gördüğümüz optimizasyon temalarının çoğu devre dışı kalır.

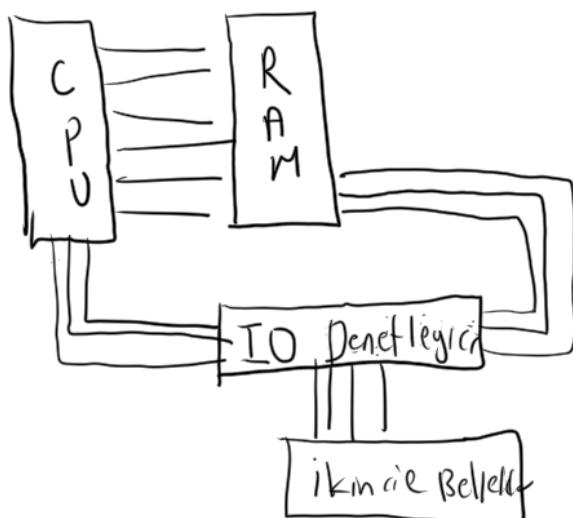
Aşağı Seviyeli Disk İşlemleri

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 elektromekanik hard diskler
- 2) Yarı iletkenlerle imal edilmiş SSD'ler, flash EPROM ve EEPROM bellekler
- 3) CDROM, DVD ROM gibi işinsal mekanizmayla çalışan elektromekanik 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 arayızları 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 konusuya okunan bilgiler doğrudan RAM'e aktarılır. Bir yazma işlem söz konusuya 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 büyülüktedir?
- 2) Okunacak ya da yazılacak bilgiler için RAM aktarım adresi neresidir?

Modern çok prosesli işletim sistemleri ikincil belleklerden bir talepte bulunduğuunda 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 büyülük değişebilir. Ancak 512 standart bir değer olarak kabul görmüştür). Sektör klasik elektomekanik 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

Elektomekanik hard disklerde bilgiler manyetik olarak bir yüzeye kaydedilir ve oradan okunur. Bir hard diskte tipik olarak platter denilen yüzeyle bir çubuga 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.

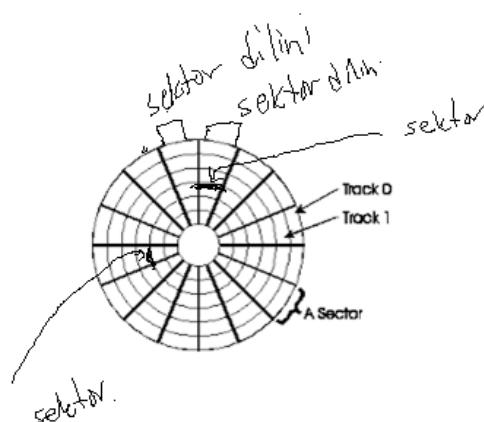


Yukarıdaki resimde üç platter vardır. Disk kafaları aynı eksene monte edilmiştir. Her yüzey için bir kafa bulunmaktadır. Yani yukarıdaki resimde görülen hard diskte toplam 6 kafa vardır. Dodlama ve geri alma disk dönerken kafalar tarafından yapılır. Hard diskler kendi içerisinde küçük bir önbelleğe de 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 yazabiliz. 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 ayrıliyordu. 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 daha 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'lereki sektör sayıları iç track'lardan daha fazladır.

Bir hard diskin hızı saniyede okunan ya da yazılan byte sayısıyla (ya da sektör sayısıyla) 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 Hızasına Gelene Kadarki Zaman (Rotational Delay): Diskin hızını etkileyen ikinci bileşen disk dönüş hızıdır. Çünkü sektör kafa hızasından geçen okuma yazma yapılır. İlgili sektörün kafa hızasından geçme zamanı diskin dönüş hızıyla ilgilidir. Bugünkü diskler tipik olarak 4200, 5400, 7200, 10000, 15000 hızlardır. Bizim notebook'larımızda tipik olarak 5400 dönüş hızları kullanmaktadır.

3) Aktarım Zamanı (Transfer Time): Bu zaman en önemsiz zamanı oluşturur. Disk dönerken kafanın bilgiyi okuyup IO denetleyicisine aktardığı zamandır. Bu zaman IO denetleyicisinin tasarımasına da hard diskin teknolojisine de bağlıdır. Örneğin SCSI denetleyiciler daha hızlı bir aktarım sunabilmektedir.

İşletim sistemi tasarımcıları disk kafa hareketini minimize etmeye odaklılardır. 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 yaklaştırılmaktadır.

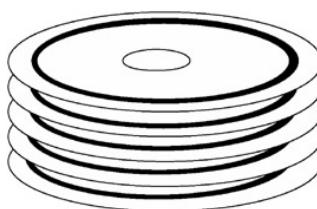
Sektörler İçin Koordinat Sistemi ve Sektör Numaralandırması

İkincil belleğin türü ne olursa olsun bugün ikincil bellek arasındaki veri aktarımı sektör düzeyinde yapılmaktadır. Sektörlerin de birer numaraları vardır. Sektörler track'lerin üzerinde bulunur. Track'teki sektör sayısı hard disktten hard diske değişebilmektedir. Ayrıca disk yüzeyleri dairesel olduğu için dış track'lereki sektör sayıları iç track'lerekine 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 oluşmaktadır. 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 sektörde 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 sıfırıncı 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>
    }
}
```

Gördüğü gibi mantıksal numaralandırma kafa hareketini minimize edecek biçimde yapılmıştır. Numaralandırmada önce en dış yüzeyin ilk track'i dolanılır, sonra diğer yüze geçirip onun en dış track'i dolanılır. Böyle böyle tüm yüzlerin en dış track'leri dolanıldıktan sonra bir sonraki track'e geçirilir. 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 hızasına karşılık gelir.

Sektör numaralandırmasında bir de silindir (cylinder) terimi ile karşılaşılmaktadır. Tüm yüzeylerin n numaralı track'lerinin kümesine 'n' numaralı silindir denilmektedir. Örneğin 6 yüzelyi (3 platter'lı) bir hard diskte 10 numaralı silindir demek tüm yüzeylerin 10 numaralı track'lerinin kümesidir.



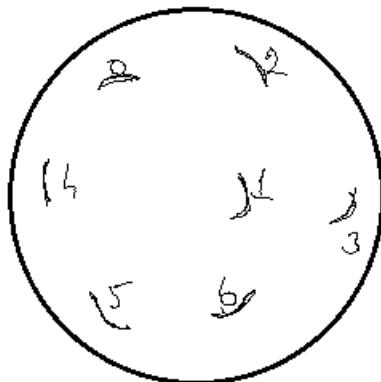
A cylinder is
the set of the
same tracks
across all
platters.

SSD'Diskler ve Sektörler

Günümüzde hard diskler gitgide azalmakta ve yerini SSD disklere bırakmaktadır. SSD diskler mekanik bir parça 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 yapılmaktadır. Ancak genel olarak SSD'lerden okuma çok hızlımasına karşın yazma göreli olarak daha yavaştır. SSD'ler mekanik bir birim olmadığı için onlarda okuma ve yazma için kafalar bulunmaz. Dolayısıyla SSD'lerde track kavramı dayoktur. 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. Kullanıcılar bu dosya kavramı biçiminde gösterir. Aslında dosyalar ardışıl byte'lardan oluşmamaktadır. Diskteki sektörlerde bulunan bilgilerden oluşmaktadır. İşletim sistemi bize o sektörlerdeki bilgiyi sanki ardışıl byte topluluğunu gibi gösterir. Örneğin 7 sektörden oluşan bir dosya aşağıdaki gibi disk yüzeyine (hatta yüzeylerine) dağılmış olabilir.



İşletim sistemi dosyaların parçalarını sektörlerde 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 olması 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 kavumlardır. Diske sektör numaralatıyla 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 dosyayı 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 büyükçe 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 gerçekleşme (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 faydalananak 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 tanıyor 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 Alocation 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 Techology 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 Linux'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 içerisinde bulundururlar. User moddan bizim bu kodları kullanabilmemizi aygit sürücülerle sağlarlar. Bu aygit sürücüler diskı sanki bir dosyamış gibi bize göstermektedir. Biz bu aygit sürücülerini Windows sistemlerindeki dosya açan CreateFile API fonksiyonuyla açarız. Sonra SetFilePointer API fonksiyonuyla ilgili sektörün başladığı pozisyon'a 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 ismiyle 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 disk (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 dizinin altında aygit sürücü biçiminde bulunmaktadır. Yani bu sistemlerde disk işlemleri tamamen dosya işlemleri gibi yapılmaktadır. Disk ya da volüme ilişkin aygit sürücü open fonksiyonuyla açılır. Iseek fonksiyonuyla konumlandırma yapılarak read ve write fonksiyonlarıyla da okuma ve yazma yapılmaktadır. Örneğin tipik olarak sata disklerde tüm diske ilişkin bu aygit sürücü dosyaları /sda, /sdb, /sdc biçimindedir. Volümlere ilişkin dosyalar da /sda1, /sda2, /sda3 biçiminde organize edilmiştir. Bu aygit sürücü dosyaları "other" için read ve write haklarına sahip değildir. Bu nedenle bu aygit sürücülerini 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);

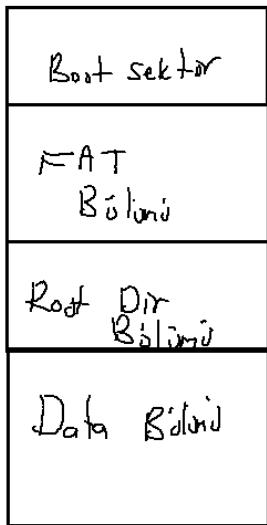
    exit(EXIT_FAILURE);
}
```

FAT Dosya Sistemlerinin Disk Organizasyonu

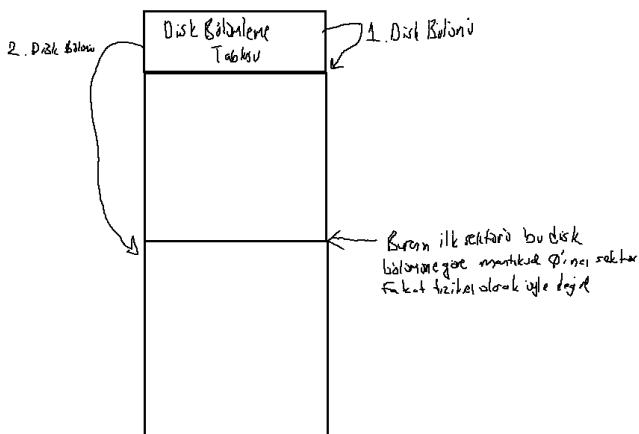
FAT dosya sistemleri Microsoft'un ilk DOS işletim sisteminden beri kullandığı temel dosya sistemidir. FAT dosya sistemi ilk yıllarda önemli bir yenilik olarak değerlendirilmiştir. Bu dosya sistemlerinde ileride ele alınacağı gibi ismine FAT (File Allocation Table) denilen önemli bir meta-data bölümü vardır. Bu bölüm dosya sistemine ismini vermiştir.

FAT dosya sisteminin FAT12, FAT16 ve FAT32 olmak üzere üç değişik biçimini 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.

FAT12 ve FAT16 dosya sistemiyle formatlanmış bir disk bölümü aşağıdaki organizasyonel bölgümlerden oluşur:



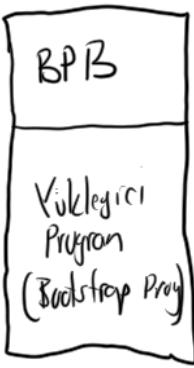
Hard disk'ler, SSD'ler ve Flash EPROM bellekler birden fazla disk bölümüne sahip olabilirler ve bu disk bölgümlerine farklı dosya sistemleri yüklenebilir. Bu medyalarda hangi disk bölgümlerinin hangi sektörden başlayıp ne kadar uzunlukta olduğu medyanın içerisinde saklanmaktadır. Disk bölgümlerinin nereden başladığı ve ne uzunlukta olduğunu tutan metatablosuna genel olarak Disk Bölümleme Tablosu (Disk Partition Table) denilmektedir. Disk bölgümlerine ayrılmış bir diskteki sektor numaralandırmaları iki biçimde yapılabilmektedir: Fiziksel numaralandırma ve mantıksal numaralandırma. Fiziksel numaralandırma kontrol kartının uyguladığı gerçek numaralandırmadır. Burada medyanın tamamı tek bir medya olarak değerlendirilir ve ilk sektor 0'dan başlatılarak her sektöre artan bir numara karşılık düşürülmüştür. Mantıksal numaralandırmada ilgili disk bölümünün başı orijin noktası olarak kabul edilir. Yani her disk bölümünü adeta ayrı bir diskmiş gibi 0'dan başlayarak numaralandırmaktadır. Örneğin:



Bir diskteki kullanıma hazırlanmış disk bölgümlerine (partitions) "volume" de denilmektedir. Microsoft "volume" sözcüğünü bu bağlamda çok kullanmaktadır.

Boot Sektör

FAT dosya sistemlerinde volüm'ün ilk sektörüne (yani mantıksal 0'inci sektörüne) boot sektör sektör denilmektedir. Boot sektör kendi içerisinde iki bölümünden 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 uzuluğundadır (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 bölgeleri iyi anlaşılsın 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
00000010	02	00	02	00	00	F8	A0	00	3F	00	FF	00	00	B8	DC	E8
00000020	00	40	01	00	80	00	29	49	73	D6	26	4E	4F	20	4E	41
00000030	4D	45	20	20	20	46	41	54	31	36	20	20	20	33	C9	
00000040	8E	D1	BC	F0	7B	8E	D9	B8	00	20	8E	CO	FC	BD	00	7C
00000050	38	4E	24	7D	24	8B	C1	99	E8	3C	01	72	1C	83	EB	3A
00000060	66	A1	1C	7C	26	66	3B	07	26	8A	57	FC	75	06	80	CA
00000070	02	88	56	02	80	C3	10	73	EB	33	C9	8A	46	10	98	F7
00000080	66	16	03	46	1C	13	56	1E	03	46	0E	13	D1	8B	76	11
00000090	60	89	46	FC	89	56	FE	B8	20	00	F7	E6	8B	5E	0B	03
000000A0	C3	48	F7	F3	01	46	FC	11	4E	FE	61	BF	00	00	E8	E6
000000B0	00	72	39	26	38	2D	74	17	60	B1	0B	BE	A1	7D	F3	A6
000000C0	61	74	32	4E	74	09	83	C7	20	3B	FB	72	E6	EB	DC	A0
000000D0	FB	7D	B4	7D	8B	F0	AC	98	40	74	0C	48	74	13	B4	0E
000000E0	BB	07	00	CD	10	EB	EF	A0	FD	7D	EB	E6	A0	FC	7D	EB
000000F0	E1	CD	16	CD	19	26	8B	55	1A	52	B0	01	BB	00	00	E8
00000100	3B	00	72	E8	5B	8A	56	24	BE	0B	7C	8B	FC	C7	46	F0
00000110	3D	7D	C7	46	F4	29	7D	8C	D9	89	4E	F2	89	4E	F6	C6
00000120	06	96	7D	CB	EA	03	00	00	20	0F	B6	C8	66	8B	46	F8
00000130	66	03	46	1C	66	8B	D0	66	C1	EA	10	EB	5E	0F	B6	C8
00000140	4A	4A	8A	46	0D	32	E4	F7	E2	03	46	FC	13	56	FE	EB
00000150	4A	52	50	06	53	6A	01	6A	10	91	8B	46	18	96	92	33
00000160	D2	F7	F6	91	F7	F6	42	87	CA	F7	76	1A	8A	F2	8A	E8
00000170	C0	CC	02	0A	CC	B8	01	02	80	7E	02	0E	75	04	B4	42
00000180	8B	F4	8A	56	24	CD	13	61	61	72	0B	40	75	01	42	03
00000190	5E	OB	49	75	06	F8	C3	41	BB	00	00	60	66	6A	00	EB
000001A0	B0	42	4F	4F	54	4D	47	52	20	20	20	0D	0A	52	65	
000001B0	6D	6F	76	65	20	64	69	73	6B	73	20	6F	72	20	6F	74
000001C0	68	65	72	20	6D	65	64	69	61	2E	FF	0D	0A	44	69	73
000001D0	6B	20	65	72	72	6F	72	FF	0D	0A	50	72	65	73	73	20
000001E0	61	6E	79	20	6B	65	79	20	74	6F	20	72	65	73	74	61
000001F0	72	74	0D	0A	00	00	00	00	00	00	00	AC	CB	D8	55	AA

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'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	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 Etiketi (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 önesi 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. Gerçek 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'in bir sonraki track başına hizalanması işleyisi hızlandırılmaktadır. Bazen de boot sektör yükleyici programı daha büyük olabilir. Bu durumda da FAT'in yeri ötelenebilir. İşte bu alanda FAT'in 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'in 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'in bir kopyası üzerinde güncelleme yapar yapmaz ikinci kopyası üzerinde de güncelleme yapar. Yani bu kopya normal durumda aynı içeriye sahiptir. Bizim de iki kopyaya birden bakmak gibi bir gereksimimiz olmaz. Burada yeniden vurgulamak gerekir: Microsoft FAT'in iki kopyasını biri bozulursa diğerinden faydalansın diye oluşturmuştur. Ancak pratikte bu durumun faydası olduğu söylenemez.

Root Girişlerinin Sayısı: Yukarıdaki şekildeki 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:

$$\text{Root Dir SektörUzunluğu} = \text{Root girişlerinin sayısı} * 32 / 512$$

FAT12 ve FAT16 sistemlerinde kök dizindeki dosya sayısının en fazla bu alanda belirtilen miktarda olabileceği 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 söylemenemez. Bu alanda eğer F8 varsa ortam hard disk'tir. F0 varsa ortam floppy'dir. Biz burada artık hep F8 görürüz.

FAT'in Bir Kopyasındaki Sektör Sayısı: Bu alanda FAT'in bir kopyasındaki sektör sayısı bulunmaktadır. Böylelikle FAT'in toplam sektör uzunluğu hesaplanabilir. FAT'in iki kopyasının bulunduğu 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 ilgili disk bölümünün başından itibaren kaçinci sektörde boot-sektörün yer aldığı bilgisi vardır. Eğer buradaki değer 0 ise boot sektör hemen ilgili disk bölümünün başındadır. Fakat bazen boot sektör hemen ilgili disk bölümünün başından başlatılmaz. Track hizalaması için biraz öteye alınabilir.

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 volume bir hard diskteyse o hard diskin kaçinci hard disk olduğu bu alanda yazmaktadır.

Genişletilmiş Boot Id'si (Extended Boot Signature): 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): 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 teil bir değer değildir. Kopya koruma gibi amaçlarla bu değerden faydallanması doğru yaklaşım değildir.)

Volüm Etiketi (Volume Label): Bu alanda 11 byte'lık volüm etiketi bulunmaktadır. Ancak volüm etiketi FAT dosya sistemlerinde kök dizinde "Volume Label" özniteligine (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.

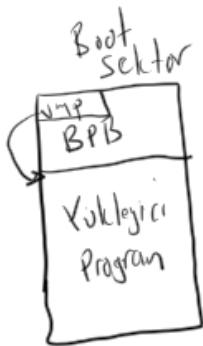
BPB bloğundaki bilgilerden hareketle tüm data alanlarının hangi mantıksal (relative) sektörden başlayıp ne uzunlukta sürdüğü belirlenebilir:

- 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'in Bir Kopyasının Sektör Sayısı ile FAT Kopyalarının Sayısı alanındaki bilgilerin çarpımıyla bulunabilir.
- 3) Root Dir bölümü FAT'in bitişinden başlar ve uzunluğu BPB'deki Root Dir Girişlerinin Sayısı alanındaki değerden faydalananarak bulunur.
- 4) Data bölümü Root Dir bölümünün bitişinden başlar.

Boot Sektör Yükleyici Programı

Boot Sektör yükleyici programı boot sektörde hemen BPB'nin bittiği yerden başlar ve boot sektörün sonuna kadar devam eder. Boot sektör yükleyici programı sembolik makine dilinde yazılmış 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 sisteme 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 yerlestirmelidir. BPB alanını okuyup bir yapıya yerlestiren bir fonksiyon şöyleden yazılabılır:

```
/* FatSys.h */

#ifndef FATSYS_H_
#define FATSYS_H_

#include <Windows.h>

/* Symbolic contants */

#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->spc = *(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;
}

```

Şöyledir 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);
    }

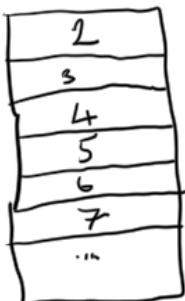
    exit(status);
}

```

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'in 4 sektörden olduğunu varsayıyalım. 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);
}

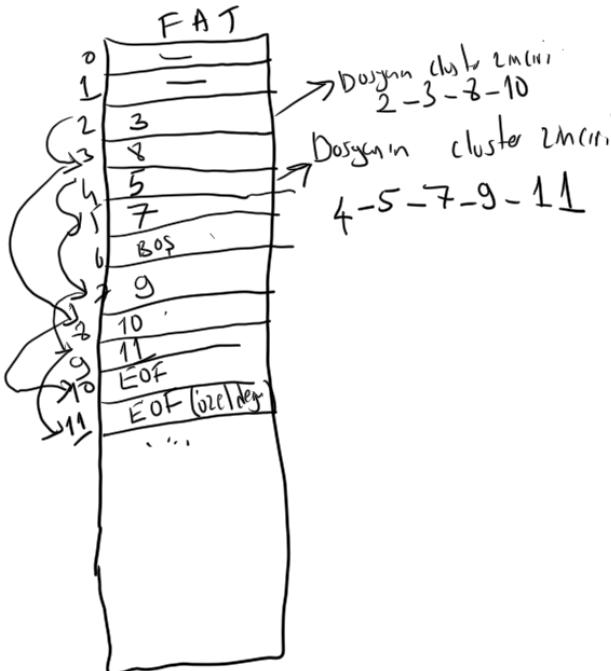
BOOL WriteCluster(const BPB *pBPB, DWORD clu, const void *buf)
{
    DWORD sect = pBPB->dataOrigin + (clu - 2) * pBPB->spc;
    return WriteSector(pBPB->hVolume, sect, pBPB->spc, buf);
}
```

Burada görüldüğü gibi önce okunacak ya da yazılacak cluster'ın data bölümünün kaçinci 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'larda 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 yapıdadır. Bir dosyanın ilk cluster bilgisi dizin girişinden alınır (bu konu ileride ele alınmaktadır). FAT tablosu FAT elemanlarından oluşmaktadır. Her FAT elemanın ilk 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 biçimini 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'in ilk iki elemanında özel değerler bulunmaktadır. İşletim sistemi bu özel değerleri (magic numbers) kontrol edebilir. (Eski DOS ve Windows bunu kontrol ediyordu. Artık Windows kontrol etmemektedir.) Bu değerler yanlış ise FAT bölümünün bozulmuş olduğu sonucu çıkarılabilir. 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:FFE] FAT32: [00000002:0FFFFFEF]	Sonraki cluster numarası. Bir FAT elemanında bulunan normaldeğer 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: OFFFFF7	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 yerleştirmez.
FAT12: [FF8:FFF]	Son cluster. Bu değerleri dosyanın cluster zincirindeki son cluster'ı belirtmektedir.
FAT16: [FFF8:FFFF]	
FAT32: [0FFFFFF8:0FFFFFFF]	

FAT32'de sanıldığın aksine her FAT elemanı 32 bit değil 28 bittir. Dolayısıyla FAT tablosunda FAT elemanlarının yüksek anlamlı 4 biti (yani yüksek anlamlı hex digit) her zaman sıfırdır.

FAT Elemanlarının Elde Edilmesi

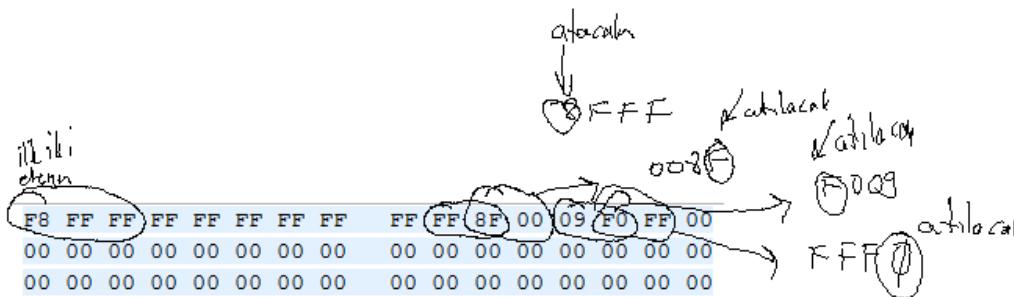
FAT tablosunun aşağıdaki gibi unsigned char türünden bir dizi içerisinde okunduğunu varsayıyalım:

```
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 elemanın FAT'teki konumuna cluster numarasının 1.5 ile çarpılmasıyla gidilir. Ancak oradan çekilen WORD değerin hangi 4 bitlik kısmının atılacağı cluster numarasının tek mi çift mi olduğuna bağlıdır. Örneğin aşağıda 12 bit için girişler verilmektedir:



12 bit FAT'te normal olarak 3 hex digit elde edilmesi gereklidir. Biz WORD çektiğimizde bu WORD değerinin uygun olan 4 bitini atmamız gereklidir. Pekiyi hangi 4 biti atmamız? İşte eğer cluster numarası çift ise bizim sonraki byte'ın düşük anlamlı hex digitini almamız, cluseter numarası tek ise bizim ilk byte'ın yüksek anlamlı hex digitini almamız gereklidir. Bunun formülüze edilmiş hali şöyle açıklanabilir: Biz her zaman FAT12'de cluster numarasını 1.5 ile çarpıp oradan WORD çekeriz. Eğer cluster numarası çift ise yüksek anlamlı 4 biti atarız, tek ise düşük anlamlı 4 biti atarız. O halde GetNextClu12 fonksiyonu şöyle yazılabılır:

```
WORD GetNextClu12(BYTE *pFat, WORD clu)
{
    if (clu % 2 == 0)
        return *(WORD *)(pFat + clu * 3 / 2) & 0xFFFF;
    else
        return *(WORD *)(pFat + clu * 3 / 2) >> 4;
}
```

Aynı fonksiyon şöyle de yazılabılır:

```
WORD GetNextCluster12(BYTE *pFat, WORD clu)
{
    WORD wordVal = *(WORD *)(pFat + clu * 3 / 2);
```

```
        return clu & 1 ? wordVal >> 4 : wordVal & 0x0FFF;
}
```

FAT32'de ilgili FAT elemanın byte offset'i cluster numarası 4 ile çarpılarak elde edilir:

```
DWORD GetNextClu32(BYTE *pFat, DWORD clu)
{
    return *(DWORD *)(pFat + clu * 4);
}
```

Yukarıdaki fonksiyonları bir döngü içerisinde çağırarak dosyanın cluster zincirini elde edebiliriz:

```
WORD *GetClusterChain12(BYTE *fat, WORD firstClu)
{
    static WORD chain[MAX_FAT12_CHAIN];
    WORD clu = firstClu, i;

    for (i = 0; clu < 0xFF8; ++i) {
        chain[i] = clu;
        clu = GetNextCluster12(pFat, clu);
    }
    chain[i] = 0;

    return chain;
}

WORD *GetClusterChain16(BYTE *pFat, WORD firstClu)
{
    static WORD chain[MAX_FAT16_CHAIN];
    WORD clu = firstClu, i;

    for (i = 0; clu < 0xFFFF8; ++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;

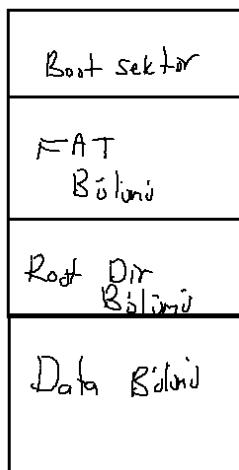
    for (i = 0; clu < 0xFFFF8; ++i) {
        chain[i] = clu;
        clu = GetNextClu32(pFat, clu);
    }
    chain[i] = 0;

    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'lük girişler tutulmaktadır. Bu girişlerin sayısının BPB alanı içerisinde tutulduğunu anımsayınız. RootDir bölümünün sektör sayısı sınırlıdır. Bu sayı formatlama sırasında belirlenmektedir.



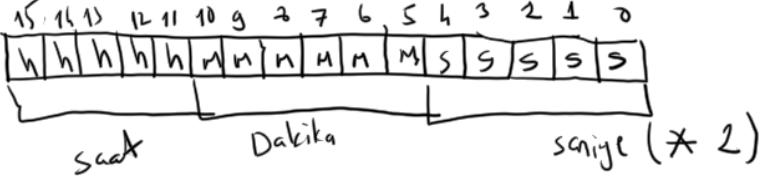
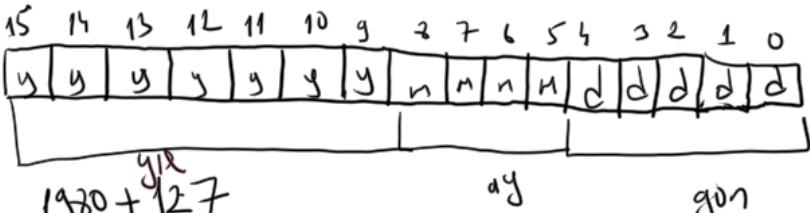
Böylece FAT12 ve FAT16'da kök dizinde belli miktarda dizin girişi bulunabilmektedir. Ancak FAT32'de organizasyon biraz değiştirilmiştir. FAT32'de kök dizinde istenildiği kadar çok dizin girişi bulundurulabilmektedir

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'lük dizin girişleri bulunmaktadır. Örnek bir 32'lük dizin girişini aşağıda görüyorsunuz:

54 45 53 54 20 20 20 20	54 58 54 20 18 04 D0 65	TEST	TXT	.Ge
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 kısayrsa geri kalan alanlar boşluk karakteriyle (0x20) doldurulmaktadır.
0B	BYTE	Dosyanın özelliği. Dosya özelliği kursun başlarında da ele alındığı gibi bit bit anlamlı bir alandır. Görüldüğü gibi girişin bir dosya mı yoksa bir dizin mi olduğu yalnızca özellik byte'ının 4'üncü bitinden anlaşılmaktadır.
[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.
		
		Yani yıl alanında yazan değere 1980 toplamak gereklidir.
2A	WORD	Dosyanın ilk cluster numarası. Bilindiği gibi dosyanın FAT zincirinin izlenebilmesi için onun ilk cluster numarasının bilinmesi gereklidir. 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. Peki操作系统 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 gelindiği 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 bulunan 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 kısası 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.

Göründüğü 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 OF 00 FB FF FF	CX.T... YYYY.. YY
FF FF FF FF FF FF FF	FF FF 00 00 FF FF FF FF	YYYYYYYYYY.. YYYY
02 47 00 5F 00 46 00 49	00 4C 00 0F 00 FB 45 00	.G._.F.I.L... GE.
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._... GI.
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 oluşturulduğuna diikat ediniz:

```
43      XT
02      GFILE_NAME_T
01      THIS_ISALON
???     THIS_IS~1TXT
```

Yol İfadelerinin Çözümlenmesi (Pathname Resolution)

Bir yol ifadesi verildiğinde işletim sisteminin yol ifadesiyle hedeflenen dosyaya ilişkin bilgileri ele geçirmesi gereklidir. Bu süreçte yol ifadelerinin çözümlenmesi (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 çözümlenir. Görelî yol ifadeleri ise prosesin kontrol bloğunda tutulan "çalışma dizininden (current working directory)" itibaren çözümlenir. 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 iin 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 girişî 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 eristikleri dizin girişlerini bir cache sistemi içerisinde saklamaktadır. Bu cache sistemine sıkılıkla "dentry cache" denilmektedir. "Dentry Cache" genellikle bir hash tablosu biçiminde oluşturulmaktadır. Hash tablosuna anahtar olarak yol ifadesi verilir. Değer olarak da dizin girişî 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, DIRENTY *dirEntry);

#endif

/* PathNameResolution.c */

#include <string.h>
#include <cctype.h>
#include "DiskIO.h"
#include "PathNameResolution.h"

/* static Function Prototypes */

DIRENTY *findDirEntry(void *cluster, size_t size, const char *fileName);

char g_cwd[MAX_PATH] = "g:\\";

BOOL PathNameLookup(const char *path, DIRENTY *dirEntry)
{
    int drive;
    char absPath[MAX_PATH];
    char *pathPos, *pathComp, *dirEntries, *cluCont;
    size_t dirEntrySize, cluSize;
    HANDLE hVolume;
    BPB bpb;
    DIRENTY *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, &bpb))
        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(bp.bpb.fatLen * bp.bps)) == NULL)
    return FALSE;

if (!ReadSector(hVolume, bp.bpb.fatOrigin, bp.bpb.fatLen, fat))
    return FALSE;

cluSize = bp.bpb.spc * bp.bpb.bps;
if ((cluCont = (char *)malloc(cluSize)) == NULL)
    return FALSE;

while ((pathComp = strtok(NULL, "\\")) != NULL) {
    clu = dentry->firstClu;

    foundFlag = FALSE;
    while (clu < 0xFFFF8) {
        if (!ReadCluster(&bp, 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 };

    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);
}

```

Pekiyi yol ifadelerinin çözümlenmesi işleminde nasıl iyileştirmeler yapılabilir? Bazıları şunlardır:

- FAT'in tamamı her defasında (yani örneğimizde PathNameLookup) fonksiyonunda okunmayabilir. Toplamda bir kez işin başında okunup global bir alana yerleştirilebilir.
- FAT'in 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 dizin girişleri için de bir cache sisteminin oluşturulması uygun olur. Böylece aynı dosya ya da dizinler 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 gereklidir. Ancak orada çalışma dizininin yalnızca yol ifadesi olarak değil dizin girişi olarak da saklanması uygun olur. Böylece doğrudan göreli yol ifadeleri o dizin 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'in sektör temelinde cache'lenmesine ilişkin bir örnek aşağıdaki gibi olabilir. Bu örnekte FAT'in son kullanılan sektörleri bir cache'te saklanmaktadır. Bu cache'ten eleman çıkartılmada LRU (Least Recently Used) algoritması kullanılmaktadır. Örnek yalnızca FAT16 sistemi için gerçekleştirilmiştir. GetNextClu16 fonksiyonunun cache'e bakarak çalışmasına 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 *fcn, *newNode;

    sect = pBPB->fatOrigin + clu * 2 / 512;
    hash = HashFunc(sect);
    fcn = g_cache[hash];

    while (fcn != NULL) {
        if (fcn->sect == sect) {
            if (g_lruTail != fcn) {
                if (fcn->lruPrev != NULL)
                    fcn->lruPrev->lruNext = fcn->lruNext;
                else

```

```

        g_lruHead = fcn->lruNext;
        if (fcn->lruNext != NULL)
            fcn->lruNext->lruPrev = fcn->lruPrev;

        fcn->lruPrev = g_lruTail;
        g_lruTail->lruNext = fcn;
        g_lruTail = fcn;
        fcn->lruNext = NULL;
    }

    return *(WORD *)(fcn->data + (clu * 2 % 512));
}
fcn = fcn->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;

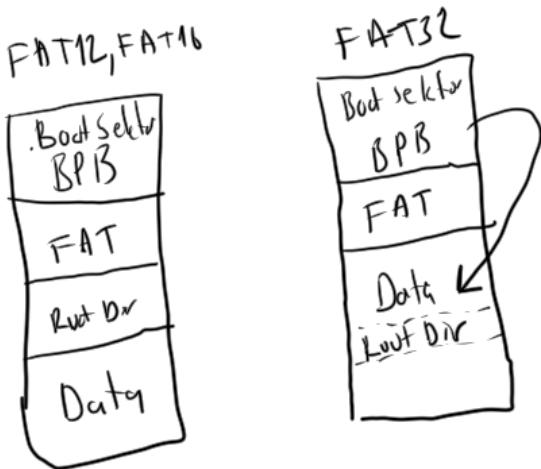
if (g_lruTail == NULL) {
    g_lruHead = g_lruTail = newNode;
    newNode->lruPrev = NULL;
}
else {
    g_lruTail->lruNext = newNode;
    newNode->lruPrev = g_lruTail;
    g_lruTail = newNode;
}
newNode->lruNext = NULL;

return *(WORD *)(buf + (clu * 2 % 512));
}

```

FAT32 Sistemlerinin Farklılıklarları

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'dakinin biraz değiştirilmiş biçimidir.

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 Yapısunın 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'sı
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'inci 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 kapsılsa hangi kopyanın aktif olduğu da burada belirtilmektedir. Ayrintılı bilgi için "FAT32 Filse System Specification" dokümanını inceleyiniz.

FAT32 Versiyon Numarası: Burada FAT32 sisteminin versiyon numarası bulunur.

Root Dir Bölümünün Cluster Numarası: Burada Root Dir bölümünün Data bölümünün kaçinci cluster'ından başladığı bilgisi bulunmaktadır. (Animsanacağı 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. Animsanacağı gibi FAT12 ve FAT16'da volümdeki boş cluster'ların sayısı herhangi bir yerde tutulmamaktadır. FAT bölümündeki 0 olan FAT elemanları sıyılarak 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'sı (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 değiştirilmemiğini kontrol amacıyla kullanılmıştı. Bugün pek bir kullanım amacı kalmamıştır. (Bu değer teil bir değer değildir. Kopya koruma gibi amaçlarla bu değerden faydalılanması doğru yaklaşım değildir.)

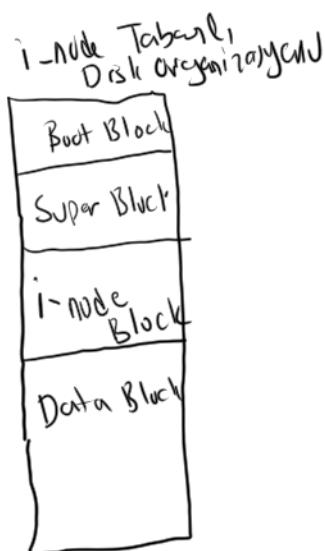
Volüm Etiketi (Volume Label): Bu bilgi de FAT12 ve FAT16 BPB'sinde vardır. Bu alanda 11 byte'lık volüme etiketi bulunmaktadır. Ancak volüm etiketi FAT dosya sistemlerinde kök dizinde "Volume Label" özniteliğine (attribute) sahip bir dizin girişi olarak da tutulmaktadır. DOS ve Windows uzun süredir volüm etiketini BOOT sektör BPB bloğunbdan 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:



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 gereklidir. 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 aygit sürücüsü her okunduğunda sıfır değerini veren bir dosya gibi davranış gösterir. 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 tanıtılmıştır:

```
sudo losetup /dev/loop0 mydisk.dat
```

Artık /dev/loop0 aygit sürücüsü mydisk.dat dosyasını kullanacak hale getirilmiştir.

3) Oluşturduğumuz volümü mkfs ile formatlayabiliriz:

```
sudo mkfs -t ext2 /dev/loop0
```

4) Artık oluşturulan dosya sistemi mount edilebilir. Tabii bunun için önce mount noktasında içi boş bir dizin açmak gereklidir. Geleneksel olarak mount edilecek aygıtlar için /mnt dizinin 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
```

Artık mount işlemini yapabiliriz:

```
sudo mount -t ext2 /dev/loop0 /mnt/mydisk
```

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 volumde 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 umount edilmelidir:

```
sudo umount /mnt/mydisk
```

Boot blokta -tipki 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. Peki süper blokta nasıl bilgiler vardır? Örneğin Ext-2 dosya sisteminde süper blok aşağıdaki elemanlara sahiptir:

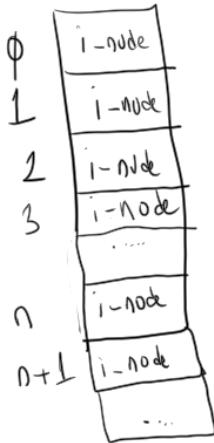
Offset (bytes)	Size (bytes)	Description
0	4	s_inodes_count
4	4	s_blocks_count
8	4	s_r_blocks_count
12	4	s_free_blocks_count
16	4	s_free_inodes_count
20	4	s_first_data_block
24	4	s_log_block_size
28	4	s_log_frag_size
32	4	s_blocks_per_group
36	4	s_frags_per_group
40	4	s_inodes_per_group
44	4	s_mtime
48	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 elemanın ilki 0 olmak üzere bir numarası vardır.

i-node Block



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 bulumaktadır. 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 gereklisiyle- bazı bilgileri de bilgileri bize vermemektedir. Ext-2 dokümanlarında i-node elemanın içeriği şöyle dokumente 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

Offset (bytes)	Size (bytes)	Description
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

Gördüğü gibi Ext-2'de bir i-node elemanı 128 byte uzunluğundadır.

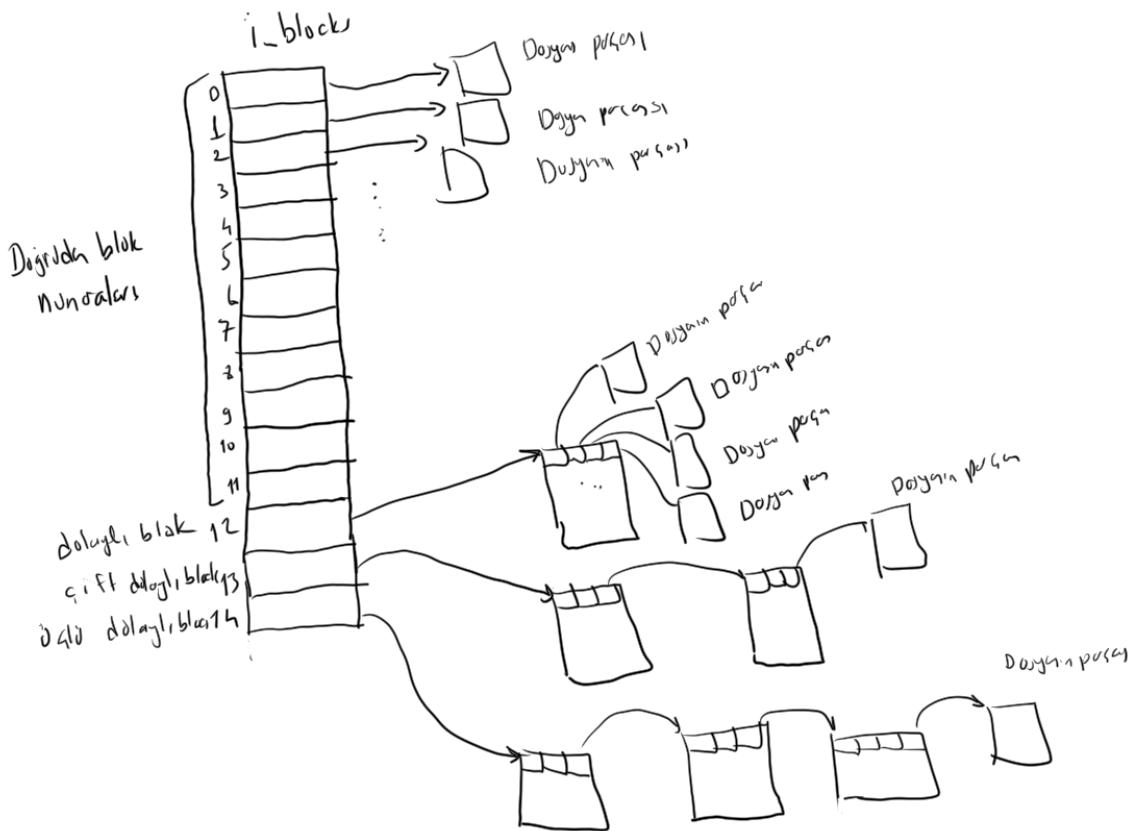
I-node tabanlı dosya sistemlerinde volümdeki toplam dosyaların sayısı i-node blok'ta ayrılmış i-node elemanlarının sayısından fazla olamaz. Çünkü her dosya için (dosyanın büyüklüğü ne olursa olsun) i-node blok'ta bir tane i-node elemanı tahsis edilmek zorundadır.

Bir dosyanın i-node numarası o dosyanın bilgilerinin i-node blok'taki hangi i-node elemanında bulunduğu belirtir. ls komutunda -i seçeneği ile dosyaların i-node numaraları elde edilebilir. Örneğin:

```
csd@csd-VirtualBox /mnt/mydisk $ ls -li
total 14
drwxr-xr-x 2 csd csd 4096 Jan 23 11:11 Music
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. Pekiyyi 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 elemanın 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 olduğu 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:

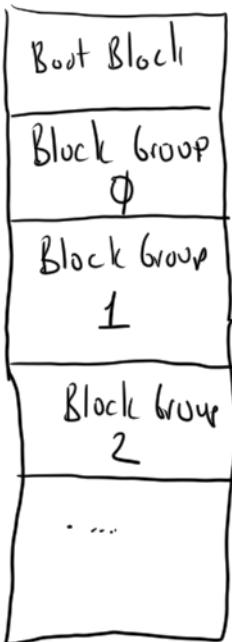
```
block_size = 1024 << s_log_block_size;
```

Bir bloğun 1024 byte (yani 2 sektör) uzunlığında olduğunu varsayıyalım. Bu durumda eğer dosya $12 * 1024$ byte'tan küçük ya da ona eşitse dosyanın bloklarının numaraları doğrudan i-node elemanın `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 varsayırsak 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 elemanda belirlenen numaralı bloğa gittiğimizde onun içerisinde 256 tane blok numarası görülür. 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.

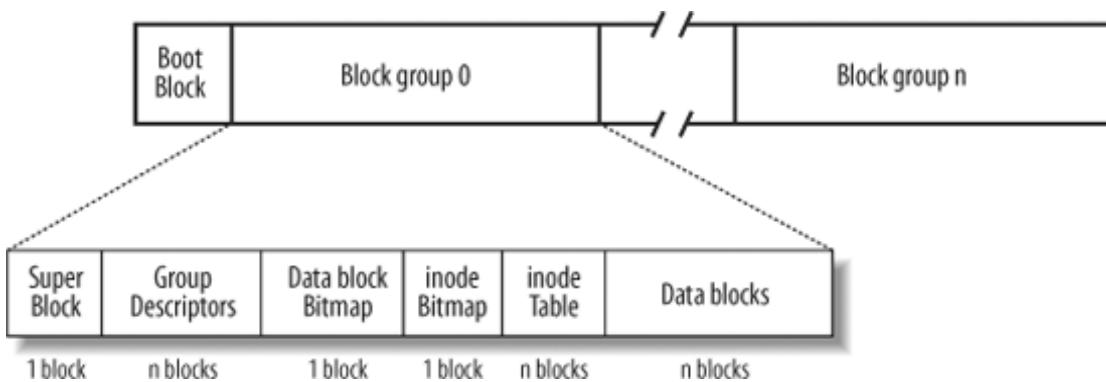
i-node tabanlı dosya sistemleri mimari olarak birbirlerine çok benzese de ayrıntılarda farklılıklar söz konusudur. Bu nedenle burada biz spesik olarak Ext-2 dosya sistemine özgü disk organizasyonu üzerinde de durmak istiyoruz.

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:



Gördüğü gibi aslında ext-2 disk organizasyonu blok grouplardan 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 grplardaki 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'inci (yani ilk) blok gruptaki süper bloğun kopyalarını tutar. Blok grubun ikinci elemanı "Grup Betimleyicilerinden (Group Descriptors)" oluşturmaktadır. 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 ise Super Blokta tutulmaktadır. Burada görüldüğü gibi her blok grubunda aslında backup amaçlı blok betimleyicilerinin de kopyası bulundurulmaktadır.

Yani blok gruplarındaki 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 grupta olduğunu şöyle tespit edebiliriz:

`n / s_inodes_per_group`

Her blok grubunda eşit sayıda i-node elemanın bulunduğu dikkat ediniz.

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 aygit 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:

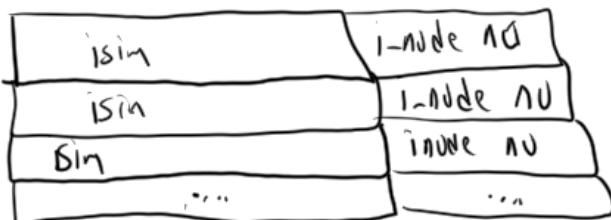
00000800	04 00 00 00 05 00 00 00	06 00 00 00	D0 01 34 00 G.4.
00000810	02 00 04 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00000820	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00000830	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

Burada volümün bir tek blok grubundan olduğunu 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 görmektesiniz:

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 01 00 10 00 00 00	cş.W.....
00001DAO	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 00ÿÈ#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 dizin içeriğinin formatı bakımından farklılıklar vardır. Ancak burada biz bu formatı genel olarak aşağıdaki biçimdeymiş gibi gösterebiliriz:



Ext-2 dosya sisteminde dizin 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

Gördüğü gibi Ext-2'de dizin 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 kısaltır. Tablodaki inode isimli eleman ilgili dosyanın i-node numarasını tutar. `rec_len` ilgili dizin girişinin kaç byte uzunlukta olduğunu belirtir. Yani sonraki giriş burada belirtilen değer kadar ileridedir. `name_len` değişken olan dosya isminin kaç karakter uzunlığında olduğunu tutar. `file_type` dosyanın türünü belirtir. Gerçi bu tür bilgisi aynı zamanda i-node elemanın içinde de vardır fakat "yol

ifadesinin çözümlenmesi (path name resolution)" işleminin hızlı bir biçimde yapılabilmesi için bu bilgi ayrıca Ext-2'de dizin girişinde de bulundurulmaktadır.

Pekiyi i-node temelli bir dosya sisteminde bir yol ifadesi nasıl çözümlenir? 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 dizin içerisinde diğer yol bileşeni, onun içerisinde de diğeri bulunarak süreç ilerletilir.

Aşağıda kök dizine ilişkin bir dizin içeriğini görüyorsunuz:

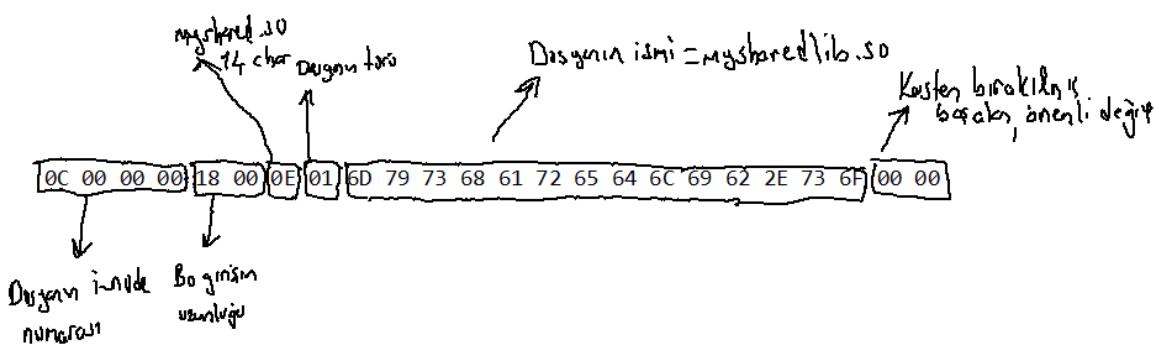
00003800	b2 00 00 00 0C 00 01 02 2E 00 00 00 02 00 00 00	[].....
00003810	0C 00 02 02 2E 2E 00 00 0B 00 00 00 14 00 0A 02
00003820	6C 6F 73 74 2B 66 6F 75 6E 64 00 00 0C 00 00 00	lost+found.....
00003830	18 00 0E 01 6D 79 73 68 61 72 65 64 6C 69 62 2Emysharedlib.
00003840	73 6F 00 00 0D 00 00 00 BC 03 08 01 74 65 73 74	so.....14....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	b2 00 00 00 0C 00 01 02 2E 00 00 00 02 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 2Emysharedlib.
00003840	73 6F 00 00 0D 00 00 00 BC 03 08 01 74 65 73 74	so.....14....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.

Diskin 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 kurabiliyoruz. Windows NTFS sistemini kullanıyorken, 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'inci sektörüne) "Master Boot Record" denilmektedir. Bu ilk sektörün yapısı şöyledir:



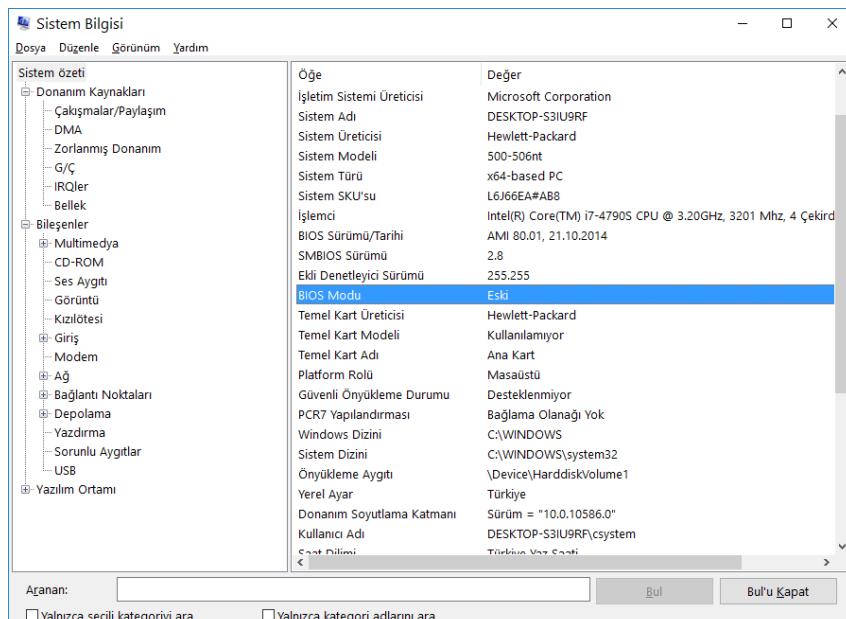
Disk Bölümleme Tablosu her biri 16 byte olan 4 girişe sahiptir. Bu durumda diskte en fazla 4 ana disk bölümü bulunabilmektedir. Daha fazla disk bölümü oluşturabilmek için "uzatılmış disk bölümü (extended partition)" kavramı uydurulmuştur. 16 byte'lık disk bölümleme tablosu elemanın formatı şöyledir:

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
12 - 15	4	Partition Size (in sectors)

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 anlaşıılır. UEFI tarzı disk organizasyonunda disk bölümleme tablosuna "GUID Bölümleme Tablosu (GUID Partition Table)" denilmektedir. Kursumuzda GUID Bölümleme Tablosu ele alınmayacaktır.) Görüldüğü gibi her disk bölümü için o disk bölümünün hangi sektörden başladığı ve ne uzunlukta olduğu bilgisi de disk bölümleme tablosu elemanında belirtilmektedir. Tablodaki diğer elemanların artık önemi kalmamıştır.

Disk bölümleme 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 yapılmıyor olabilir. Bunun için "CMOS Setup'a girişi UEFI desteğini açmak gerekebilir.

MBR sektörünün başında küçük bir "boot loader" programı bulunur. Bu program disk bölümleme 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 sektörə zaten "boot sector" deniliyordu)

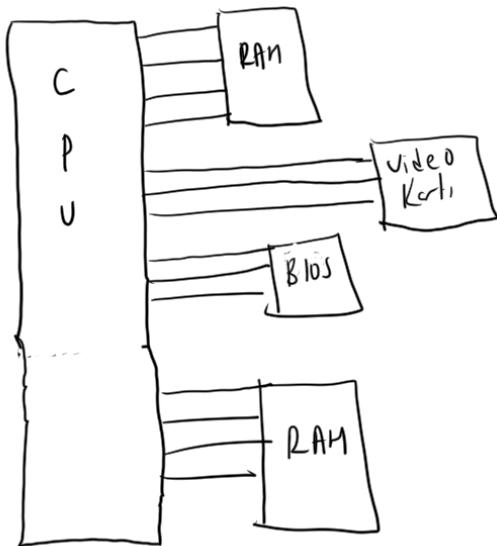
Biz burada disk bölümleme tablosundaki genişletilmiş (extended) disk bölümlerinin nasıl oluşturulduğu ve GUID disk bölümleme tablosunun yapısını ele almayacağız. Bu konu "Sistem Programalama ve İleri C Uygulamaları II" numaralı kursun konusu içerisindeindedir.

Intel Tabanlı Bilgisayarların Boot Süreci

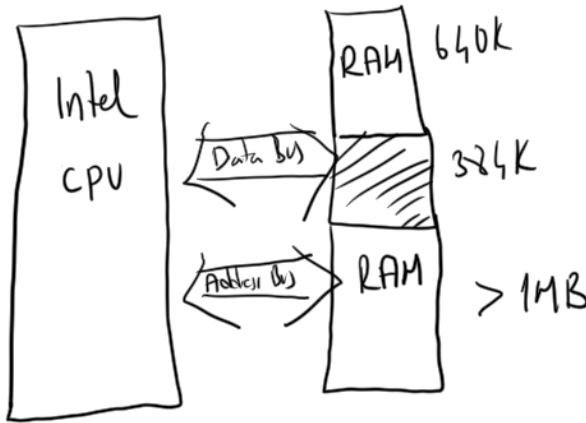
Bugün kullandığımız Intel tabanlı masaüstü ve dizüstü bilgisayarlarımıza açtığımızda işletim sisteminin otomatik olarak yüklediğini görürüz. Peki bu süreç nasıl gerçekleşmektedir? İşte çok ayrıntılarına girmeden PC'lerimizdeki tipik boot süreci şöyledir:

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 beklete 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 elektroniksel düzeyde kolay bir biçimde yapılmaktadır. Örneğin:

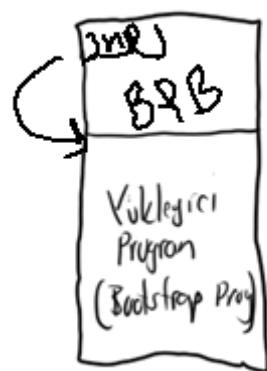


Tabii bizim sistem programcısı olarak adres alanının hangi kısımlarının hangi aygıtlar içerisindeki belleklere bağlı olduğunu bilmemiz gerekebilir. Örneğin PC mimarisinde ilk 1MB belleğin son 384K'sı başka aygıtlara yönlendirilmiştir. (Bu nedenle DOS işletim sistemi $1024K - 384K = 640K$ kullanabiliyordu). Yani PC RAM'ı 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 aygıtını olarak seçtiğini düşünelim.
- 3) BIOS'taki bootstrap kod boot aygıtının (örneğin C sürücüsünün) ilk sektörünü bellege yükler (7C00 adresine) ve oradaki koda jump eder. Böylece diskin MBR sekötüründeki "yükleyici program" çalıştırılmış olur.
- 4) MBR'deki yükleyici program da aktif disk bölümünü tespit ve onun ilk sektörünü bellege 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 sektörde 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) bellege okuyarak oraya jump eder. Buradaki program aktif disk bölümünü tespit ederek onun ilk sektörünü bellege okur ve oraya jump eder. İşletim sistemini de oradaki kod yükler.

Örnek Çalışma: Bilgisayar Açıldığında Bizim Yerleştirdiğimiz Bir Programın Çalıştırılması

Bunu sağlamadan en pratik yolu sembolik makine dilinde bir program yapıp onu hard diskin, flash belleğin ya da CD'nin ilk sektörüne yerleştirmektir. Ve makineyi o sürücüden boot edilecek biçimde "CMOS Setup"tan ayarlamaktır. Ekrana Intel gerçek modda merhaba dünya yazısını çıkartan bir nasm programı şöyle yazılabılır:

```
[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.

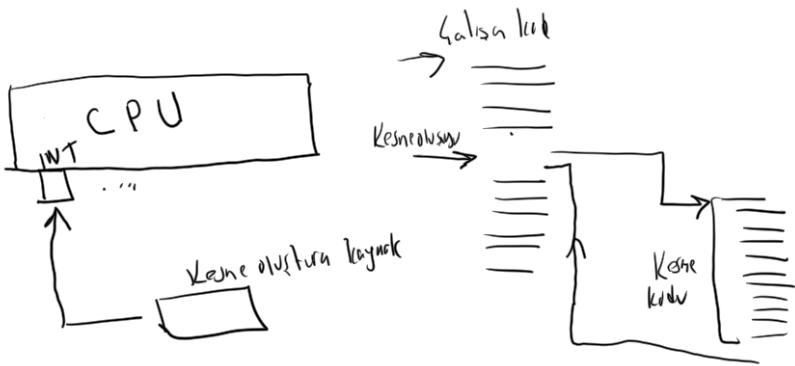
Kesmeler (Interrupts)

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".

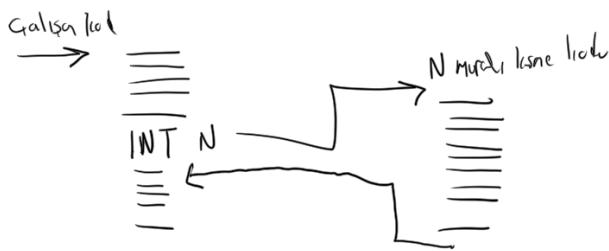
Kesme işlemciye özgü bir kavramdır. İşlemcilerin hemen hepsinde (mikrodenetleyiciler de dahil olmak üzere) çeşitli biçimlerde kesme kavramları vardır. Kesmeler fonksiyon çağrılarına (yani CALL işlemine) benziyor olsa da aslında çeşitli bakımlardan fonksiyon çağrılarından farklıdır.

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 programçının kendisidir. Zaten yazılım kesmelerinin mekanizma olarak fonksiyon çağrılarından (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" denir. Sayfalama mekanizması ve koruma mekanizması hep bu tür içsel kesmelerle yönetilmektedir. Örneğin işlemci yetkisi olmayan kod bir bellek alanına erişmeye çalıştığında "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.

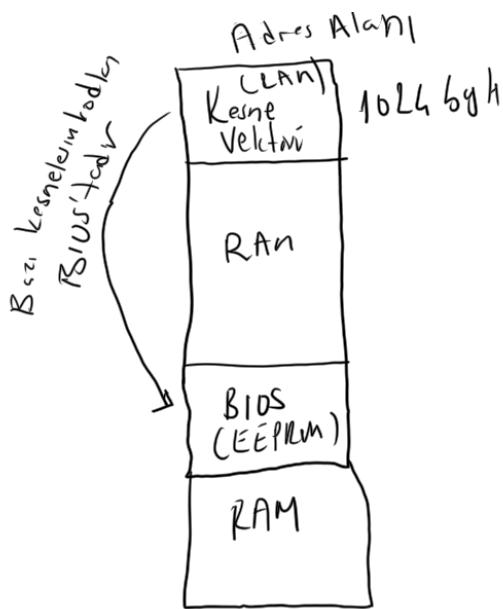
Kesmenin cinsi ne olursa olsun bir kesme oluştduğunda araya girilip çalıştırılan koda "kesme kodu (interrupt handler)" denilmektedir. Pek çok işlemcide kesmelerin numaraları vardır. Yani numaralar sayesinde çok sayıda kesme varmış gibi bir etki oluşturulabilmektedir. Örneğin Intel işlemcilerinde toplam 256 kesme vardır. Zaten örneğin INT makine komutu kesme numarasını argüman olarak alır. Örneğin:

```
....  
int      0x10          ; 10h numaralı kesme yazaılımsal olarak çağrılıyor  
....
```

Geleneksel olarak kesme numaraları Intel'de 16'lık sistemde belirtilmektedir.

Peki bir kesme oluştduğunda kimin kodu çalıştırılmaktadır? İşte Intel işlemcilerinde bir kesme oluştduğunda işlemci hangi kodu çalıştıracağına 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 oluştduğunda 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 gereklidir. 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 vektöryle oynayarak kendi kodunun çalışmasını sağlayabildiğine dikkat ediniz. Kullandığımız PC'ler reset edildiğinde kesme vekktörü de sıfırlanmış durumdadır. Çalışma BIOS'ta naşlar 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 süreçte kesmenin kancalanması (hook edilmesi) denilmektedir. DOS zamanındaki pek çok virus 8h gibi 13h gibi BIOS kesmelerini kancalıyor ve bulaşmayı böyle sağlıyordu.

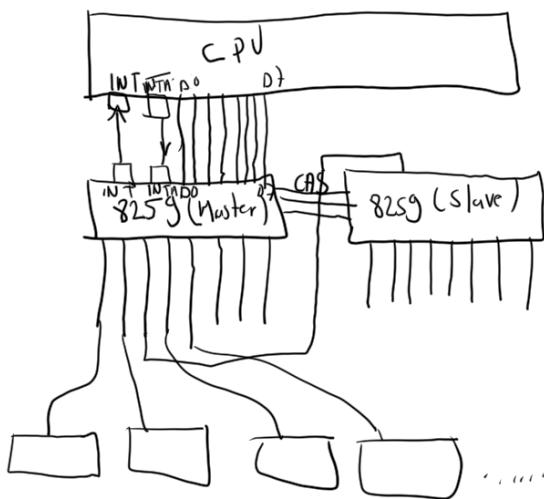
Intel işlemcileri reset edildiğinde "gerçek mod (real mode)" denilen bir modda çalışmaya başlar. Gerçek mod işlemcisinin 16 bit 8086 gibi (yani DOS'un kullanıldığı işlemci) çalıştığı moddur. Intel işlemcilerinin "korumalı mod (protected mode)" denilen diğer bir çalışma modu daha vardır. Örneğin Windows, Linux korumalı modda çalışmaktadır. Bu işletim sistemleri yüklenirken işlemci gerçek moddan korumalı moda geçirilmektedir.

Korumalı modda kesme vektörünün formatı ve yeri değişmektedir. Korumalı 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 korumalı moda geçmeden önce kesme betimleyici tablosunu oluşturur. IDTR yazmacının burayı göstermesini sağlar.

Peki korumalı 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 aygit sürücü oluşturmak gereklidir. Tabii aygit sürücülerin belli mimarileri vardır. Biz aygit 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 aygit sürücü faydalansın 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 uyarıdağında kesme denetleyicisi de CPU'yunun INT ucunu uyarmaktadır. AT'lerle (PC'lerin 80286 işlemciyle 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 çalıştığı 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 yapılmaktadır. CPU'yu yeniden donanım kesmelerine açmak için ise STI (Set Interrupt Flag) makine komutu kullanılabilir.

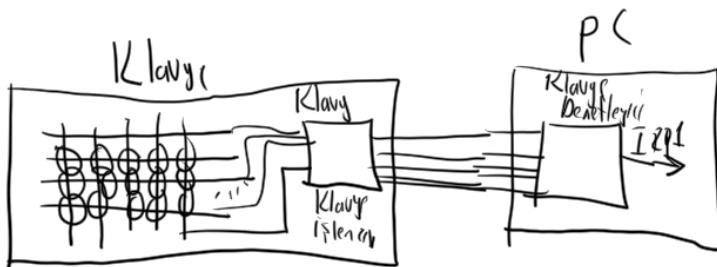
Bugünkü PC mimarisinde kesme denetleyicisinin giriş uçlarından her birine IRQ denilmektedir. Birinci kesme denetleyicinin ilk ucu IRQ0 olmak üzere her IRQ'ya bir numara verilmiştir. IRQ2 hattı kaskat bağlantı için kullanıldığından bu hatta bir donanım birimi bağlı değildir.

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 programlanaktadır.

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 kullanılabilmektedir.

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 deversesi bir çeşit matris gibidir. Tellerin kesişim noktasında tuşlar vardır. Böylece elektriksel olarak hangi tuşa basıldığı anlaşılmaktadır. Bu matrisin uçları klavye içerisindeki kalvye işlemcisine (keyboard encoder) giridi olarak verilir. Klavye işlemcisi hangi tuşa basılmış olduğunu tespit eder ve bunu klavye kablosu yoluyla bilgisayar tarafından kalvye 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 aldığında tuşun hangi tuş olduğunu anlar. İşletim sistemleri bölgesel ayarlara bakarak o tuşun aslında hangi karakterle (klavye üzerinde gördüğümüz simbol) 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 tarafından 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 sonmra 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 ilettilir. Oluşan kesmenin kodu bu tuşun NUM-LOCK tuşu olduğunu anlar bu kez klavye içerisindeki işlemciyi programlayarak bu işgi yakılmasını sağlar.

Klavypedeki tuşlara yönelik önemli bir durum da "typematic" sürecidir. Bir tuşa basıp parmağımızı o tuşa beklettiğimizde sürekli o tuşa basılmış 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 tarafındanklavye denetleyici yoluyla (tabii dolayısıyla yazılım yoluyla) programlanabilmektedir.

IRQ2: Bu hat kaskat bağlantından 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 bilgisayarlarımıza 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 kullanılabilmektedir.

IRQ5: Bu eski PC'lerdeki LPT2 olarak bilinen paralel port işlemcisine bağlıdır. Paralel portlar eskiden bilgisayar yazıcı haberleşmesi için kullanıyordu. Artık uzunca bir süredir paralel portlar da bilgisayarlarımıza bulunmamaktadır. Artık bilgisayar yazıcı arasındaki bağlantılar için yine USB portlarını kullanmaktadır. IRQ5 günümüzde artık ses kartları tafadan kullanılmaktadır. Ses kartları da bilgisayarda kesme oluşturabilmektedir.

IRQ6: Eskiden floppy disketler vardı. Bu disketler floppy sürücüsüne takılarak çalıştırılırlardı. İlk floppy sürücüsü için Microsoft A harfini ikinci floppy sürücüsü içi B harfini kullanıyordu. Hard disklere de C harfi verilmiştir. 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 bilgisayarlarımıza 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 basıldığından çıkan setup ekranındaki bilgiler aynı işlemcinin üzerindeki belleğe yazılmaktadır. Tabii bu Real Time Clock denetleyicisi bir pille beslenmektedir. Genellikle bu pil bilgisayarnın 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 bilgisayarlarımıza kullandığımız PCI slotlar aslında IRQ hatlarına da bağlıdır. İşte PCI kartları tasarlayan donanımcılar birtakım olaylarda PCI slot yardımıyla IRQ oluturabilmektedir.

IRQ12: Fare dediğimiz aygit 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 portları kullanılmaktadır.

IRQ13: Bu IRQ hattına Intel'in 80387, 80487 matematik işlemcileri bağlıdır. Matematik işlemci sorunla karşılaşlığında ana işlemciye bu IRQ hattını kullanarak kesme gönderiyordu. Intel'in 80486 DX modeliyle modeliyle birlikte matematik işlemciler de ana işlemci ile aynı entegre devre içeresine 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.