İşletim Sistemlerine Giriş

İşletim Sisteminin Tanımı

Donanımsal ve Yazılımsal Kaynaklar

Kısa Tarihçe

- Toplu programlama
- Çoklu programlama
- İnteraktif sistem
- Zaman paylaşımlı sistemler

İşletim Sisteminin Tanımı

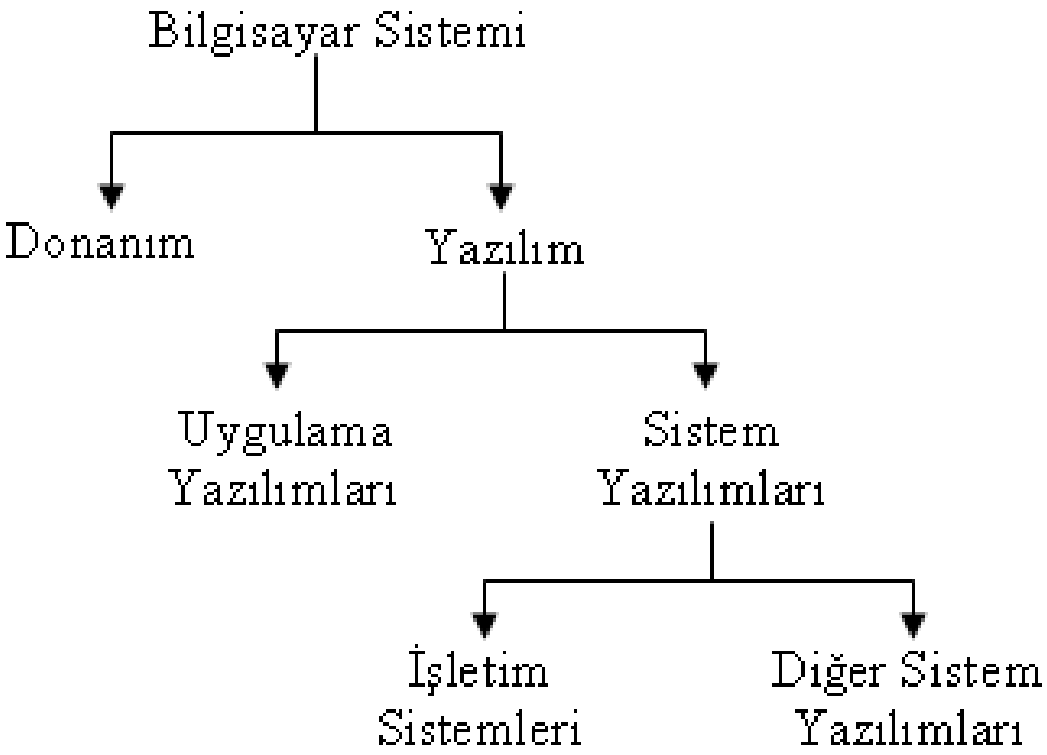
İşletim sistemi, bilgisayarın kullanıcısı ile bilgisayarın donanımı arasında hizmet veren bir programdır. Kullanıcılarına programlarını çalıştırabilecekleri bir ortam sunar. Ana hedefi, bilgisayar sistemini oluşturan donanım ve yazılım kaynaklarının kullanıcılar arasında **fonksiyonel, güvenilir ve verimli** bir şekilde kullanılmasını sağlamaktır.

Donanımsal ve Yazılımsal Kaynaklar

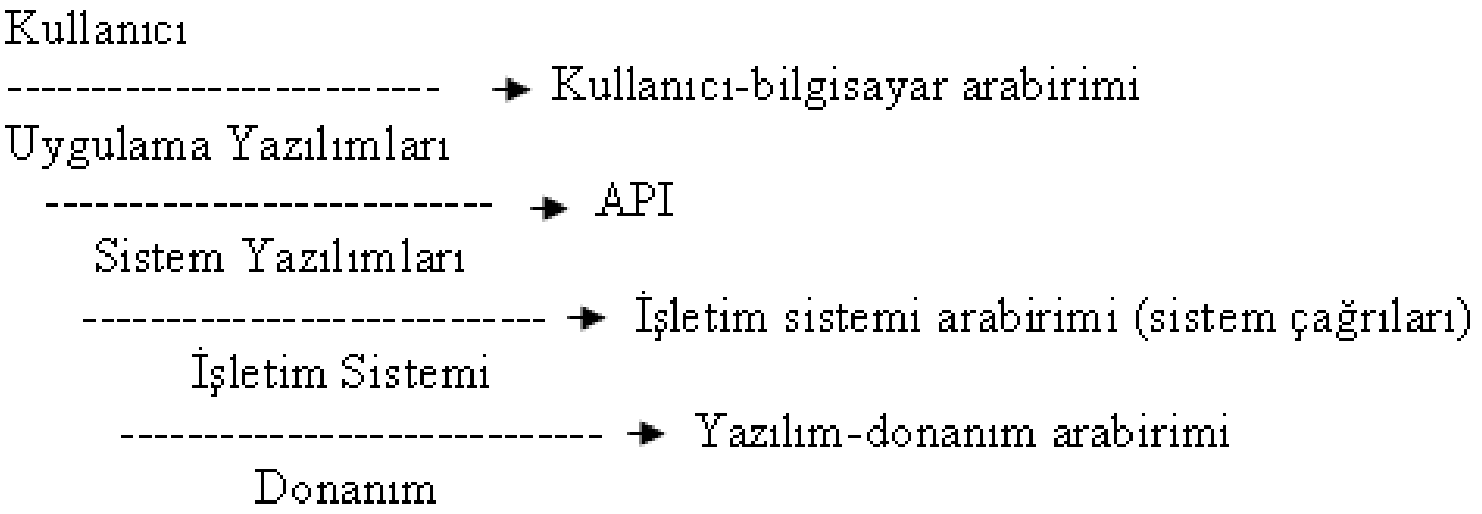
- Bir bilgisayar sisteminin en önemli kaynaklarından biri olan donanım; merkezi işlem birimi (Central Processing Unit - CPU), ana bellek (Main Memory) ve giriş/çıkış (I/O) birimlerinden (disk, disket, klavye, monitör vs.) oluşur.

- Diğer önemli kaynak ise yazılımdır ve iki kısımda incelenir. Bunlar uygulama yazılımları ve sistem yazılımlarıdır.

Yazılımsal Kaynaklar



Katmanlı Gösterim



Kısa Tarihçe

İşletim sistemi ve bilgisayar mimarisinin gelişimi arasında yakın bir ilişki vardır.

1. Jenerasyon (1945-1955): Vakum tüpler kullanılarak sayısal makineler geliştirilmiştir. Bu bilgisayarlar bir konsol üzerinden çalıştırılan çok büyük makineler idi. Programlama kontrol panelinde bulunan soketlere ilgili kablolar takılarak makine dili ile yapılırdı.

Kısa Tarihçe

2. Jenerasyon (1955-1965): Transistörlerin icadı ile başlayan dönemdir. Ortak giriş cihazları kart okuyucular ve teyp sürücüleri, ortak çıkış cihazları ise teyp sürücüleri ve satır yazıcılardı. Bu sistemlerin kullanıcıları, bilgisayar sistemiyle direkt olarak etkileşim kuramazlardı. Kullanıcı, program, data ve bazı kontrol bilgilerini içeren bir işi bilgisayar operatörüne verir, bir zaman sonra da programın sonuçlarını içeren çıktıları alırdı.

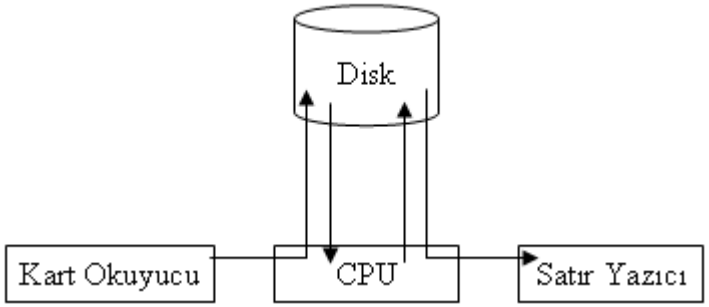
İşletim sistemi oldukça basitti. Ana görevi bir işten diğerine kontrolü vermekti. İşletim sistemi daima hafızada yer alırdı. İşlemleri hızlandırmak için benzer gereksinimleri olan programlar bir grup olarak toplu bir biçimde bilgisayarda çalıştırılırdı. Bu dönemde özellikle bilimsel işler için Fortran dili kullanıldı. Bu derleyici bir kere yüklendikten sonra çok sayıda programcının işleri teyp sürücüsüne sırasıyla aktarılır ve işletilirdi. Bu işletim sistemi stratejisi **toplu işlem (batch)** olarak anılır.

Kısa Tarihçe

3. Jenerasyon (1965-1980): Disk teknolojisinin gelişiminden önce delikli kartlar direkt olarak hafızaya alınıyor, daha sonrada işler sırasıyla çalıştırılıyordu. Delikli kart ya da teyp sürücüden okuma/yazma işlemi boyunca CPU tamamen bekleme durumunda kalıyordu.

Bu teknolojiyle delikli kartlar kart okuyucudan direkt olarak diske aktarılmış ve kartların imajlarının yerleri işletim sistemi tarafından bir tabloda tutulmuştur. Bir iş çalıştırıldığında kart okunacağı zaman disk bölgesinden okunuyor benzer şekilde iş yazıcıdan bir satır çıkaracağı zaman da sistem tamponuna kopyalanıyor, sistem tamponundan (buffer) diske yazılıyor ve iş tamamlanınca çıktısı basılıyordu. Bu teknik spooling (Simultaneous Peripheral Operation Off-Line) olarak bilinir.

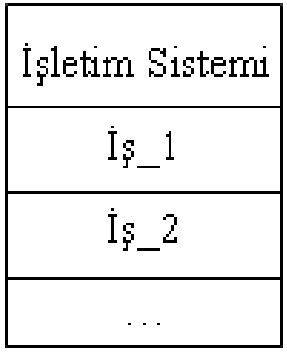
Spooling



Spooling tekniği, iş havuzu kavramını doğurmuştur. İş havuzu, işletim sisteminin CPU verimliliğini arttırmak için hangi işin sonraki adımda işletileceğini seçmesine imkân tanır.

Çoklu Programlama (Multiprogramming)

İşletim sistemi aynı zamanda birkaç işi hafızada tutar. Bu işler, iş havuzunda tutulan işlerin bir alt kümesidir. İşletim sistemi hafızadaki işlerden birini seçer ve çalıştırır. Çalıştırılacak işler olduğu sürece CPU boşta kalmayacaktır.



Çoklu Programlama (Multiprogramming)

Çoklu programlama mantığıyla birlikte iş yönetimi, bellek yönetimi ve CPU planlaması gibi kavramlar ortaya çıkmıştır.

Çoklu programlama tekniği CPU, bellek ve I/O cihazlarının verimli bir şekilde kullanılmasını sağlamıştır. Ancak, işler bitene kadar kullanıcı sisteme hiçbir şekilde müdahale edemez. Bir programın sağlıklı bir şekilde çalışabilmesi için hatasız olarak derlenmesi gerekir.

İnteraktif Sistem

Etkileşimli (interactive) bilgisayar sistemlerinde kullanıcı ve sistem arasında gerçek zamanlı olarak iletişim kurma imkanı sağlanır. Kullanıcı işletim sistemine veya programa komutlar verir ve derhal cevap alır. Genellikle klavye giriş için, monitör de çıkış için kullanılır.

İlk bilgisayarlar tek kullanıcılı interaktif sistemlerdi. Bu ortam, programcılara programlarını test edip geliştirmelerine büyük esneklik sağlamıştı. Ancak bazı hareketlerin programcı veya operatör tarafından yapılması gerektiğinden CPU'nun boşta kaldığı süre oldukça fazla idi.

Zaman Paylaşımlı Sistem (Time-sharing)

Zaman paylaşımlı sistemler, bilgisayar sisteminin makul bir bedele interaktif kullanımı için geliştirilmiştir. Her bir kullanıcının bellekte en az bir programı vardır. Çalışmakta olan programa proses denir. Bir proses çalıştığı zaman, çalışması bitebilir ya da I/O işlemine ihtiyaç duyabilir. İnteraktif sistemde klavyeden giriş yapıldığından ve bir insanın da saniyede 5 karakter bastığı düşünüldüğünde bu hız CPU'ya göre oldukça düşük kalacaktır.

Sistem, hızlı bir şekilde kullanıcılar arasında anahtarlama yaptığından her kullanıcıya kendi bilgisayarıymış izlenimi verir.

Kısa Tarihçe

4. Jenerasyon (1980 - Günümüze): Büyük ölçekli entegre devrelerinin üretimi ile birlikte tek bir kullanıcıya adanan kişisel bilgisayarlar (PC) dönemi başlamıştır. Giriş/çıkış cihazları da oldukça değişmiştir. Kart okuyucular yerlerini klavye ve fareye, satır yazıcılar da monitör ve hızlı yazıcılara bırakmıştır. Artık, bilgisayar hakkında çok az bilgiye sahip kullanıcıların dahi bunlar üzerinde çalışan programları rahatlıkla kullanabilmelerine olanak sağlanmıştır.

4. Jenerasyon

1985'li yıllarda ağ teknolojileri ve PC'lerdeki gelişimin neticesinde ağ işletim sistemleri (Network operating systems) ve dağıtık işletim sistemleri (Distributed operating systems) ortaya çıkmıştır.

Network işletim sisteminde kullanıcılar, ortamda bulunan çok sayıda bilgisayarın varlığından haberdardır, uzaktaki makinelere bağlanabilir ve dosyalarını bir bilgisayardan diğerine kopyalayabilirler.

Dağıtık işletim sistemlerinde ise aslında ortamda çok sayıda işlemci olmasına karşın kullanıcılarına geleneksel tek işlemcili gibi gözükmesidir. Kullanıcılar, dosyalarının bulunduğu ortamdan ve programlarının nerede çalıştırıldığından habersizdir.

BİLGİSAYAR SİSTEMLERİNİN YAPISI

- Mikroişlemci
  - CPU (Central Processing Unit)
  - ALU (Arithmetic Logic Unit)
  - Kontrol Ünitesi
  - Program Sayacı ve İşlemci Modları
- Ana Bellek (Main Memory)
- Giriş/Çıkış Cihazları
  - Kontrolör ve Sürücü Yazılımları
  - Arabirimler
  - Kontrolör Arabirimi
  - Doğrudan Bellek Erişimi-DMA
  - Kesmeler
  - Memory-Mapped I/O

BİLGİSAYAR SİSTEMLERİNİN YAPISI

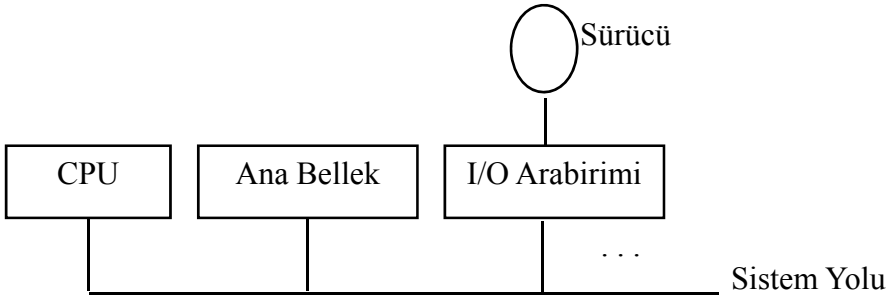
Bilgisayarlar ortaya çıkmadan önce işe özgü elektronik devre tasarımları yapılırdı. von Neumann mimarisi ile birlikte sabit elektronik devreler kullanılarak bu devrelerin yerine getireceği işlemler değişken programlarla tanımlanmıştır.

von Neumann mimarisi bilgisayarlar, günümüz bilgisayar sistemlerinin temelini oluşturur.

Bilgisayar Mimarisinin Bileşenleri

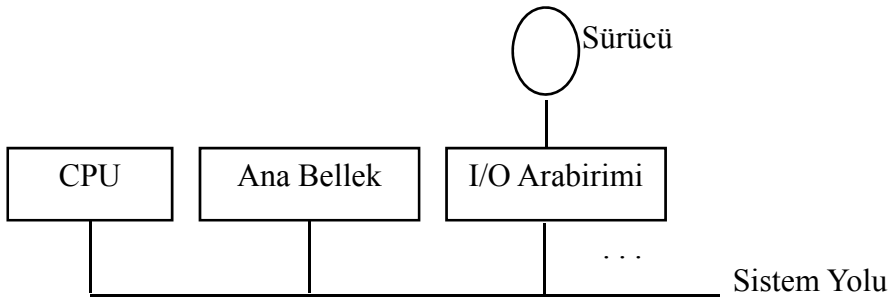
Bu mimari dört kısımdan oluşur;

- ✓ İşlemci (Central Processing Unit-CPU),
- ✓ Ana bellek (main memory) ve
- ✓Giriş/çıkış (Input/Output-I/O) cihazları.



CPU

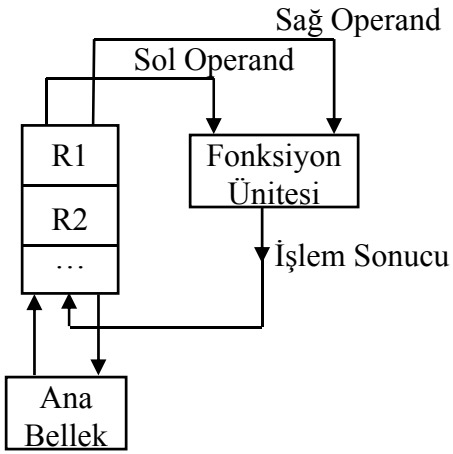
Bir bilgisayar sisteminin beynidir. Hangi komutların işletileceğine karar verir, bu komutları çözer ve işletir. Her işlemci kendisine has komut setine sahiptir. Bir CPU, aritmetik lojik ünitesi (Arithmetical Logical Unit-ALU) ve kontrol ünitesinden oluşur.



ALU

ALU, aritmetik ve lojik işlemlerin yapıldığı bir fonksiyon ünitesine sahiptir.

Örneğin (x+y) toplama işleminde sol operand x, sağ operand y ve işlem + ‘dır.



z = x+y işlemi assembly dilinde;

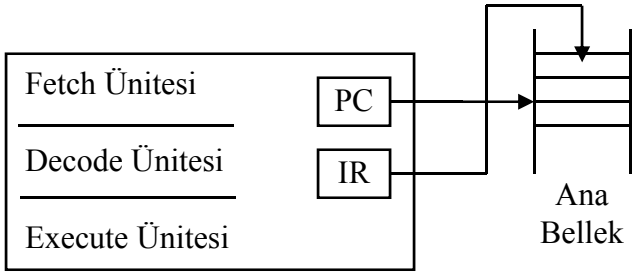
load R1, x (x’i bellekten R1’e yükle)
load R2, y (y’yi bellekten R2’e yükle)
add R1, R2 (Toplama işleminin sonucu R1’e)
store R1, z (R1’i bellek hücresi z’ye yaz)

Kontrol Ünitesi

Kontrol ünitesi 3 kısımdan oluşur;

- Fetch (alıp getirme) ünitesi
- Decode (kod çözme) ünitesi
- Execute (icra etme) ünitesi

Ayrıca bazı kaydedicilere sahiptir. Bunlardan en önemlileri; program sayacı (Program Counter-PC) ve komut kaydedicisi (Instruction Register-IR)'dir.



Program Sayacı ve İşlemci Modları

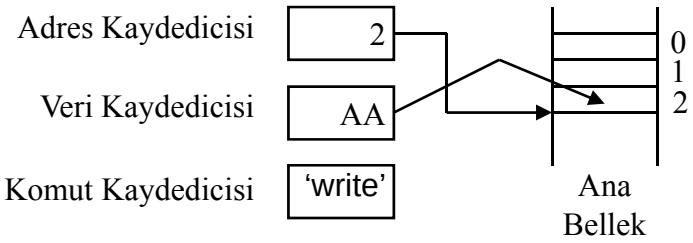
Bilgisayar açıldığı zaman program sayacına ilk değeri, boot işlemiyle yüklenir. Daha sonra komutlar sırasıyla ana bellekten okunur ve işletilir. Şayet bir programın çalıştırılması isteniyorsa işletim sisteminin program sayacına programın başladığı satırın adresini yüklemesi kafidir.

Günümüz işlemcileri supervisor veya kullanıcı modunda çalışabilmektedir. İşletim sistemi supervisor modda çalışır. Böylelikle işlemcinin komut setindeki her komutu çalıştırabilme, dolayısıyla da donanımsal kaynakların her türlü özelliğini kullanabilme yetkisine sahip olmaktadır.

Ana Bellek

Uçucu (volatile) bir yapıya sahiptir. Ana bellek üç kaydediciden oluşan bir arabirime sahiptir.Bunlar;

- Adres kaydedicisi (Memory Address Register-MAR)
- Veri kaydedicisi (Memory Data Register-MDR)
- Komut kaydedicisidir (Command Register).



Sözcük (Word) Kavramı

Günümüz bilgisayarlarında bellek hücresi genellikle 8 bitliktir. Her bir hücre 8 bit olmasına rağmen 2 veya 4 byte'lık kısımlar bir bütün olarak CPU üzerinde işlenebilir. CPU'nun işleyebildiği veri, 'word' olarak isimlendirilir.

CPU tasarımına ve makine komutlarına göre word genişliği değişiklik gösterir.

Giriş/Çıkış Cihazları

Giriş Cihazı: Dış dünyadan CPU'ya veri transfer eden alt birim giriş cihazıdır. Örneğin klavye, fare, mikrofon gibi.

Çıkış Cihazı: CPU'dan dış dünyaya veri transfer eden alt birim ise çıkış cihazıdır. Örneğin ekran, hoparlör, yazıcı gibi.

Giriş/Çıkış Cihazı: Seri ve paralel portlar, kızılötesi alıcı ve vericiler, network arabirim kartları hem giriş hem de çıkış yapabilen cihazlar kategorisindedir.

Kontrolör ve Sürücü Yazılımları

Her bir giriş/çıkış (I/O) cihazı, fiziksel cihazın kendisinin ve bu cihazın detaylı işletimini kontrol eden bir kontrolör içerir.

Çok çeşitli cihaz kontrolörleri vardır ve işletim sistemi bunların düzgün bir şekilde çalışmasından sorumludur. İşletim sistemi tüm bu kontrolörler arasındaki farklılıkları gizleyerek ortak bir arabirim sunar.

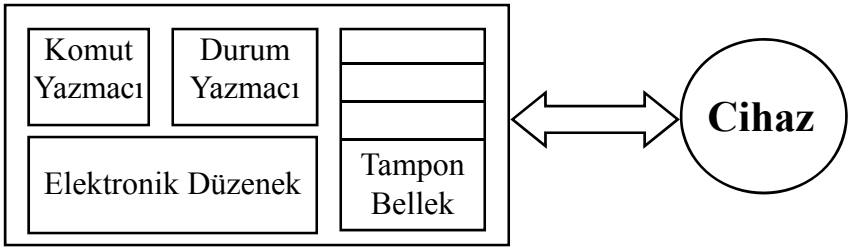
Aygıt sürücü yazılımları, cihazın kontrolörüyle etkileşim kurarak cihazı kontrol eder. Aygıt sürücüleri, uygulama yazılımlarına ortak bir arabirim sunarak kontrolör yönetiminin detaylarını gizler.

Arabirimler

Cihaz ile kontrolör arasında cihazın işleyişine bağlı olarak seri ya da paralel bağlantı olabilir. Genellikle cihaz ile kontrolör arasındaki mesafe az ise paralel, çok ise seri bağlantı tercih edilir.

Kontrolör ve cihaz üretimi yapan firmalar farklı olabileceğinden ANSI, IEEE ve ISO standartlarına uygun arabirimler geliştirilmektedir. Örneğin disk sürücüyü disk kontrolörüne bağlamada IDE (Integrated Drive Electronics) veya SCSI (Small Computer System Interface) arabirimleri yaygın olarak kullanılır.

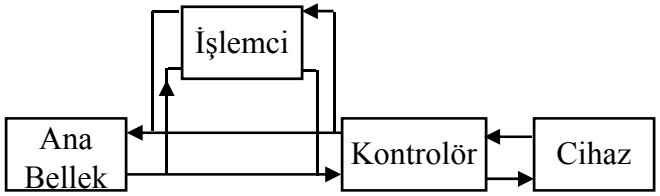
Kontrolör Arabirimi



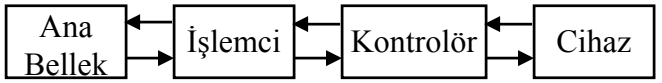
Şayet cihaz yazılımla kontrol edilecekse sürekli olarak kontrolörün durum yazmacı kontrol edilmelidir. Durum yazmacındaki ‘busy’ ve ‘done’ bayrakları kontrol edilerek veri transferi yapılabilir. Şayet bu iki bayrak da ‘false’ değerine sahipse yazılımla komut yazmacına ‘yaz’ (veya oku) komutu verilerek cihaz aktive edilir. I/O komutu geldiğinde ise busy bayrağı set edilerek işlem başlatılır. İşlem tamamlandığında done bayrağı set edilir, busy bayrağı ise temizlenir. İşlem akışına göre veriler ya tampon bellekten cihaza ya da cihazdan tampon belleğe aktarılır.

Doğrudan Bellek Erişimi-DMA

Birçok durumda veri cihazdan okunduğu zaman, programcı belleğe kopyalamak ister. Benzer şekilde veri, cihaza yazılacağı zaman da hafızadan okunacaktır. Burada CPU'nun görevi sadece cihaz ile bellek arasında bilgiyi kopyalamaktır.



DMA erişimi



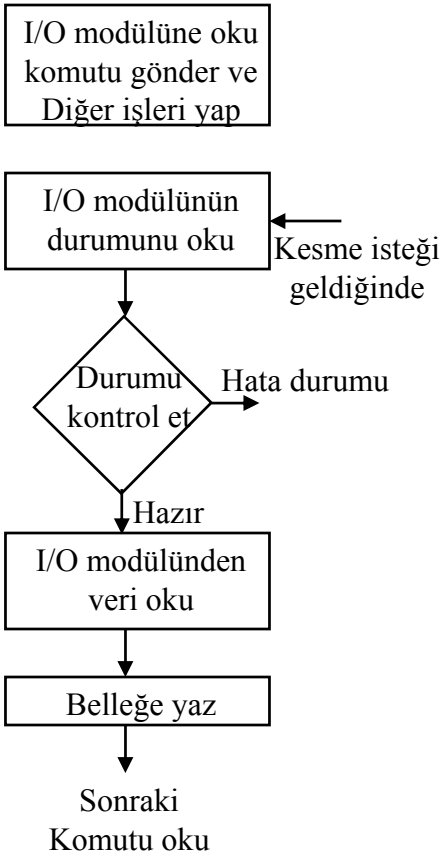
Geleneksel yöntem

Kesmeler (Interrupt)

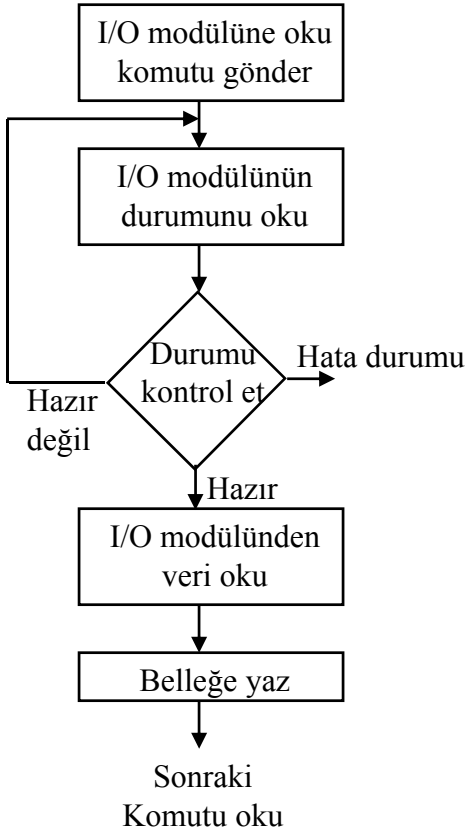
Bir aygıt sürücü yazılımı, cihaza ‘işleme başla’ komutunu verince, uygulama programı I/O işlemi bitinceye kadar ilerleme kaydedemez. Bu yaklaşım direkt I/O kutuplama (polling) olarak bilinir.

Oysa CPU, sürekli olarak kontrolörün bayraklarını kontrol etmek yerine kontrolöre bir komut verir ve çoklu programlama mantığı gereğince başka işlemler yapabilir. I/O modülü veri transferi yapmak istediğinde işlemciye bir kesme isteği gönderir, işlemeci de veri transferini yapar ve bir önceki işleme geri döner.

Kesmeler



Kesme kullanarak I/O



Kesme kullanılmadan I/O (polling)

Memory-Mapped I/O

Kontrolörün kaydedicilerinden okuma/yazma yapmak için geleneksel olarak makine komut setinde özel I/O komutları vardır. Örneğin,

input cihaz_adresi

output cihaz_adresi

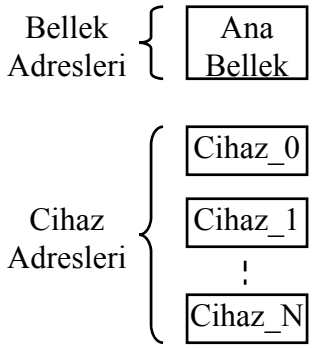
copy_in CPU_kaydedicisi, cihaz_adresi, kontrolör_kaydedicisi

copy_out CPU_kaydedicisi, cihaz_adresi, kontrolör_kaydedicisi

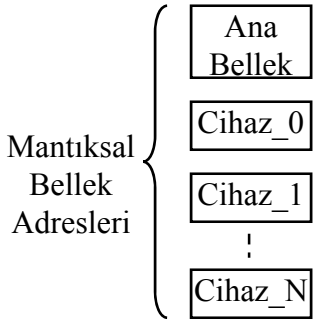
test CPU_kaydedicisi, cihaz_adresi

Memory-Mapped I/O

I/O cihazlarına, mantıksal bellek adresleriyle erişim metodunda, cihazların özel adresleri yerine mantıksal bellek adresleriyle erişilir. Erişim herhangi bir bellek bölgesine erişir gibi yapılır. Geleneksel yöntemle cihazın kaydedicilerini okumak ve yazmak bu metoda göre ek I/O komutları gerektirecektir. Bu metot işlemcide işlenen komut sayısını önemli ölçüde azaltacaktır.



Geleneksel yaklaşım



Memory-mapped I/O

Diyelim 0x10 adresine sahip cihazımız, mantıksal adres uzayında, 0xAA100 ile 0xAA10F adresleri arasında komut, durum ve 14 veri kaydedicisine sahip olsun. Bu durumda, kontrolörün 2 numaralı kaydedicisinin içeriğini R3 kaydedicisine aktarmak istiyorsak, ‘load R3, 0xAA102’ komutunu kullanmak yeterli olacaktır.

PROSES YÖNETİMİ

Proses Kavramı

Klasik ve Modern Prosesler

Sistem Çağrıları

Kullanıcı ve Supervisor Mod

Proses Kontrol Bloğu

Proses Durum Diyagramları

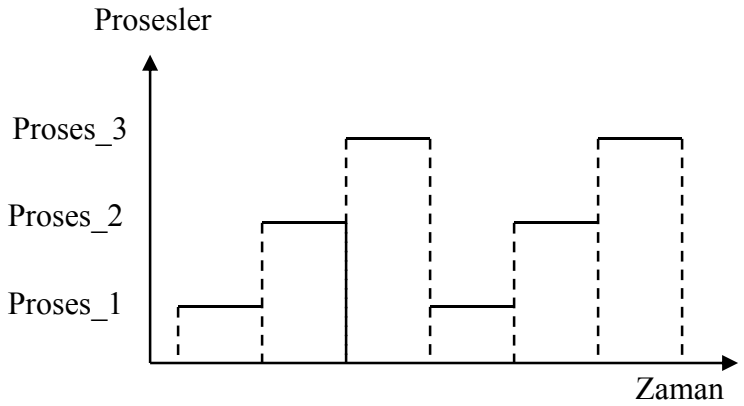
- İki durumlu proses modeli
- Üç durumlu proses modeli

Parent (anne) ve child (çocuk) proses kavramı

PROSES KAVRAMI

Proses, çalışır durumda olan programa verilen bir isimdir. CPU tarafından işletilmeyen bir program pasif bir yapıya sahip olup komutlar dizisinden ibarettir. Bu komutlar işletime alınmak suretiyle işlevlerini yerine getirebilirler.

Tüm modern bilgisayarlar aynı anda birkaç iş yapabilir; kullanıcı programı çalışıyorken diskten okuma, bir text dokümanı ekranda gösterme veya çıktı işlemlerini de gerçekleştirebilir. Çoklu programlamayı destekleyen sistemlerde CPU, bir programdan diğer bir programa belli zaman dilimlerinde anahtarlanır.



Klasik ve Modern Prosesler

Klasik proses; von Neumann mimarisi bilgisayarlarda işletilmekte olan programa ait bir tabirdir. Daha sonraları modern proses ve thread (iplik) kavramı ortaya çıkmıştır.

Programcılar programlarının değişik kısımlarını bir thread kümesi olarak tek bir klasik proses çatısı altında geliştirebilirler. Klasik prosesler, tek bir threade sahip modern prosesler olarak düşünülebilir. Klasik prosesler bir arada çalışabilir ancak aynı veri yapılarını paylaşmazlar.

Proses-Thread

Çok işlemcili sistemlerde, her bir işlemciye bir thread atamak suretiyle gerçek paralellik sağlanabilir. Tek işlemcili sistemlerde ise, bir prosese adanan sürede, prosesin sahip olduğu thread'ler zaman paylaşımlı olarak çalışırlar dolayısıyla gerçek paralellik sağlanamaz.

Günümüz işletim sistemlerinden olan FreeBSD Unix, modern prosesleri destekler şekilde tasarlanmamasına rağmen, Linux ve Solaris gibi işletim sistemleri klasik proses kavramı üzerine inşa edilmiş ve bazı eklentilerle modern proses ve thread kavramlarını destekler hale getirilmiştir. Mach ve Windows işletim sistemleri modern proses ve thread kavramı üzerine kurulmuştur. Unix ailesinin klasik prosesleri, Windows ailesinin de modern prosesleri desteklemek üzere tasarlandığı söylenebilir.

Sistem Çağrıları

Bilgisayar ilk açıldığı zaman işletim sistemi ana belleğe yüklenir ve çalışmaya başlar. İşletim sistemi bir programı çalıştırmak üzere seçtiğinde o programın bulunduğu ana bellek bölgesine dallanır ve programı işletir. Belirli bir zaman diliminden sonra işletim sistemi tekrar CPU'nun kontrolünü kazanır ve kendi kodunu çalıştırır. Ve bu durum sürekli olarak devam eder.

Proses ve threadlerin oluşturulması ve sonlandırılması, proseslere kaynak tahsisi, proseslerin giriş/çıkış işlemleri için aygıt yöneticisiyle ve belleğe yüklenmesi için de bellek yöneticisiyle işbirliği yapmak proses yöneticisinin sorumluluğundadır.

Örneğin, Unix'te fork(), Windows'ta ise CreateProcess() sistem çağrıları proses oluşturmak için kullanılır.

Benzer şekilde Linux'ta pthread_create() ve Windows'ta CreateThread() komutları, thread oluşturmak için kullanılır.

Kullanıcı ve Supervisor Mod

İşletim sistemi, kaynak paylaşımı (sharing) ve korumanın (protection) sağlanabilmesi için giriş/çıkış komutları gibi ayrıcalıklı komutları kendi bünyesinde işler.

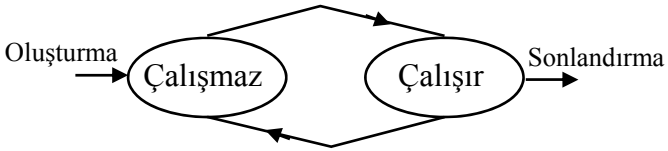
Bir kullanıcı programındaki tüm komutlar kullanıcı modu komutlarıdır ve donanım üzerinde direkt olarak icra edilir. Kullanıcı programı, örneğin proses oluşturmak için fork() komutunu kullandığında donanım tarafından direkt olarak değil CPU'nun supervisor modunda işletim sistemi fonksiyonu olarak işletilir.

Proses Kontrol Bloğu

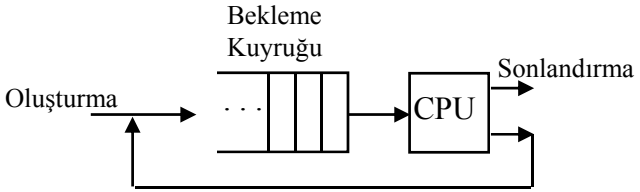
Bir prosesi sadece program kodlarından ibaret olduğunu düşünmek yeterli olmaz. Çoklu programlamada, değişik zaman dilimlerinde farklı prosesler çalıştığından proseslere ait bilgilerin tutulduğu proses kontrol blokları vardır. Bu bloklar, prosesin ana bellekteki yeri, proses aktivitesi, program sayacının değeri, işlemcinin kaydedicileri, tahsis edilen kaynaklar, açık durumdaki dosyalar, hafıza limitleri gibi bilgileri içerirler.

İki durumlu proses modeli

Herhangi bir zaman diliminde bir proses CPU tarafından ya işletiliyordur ya da işletilmiyordur. Yani proses ya çalışır ya da çalışmaz durumdadır.



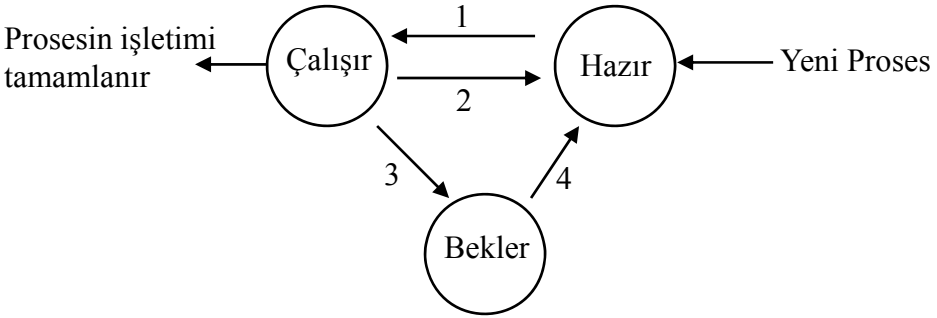
Durum geçişleri

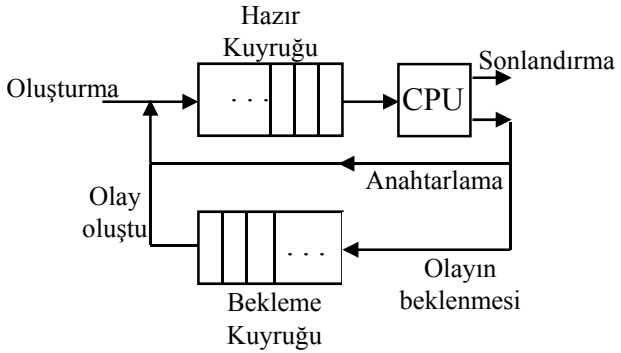


Kuyruk diyagramı

Üç durumlu proses modeli

Eğer tüm prosesler çalışmaya hazır durumundaysa, kuyruk yapısı iki durumlu modeldeki gibi düşünülebilir. Fakat giriş/çıkış işleminin sonlanmasını bekleyen bir proses olduğunda kuyruktan sıra ile prosesleri işletime sokmak mümkün olmayacaktır. İki durumlu modeldeki çalışmaz durumu, hazır ve bekler durumu olarak genişletilebilir.

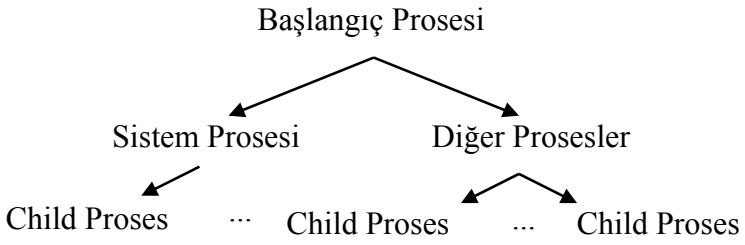




Bir olay vuku bulduğunda (I/O, semafor,…) işletim sistemi bekleme kuyruğunu tarayacak, bu olayı bekleyen prosesleri tespit edecek hazır kuyruğuna aktaracaktır. Kuyrukta yüzlerce proses olduğu düşünüldüğünde her bir olay için bir kuyruk tahsisi yapmak daha verimli bir kullanım olacaktır.

Parent (anne) ve child (çocuk) proses

Bir proses tarafından oluşturulan proses child proses, child prosesi üreten proses de parent proses olarak isimlendirilir. Proses hiyerarşisinde parent proses çok sayıda child prosese sahip olabilirken child proses sadece bir parent prosese aittir.



Genellikle işletim sistemi yüklendiğinde bir başlangıç prosesi otomatik olarak oluşturulur ve bu proses tüm proses hiyerarşisinin kökü olur. Bir proses oluşturulduğunda bu köke eklenir.

CPU Planlaması (Scheduling)

Bağlam Anahtarlama (Context Switching)

Planlama çeşitleri

- Kesmeyen (Non Preemtive) Planlama
- Kesen (Preemtive) Planlama

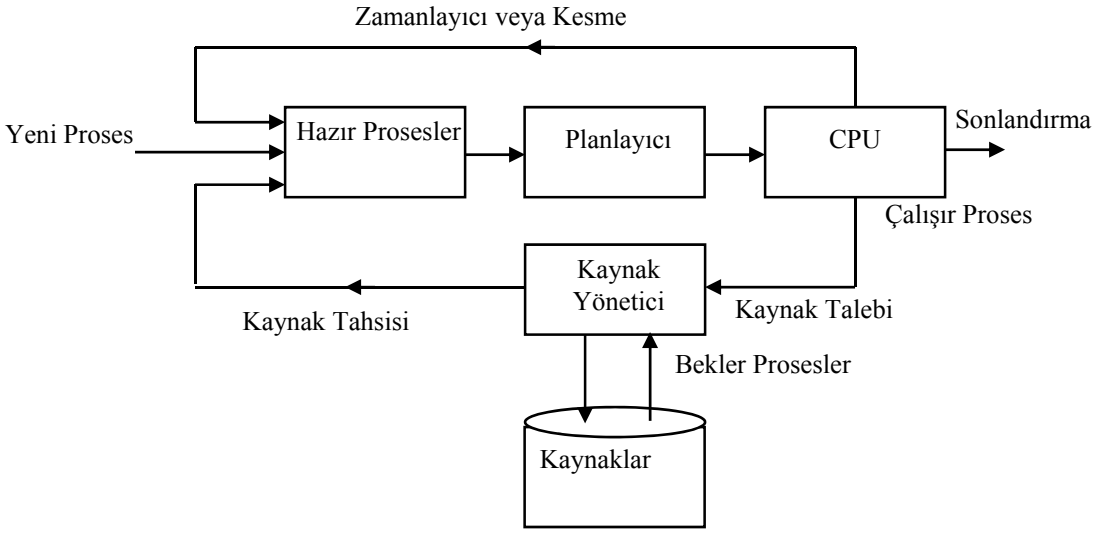
Planlayıcı tasarımında göz önünde bulundurulması gereken kavramlar

Kesmeyen Planlamada Algoritma Seçimi

- First Come First Served-FCFS
- Shortest Job First-SJF
- Priority Scheduling
- Deadline Scheduling

CPU Planlaması (Scheduling)

Bir bilgisayar sistemindeki paylaşılan kaynakların (CPU, Bellek, …) kullanımını planlamak önemli bir konudur. Çoklu programlamayı destekleyen işletim sistemlerinde birden fazla prosesin ana belleğe yüklenmesine ve CPU'yu zaman paylaşımlı olarak kullanmasına olanak tanınır.



Bağlam Anahtarlama (Context Switching)

CPU, bir prosesten diğerine anahtarlandığında data ve durum kaydedicilerinin kaydedilmesi gereklidir.

Örneğin 1 birimlik bilginin yüklenmesi için 20 nanosaniyeye (ns) ihtiyaç duyulduğu ve 50 tane de kaydedici olduğu düşünülürse, bu kaydedicilerin kaydedilmesi 1 milisaniye (ms) alacaktır. Ayrıca yeni yüklenecek prosesin kaydedicileri de 1 ms'de CPU'nun kaydedicilerine aktarılacağından bu süreç 2 ms sürecektir. Bu süre zarfında işe dönük birçok komut işletilebilecekken, anahtarlamaya geniş bir zaman ayrılması gerekmektedir.

Planlama Çeşitleri

1. Kesmeyen (Non Preemtive) Planlama

CPU'nun, prosesin kendi isteği dışında başka bir prosese atanmadığı işletim şeklidir. Şayet çalışan proses hiçbir kaynak talebi yapmadan sonsuz döngüye girerse, CPU'yu hiçbir zaman bırakmayacaktır. Ancak çalışan proseslerin kesmelere açık olması şartıyla bu durumun üstesinden gelinir.

2. Kesen (Preemtive) Planlama

Bir proses, kendi isteğinin dışında CPU'yu bırakmak durumunda kalabiliyorsa bu işletim türü kesendir.

Performans Değerlendirmesi İçin Strateji Seçimi

CPU'yu kullanmak için birbiriyle yarışan proseslerin CPU'ya tahsisi, her bir prosese eşit zaman dilimleri verilerek ya da proseslere öncelik düzeyi atanarak yapılabilir.

Gerçek zamanlı (Real-time) sistemlerde bir işin belli bir zaman içerisinde gerçekleştirilmesi önemliyken,

zaman paylaşımlı (Time-sharing) sistemlerde CPU'nun proseslere veya kullanıcılara eşit olarak paylaştırılması,

interaktif sistemlerde (Interactive) kullanıcının verdiği komutlara alınan yanıt süresi,

toplu işlemlerde (Batch) ise bir işin tamamlanma süresi (çünkü kullanıcı ancak bu süre sonunda yanıt alır) ön plandadır.

Planlayıcı Tasarımında Göz Önünde Bulundurulması Gereken Kavramlar

- CPU verimliliği (CPU efficiency)
- Servis süresi (Service time)
- Yanıt süresi (Response time)
- Bekleme süresi (Waiting time)
- İşin tamamlanma süresi (Turn-around time)

Kesmeyen Planlamada Algoritma Seçimi

1. İlk gelene önce servis verilen algoritma (First Come First Served -FCFS)

Proseslerin hazır kuyruğuna geliş sırasına göre işletildiği yani proseslere aynı öncelik düzeyinin atandığı bir algoritmadır.

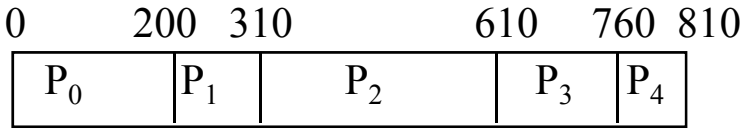
Örnek: t=0'da hazır kuyruğuna P_0, P_1, P_2, P_3 ve P_4 sırasında gelen 5 proses olsun ve bu proseslerin servis süreleri T ile gösterilsin. Her bir prosesin bekleme ve tamamlanma sürelerini FCFS algoritmasına göre hesap edelim.

i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50

Örnek (Devamı)

i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50

Her bir prosesin bitiş süresi;



Grafik gösterimi

$T_{toplam}(P_0)=T(P_0)=200$

$T_{toplam}(P_1)= T_{toplam}(P_0)+T(P_1)=200+110=310$

$T_{toplam}(P_2)= T_{toplam}(P_1)+T(P_2)=310+300=610$

$T_{toplam}(P_3)= T_{toplam}(P_2)+T(P_3)=610+150=760$

$T_{toplam}(P_4)= T_{toplam}(P_3)+T(P_4)=760+50=810$

Ortalama bitiş süresi = $\sum_{i=0}^{4}(T_{toplam}(P_i))/5 = (200+310+610+760+810)/5=538$

Her bir prosesin bekleme süresi;

$T_{bekleme}(P_0)=0$

$T_{bekleme}(P_1)= T_{toplam}(P_0)=200$

$T_{bekleme}(P_2)= T_{toplam}(P_1)=310$

$T_{bekleme}(P_3)= T_{toplam}(P_2)=610$

$T_{bekleme}(P_4)= T_{toplam}(P_3)=760$

Ortalama bekleme süresi = $\sum_{i=0}^{4}(T_{bekleme}(P_i))/5 = (0+200+310+610+760)/5=376$

Kesmeyen Planlamada Algoritma Seçimi

2. İşletim süresi en kısa olan işe önce servis verilen algoritma (Shortest Job First-SJF)

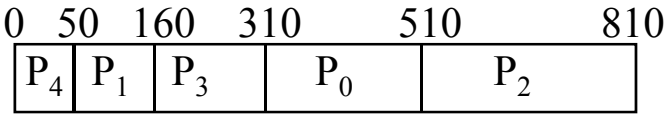
Minimum servis süresi gerektiren proses en yüksek önceliğe sahiptir. Servis süresi daha az olan prosese öncelik verildiğinden ortalama bekleme süresini minimize eden bir algoritmadır. Ancak, kısa servis süresi olan prosesler önce servis alacağından uzun servis süresi olan proseslerin uzun süre hazır kuyruğunda beklemeleri olasıdır. FCFS algoritmasının aksine proseslerin hazır kuyruğuna geliş sırasının bir önemi yoktur.

Örnek: FCFS algoritmasındaki örnek göz önünde bulundurulduğunda, servis süresi en az olan P_4, daha sonra P_1, P_3, P_0 ve P_2 işletime alınacaktır.

i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50

Örnek (Devamı)

i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50



Her bir prosesin bitiş süresi;

$T_{toplam}(P_0)=T(P_0)+ T(P_4)+ T(P_1)+ T(P_3)=200+50+110+150=510$
$T_{toplam}(P_1)= T(P_1)+ T(P_4)=110+50=160$
$T_{toplam}(P_2)= T(P_2)+ T(P_4)+ T(P_1)+ T(P_3)+ T(P_0)=300+50+110+150+200=810$
$T_{toplam}(P_3)= T(P_3)+ T(P_4)+ T(P_1)=150+50+110=310$
$T_{toplam}(P_4)= T(P_4)=50$

Ortalama bitiş süresi = $\sum_{i=0}^{4}(T_{toplam}(P_i))/5 = (510+160+810+310+50)/5=368$

Herbir prosesin bekleme süresi;

$T_{bekleme}(P_0)= 310$
$T_{bekleme}(P_1)= 50$
$T_{bekleme}(P_2)= 510$
$T_{bekleme}(P_3)= 160$
$T_{bekleme}(P_4)= 0$

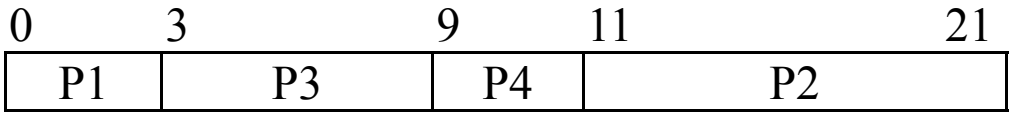
Ortalama bekleme süresi = $\sum_{i=0}^{4}(T_{bekleme}(P_i))/5 = (310+50+510+160+0)/5=206$

Örnek:

4 prosesin hazır kuyruğuna geliş zamanları ve servis süreleri aşağıda verilmiştir.

Proses	Geliş Zamanı	Servis Süresi
P1	0	3
P2	1	10
P3	2	6
P4	4	2

Kesmeyen yapıdaki SJF (Shortest Job First) algoritmasına göre proseslerin işletiminin grafiksel gösterimi,



Ortalama Bekleme Zamanı = $[T_{bekleme}(P1)+ T_{bekleme}(P3)+ T_{bekleme}(P4)+ T_{bekleme}(P2)]/4$

Ortalama Bekleme Zamanı = (0-0 + 3-2 + 9-4 + 11-1) / 4 = (1+5+10)/4 = 16/4 = 4

Bitiş süreleri hesaplanırken de, grafiğe bakılır ve prosesin bittiği süreden geliş zamanı çıkarılır. Daha sonra da ortalamaları alınır.

Kesmeyen Planlamada Algoritma Seçimi

3. Öncelik tabanlı planlama algoritması (Priority Scheduling)

Bu algoritmada proseslere kullanıcı tarafından (veya işletim sistemi tarafından) belirlenen öncelik düzeyi atanır. Genellikle bu planlama algoritmasında öncelik düzeyi sabit bir değerdir ve sayıca düşük öncelik değerine sahip proses yüksek önceliklidir.

Örnek: 5 prosese ait servis süreleri ve öncelik düzeyleri aşağıdaki gibi olsun. Her bir prosesin bekleme ve tamamlanma sürelerini hesap edelim (Tüm proseslerin t=0'da hazır kuyruğuna geldiği farz edilecektir).

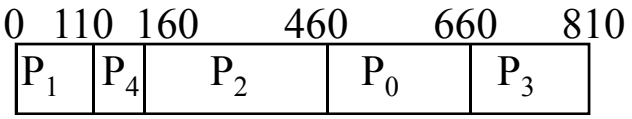
i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50
Önceliği	4	1	3	5	2

P_1 prosesi en yüksek öncelikli olduğundan ilk önce işletilecektir. Daha sonra da sırasıyla P_4, P_2, P_0 ve P_3 işletilecektir.

Örnek (Devamı)

i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50
Önceliği	4	1	3	5	2

Her bir prosesin bitiş süresi;



$Ttoplam(P_0)=T(P_0)+ T(P_1)+ T(P_4)+ T(P_2)=200+110+50+300=660$
$Ttoplam(P_1)=T(P_1)=110$
$Ttoplam(P_2)=T(P_2)+ T(P_1)+ T(P_4)=300+110+50=460$
$Ttoplam(P_3)=T(P_3)+ T(P_1)+ T(P_4)+ T(P_2)+ T(P_0)=150+110+50+300+200=810$
$Ttoplam(P_4)=T(P_4)+ T(P_1)=50+110=160$

Ortalama bitiş süresi $= \sum_{i=0}^{4}(T_{toplam}(P_i))/5 = (660+110+460+810+160)/5= 440$

Herbir prosesin bekleme süresi;

$T_{bekleme}(P_0)= 460$
$T_{bekleme}(P_1)= 0$
$T_{bekleme}(P_2)= 160$
$T_{bekleme}(P_3)= 660$
$T_{bekleme}(P_4)= 110$

Ortalama bekleme süresi $= \sum_{i=0}^{4}(T_{bekleme}(P_i))/5 = (460+0+160+660+110\ )/5=278$

Kesmeyen Planlamada Algoritma Seçimi

4. Proseslerin işletimi için belli bir mühlet tanınan planlama (Deadline Scheduling)

Özellikle gerçek zamanlı sistemlerde bazı proseslerin belli bir zaman içerisinde tamamlanması gereklidir. İşlerin bitiş ve bekleme sürelerinden çok, maksimum servis süreleri içerisinde bitirilmesi esastır.

Örnek: 5 prosese ait servis süreleri ve bu proseslerin işletilmesi için tanınan maksimum süre aşağıda verilen tablodaki gibi olsun.

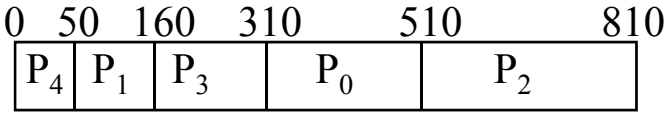
i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50
Max. Süre	650	450	-	350	100

Örnek (Devamı)

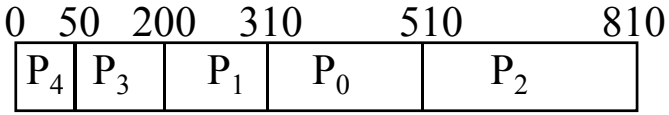
Bu planlamada önemli olan maksimum servis süresinde proseslerin işletimini tamamlamak olduğundan değişik planlamalar yapılabilir.

i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50
Max. Süre	650	450	-	350	100

Prosesler P_4, P_1, P_3, P_0 ve P_2 sırasıyla işletilebilir:



Ayrıca, maksimum servis süresi en az olanın önce işletilmesi yaklaşımı da vardır. Ama yine proseslerin maksimum servis süreleri içerisinde işletimi esastır. Bu durumda da prosesler P_4, P_3, P_1, P_0 ve P_2 sırasıyla işletilebilir:



CPU Planlaması (Scheduling)

Kesen Planlamada Algoritma Seçimi

SJF Algoritması

Öncelik Tabanlı Algoritma

Round Robin Algoritması

Çok Kuyruklu Planlama (Multiple Queues)

- Çok kuyruklu geri beslemeli planlayıcı

Fair-Share Planlaması

Kesen Planlamada Algoritma Seçimi

Kesen planlamada CPU yüksek öncelikli proseslere tahsis edilir. Yüksek öncelikli proses, CPU talebinde bulunduğunda tüm diğer düşük öncelikli prosesler CPU'yu bırakmak durumunda kalırlar.

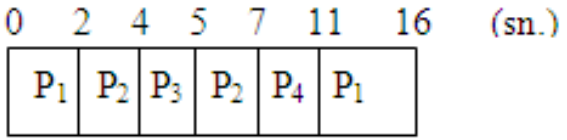
Kesmeyen planlamadaki SJF ve öncelik tabanlı algoritmalar, kesen planlamada da aynı adlarla anılırlar. Aralarındaki fark; kesen planlamada her zaman, yüksek öncelikli proses çalışır durumdadır.

Kesen planlamada CPU'nun prosesler arasındaki anahtarlanması, kesmeyen planlamaya göre daha sık olur.

Kesen Yapıda SJF Algoritması

Örnek: Dört prosesin hazır kuyruğuna geliş zamanları ve servis süreleri aşağıdaki gibidir. Kesen (preemtive) yapıdaki SJF (Shortest Job First) planlama algoritmasına göre her bir prosesin ortalama bekleme süresini hesap edelim.

Proses	Geliş Zamanı (sn.)	Servis Süresi (sn.)
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



Grafiksel gösterim

Bekleme süreleri;

$T_b(P_1)=11-2=9$ sn.
$T_b(P_2)=5-4=1$ sn.
$T_b(P_3)=0$ sn.
$T_b(P_4)=7-5=2$ sn.

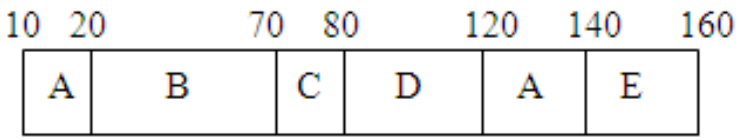
$T_{ort}=(9+1+0+2)/4=3$ sn. olarak bulunur.

Kesen Yapıda Öncelik Tabanlı Algoritma

Örnek: 5 prosesin hazır kuyruğuna geliş zamanları ve öncelik düzeyleri aşağıdaki tabloda verilmiştir. Buna göre, her bir prosesin ne zaman işletime alındığını ve ne zaman işletiminin tamamlandığını, kesen yapıdaki öncelik tabanlı (Priority) planlama algoritmasına göre bulalım.

(Not: Önceliği, sayıca büyük olan proses, daha öncelikli ele alınacaktır.)

Proses	Geliş Zamanı	Servis Süresi	Önceliği
A	10	30	1
B	20	50	5
C	30	10	3
D	40	40	2
E	50	20	0



Grafiksel gösterim

Grafiksel gösterimden;

Proses	İşletime alındığı zaman	İşletiminin tamamlandığı zaman
A	10	140
B	20	70
C	70	80
D	80	120
E	140	160

Round Robin Algoritması

En sık kullanılan planlama algoritmalarından biridir. CPU talebinde bulunan tüm proseslere eşit işlem zamanı tanınması prensibi vardır. Eşit işlem zamanı programlanabilen bir iç zamanlayıcıyla sağlanır.

Bu algoritmada hazır kuyruğundaki prosesler CPU'ya sırayla anahtarlanır. İşlem süresi biten proses, hazır kuyruğunun sonuna aktarılır ve kuyruğun başındaki proses CPU'ya anahtarlanır.

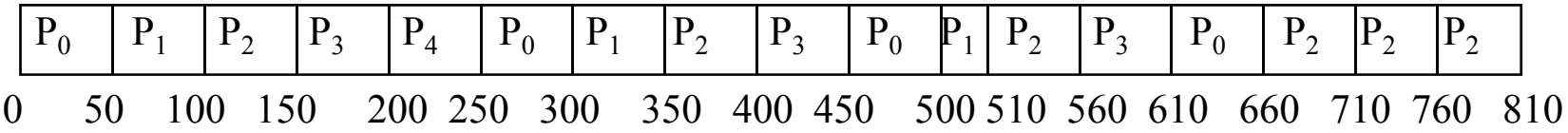
Örnek: t=0'da hazır kuyruğuna P_0,P_1,P_2,P_3,P_4,P_5 sırasıyla gelen 5 prosesin servis süreleri aşağıdaki gibi olsun. İç zamanlayıcı kesme süresinin 50 birim olduğu varsayılırsa her bir prosesin ortalama bitiş ve yanıt sürelerini hesap edelim.

i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50

Örnek (Devamı):

i	0	1	2	3	4
$T(P_i)$	200	110	300	150	50

İşleyişin grafiksel gösterimi:



Bitiş süreleri:

$T_{toplam}(P_0)=660$
$T_{toplam}(P_1)=510$
$T_{toplam}(P_2)=810$
$T_{toplam}(P_3)=610$
$T_{toplam}(P_4)=250$

} Ortalama bitiş süresi =(660+510+810+610+250)/5=568

Yanıt süreleri;

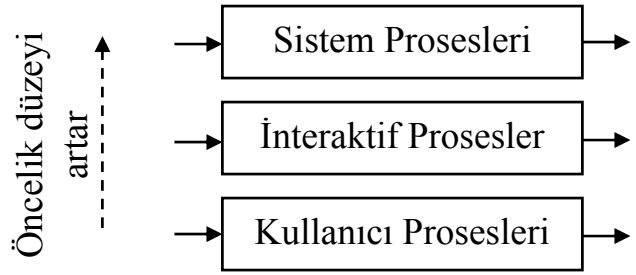
$T_{yanıt}(P_0)=0$
$T_{yanıt}(P_1)=50$
$T_{yanıt}(P_2)=100$
$T_{yanıt}(P_3)=150$
$T_{yanıt}(P_4)=200$

} Ortalama yanıt süresi =(0+50+100+150+200)/5=100

Çok Kuyruklu Planlama (Multiple Queues)

Diğer planlama algoritmalarında, tüm proseslerin tek bir hazır kuyruğunda bulunduğu varsayılarak bu prosesler arasında seçim yapılmaktaydı. Oysa ele alınan görevler çok farklı nitelikte olabilir.

Bu planlama iki aşamalı olarak düşünülür. Planlayıcı, ilk olarak hangi öncelik düzeyinden seçim yapacağına daha sonra da seçimini yaptığı öncelik düzeyinden hangi prosesin seçeceğine karar verir. Her bir aşamada farklı algoritmalar kullanılabilir.



Üç seviyeli hazır kuyruğu

Çok Kuyruklu Geri Beslemeli Planlayıcı

Çok kuyruklu planlamada prosesler belli bir öncelik düzeyindeki kuyruğa atılır ve yeri değişmez. Çok kuyruklu geri beslemeli planlayıcı yapısında ise prosesler ilk olarak zaman dilimi az olan en yüksek öncelikli kuyruğa konur. Bu zaman diliminde proses tamamlanmazsa zaman dilimi biraz daha fazla olan bir alt kuyruğa aktarılır. Şayet üç seviyeli bir kuyruk yapısı varsa ve alt kuyrukta da prosesin işletimi tamamlanmamışsa en alt seviyedeki kuyruğa konur.

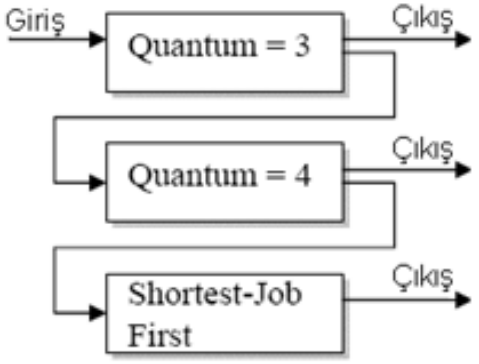
Bu planlayıcı kesen yapıya sahiptir. Üst seviyedeki kuyruktaki işler öncelikle işletilir. Yani alt seviyedeki bir iş işletilirken üst seviyedeki kuyruğa bir iş geldiğinde işin işletimi yarıda kesilir ve yeni gelen iş işletime alınır.

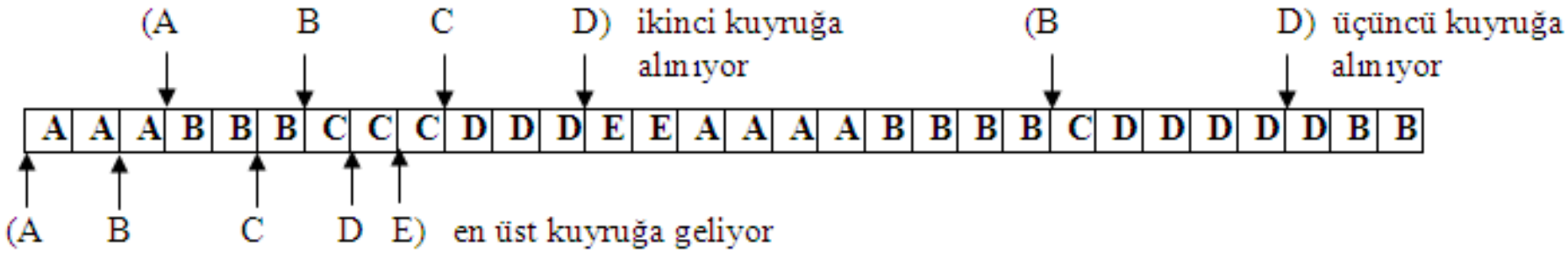
Çok Kuyruklu Geri Beslemeli Planlayıcı

Örnek: Çok kuyruklu geri beslemeli planlama algoritmasının kullanıldığı bir sistemde, proseslerin kuyruğa geliş zamanları ve servis süreleri aşağıdaki gibidir. Proseslerin ne şekilde işletileceğini grafiksel olarak gösterelim.

(Not: Üst iki sevide, prosesler kuyruğa geldikleri sırada işletime alınmaktadır)

Proses ID	Geliş Zamanı	Servis Süresi
A	0	7
B	2	9
C	5	4
D	7	8
E	8	2





Fair-Share Planlaması

Daha önce bahsedilen planlama algoritmalarında, proseslerin CPU'ya tahsisi, bilgisayar sistemini kullanan kullanıcıların sayısından bağımsız olarak yapıldı. Örneğin birinci kullanıcının 7 prosesi, ikinci kullanıcının ise 3 prosesi olduğunu ve sistemimizin Round Robin planlama algoritmasını kullandığını farz edersek, CPU'nun %70'ini birinci kullanıcının, %30'unu ise ikinci kullanıcının proseslerinin kullandığını söyleyebiliriz.

Örneğin bir sistem, iki kullanıcıya sahip olsun. Birinci kullanıcının 2 prosesi (P1_1,P1_2), ikinci kullanıcının da 2 prosesi (P2_1, P2_2) olduğunu varsayarsak ve kullanıcılara eşit olarak CPU'yu paylaştırmak istiyorsak, Round Robin algoritması şu şekilde çalıştırılabilir;

P1_1 P2_1 **P1_2** P2_2 **P1_1** P2_1 **P1_2** P2_1…

Şayet birinci kullanıcıya iki kat daha fazla süre vermek istersek, Round Robin algoritması şu şekilde çalıştırılabilir;

P1_1 P1_2 P2_1 **P1_1 P1_2** P2_2…

Prosesler Arası Senkronizasyon

Paylaşılır ve Paylaşılmaz Kaynak

Kritik Kaynak ve Kritik Kesim

Karşılıklı Dışlama

Kritik Bölge Problemlerinin Çözümü

- Kesmeler aktif ve pasif yapılarak
- Ortak değişken kullanımı
- Özel makine komutlarının kullanımı

Prosesler Arası Senkronizasyon

Çoklu programlama (multiprogramming), bir proses bitmeden diğer bir prosesin işletilmesine, dolayısıyla programcılara bir problemin çözümünde bir grup ilişkili prosesi paralel olarak oluşturabilmelerine olanak sağlamıştır. Örneğin bazı prosesler giriş/çıkış işlemi yapıyorken diğer bir proses CPU'yu kullanıyor olabilir. Paralel çalışan prosesler birbirleriyle etkileşebilirler. Ancak, iki ya da daha çok bir arada çalışan prosesin paylaşılan kaynakları ortaklaşa kullanmalarından dolayı senkronizasyon problemleri ortaya çıkar.

Paylaşılır ve Paylaşılmaz Kaynak

Bir kaynağın paylaşılabilir olması, bir prosesin işletimi tamamlanmadan diğer proseslerin de bu kaynağı bir aksaklık oluşturmayacak şekilde kullanabilmesi demektir. Bu bağlamda CPU, birincil ve ikincil bellekler paylaşılan kaynaklar grubundandır. Fakat yazıcı bu gruptan değildir (Bir iş tamamlanmadan diğer bir işin başlaması mümkün değildir).

Şayet değişken, dosya ya da tampon bölgelerin paylaşımı söz konusu olduğunda, bunlar üzerinde yapılan işlemin türüne (okuma, yazma) göre paylaşılır ya da paylaşılmaz olurlar.

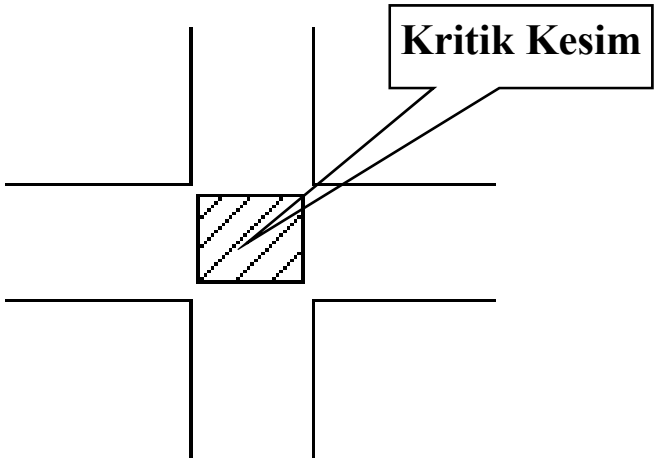
Kritik Kaynak ve Kritik Kesim

Paylaşılmayan kaynaklar, erişimin denetim altında gerçekleştirilmesi gereken **kritik kaynaklardır.** Yani bu kaynaklara erişmeden önce kullanılmadığından emin olmak gerekir. Bir proses kritik bir kaynak üzerinde işlem yapıyorsa ve diğer bir proses de bu kaynağa erişmek istiyorsa, kaynağın kullanılabilirlik durumunun denetlenmesi ve gerekirse bu isteğin ertelenmesi prosesler arası senkronizasyon kapsamında değerlendirilir.

Bir programda, kritik kaynağın kullanıldığı program kesimleri **kritik kesim** (critical section) olarak anılır.

Karşılıklı Dışlama

Kritik bir kaynağı kullanan proses, kritik kesimine girdiğinde yine bu kaynakla ilişkili başka bir prosesin kritik kesimine girmesinin engellenmesi karşılıklı dışlama (mutual exclusion) olarak bilinir. Örneğin iki yolun kesişiminden meydana gelen bir kavşakta iki aracın geçişi söz konusu olduğunda kesişim bölgesi kritik kesimdir.



Örnek:

Bu örneğimizde, her iki prosesin de bir ortak değişken (toplam değişkeni) üzerinde yazma işlemi yapacağı ve bu proseslerin aynı anda çalıştırıldığı düşünülerek, bu ortak değişkene kontrollü erişim yapılmadığı taktirde neler olacağı incelenecektir.

'toplam' paylaşılan değişken, 'degisim' yerel değişken ve iki prosese ait program parçacıkları aşağıdaki gibi olsun:

Proses 0:

```
. . .
toplam=toplam + degisim;
. . .
```

Proses 1:

```
. . .
toplam=toplam - degisim;
. . .
```

Örnek: (Devamı)

Programın assembly kodu karşılıkları aşağıdaki gibidir:

Proses 0:

```
load R1, toplam
load R2, degisim
add R1, R2
store R1, toplam
```

Proses 1:

```
load R1, toplam
load R2, degisim
sub R1, R2
store R1, toplam
```

Aslında Proses 0 ya da Proses 1'in programına bakılarak kritik kesim problemini tespit etmek mümkün değildir. Problem, programdan değil, paylaşımdan kaynaklanmaktadır. Bu kritik kesim probleminden kaçınmak için sadece bir prosesin kritik kesimine girmesine izin verilmelidir.

Kritik Bölge Problemlerinin Çözümü

1. Kesmeler Aktif ve Pasif Yapılarak: Bir proses kritik kesimine girince sistem kesilmelere karşı kapatılarak diğer proseslerin ilgili kritik kesimine girmesi engellenebilir ve kritik kesim tamamlanınca da tekrar kesilmelere açılabilir.

Proses 0:

```
. . .
kesmeler_pasif;
toplam=toplam + degisim;
kesmeler_aktif;
. . .
```

Proses 1:

```
. . .
kesmeler_pasif;
toplam=toplam - degisim;
kesmeler_aktif;
. . .
```

İşletim sistemi, kullanıcı mod programlara kesmeleri aktif ya da pasif yapma yetkisi vermez.

Kritik Bölge Problemlerinin Çözümü

2. Ortak Değişken Kullanımı: Kesmelerin aktif ya da pasif yapılmasını gerektirmeyen bir çözümdür. Başka bir paylaşılır değişken kullanılarak prosesler arasında koordinasyon sağlanır. Bu değişkenin ismi 'lock' olsun. Prosesler kritik kesimlerine girerken bu değişkeni 'True', kritik kesimlerinden çıkarken de 'False' yapmak suretiyle birlikte çalışırlar.

```
Shared boolean lock=False;
Shared double toplam, degisim;
```

Proses 0:

```
. . .
While(lock);
lock=True;
//kritik kesime giriş noktası
toplam=toplam + degisim;
lock=False;
//kritik kesimden çıkış
. . .
```

Proses 1:

```
. . .
While(lock);
lock=True;
//kritik kesime giriş noktası
toplam=toplam - degisim;
lock=False;
//kritik kesimden çıkış
. . .
```

Kritik Bölge Problemlerinin Çözümü

Test edip set etme işlemi çok kısa bir zaman alır. Normal kritik kesim probleminin işletim süresi uygulama yazılımına bağlı iken, 'lock' kritik kesim probleminin çözümü için kesmelerin pasif edilmesi düşünülebilir. Kesme komutları ayrıcalıklı komutlar olduğundan kritik kesime girerken enter() ve çıkarken de exit() işletim sistemi çağrıları kullanılabilir.

Proses 0:

```
. . .
Enter(lock);
toplam=toplam + degisim;
Exit(lock);
. . .
```

Proses 1:

```
. . .
Enter(lock);
toplam=toplam - degisim;
Exit(lock);
. . .
```

Sistem Çağrıları

```
Enter(lock) {
Kesmeler_pasif;
While(lock) {
            Kesmeler_aktif;
            Kesmeler_pasif;
            }
lock=True;
kesmeler_aktif;
}
```

```
Exit(lock) {
Kesmeler_pasif;
lock=False;
kesmeler_aktif;
}
```

Kritik Bölge Problemlerinin Çözümü

3. Özel Makine Komutlarının Kullanımı: Karşılıklı dışlama problemlerinin çoğu, bir değişkenin değeri sınandıktan sonra değerinin değiştirilmesi esnasında kesmeler sebebiyle işleminin yarıda kalmasından kaynaklanmaktadır. Bu iki işlem, tek makine saykılında icra edilebilir. Örneğin test edip set etmeye yarayan makine komutunun yapısı;

```
Boolean test_set (int i)
{
        if (i == 0) {i=1; return true;}
                else {return false;}
}
```

Ya da bir değişken ve kaydedicinin içeriklerini değiş tokuş eden ‘exchange’ komutunun yapısı;

```
void exchange (int kaydedici, int degisken)
{       int temp;
        temp=degisken;
        degisken=kaydedici;
        kaydedici=temp;
}
```

Örnek: test_set komutunun programda kullanılması

```
Boolean test_set (int i)
        {
                if (i == 0) {i=1; return true;}
                        else {return false;}
        }
```

```
int i=0; // tüm proseslerin paylaştığı değişken
…
while (!test_set(i));
//kritik kesim
i=0;
//programın kritik olmayan diğer kısımları
…
```

Kritik kesim içeren her bir prosesin program parçacığı

Örnek: exchange komutunun programda kullanılması

```
void exchange (int kaydedici, int degisken)
        {       int temp;
                temp=degisken;
                degisken=kaydedici;
                kaydedici=temp;
        }
```

```
i=0; // tüm proseslerin paylaştığı değişken
…
kaydedici=1;
While (kaydedici!=0)
    exchange(kaydedici, i);
//kritik kesim
exchange(kaydedici, i);
//programın kritik olmayan diğer kısımları
…
```

Kritik kesim içeren her bir prosesin program parçacığı

PROSESLER ARASI SENKRONİZASYON (Devamı)

Kritik Kesim Problemlerinin Çözümü

- **Diğer Yazılımsal Yaklaşımlar**
  - Dekker Algoritması ve Gelişim Süreci
  - Peterson Algoritması

- **Semaforlar**
- **Monitörler**

Kritik Bölge Problemlerinin Çözümü

Dekker Algoritması ve Gelişim Süreci:

İki proses için geliştirilmiş bir metottur. Yani ikiden fazla sayıda proses varsa uygulanabilirliği yoktur. Bu algoritmanın gelişim sürecinin incelenmesi, eş zamanlı çalışan proseslerde karşılaşılan problemleri anlayabilmek için faydalı olacaktır.

İlk Aşama:

'turn', paylaşılan global bir değişken olsun. Kritik kesimine girmek isteyen proses ilk olarak bu değişkeni sınayacak, şayet 'turn' değişkeninin değeri prosesin numarasına eşitse kritik bölgesine girecek aksi taktirde bekleyecektir. Kritik kesim tamamlanınca da 'turn' değişkeninin değeri diğer prosesin işletimine izin verecek şekilde ayarlanacaktır.

```
Shared int turn=0;
```

Proses 0:	Proses 1:

. . .
While (turn!=0);
//kritik kesim;
turn=1;
. . .
	. . .
	While (turn!=1);
	//kritik kesim;
	turn=0;
	. . .

Bu yaklaşım karşılıklı dışlamayı sağlar ancak bazı aksaklıkları vardır: Prosesler sıra ile çalışmak zorundadır.

İkinci Aşama:

İlk aşamada proseslerin sıra ile çalışma zorunluluğu vardı. Her iki prosese de birer değişken tahsis etmek suretiyle bu durum aşılabilir. Bu değişken iki elemana sahip bir dizi olsun.

```
Shared boolean flag[2]={False,False};
```

Proses 0:	Proses 1:

. . .
While(flag[1]);
flag[0]=True;
//kritik kesim;
flag[0]=False;
. . .
	. . .
	While(flag[0]);
	flag[1]=True;
	//kritik kesim;
	flag[1]=False;
	. . .

Bu yaklaşım karşılıklı dışlama şartını sağlamamaktadır. Şöyle ki; Proses 0 'while' döngüsünde 'flag[1]' değişkenini 'False' olarak, daha sonra da Proses 1 'flag[0]' değişkenini 'False' olarak tespit ederse, her iki proses de kritik kesimine girecektir.

Üçüncü Aşama:

Bir proses diğer prosesin durumunu kontrol ettikten sonra (kendi durumunu değiştirmeden hemen önce) bir kesilme meydana gelirse her iki prosesin de kritik kesimine girme durumu oluşabiliyordu. 'flag' değişkenin değerinin 'while' döngüsünden önce set edilmesi durumunda program parçacıklarının işleyişini inceleyelim.

```
Shared boolean flag[2]={False,False};
```

Proses 0:

```
. . .
flag[0]=True;
While (flag[1]);
//kritik kesim;
flag[0]=False;
. . .
```

Proses 1:

```
. . .
flag[1]=True;
While (flag[0]);
//kritik kesim;
flag[1]=False;
. . .
```

Karşılıklı dışlama şartı sağlanmıştır. Ancak, her iki prosesin de 'flag' değişkenlerinin değerini 'True' yaptığını ve birinin 'while' döngüsüne girdiğini düşündüğümüzde kilitlenme meydana gelecektir. Bunun sebebi; her iki prosesin de birbirlerinin 'flag' değişkenlerinin 'False' olmasını beklemesidir.

Dördüncü Aşama:

Üçüncü yaklaşımda bir proses diğer prosesin durumunu bilmeksizin durumunu set edebiliyordu.

```
Shared boolean flag[2]={False,False};
```

Proses 0:	Proses 1:

. . .
flag[0]=True;
While (flag[1])
{ flag[0]=False;
//delay;
flag[0]=True;
}
//kritik kesim;
flag[0]=False;
. . .
	. . .
	flag[1]=True;
	While (flag[0])
	{ flag[1]=False;
	//delay;
	flag[1]=True;
	}
	//kritik kesim;
	flag[1]=False;
	. . .

Bu yaklaşım, karşılıklı dışlama şartı sağlamaktadır. Ancak, livelock oluşabilir.

Doğru Çözüm (Dekker'in Algoritması):

Dördüncü aşamada 'while' döngüsü içerisinde diğer prosesin kritik kesimine girmesine izin verilmekte fakat bu aktivitenin düzene koyulması gerekmektedir.

```
Shared boolean flag[2]={False,False};
Shared int turn=1;
```

Proses 0:

```
. . .
flag[0]=True;
While (flag[1])
     if (turn = = 1)
       { flag[0]=False;
              While (turn = = 1);
          flag[0]=True;
       }
//kritik kesim;
flag[0]=False;
turn=1
. . .
```

Proses 1:

```
. . .
flag[1]=True;
While (flag[0])
     if (turn = = 0)
       { flag[1]=False;
              While (turn = = 0);
          flag[1]=True;
       }
//kritik kesim;
flag[1]=False;
turn=0;
. . .
```

Peterson Algoritması

Dekker'in algoritması karşılıklı dışlama problemini çözmüştür. Ancak program oldukça karmaşıktır. Peterson'un çözümünün anlaşılması daha kolaydır ve ikiden fazla proses için genelleştirilebilir.

```
Shared boolean flag[2]={False,False};
Shared int turn;
```

Proses 0:

```
. . .
flag[0]=True;
turn=1;
While(flag[1] && turn = = 1);
//kritik kesim;
flag[0]=False;
. . .
```

Proses 1:

```
. . .
flag[1]=True;
turn=0;
While(flag[0] && turn = = 0);
//kritik kesim;
flag[1]=False;
. . .
```

Semaforlar

Özellikle karşılıklı dışlama kapsamında önceki kısımlarda ele alınan yazılımsal yaklaşımlarda prosesler, kaynağın serbest olup olmadığını sürekli olarak sınamak suretiyle merkezi işlem birimini gereksiz yere meşgul ediyorlardı. 1965 yılında Edsger Dijkstra'nın prosesler arasındaki senkronizasyonu sağlamak için geliştirdiği semafor yapısıyla bu problem ortadan kalkmıştır.

Dijkstra çalışmasında, Almanca 'Proberen - test et' manasına gelen P operatörünü ve 'Verhogen - arttır' manasına gelen V operatörünü pozitif tamsayı tipinde değişken olan 's' semoforu üzerinde uygulamıştır.

V(s) : < s = s+1 >

P(s) : < while (s < = 0) {bekle}; s = s-1 >

Örnek:

Kritik kesimin anlatıldığı kısımda ele alınan ‘toplam’ değişkenlerinin güncellenmesine dair örneği göz önünde bulunduralım. Bu örnekte kullanılan semafor ya 0 ya da 1 değerini almaktadır. Bu tip semaforlar, ikili (binary) semaforlar olarak adlandırılır.

```
Semaphore mutex=1;
```

Proses 0:

```
. . .
P(mutex);
//kritik kesime giriş
toplam=toplam + degisim;
V(mutex);
//kritik kesimden çıkış
. . .
```

Proses 1:

```
. . .
P(mutex);
//kritik kesime giriş
toplam=toplam - degisim;
V(mutex);
//kritik kesimden çıkış
. . .
```

Örnek: Üretici/Tüketici (Producer/Consumer ya da Bounded Buffer) Problemi

Üretici-Tüketici probleminde, üreticinin görevi yeni bir ürün ürettiğinde ortaklaşa kullanılan bir alana bırakmaktır, tüketicinin görevi ise bu alandan alarak bu ürünü kullanmaktır. Bu ortak alanın belli bir büyüklük sınırı vardır. Değişik sayıda üretici ve tüketici olabilir, problemin daha kolay anlaşılması için bir üretici ve bir tüketici olduğu kabul edilecektir.

Ortak kullanılan alana aynı anda sadece bir erişim hakkı (üretici ya da tüketici) verilir. Bunu sağlamak için *mutex* (mutual exclusion'ın kısaltması) isimli semafor kullanılır. *mutex* sadece iki değer alır; 1 ise kritik kesime giriş yapılabilir, 0 ise beklenir. İkinci koşul, ortak alan boş ise tüketicinin bekletilmesidir, bu amaç için *full* semaforu kullanılır. Üçüncü koşul ise, ortak alanın dolması durumunda üreticinin bekletilmesidir, *empty* semaforu kullanılır. Proseslerin N boyutlu bir bellek alanını ortaklaşa kullandığı varsayılırsa, *full* ve *empty* semaforları 0 ile N arasındaki tamsayı değerlerini alabilen **sayan (counting) semaforlardır.**

Örnek: (Devamı)

```
int N=100;
Semaphore mutex=1; Semaphore full=0; Semaphore empty=N;
```

Producer:

```
int item;
. . .
produce_item(item); //üretim işlemi
P(empty);//bellek bölgesi doluysa proses bloklanır
P(mutex); //kritik kesime giriş
enter_item(item); //ürünü bellek bölgesine koy
V(mutex); //kritik kesimden çıkış
V(full); //1 bellek bölgesinin daha kullanıldığını
//ifade eder
. . .
```

Consumer:

```
int item;
. . .
P(full); // bellek bölgesi boşsa prosesi bloklar
P(mutex); //krtik kesime giriş
remove_item(item); //ürünü tampondan al
V(mutex); //kritik kesimden çıkış
V(empty); //1 bellek bölgesinin tekrar
//kullanılabileceğini ifade eder.
consume_item(item);//tüketim işlemi
. . .
```

Örnek: Semaforların akış denetiminde kullanılması

Bir hesaplama işleminde, hesap yükünün 2 prosese bölüştürüldüğünü ve hesaplama adımlarının belli bir sıra dahilinde olmasını istediğimizi düşünelim.

İlk olarak Proses_a'nın *x* değişkeni üzerinde işlem yapması ve Proses_b'nin bunu beklemesi,

daha sonra da Proses_b'nin *y* değişkeni üzerinde işlem yapması ve bu sırada Proses_a'nın beklemesi isteniyor.

Bu problem, kritik bölge probleminden ziyade prosesler arasındaki senkronizasyonla ilgilidir. Semaforlar bu tip senkronizasyon işlemleri için de kullanılabilir.

Örnek: (Devamı)

```
Semaphore S1=0; Semaphore S2=0;
```

Proses_a:

```
. . .
write(x); //x'i oluştur
V(S1);//Proses_b'ye sinyal gönder
Hesaplama_a(); //diğer hesaplamalar
P(S2);//Proses_b'den sinyal bekle
read(y);//y'yi kullan
. . .
```

Proses_b:

```
. . .
P(S1);//Proses_a'dan sinyal bekle
read(x); //x'i kullan
Hesaplama_b();//diğer hesaplamalar
write(y);//y'yi oluştur
V(S2);//Proses_a'ya sinyal gönder
. . .
```

Monitörler

Prosesler arasındaki senkronizasyon problemini çözmede kullanılan, programlama dili desteği gerektiren üst seviyeli bir yapıdır.

Semaforlarla çözülen senkronizasyon problemleri monitörle de çözülebilir.

Monitörler, proseslerin kritik kesimlerinin toplandığı merkezi bir denetim sağlama aracıdır. En önemli özelliği, bir anda sadece bir işlem monitörde aktif olabilir.

```
monitor ortak_değişken {
private:
   int toplam;
public:
   ekle(int degisim) {toplam=toplam+degisim;};
   çıkart(int değişim) {toplam=toplam-degisim;};
}
```

PROSESLER ARASI İLETİŞİM -IPC

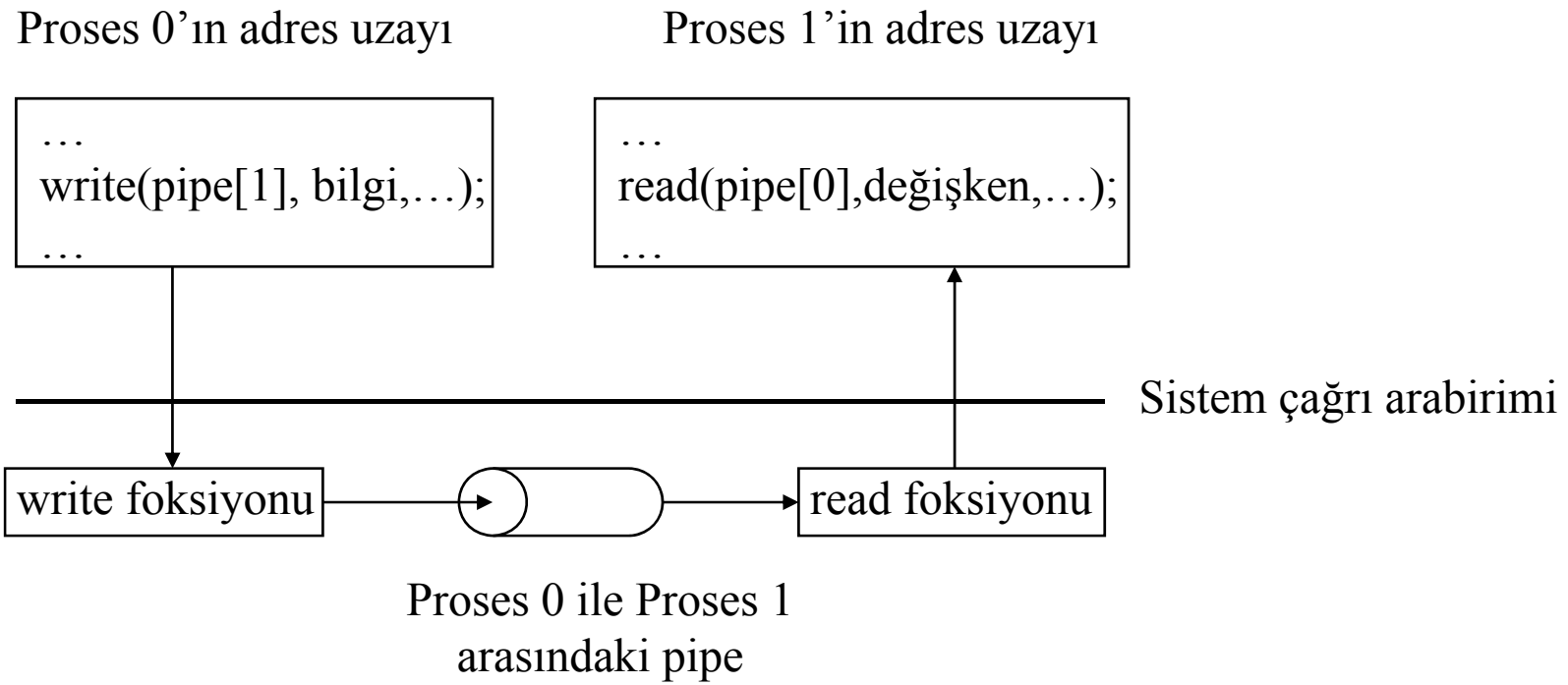
- Pipe Modeli
- Mesaj Geçişi
  - Posta kutusu ve Port
  - Üretici /Tüketici probleminin mesaj geçişiyle çözümü

Prosesler Arası İletişim -IPC

Şimdiye kadar anlatılan mekanizmalarda prosesler arasındaki bilgi paylaşımı, paylaşılır adres veya bellek bölgeleri vasıtasıyla yapıldı. Özellikle farklı bilgisayarlar üzerindeki prosesler söz konusu olduğunda, bu prosesler arasında paylaşılır bellek bölgeleri olmadığından, bilgi paylaşımı ve akış denetimi için farklı mekanizmaların kullanılmasına ihtiyaç vardır.

Pipe Modeli

İlk olarak Unix'te uygulanan, farklı adres uzaylarında paylaşım sağlayan ve "first-in first-out" mantığıyla çalışan çekirdek seviyesinde bir veri yapısıdır.

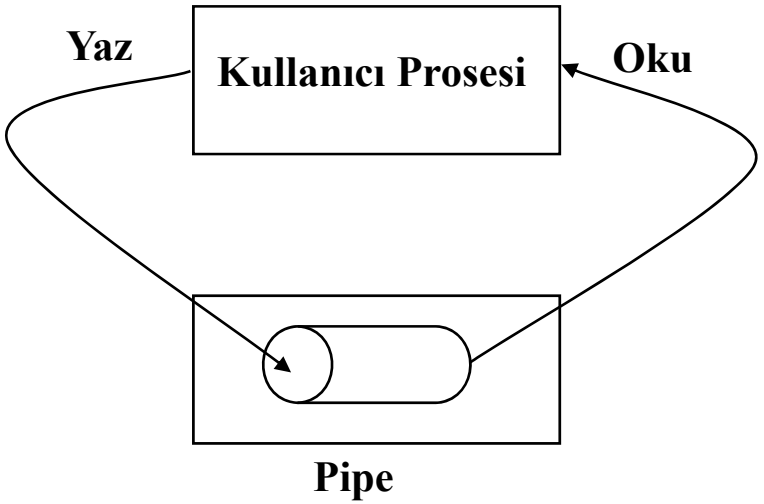


Proses 0 ile Proses 1 arasındaki pipe

- Genellikle bir proses pipe'e veri yazarken, diğeri okur.
- Pipe dolu ise, pipe'a yazmaya çalışan proses bloke olur.
- Pipe boş ise, pipe'dan okuma yapmaya çalışan proses bloke olur.

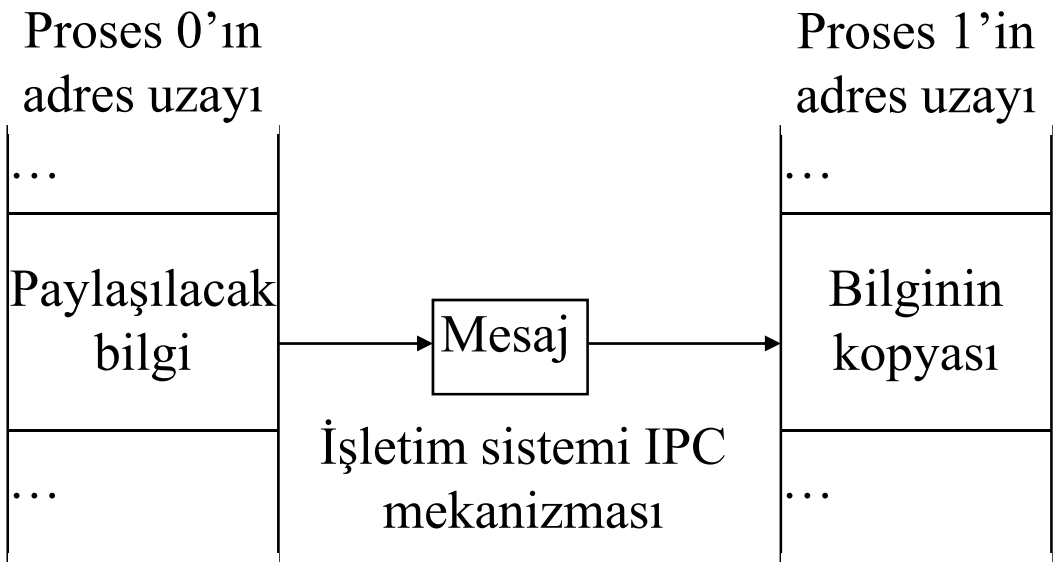
Örnek: Tek bir kullanıcı prosesinden, pipe'a yazma ve aynı prosesten okuma işlemi.

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

#define mesaj_uzunlugu  10;

char *mesaj1 = "merhaba";
char *mesaj2 = "MERHABA";

main(void) {
   char gelen_mesaj[mesaj_uzunluğu];
   int p[2];
   pipe(p);

   /* pipe'a yazılıyor */
   write(p[1], mesaj1, mesaj_uzunluğu);
   write(p[1], mesaj2, mesaj_uzunluğu);

   /*pipe'tan okunuyor */
   for(j=0; j<2; j++)
   {   read(p[0], gelen_mesaj, mesaj_uzunluğu);
       printf("%s\n", gelen_mesaj);
   }
 }
```

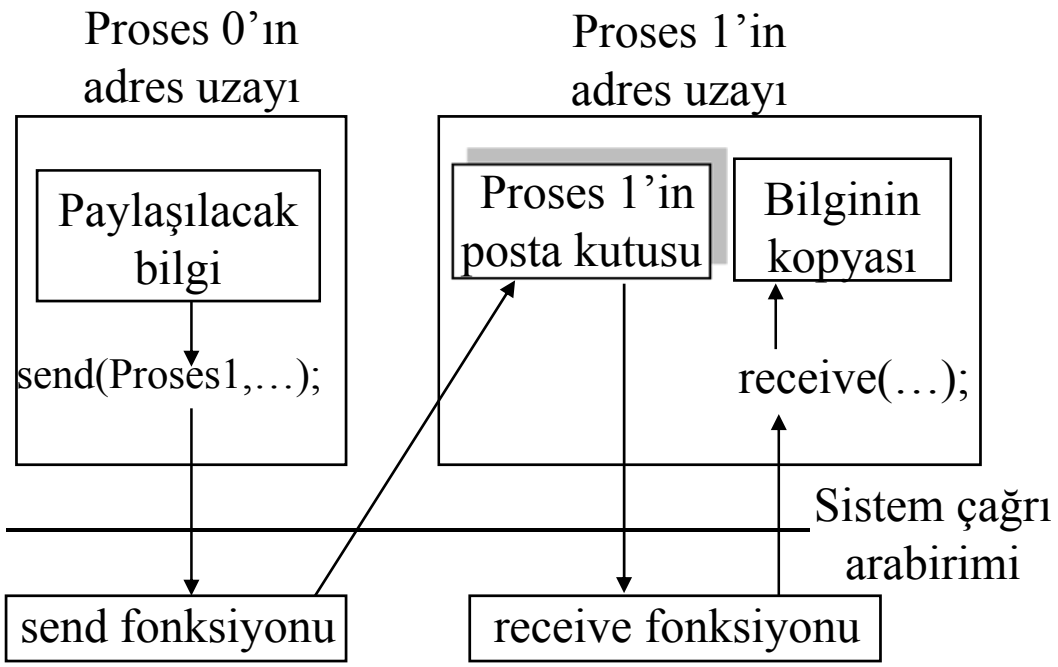


Mesaj Geçişi

Birine bir mesaj bırakmak istediğimizde, alıcının anlayabileceği bir formda mesajı oluşturur, alıcının mesajlarını tuttuğu posta kutusuna göndeririz. Mesaj geçişiyle prosesler arası etkileşim de aynı şekilde olur.

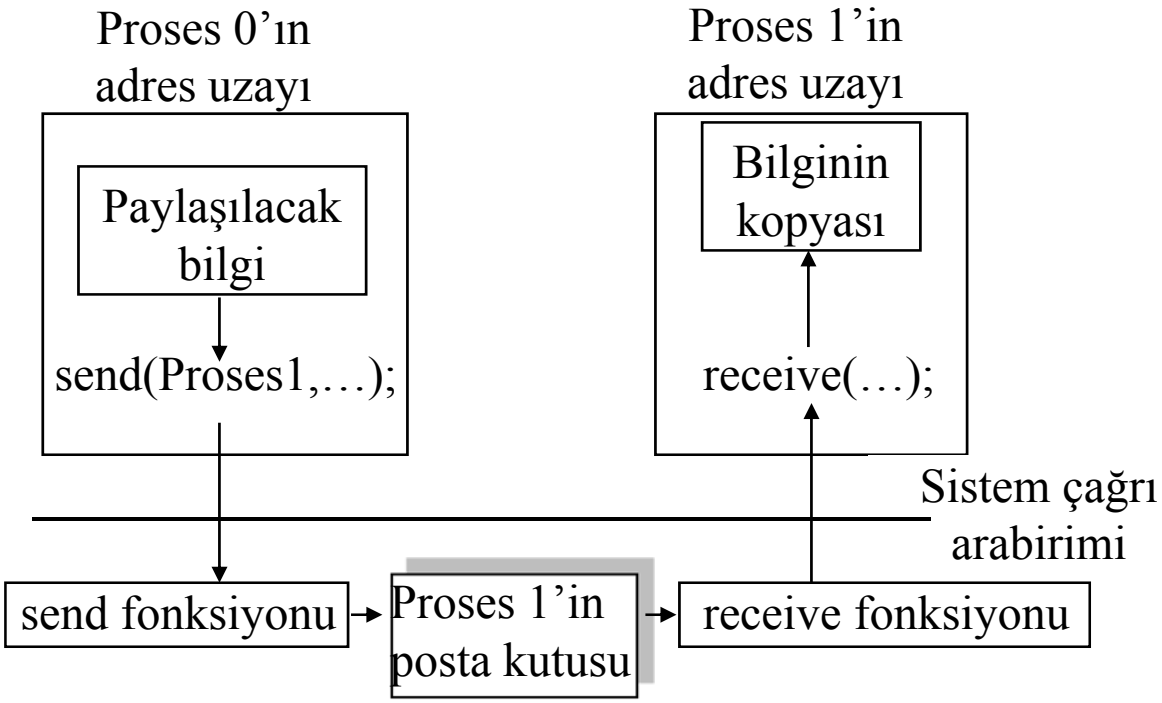


Posta kutusunun kullanıcı prosesin adres uzayında oluşturulması



Posta kutusunun sistem uzayında oluşturulması

Posta kutularının kullanıcı adres uzayında oluşturulması durumunda bazı problemlerle karşılaşılabilir; kazara posta kutusunun dolu kısımlarının üstüne yazılması durumunda mesajlar kaybolabilir.



send( ) ve receive( ) işlemleri

send() ve *receive()* işlemleri için 2 seçenek vardır;

send() işlemi, senkron (synchronous) veya asenkron (asynchronous) yapıda,

receive işlemi ise bloklanan (blocking) veya bloklanmayan (nonblocking) yapıda olabilir. Bu sistem çağrılarına, kütüphaneler vasıtasıyla erişim yapılır.

send() işlemi

Asenkron send() işlemi, mesajı alıcının posta kutusuna gönderir ve alıcının mesajı okumasını beklemeden işlemine devam eder.

Senkron send() işleminde, alıcı proses mesajı başarılı bir şekilde alıncaya kadar gönderici proses bloklanır.

Senkron send() işleminin zayıf (weak) ve güçlü (strong) formu vardır; zayıf formda, alıcının posta kutusuna güvenli şekilde gönderildikten sonra işletim devam eder,

güçlü formda ise alıcı proses posta kutusundan mesajı alıncaya kadar gönderen proses bloklanır. Özellikle prosesler arası senkronizasyonda güçlü senkron mod kullanılır. zayıf form bu gereksinimi karşılamaz ancak asenkron send() işlemine göre daha güvenlidir.

receive() işlemi

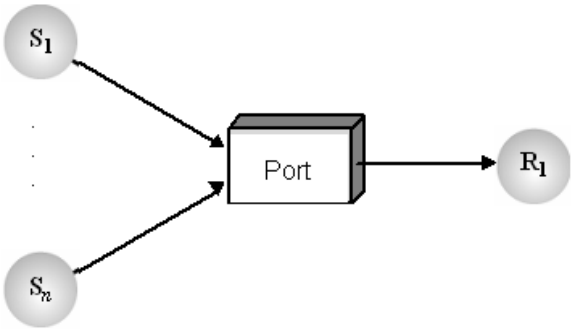
Bloklanan receive() işleminde, alıcının posta kutusuna mesaj daha önceden gönderilmişse mesaj alınır ve işletime devam edilir. Mesaj yoksa alıcı proses bloklanır. Bloklanmayan yapıda ise posta kutusunda mesaj olmasa bile proses işletimine devam eder.

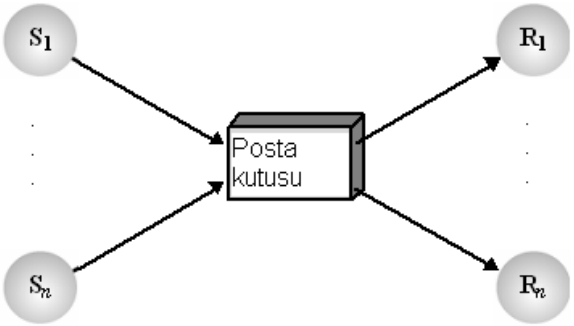
Uygulamada özellikle iki kombinasyon kullanılır:

- Senkron send()-bloklanan receive(): Hem alıcı hem de gönderici, mesaj iletilene kadar bloklanır. Prosesler arasında sıkı bir senkronizasyon denetimi sağlar.
- Asenkron send()-bloklanan receive(): Gönderici işletimine devam etmesine rağmen, alıcı mesaj alınıncaya kadar bloklanır. Özellikle servis sağlamaya yarayan server prosesleri bu türdendir.

Posta kutusu ve Port

Gönderici ya da alıcı prosesler posta kutusuna tekli ya da çoklu olarak erişebilirler. Gönderici ve alıcı arasında bire bir bağlantı özel bir bağlantı şeklidir; diğer proseslerin müdahalesinden arınmış bir yapıdır. Gönderici proseslerin çoklu olarak posta kutusuna ulaştığı ve tekli alıcının olduğu yapı *client/server* mimarisi olarak bilinir ve bu sistemlerde posta kutuları *port* olarak anılırlar.





Örnek: Üretici Tüketici probleminin mesaj geçişiyle çözümü

```
Const int kapasite=…;//tutulacak mesaj sayısı

void main(){
  create_mailbox(üretici_pkutusu);
  create_mailbox(tüketici_pkutusu);

  for(int i=1; i<=kapasite; i++)
      send(üretici_pkutusu,NULL)
}
```

```
üretici(){
message ü_msg;
 while(True){
   receive(üretici_pkutusu,ü_msg);
   ü_msg=üret();
   send(tüketici_pkutusu,ü_msg);
 }
}
```

```
tüketici(){
message t_msg;
 while(True){
   receive(tüketici_pkutusu,t_msg);
   tüket(t_msg);
   send(üretici_pkutusu,NULL);
 }
}
```

Kilitlenme (Deadlock)

- **Kilitlenme için Gerekli Koşullar**
- **Kilitlenme Durumunda Kullanılan Yaklaşımlar**
  - Sistemin kilitlenme durumuna girmemesini sağlamak.
    - Kilitlenmeyi önlemek
    - Kilitlenmeden kaçınmak
  - Sistem kilitlenme durumuna girdiyse bu durumdan kurtulmasını sağlamak.

Kilitlenme (Deadlock) Nedir?

Sistem kaynaklarını ortak olarak kullanan veya birbiri ile haberleşen bir grup prosesin kalıcı olarak bloke olması durumuna kilitlenme adı veriler.

Üç prosese ait kod blokları aşağıdaki gibi olsun;

Proses 1

```
…
Request(Kaynak_1);
//Kaynak_1 tutuluyor
…
Request(Kaynak_2);
```

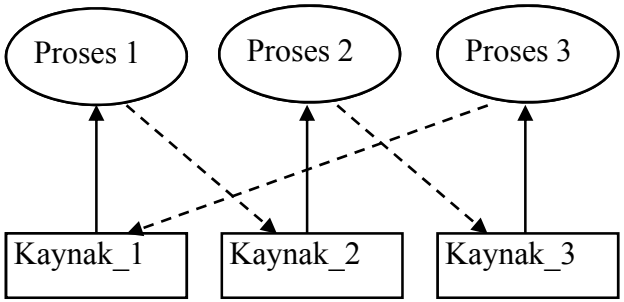
Proses 2

```
…
Request(Kaynak_2);
//Kaynak_2 tutuluyor
…
Request(Kaynak_3);
```

Proses 3

```
…
Request(Kaynak_3);
//Kaynak_3 tutuluyor
…
Request(Kaynak_1);
```

Diğer bir gösterimle;



Kilitlenmeler Ne Tip Kaynaklar Üzerinde Olur?

Kilitlenmeler tekrar kullanılabilir kaynaklar (birincil ve ikincil bellekler, I/O cihazları, dosyalar, semaforlar) ya da tüketilir kaynaklar (kesmeler, mesajlar, I/O cihazlarının tamponları) üzerinde oluşabilir.

Örneğin iki prosesin mesaj geçişi yoluyla iletişim kurduğu ve her iki prosesin de birbirinden gelecek mesajı beklemesi durumunda kilitlenme yaşanacaktır.

Proses_1

```
Receive(Proses_2);
…
Send(Proses_2);
…
```

Proses_2

```
Receive(Proses_1);
…
Send(Proses_1);
…
```

Kilitlenme Durumları

Genel olarak iki tip kilitlenme durumu olabilir;

- Bir proseste yer alan thread'ler arasında
- Farklı proseslerdeki thread'ler arasında

Ayrıca uygulama programları ve işletim sistemi fonksiyonları arasında da kilitlenme meydana gelebilir.

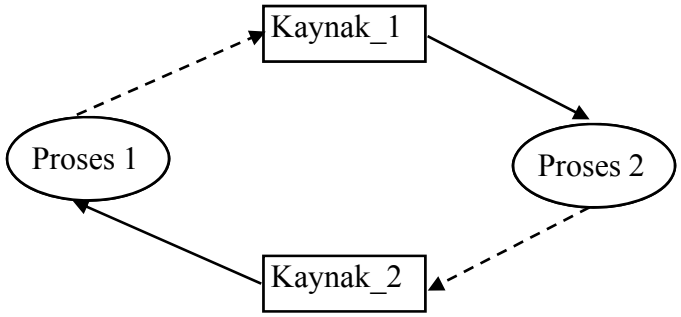
Kilitlenme için Gerekli Koşullar

1. Karşılıklı dışlama (mutual exclusion): En azından bir kaynağın, diğer prosesler tarafından kullanılamadığı bir durumda kilitlenmeden söz etmek mümkündür.

2. Sahiplenme ve bekleme (hold and wait): Bir proses en azından bir kaynağı elinde tutmalı ve diğer prosesler tarafından tutulan ek bir kaynağa gereksinim duymalıdır.

3. Geri alınamaz kaynak(no preemption): Bir prosese atanan kaynaklar, işletim sistemi tarafından prosesin kendi isteği dışında elinden alınamıyorsa.

4. Döngüsel bekleme (circular waiting): Proseslerin gereksinim duyduğu kaynakları, karşılıklı olarak birbirlerinden beklemesi durumudur.



Kilitlenme Durumunda Kullanılan Yaklaşımlar

- Sistemin kilitlenme durumuna girmemesini sağlamak.
  - –Kilitlenmeyi önlemek
  - –Kilitlenmeden kaçınmak

- Sistem kilitlenme durumuna girdiyse bu durumdan kurtulmasını sağlamak.

Kilitlenmeleri Önleme (Deadlock Prevention)

İlk üç gerekli şarttan birinin elimine edilmesine yönelik endirekt yöntem ya da döngüsel beklemenin engellendiği direkt yöntem mevcuttur.

Karşılıklı dışlama: Karşılıklı dışlama şartı, paylaşılamaz kaynaklar için gereklidir.

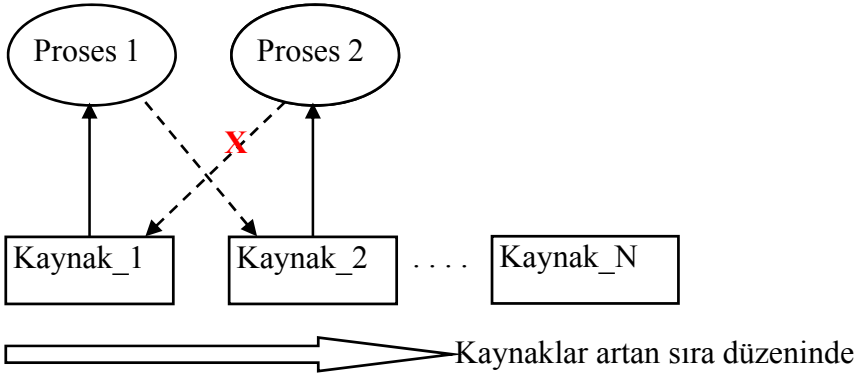
Sahiplenme ve bekleme: Bir proses bir kaynak talep ettiğinde sahip olduğu diğer kaynakları tutmaması sağlanırsa bu şartın oluşması engellenebilir. İki yaklaşım olabilir;

- Bir prosesin gereksinim duyacağı tüm kaynaklar işletime başlamadan atanır.
- Bir proses ek bir kaynak talep etmeden önce, mevcut tüm kaynaklarını bırakır.

Kilitlenmeleri Önleme

Geri alınamaz kaynak: Proseslere atanan kaynakların, proseslerin kendi istekleri dışında da bırakması sağlanırsa kilitlenme önlenebilir. Paylaşılamayan kaynakların kullanımı tamamen sonlanmadan bir prosesten alınıp diğer bir prosese tahsisi yapılamayacağından, bu tür kaynaklar için bu koşul devre dışı bırakılamaz.

Döngüsel bekleme: Kaynakların belli bir sırada tahsis edilip bırakılması suretiyle döngüsel beklemenin önüne geçilebilir. Bu yöntemde de kaynakların verimsiz kullanımı söz konusu olur.



Kaynakların tahsisi (x, bu kaynak talebinin karşılanmayacağını belirtmektedir)

Kilitlenmeden Kaçınma (Deadlock Avoidance)

Kilitlenmeden kaçınmada yatan temel felsefe, proseslerin kaynak taleplerinin gelecekte kilitlenmeye neden olup olmayacağının dinamik olarak belirlenmesidir.

İki yaklaşım vardır;

- Kaynak istekleri bir kilitlenmeye neden olacaksa prosesi başlatma!

- Proseslerin ek kaynak talepleri kilitlenmeye neden olacaksa kaynak tahsisi yapma!

Prosesin İşletimini Başlatmama Durumu

n proses ve *m* farklı kaynağa sahip bir sistem düşünelim.

$(P_1, P_2, \ldots, P_n)$:Sistemdeki prosesler

$(K_1, K_2, \ldots, K_m)$: Sistemdeki her bir kaynağın toplam miktarı

$(H_1, H_2, \ldots, H_m)$: Proseslere atanmamış her bir kaynağın toplam miktarı

$$\begin{pmatrix} M_{11} & M_{12} & \cdots & M_{1m} \\ M_{21} & M_{22} & \cdots & M_{2m} \\ \cdots & \cdots & & \cdots \\ M_{n1} & M_{n2} & \cdots & M_{nm} \end{pmatrix}$$:Proseslerin her bir kaynağa olan maksimum ihtiyaçları

$$\begin{pmatrix} T_{11} & T_{12} & \cdots & T_{1m} \\ T_{21} & T_{22} & \cdots & T_{2m} \\ \cdots & \cdots & & \cdots \\ T_{n1} & T_{n2} & \cdots & T_{nm} \end{pmatrix}$$: Proseslere şu an atanmış kaynaklar

Prosesin İşletimini Başlatmama Durumu

Bu durumda şu ilişkiler yazılabilir;

i) $K_i = H_i + \sum_{k=1}^{n} T_{ki}$: Kaynaklar ya hazır ya da tahsis edilmiştir

ii) $M_{ki} \leq K_i$: Prosesler sistemdeki toplam kaynak miktarından daha fazla kaynak talebinde bulunamazlar

iii) $T_{ki} \leq M_{ki}$: Prosesler talep ettiklerinden daha fazla kaynak talebinde bulunamazlar

Bu üç ilişkiden yola çıkarak kilitlenmeden kaçınmak için gerekli olan eşitsizliği yazabiliriz;

$$M_{(n+1)i} + \sum_{k=1}^{n} M_{ki} \leq K_i$$

Kaynak Tahsisi Yapmama Durumu

Bu yaklaşım, Dijkstra tarafından önerilen Banker Algoritmasıyla özleşmiştir; belli bir sermayeye sahip bir bankerin iflas etmeden müşterilerine kredi imkanı tanıması esastır. Müşterinin tüm gereksinim duyduğu krediyi almadan, geri ödeme garantisi yoktur. Şayet bir miktar kredi talep edildiğinde diğer müşterilerin ihtiyaçlarının karşılanamaması riski varsa kredi verilmemektedir.

Sistemin içinde bulunabileceği iki durum vardır; güvenli ve güvenli olmayan durum. Güvenli durumda, tüm proseslerin kilitlenme oluşturmayacak şekilde sonlanmasına olanak veren bir işletim şekli mevcuttur. Böyle bir işletim şekli yoksa güvenli olmayan durumdur.

Banker Algoritması

Örnek: Bir sistemin 4 prosese ve 3 tip kaynağa sahip olduğunu düşünelim. Proseslerin bu kaynaklardan maksimum talep edeceği ve şu an atanmış miktarlar aşağıdaki gibi olsun;

(6,2,3) : Sistemdeki her bir kaynağın toplam miktarı

Proseslerin her bir kaynağa olan maksimum ihtiyaçları

$$\begin{pmatrix} 4 & 2 & 2 \\ 3 & 0 & 3 \\ 2 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Proseslere şu an atanmış kaynaklar

$$\begin{pmatrix} 2 & 1 & 0 \\ 1 & 0 & 2 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Örnek: (Devamı)

- Prosesler tarafından kullanılmayan kaynak miktarları;

(6,2,3) - (5,1,2) = (1,1,1)

$$\begin{pmatrix} 2 & 1 & 0 \\ 1 & 0 & 2 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \textbf{Proseslere şu an atanmış kaynaklar}$$

$$+$$

5 1 2

- Proseslerin ihtiyaç duyacakları her bir kaynağa ait miktarlar;

$$\begin{pmatrix} 4 & 2 & 2 \\ 3 & 0 & 3 \\ 2 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} - \begin{pmatrix} 2 & 1 & 0 \\ 1 & 0 & 2 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 2 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{matrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{matrix}$$

Örnek: (Devamı)

Proseslerin geriye kalan kaynak ihtiyaçları

$$\begin{pmatrix} 2 & 1 & 2 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{matrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{matrix}$$

Proseslere atanmış kaynaklar

$$\begin{pmatrix} 2 & 1 & 0 \\ 1 & 0 & 2 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

- İlk olarak P_3 prosesine gereksinim duyduğu kaynakları atayalım;
  P_3'ün işletimi tamamladığında toplam kaynak miktarı (1,1,1)+(1,0,0)=(2,1,1) olacaktır.

- Şimdi de P_2 prosesine gereksinim duyduğu kaynakları atayalım;
  P_2'nin işletimi tamamlandığında toplam kaynak miktarı (2,1,1)+(1,0,2)=(3,1,3) olacaktır.

- Benzer şekilde P_1 ve P_4'ün gereksinimleri de karşılandığından bu durum güvenli bir durumdur.

  $P_3 \rightarrow P_2 \rightarrow P_1 \rightarrow P_4$ sırasında işletilebilir. Başka sıra düzenleri de olabilir.

  Ama **P_1** veya **P_2** ile işletim başlatılamaz.

Kilitlenmeyi Algılama (Deadlock Detection)

Kilitlenmeyi önleme ve kilitlenmeden kaçınma algoritmaları kullanılmıyorsa kilitlenme durumu oluşabilir. Bu durumda, kilitlenme durumunun oluşup oluşmadığını kontrol eden bir algoritmaya ve bu durumdan kurtulmaya yarayacak bir algoritmaya gereksinim vardır.

İki şekilde çözüm üretilebilir;

• Döngüsel beklemeye neden olan bir ya da daha çok prosesin çalışması yarıda kesilebilir.

• Kilitlenen proseslerden bazı kaynaklar geri alınabilir.

Kilitlenmeyi Algılama

Proseslerin sonlandırılması:

İki metot kullanılabilir.

- Kilitlenen tüm prosesler sonlandırılır.
- Kilitlenme durumundan çıkılıncaya kadar prosesler birer birer sonlandırılır.

Kaynakların geri alınması:

Kilitlenme giderilene kadar bazı kaynaklar proseslerden alınır ve diğer proseslere verilir. Kaynaklar proseslerin elinden alındığında normal çalışmaları devam edemeyecektir. Proseslerin güvenli olduğu durumlar belli aralıkla kaydedilebilir ve tekrar bu noktadan çalıştırılmaya başlanabilir.

Karma Yaklaşımlar

Araştırmacılar kilitlenmeyi önleme, kilitlenmeden kaçınma ve kilitlenmeyi algılama algoritmalarının tek başlarına işletim sisteminde karşılaşılan kaynak tahsisi problemini çözmediği sonucuna varmışlardır. Bu yaklaşımların kombine kullanılması ön plandadır. Kaynakların hiyerarşik düzende sınıflara ayrılması, belli sıra dahilinde kaynak tahsisi yapılması ve her bir sınıfa da en uygun tekniğin kullanılması önerilmiştir.

BELLEK YÖNETİMİ

Bellek Türleri

Bellek Organizasyonu

Adres Soyutlaması

Adres Uzayının Yönetimi

Belleğin Bölümlenmesi (Partitioning)

- Statik Bölümleme
- Dinamik Bölümleme
- Buddy Yerleştirme Algoritması

Bellek Türleri

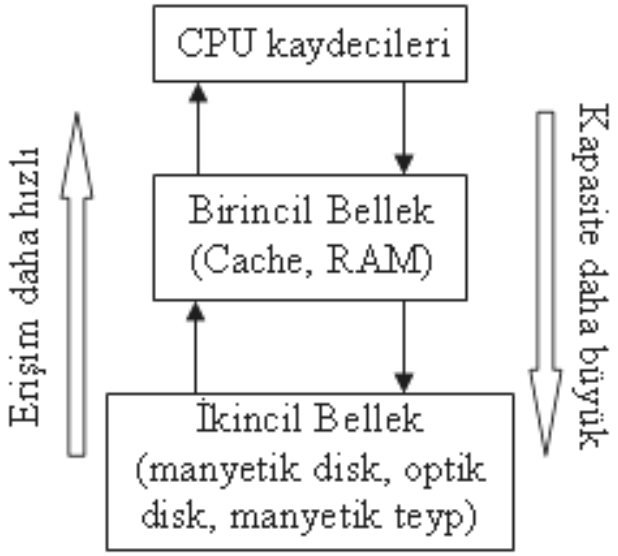
Bellek sistemi birincil ve ikincil bellekler olarak ikiye ayrılır.

Birincil bellek CPU tarafından kullanılmakta olan bilgiyi tutar. İkincil bellekler ise depolama alanı olarak kullanılır. Birincil bellekler, ikincil belleklere göre daha hızlı erişilebilen, uçucu ve her erişimde bir sözcük (word) okunabilen bir yapıya sahiptir.

İkincil bellekler ise verilere bloklar halinde erişilebilen kalıcı tip belleklerdir.

Bellek Organizasyonu

von Neumann mimarisine sahip bir bilgisayar sisteminde bellek en az üç seviyeden oluşur; en üst seviyede CPU kaydedicileri, orta seviyede ana (birincil) bellek ve en alt seviyede de ikincil bellektir.

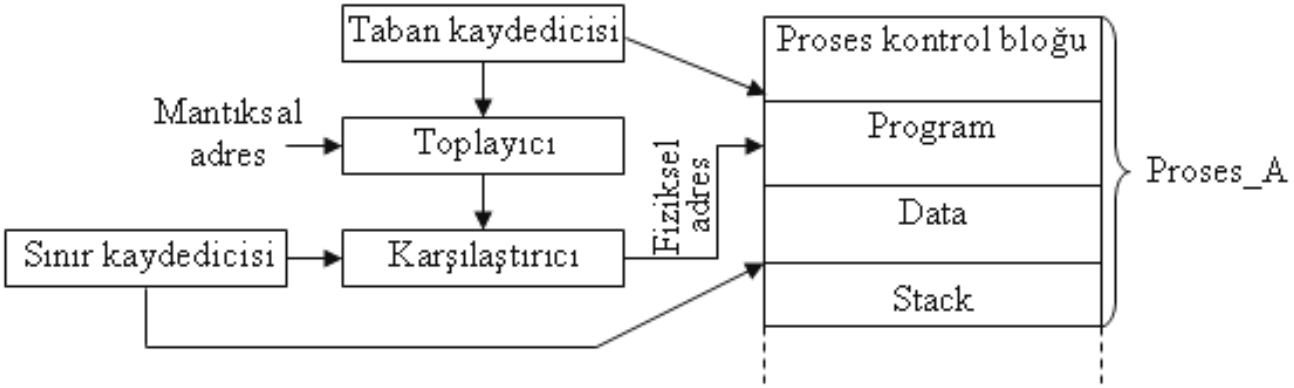


Bellek organizasyonu

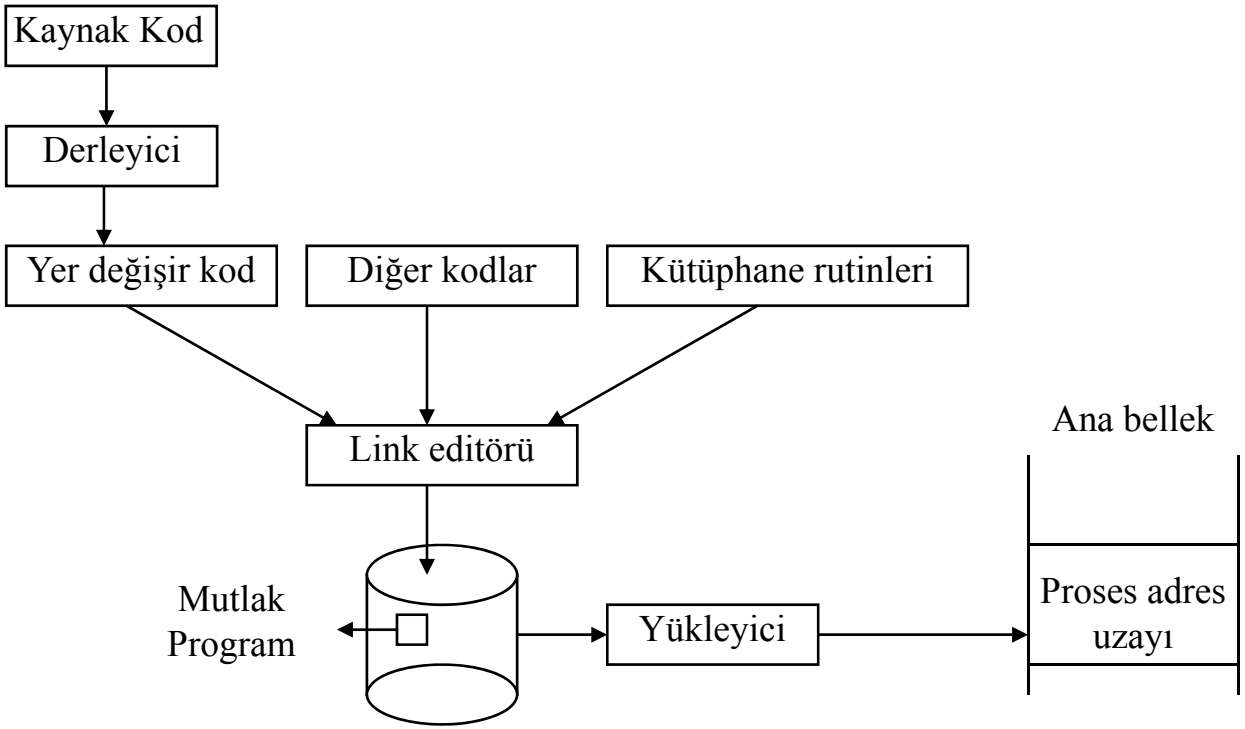
Adres Soyutlaması

İşletim sistemlerinin gelişiminin ilk yıllarında programcının adres uzayı fiziksel ana bellek idi. Yani çalışan program, bilgisayarın ana belleğine doğrudan erişebiliyordu. Çoklu programlamayı destekleyen işletim sistemleri ile birlikte fiziksel ana belleğe doğrudan erişmek yerine mantıksal ana bellek adreslerinin kullanımı ortaya çıkmıştır.

Örneğin bir prosese ana bellekte 0x10000-0x20000 arası alanlar tahsis edildiyse ve ana bellekteki 0x11000 alanına erişim yapılacaksa bunun mantıksal adresi en yalın haliyle 0x01000 olacaktır.



Adres Uzayının Yönetimi



Bellek Yöneticisinin Sorumlulukları

1. **Yeniden yerleştirme (Relocation): İ**şlemci ve bellek verimliliğini arttırmak için prosesler işletimlerinin değişik aşamalarında ikincil belleğe atılıp daha fazla sayıda hazır prosesin bellekte tutulması sağlanabilir ve daha sonra tekrar farklı bir konumdan ana belleğe geri yüklenebilir.

2. **Koruma (Protection):** Her bir proses, diğer prosesler tarafından yapılacak istenmeyen girişimlere karşı korunmalıdır.

3. **Paylaşım (Sharing):** Bazı işler için işbirliği halinde olan proseslerin, aynı veri yapılarını paylaşmak zorunda olabilirler.

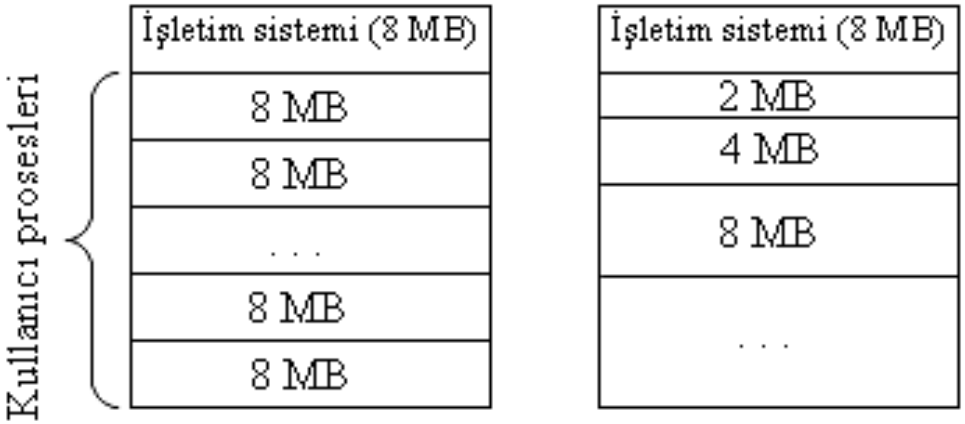
4. **Mantıksal organizasyon**

5. **Fiziksel organizasyon**

Belleğin Bölümlenmesi (Partitioning)

Bellek yöneticisinin temel işlevi, programları ana belleğe getirmektir. Belleği yönetmenin en basit şekli, belleği belli sınırlar dahilinde kesimlere ayırmaktır.

Statik Bölümleme (Fixed partitioning)



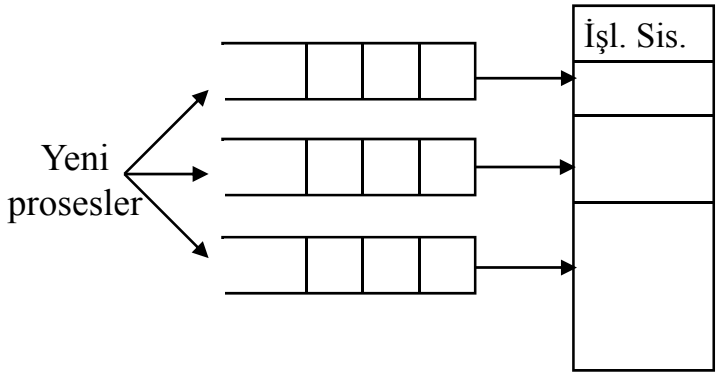
Bu bölümleme şeklinin iki dezavantajı vardır:

- Program bölümlenmiş alana sığmayabilir. Bu durumda programcı modüler yapıda programını hazırlamalı ve bir modül gerekli olduğunda yüklenmelidir.
- Bellek verimli kullanılmaz. Bir program ne kadar küçük olursa olsun tüm bölümü kullanır, bu durum iç parçalanma (internal fragmentation) olarak bilinir.

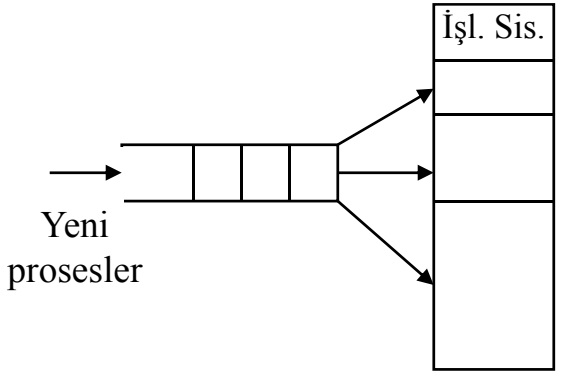
Proseslerin Bölümlere Yerleştirilmesi

Eşit büyüklükte bölümlemede proseslerin belleğe yerleştirilmesi önemsiz bir olgudur. Boş bölüm olduğu müddetçe, bir proses bu bölümlerden herhangi birine yüklenebilir.

Eşit olmayan bölümlemede prosesleri bölümlere yerleştirmenin iki yolu vardır. En basiti, prosesin sığabileceği en küçük bölüme koymaktır. Bu durumda her bir bölüm için bir kuyruk yapısı gereklidir.



Her bölüm için bir proses kuyruğu



Tüm bölümler için tek proses kuyruğu

Dinamik Bölümleme

Bölümler değişken uzunlukta ve sayıdadır. Bir proses ana belleğe getirileceği zaman, gereksinim duyduğu alan kadar yer tahsis edilir.

Proseslerin dinamik bölümlere yerleştirilmesi

Best-fit algoritmasında, ana bellekteki boş alanları tutan liste baştan sona taranır. Gereksinim duyulan alana en yakın alan seçilir.

First-fit algoritmasında, liste başından itibaren boyu uyan ilk boş alan kullanılır.

Next-fit algoritmasında, liste en son yerleştirilen yerden itibaren taranır. Boyu uyan ilk boş alan kullanılır.

Worst-fit algoritmasında, listeden boyu en büyük boş alan seçilir.

Proseslerin dinamik bölümlere yerleştirilmesi

Bu algoritmalarla ilgili olarak bazı genel yorumlar yapılmıştır;

- First-fit en basit, genellikle en iyi ve en hızlı olanıdır.
- Next-fit, first-fit'e göre biraz daha kötüdür, daha fazla sıkıştırma gerektirir.
- Best-fit, isminin aksine en kötü performansa sahiptir. Proses yerleştirildiğinde geriye kalan boş alan pek kullanılamayacağından çok daha sık sıkıştırma yapılmalıdır.
- Worst-fit işlemi sonucunda best-fit'e göre daha büyük alan geride kalacağından, bu alana prosesler yerleştirilebilecektir.

Proseslerin dinamik bölümlere yerleştirilmesi

Örnek : Bellekte boş bölgelerin büyüklükleri sırasıyla 10K, 4K, 20K, 18K, 7K, 9K, 12K ve 15K'dır. Büyüklükleri 12K, 10K ve 9K olan üç proses belleğe yüklenmek isteniyor. Aşağıdaki algoritmalara göre bu üç işi sırasıyla bellekteki boşluklara yerleştiriniz.

a. İlk Uygun (First-Fit) **b.** Sıradaki Uygun (Next-Fit)

c. En Uygun (Best-Fit) **d.** En Az Uygun (Worst-Fit)

Çözüm :

10K	4K	20K	18K	7K	9K	12K	15K

LİSTE

a. 12K üçüncü boşluğa yerleştirilir. Liste 10K, 4K, 8K, 18K, 7K, 9K, 12K, 15K olur.
10K ilk boşluğa yerleştirilir. Liste 4K, 8K, 18K, 7K, 9K, 12K, 15K olur.
9 K üçüncü boşluğa yerleştirilir. Liste 4K, 8K, 9K, 7K, 9K, 12K, 15K olur.

b. 12K üçüncü boşluğa yerleştirilir. Liste 10K, 4K, 8K, 18K, 7K, 9K, 12K, 15K olur.
10K dördüncü boşluğa yerleştirilir. Liste 10K, 4K, 8K, 8K, 7K, 9K, 12K, 15K olur.
9K altıncı boşluğa yerleştirilir. Liste 10K, 4K, 8K, 8K, 7K, 12K, 15K olur.

c. 12K yedinci boşluğa yerleştirilir. Liste 10K, 4K, 20K, 18K, 7K, 9K, 15K olur.
10K ilk boşluğa yerleştirilir. Liste 4K, 20K, 18K, 7K, 9K, 15K olur.
9K beşinci boşluğa yerleştirilir. Liste 4K, 20K, 18K, 7K, 15K olur.

d. 12K üçüncü boşluğa yerleştirilir. Liste 10K, 4K, 8K, 18K, 7K, 9K, 12K, 15K olur.
10K dördüncü boşluğa yerleştirilir. Liste 10K, 4K, 8K, 8K, 7K, 9K, 12K, 15K olur.
9K sekizinci boşluğa yerleştirilir. Liste 10K, 4K, 8K, 8K, 7K, 9K, 12K, 6K olur.

Buddy Yerleştirme Algoritması

Bu sistemde bellek 2^K lık bloklar şeklindedir.

Başlangıçta, tüm bellek 2^U luk tek bir blok olarak düşünülür. Şayet bellek gereksinimi, $2^{U-1}<S\leq 2^U$ ise tüm bellek tahsis edilir. Aksi durumda gereksinim,

$2^{U-2}<S\leq 2^{U-1}$ ise bellek 2^{U-1} lik iki kısma bölünür ve bir kısım tahsis edilir. Bu bölme işlemi prosesin gereksinimi karşılanıncaya kadar devam eder. Prosesin işletimi bitince de bölünmüş kısımlar birleştirilir.

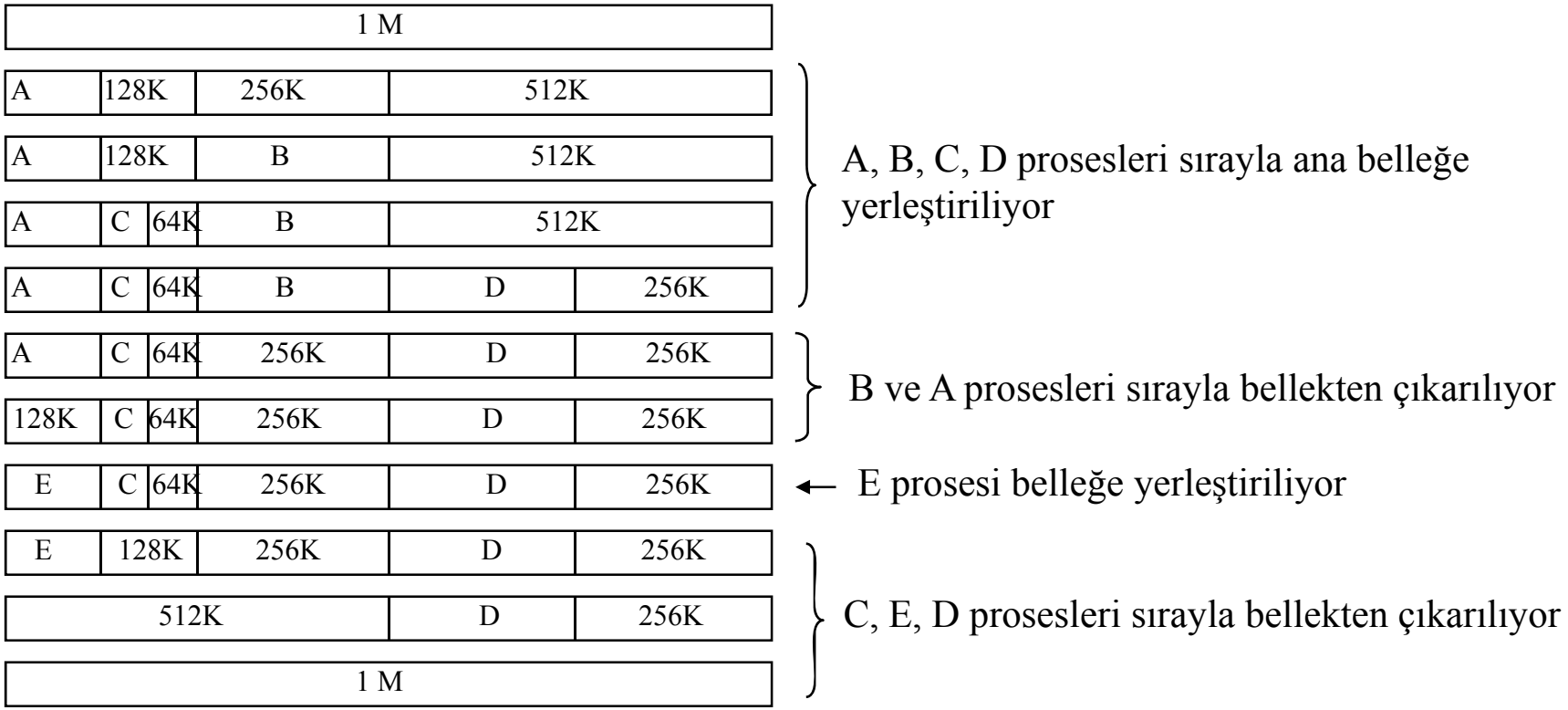
Buddy Yerleştirme Algoritması

Örnek: 5 prosesin (Proses_A 100K, Proses_B 240K, Proses_C 64K, Proses_D 256K, Proses_E 75K) aşağıda verilen sırada Buddy sistemi kullanılarak sığası 1 MB olan ana belleğe yerleştirilip çıkarılması isteniyor.

- İlk olarak A, B, C, D prosesleri sırayla ana belleğe yerleştiriliyor,
- B ve A prosesleri sırayla bellekten çıkarılıyor,
- E prosesi belleğe yerleştiriliyor,
- C, E, D prosesleri sırayla bellekten çıkarılıyor.

Örnek: (Devamı)

Proses_A 100K, Proses_B 240K, Proses_C 64K, Proses_D 256K, Proses_E 75K



BELLEK YÖNETİMİ

Sayfalama Tekniği (Paging)

Segmantasyon (Segmentation)

Sanal Bellek (Virtual Memory)

Sayfalama (Paging) Tekniği

Bellek yönetiminin etkin olabilmesi için belleğin, küçük ve eşit boylu parçalara bölünmesi ve her bir işin de aynı boyda parçalara ayrılması mantığını güden teknik sayfalama olarak bilinir.

Eşit boydaki iş parçalarına **sayfa (page),**
eşit boydaki bellek parçalarına da **çerçeve (frame)** denir.

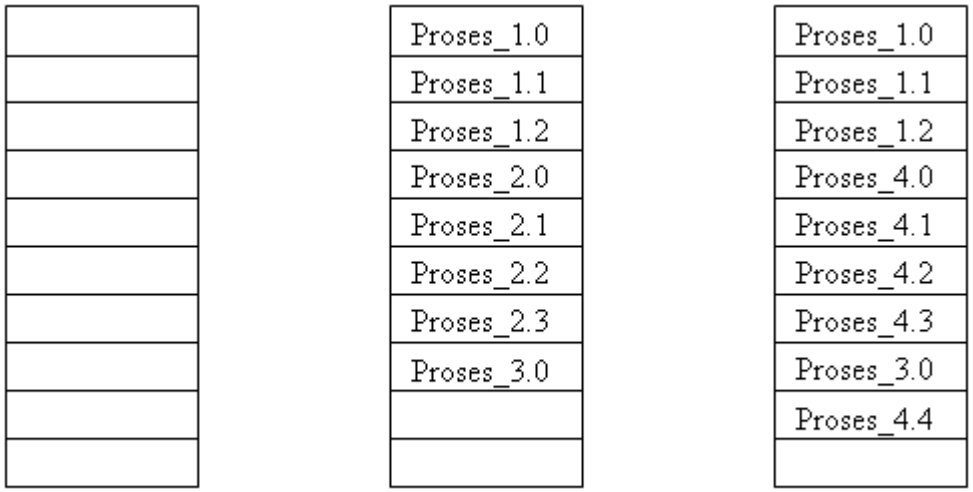
Programdaki her bir mantıksal adres, sayfa numarasını ve bu sayfa içerisindeki yerini gösteren ofset (offset) değerini içerir.

İşletim sisteminin görevi, her iş için sayfa tablolarını (işlerin her sayfasının hangi çerçevede olduğu bilgisini içerir) tutmaktır.

Örnek

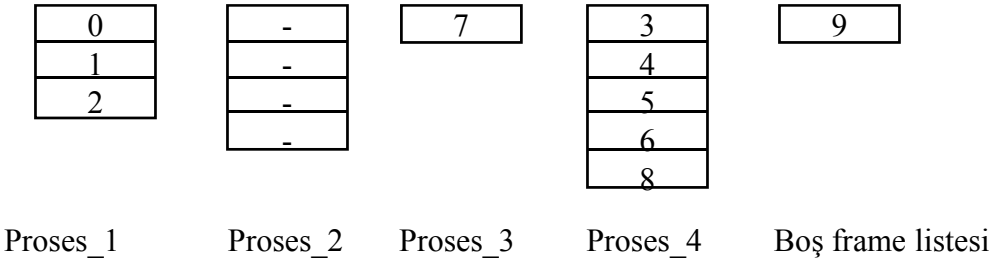
Proses_1, 3 sayfa
Proses_2, 4 sayfa
Proses_3, 1 sayfa
Proses_4, 5 sayfa

10 boş frame



(a) Başlangıçta bellekte 10 boş frame mevcut
(b) Proses_1, Proses_2, Proses_3'ün yüklenmiş hali
(c) Proses_2'nin çıkartılmış, Proses_4'ün yüklenmiş hali

Yukarıdaki örneğin sayfa tabloları aşağıdaki gibidir;



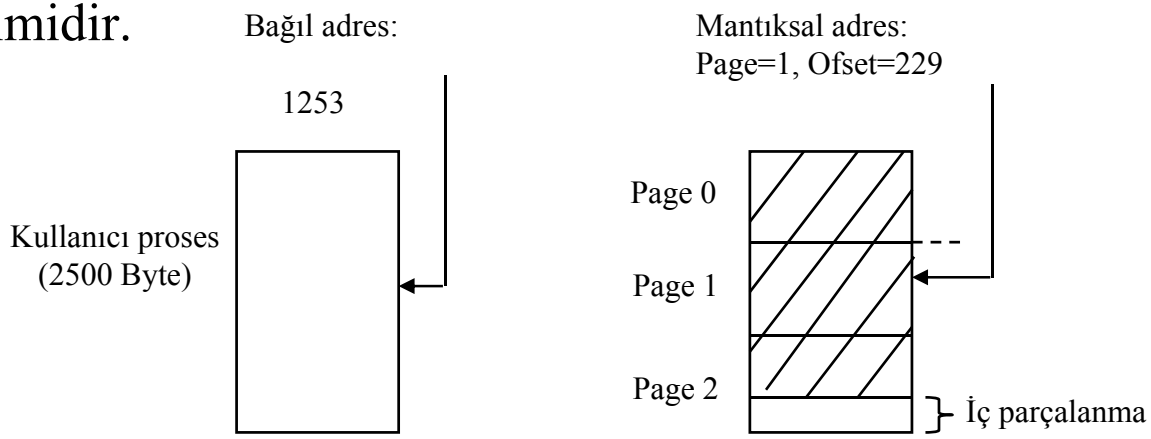
Fiziksel Adresin Mantıksal Adres Eşdeğeri

Örnek: Mantıksal adreslerin 16 bit ve sayfa boyutunun 1 KB olduğunu varsayalım. Program başına göre bağıl adresi 1253 olan bir bellek erişiminin mantıksal adres eşdeğerini bulalım.

16 bit ile 64 KB'lik bir alan adreslenebileceğinden 64 tane frame mevcuttur. 64 sayfa 6 bit ile temsil edilebilir, geriye kalan 10 bit de ofset değeridir.

sayfa_no=1 ofset=229

$(1253)_{10}$ adresinin binary karşılığı $(\mathbf{000001}0011100101)_2$'dir. En anlamlı 6 bit sayfa numarası olacağından, bu erişim birinci sayfada ve ofset değeri 229 olan bellek erişimidir.



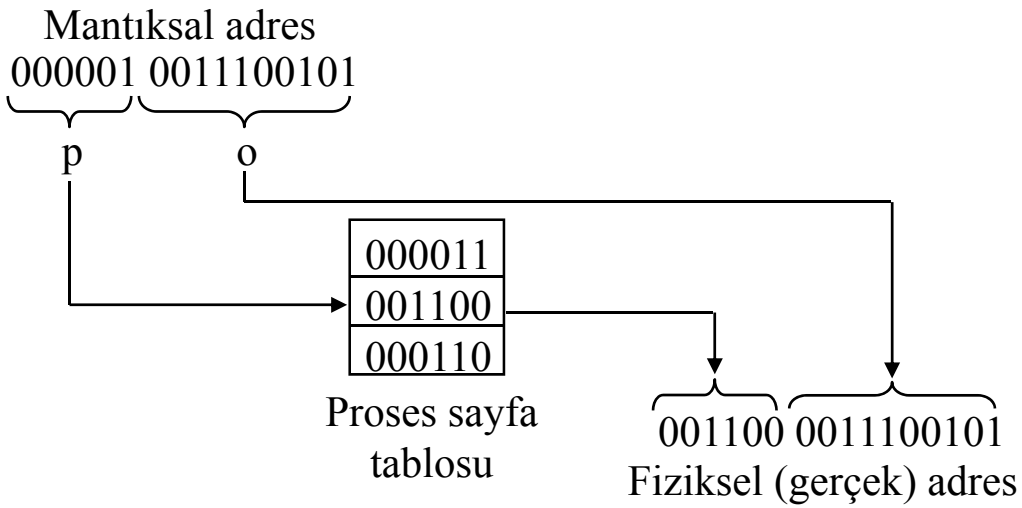
Adres Dönüşümü

Adres uzayı (p+o) bitten oluşsun (burada p: sayfa numarası, o: ofset değeri). Yukarıdaki örneğe göre p=6 ve o=10 olacaktır. Dönüşüm için gerekli adımlar,

- Mantıksal adresin en anlamlı 6 biti ayrılır,

- Elde edilen sayfa numarası sayfa tablosunu indekslemek için kullanılır. frame numarası (f) bulunur,

- frame'in başlangıç adresi $f \times 2^o$ olacaktır ve gerçek adres de bu değere ofset değerinin eklenmesiyle bulunur.



Segmantasyon (Kesimli Bellek-Segmentation)

Segmantasyonda belleğin mantıksal olarak bölümlenmesi esastır. Program ve ilişkili verileri çok sayıda segmente bölünür. Tüm segmentlerin eşit boyda olma zorunluluğu yoktur. Sayfalamada olduğu gibi mantıksal adresler segment numarası ve ofset ile ifade edilir.

Segmentelerin boyu eşit olmadığından dinamik bölümlemeye benzer. Dinamik bölümlemeye göre farkı; program birden fazla bölüm işgal edebilir ve bu bölümler bitişik olmak zorunda değildir. İç parçalanma da yoktur. Ancak dinamik bölümlemede olduğu gibi dış parçalanmalar mevcuttur.

Sayfalamaya benzer şekilde, her bir proses için bir segment tablosu ve boş bellek bilgisinin tutulduğu bir tablo mevcuttur. Segment tablosundaki her bir satır, ilgili segmentin ana bellekteki başlangıç adresini ve segmentin uzunluğunu tutar.

Segmantasyon

Örnek: Mantıksal adreslerin 16 bit olduğu bir sistemde, 4 bitin segment numarasını gösterdiği farz edilirse 12 bit ofset değeri olacaktır. Dolayısıyla maksimum segment uzunluğu 4096 byte olacaktır.

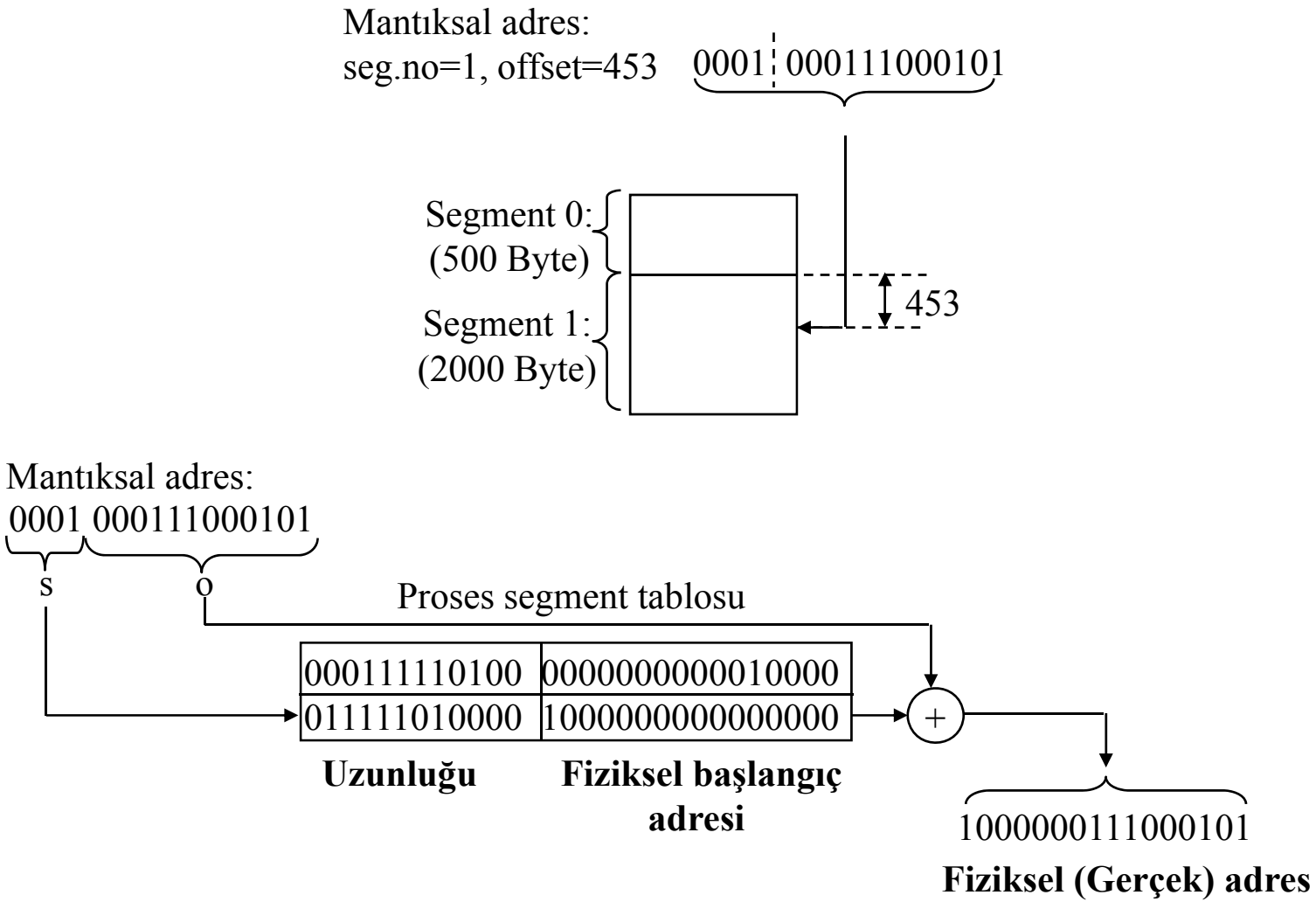
Adres dönüşümü için yapılacak işlemler;

- Mantıksal adresin en anlamlı 4 biti ayrılır,

- Elde edilen segment numarası segment tablosunu indekslemek için kullanılır ve segmentin fiziksel başlangıç adresi bulunur,

- Fiziksel adres, segmentin başlangıç adresine ofset değerinin eklenmesiyle bulunur.

Örnek: (Devamı)



Sayfalama ve Segmantasyon Arasındaki Farklar

1. Sayfa büyüklüğü bilgisayarın donanımına bağlıdır, segmentler ise kullanıcı ya da derleyiciler tarafından belirlenir.

2. Sayfalamada bellek fiziksel olarak eşit uzunluktaki parçalara bölünür. Segmantasyonda ise kavramsal bütünlük oluşturan kesimler belleği oluşturur.

3. Sayfalama programcıya yansıtılmaz, donanımsal olarak gerçeklenir. Segmantasyonda daha karmaşık bir yapı söz konusudur. Segmentler, kullanıcı direktifleriyle veya derleyici vasıtasıyla oluşturulur.

4. Sayfalamada iç parçalanma, segmantasyonda ise dış parçalanma olur.

SANAL BELLEK (Virtual Memory)

Sanal bellek kavramında, proseslere tüm adres uzayını kapsayacak şekilde, ana bellek yerine disk üzerinde yer ayrılır ve parçalar gerekli olduğunda ana belleğe yüklenir ve işletime alınır.

Herhangi bir zaman diliminde prosesin ana bellekte bulunan kısmına **yerleşik set** (resident set) denir.

Şayet işlemci, ana bellekte olmayan bir mantıksal adresle karşılaşırsa, bellek erişim hatası kesmesini üretecektir.

Bu yapı sayesinde;

1. Daha fazla sayıda proses bellekte tutulabilir.
2. Bir prosesin boyutunun ana belleğin sığasından daha fazla olması durumu programcı için önemsizdir. İşletim sistemi otomatik olarak prosesin gerekli parçalarını belleğe yükler.

Sayfalama ve Sanal Bellek

Bir bilgisayar sisteminde sanal bellek oluşturabilmek için sayfalama veya segmantasyon teknikleri kullanılabilmektedir.

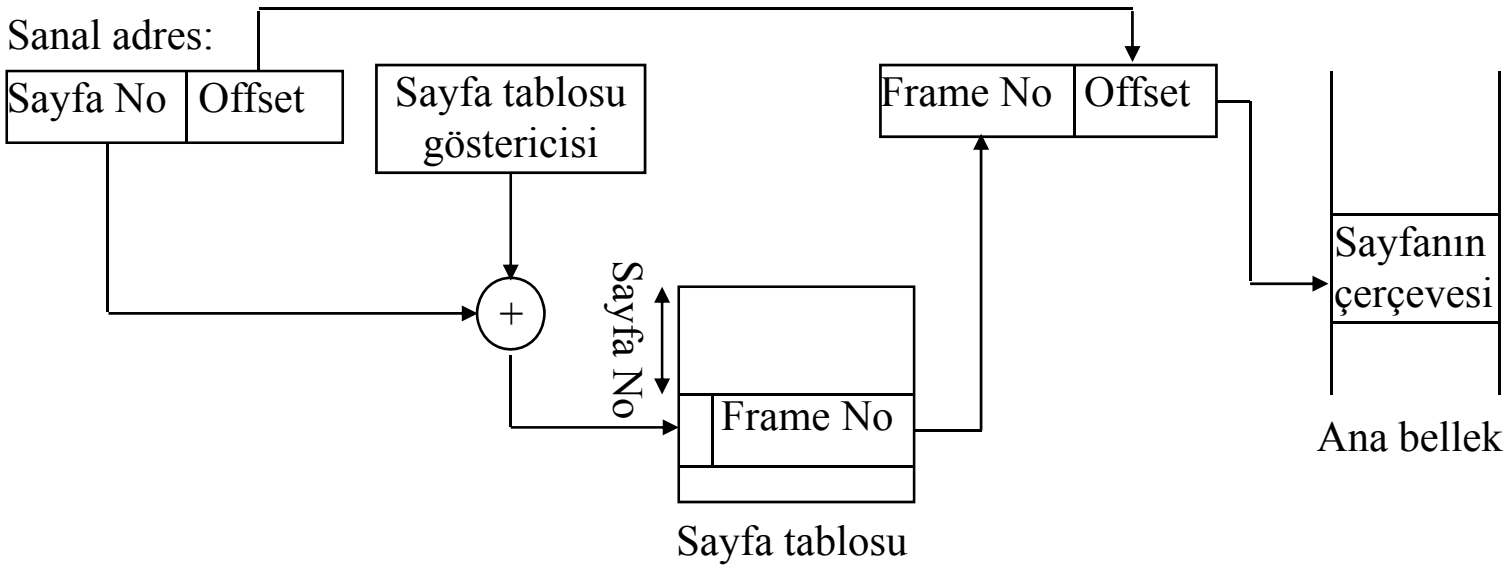
Mantıksal Adres:

Sayfa No.	Offset

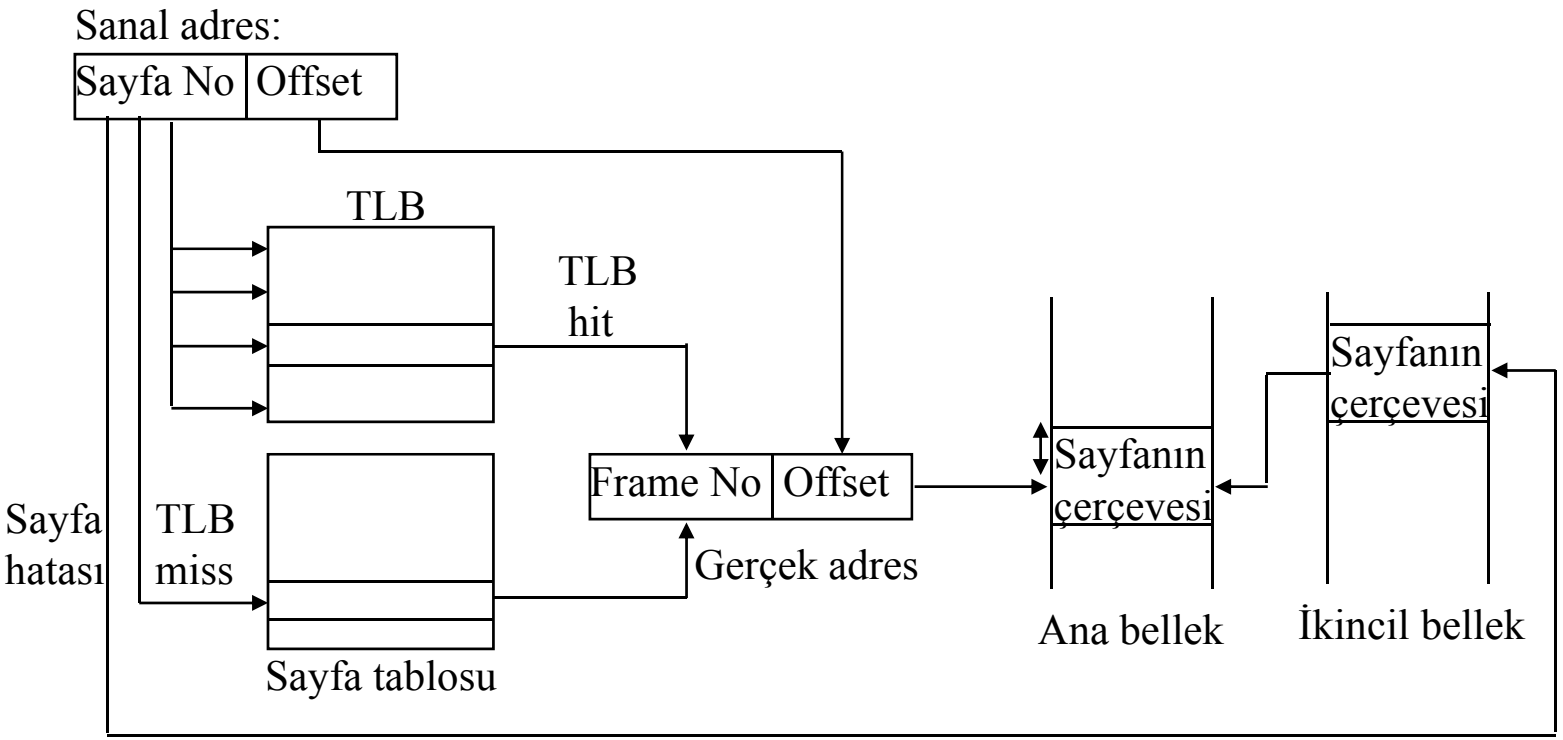
Sayfa tablosunun formatı:

M	D	Diğer bitler	Frame No

Dönüşüm için gerekli mekanizma:



Sayfalamada Translation Lookaside Buffer (TLB) Kullanımı



Sayfa Boyutunun Seçimi

Önemli bir donanım tasarım kriteridir. Göz önünde bulundurulması gereken birkaç faktör vardır, bir tanesi iç parçalanmadır. İç parçalanmalar, daha büyük boyutlu sayfalarda daha fazla olacaktır.

Küçük boyutlu sayfalar kullanılırsa, prosesler daha fazla sayfa gerektirecek, dolayısıyla sayfa tablolarının boyutu büyük olacaktır.

4 GB sanal bellek kullanımında (2^{32}), sayfa boyutunun 1 KB (2^{10}) olması durumunda,

Sistemde $2^{32}/2^{10}=2^{22}$ sayfa vardır.

2^{22} sayfanın bilgisini, 1 KB'lık sayfalarda tutmak için $2^{22}/2^{10}=2^{12}$ yani 4K sayfaya ihtiyaç vardır (Bilginin 1 Byte yer işgal ettiği düşünülmüştür).

Yani belleğin 4096 (4K) çerçevesi, sadece sayfa tablosunu tutmak için kullanılmalıdır. Sayfa boyutu 8 KB seçilirse bu sayı, 512 çerçeve olacaktır (2^{22} adet sayfayı tutmak için).

Sayfa Boyutunun Seçimi

Sayfaların ikincil bellekten ana belleğe yüklenmesi için geçen zaman;
transfer, gecikme ve arama sürelerinin toplamıdır.

Transfer süresi transfer edilecek bilginin miktarı ile orantılıdır. Ancak arama ve bekleme süreleri transfer süresini gölgede bırakır.

Örneğin transfer hızı 2 MB/sn olan bir disk için, 512 byte'lık bilginin aktarım süresi 0,25 msn'dir.

Buna karşılık gecikme ve arama sürelerinin toplamının yaklaşık 25 msn aldığı düşünülürse, transfer süresi toplam sürenin sadece %1'i kadar olacaktır. Sayfa boyutunun iki katına çıkması bu süreyi oldukça az etkileyecektir.

Oysa 512 byte'lık iki sayfanın aktarılması, toplam süreyi iki kat daha fazla etkileyecektir. Dolayısıyla daha büyük boyutlu sayfalar daha avantajlıdır.

- Eğer sayfa boyutu çok küçük olursa, daha çok sayıda sayfa hatası oluşabilecektir.

Bellek Yönetimi (Devamı) ve Disk Planlaması

- Segmantasyon ve Sanal Bellek
- Sayfalama ve Segmantasyonun Birlikte Kullanımı
- Sayfaların Belleğe Getirilme Biçimi

- Disk Planlaması (Disk Scheduling)
  - Disk Planlaması İçin Kullanılabilecek Yöntemler

Segmantasyon ve Sanal Bellek

Segmantasyonda programcı, belleğin çoklu adres uzayı (komut, veri, yığın) ya da segmentlerden oluştuğunu düşünür. Segmentler, eşit olmayan uzunluklarda ve dinamiktir. Bellek erişimleri, segment numarası ve ofset yardımıyla olur. Segment tablosu, segmentlerin başlangıç adreslerini ve boyutlarını tutar. Aynı yapı segmantasyona dayalı sanal bellekte de geçerlidir. Ancak daha karmaşık yapıdadır:

Mantıksal Adres:

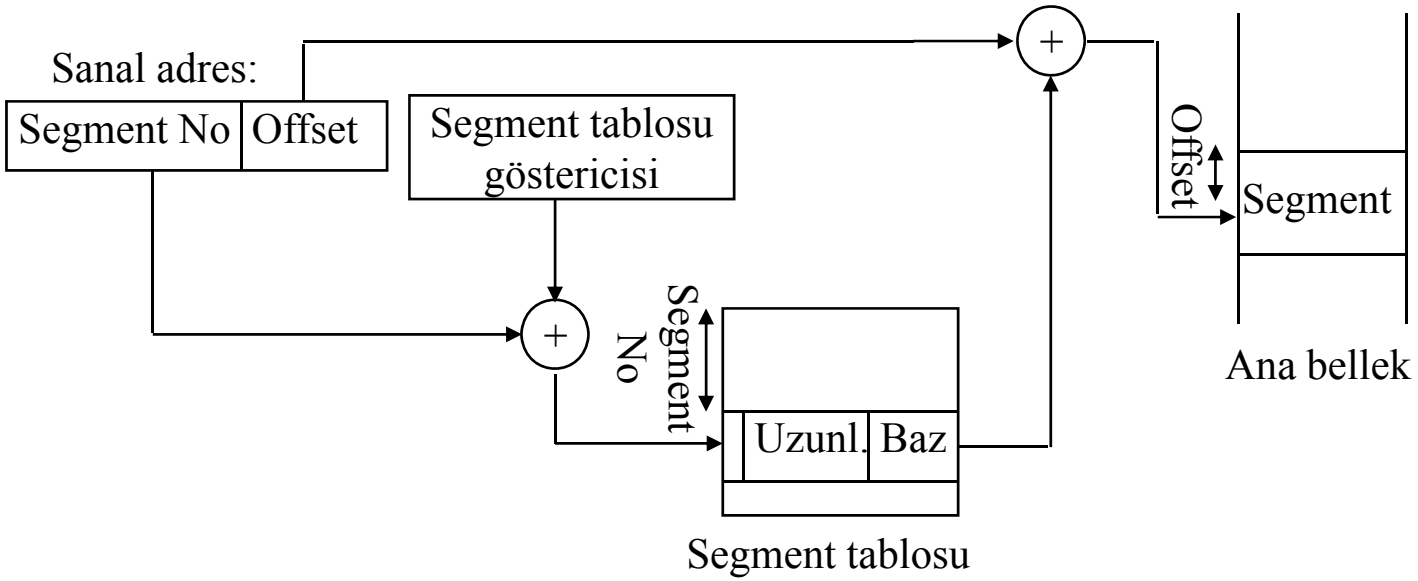
Segment No.	Offset

Segment tablosunun formatı:

M	D	Diğer bitler	Uzunluk	Segment adresi

Dönüşüm Mekanizması

Bellekten bir kelime okunmak istendiğinde, segment numarası ve ofset değerini içiren mantıksal ya da sanal adreslerden segment tablosu kullanarak fiziksel adreslere geçilmelidir. Proses çalışırken ilgili segment tablosunun başlangıç adresi bir kaydedicide tutulur. Sanal adresin segment numarası kısmı, bu tabloyu indeksler. Tablo, segmentin başlangıç adresini tutar. Bu başlangıç adresine ofset değerinin eklenmesiyle gerçek adrese erişilir.



Sayfalama ve Segmantasyonun Birlikte Kullanımı

Mantıksal adres uzayı kesimlere, kesimler de eşit uzunluktaki sayfalara ayrılır. Programcı açısından mantıksal adresler, segment numarası ve ofset değerinden, sistem açısından ise segment numarası, sayfa numarası ve ofset değerinden oluşur.

Basit segmantasyonda segment tablosu, segmentlerin uzunluklarını ve başlangıç adreslerini tutmaktaydı. Bu teknikte ise sayfa tablolarınınbaşlangıç adreslerini tutmaktadır. Segment tablosunda mevcut ve değişiklik bitine ihtiyaç yoktur (sayfa seviyesinde yapıldığından). Koruma ve paylaşım için ise diğer bitler kullanılabilir. Sayfa tabloları, içerik bakımından basit sayfalama tekniğiyle aynı yapıya sahiptir.

Mantıksal Adres:

Segment No.	Sayfa No.	Offset

Segment tablosunun formatı:

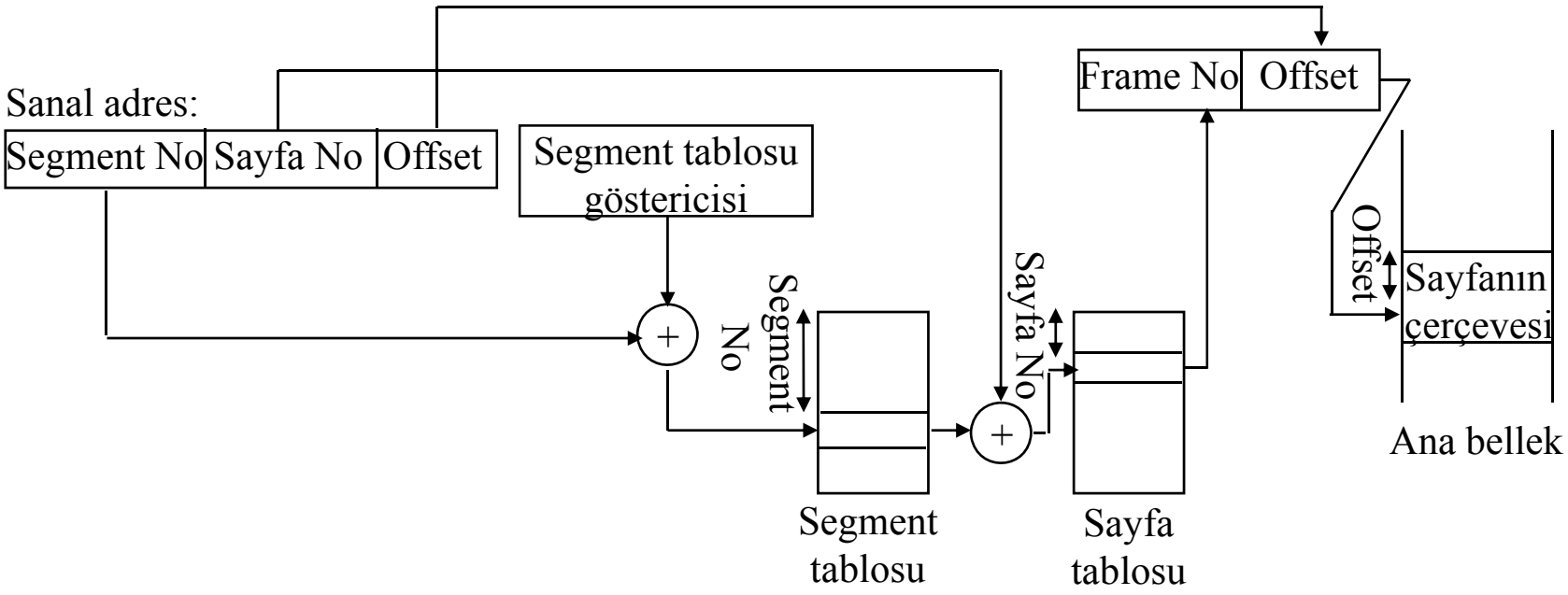
Diğer bitler	Uzunluk	Sayfa adresi

Sayfa tablosunun formatı:

M	D	Diğer bitler	Frame No

Dönüşüm Mekanizması

Her prosesle ilgili olarak bir segment tablosu, her proses segmenti için bir tane olmak üzere çok sayıda sayfa tablosu vardır. Mantıksal adres sunulduğunda işlemci, segment numarasını kullanarak proses segment tablosunu indeksler ve bu segmentin sayfa tablosu bulur. Mantıksal adresin sayfa numarası kısmı, sayfa tablosunu indeksler ve frame numarası elde edilir. Ofset değeriyle kombine edilerek gerçek adrese erişilir.



Sayfaların Belleğe Getirilme Biçimi

- **Demand paging:** Bir sayfa, referans edildiği zaman belleğe getirilir. Başlangıçta sayfa hataları oluşacak fakat zamanla bu hatalar azalacaktır.
- **Prepaging:** Talep edilmeden birden çok sayıda sayfa ana belleğe getirilir.

Proseslere Çerçeve (frame) Ataması

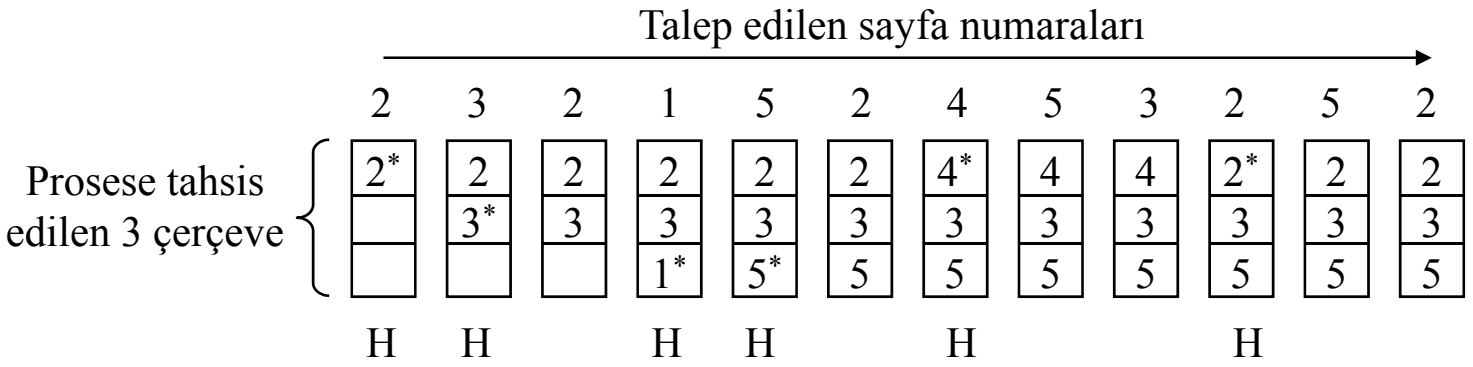
- **Statik yer atama:** Proseslere önceden belirlenen sayıda çerçeve atanır. Bu sayı uygulamanın türüne (etkileşimli, toplu işlem ya da uygulama prosesi) göre değişebilir.
- **Dinamik yer atama:** Prosesin işletimi boyunca tahsis edilen frame sayısı değişiklik gösterir. Sayfa hatalarının fazla olması durumunda ek frame tahsisi, az olması durumunda ise frame azaltılması yapılır.

Statik Yer Atamada Sayfa Çıkarma Algoritmaları

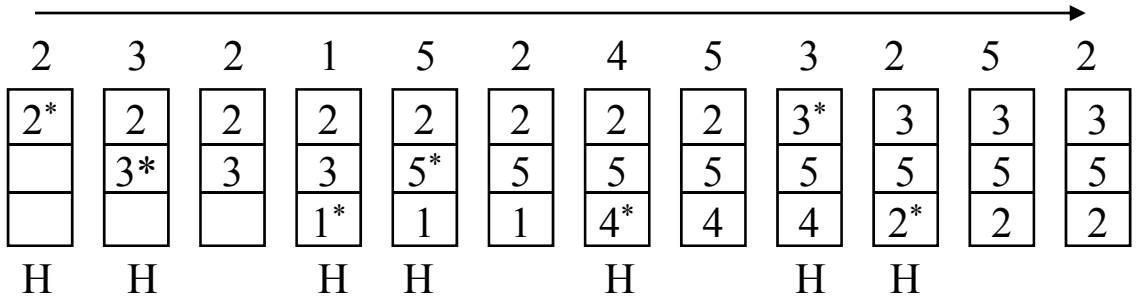
Temel Algoritmalar:

1. Optimal: Bu algoritma, en geç erişilecek sayfayı seçer. İşletim sisteminin gelecek olaylar hakkında kesin bilgi sahibi olmasını gerektirir ki uygulanabilirliği yoktur. Bu algoritma en az sayıda sayfa hatası üreteceğinden, diğer algoritmalarla kıyas için kullanılabilir.

Örnek: 5 sayfadan oluşan bir prosese ana bellekte 3 frame atansın. Erişilecek adresler, sırasıyla 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2 sayfalarında olsun. Optimal çıkarma algoritmasına göre hangi frame'lere hangi sayfaların atandığını grafiksel olarak gösterelim:

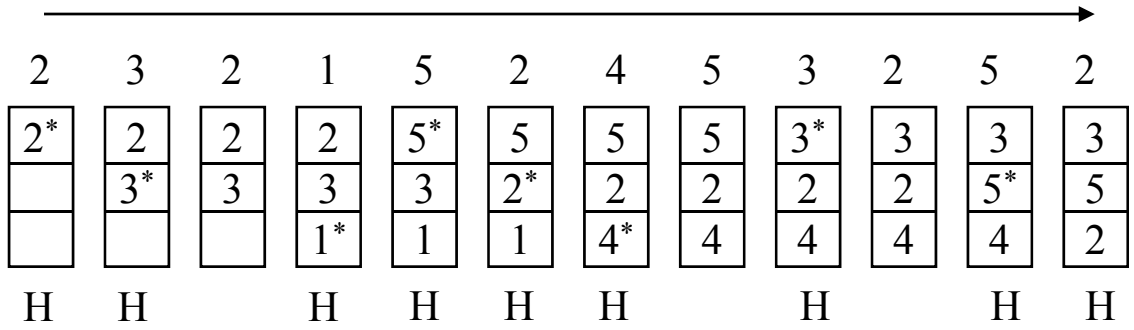


2. Least Recently Used (LRU)-En erken erişilmiş sayfayı çıkarma: En uzun zamandır erişim olmamış olan sayfanın seçilmesi mantığına dayanır. Yerellik prensibine göre yakın zamanda da bu sayfaya erişim olmayacaktır. Performansı, optimal algoritmaya en yakın olanıdır. Her sayfada, son erişim zamanı bilgisi tutulur. Sisteme ek bir yük getirir.



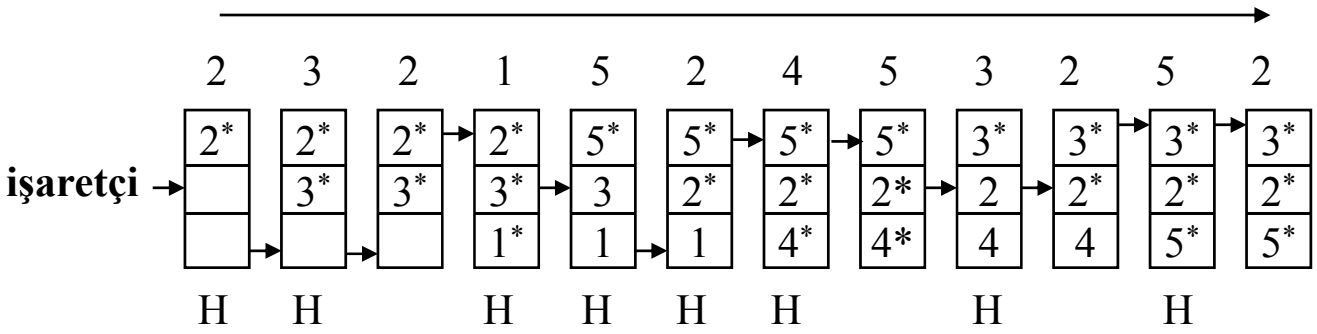
Temel Algoritmalar

3. FIFO-İlk giren ilk çıkar: Gerçeklenmesi en kolay algoritmadır. İşe atanmış frame'ler döngüsel bir kuyruk olarak ele alınır. Sayfalar sıralı olarak bellekten atılır. Uzun süre önce belleğe alınan sayfa artık kullanımda değildir. Ancak bu yapılan çıkartım her zaman doğru olmaz. Programın belli bölgeleri sürekli olarak kullanımda olabilir.



Temel Algoritmalar

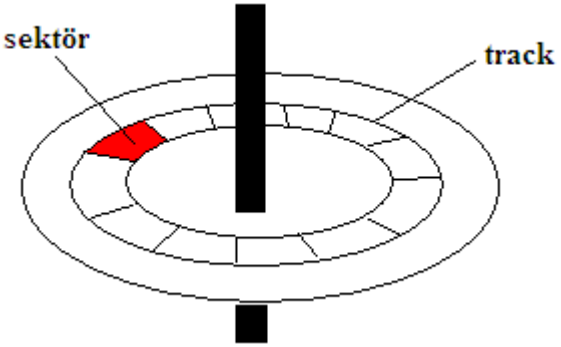
4. Clock-Saat: Her frame için, kullanım biti adında ek bir bit alanı kullanılır. Sayfa belleğe yüklendiğinde kullanım bitine 1 yüklenir. Ayrıca, sayfaya erişimde de kullanım biti 1 yapılır. Dairesel bir kuyruk yapısı vardır. Bu kuyrukla ilgili işaretçi, en son belleğe alınan sayfanın bir sonrasını gösterir. Bellekten atılacak sayfa belirlenirken, işaretçinin pozisyonuna bağlı olarak, kullanım biti 0 olan ilk sayfa seçilir. Kullanım biti 0 olan yoksa tüm kullanım bitleri sıfırlanır ve göstericinin pozisyonuna göre ilk sayfa çıkartılır.



*: kullanım bitinin 1 olduğunu gösteriri.
Ok, işaretçinin bulunduğu konumdur.
H: Sayfa hatası olduğunu gösterir.

DİSK PLANLAMASI (Disk Scheduling)

İşletim sisteminin sorumluluklarından birisi de donanımın verimli kullanılmasını temin etmektir. Disk için esas olan, hızlı erişim ve bant genişliğidir.



Erişim zamanı iki bileşenden oluşur:

- **Arama zamanı (seek time):** Disk kolunun dolayısıyla kafanın istenen sektörü içeren silindir üzerine hareket ettirmek için geçen süredir.

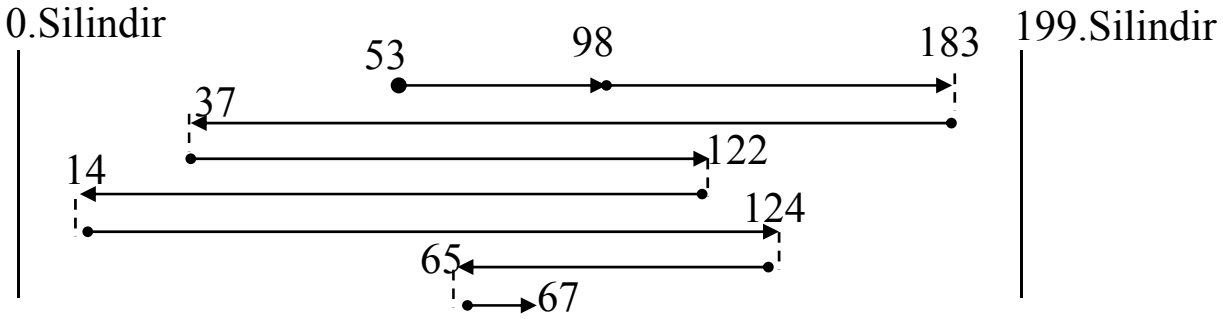
- **Döngüsel gecikme (latency time):** İlgili sektörün kafanın altından geçmesinin beklendiği süredir.

Bant genişliği: Transfer edilen toplam byte'ın, verinin talep edilmesiyle transferin tamamlandığı ana kadar geçen süreye bölünmesiyle bulunur.

Disk Planlaması İçin Kullanılabilecek Yöntemler

1. FCFS (first-come, first-served) planlaması: Disk planlamasının en basit şeklidir. Adil bir planlama gibi gözükse de en hızlı servisi sağlamaz.

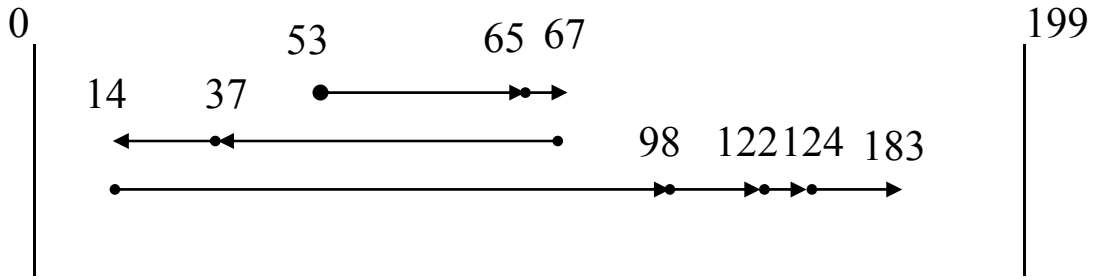
Örneğin gerekli blokların yer aldığı silindirler disk kuyruğunda şu sırada olsun: **98, 183, 37, 122, 14, 124, 65, 67**. Diskin de toplam 200 silindire sahip olduğunu düşünelim. Diskin kafasının başlangıçta 53 nolu silindirde olması durumunda kafanın ilk hareketi 98 nolu silindire daha sonra da sırasıyla 183, 37, 122, 14, 124, 65, 67 nolu silindirlere doğru olacaktır. Toplam kafa hareketi hesaplandığında 640 silindirdir.



Disk Planlaması İçin Kullanılabilecek Yöntemler

2. SSTF (shortest-seek-time-first) planlaması: Arama süresi, kafanın üzerinden geçtiği silindir sayısı arttıkça artacağından bu algoritma, kafanın mevcut pozisyonuna en yakın talebin yanıtlanması esasına dayanır.

Talep edilen silindirlerin sırası: 98, 183, 37, 122, 14, 124, 65, 67



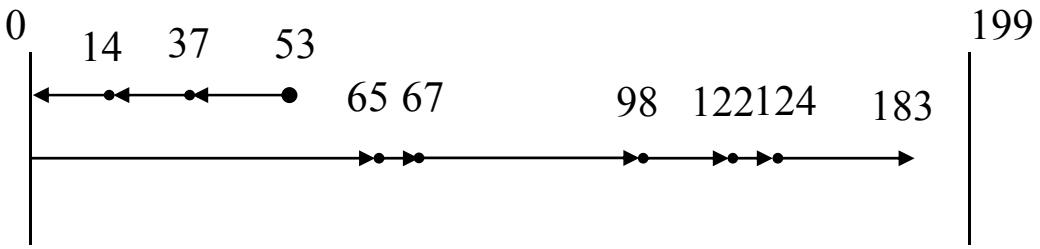
Toplam kafa hareketi 236 silindirdir. Bu algoritma, FCFS'ye göre önemli bir iyileştirme sağlar. Bu algoritma SJF (shortest job first) planlamasının bir formudur ve SJF gibi açlık (starvation) problemine sebebiyet verebilir.

Disk Planlaması İçin Kullanılabilecek Yöntemler

3. Scan planlaması: Disk kolu belli bir yönde harekete başlar ve bu yol üzerindeki tüm talepler sırasıyla karşılanır. Diskin sonuna ulaşıldığında da disk kolu yön değiştirir ve bu yöndeki talepler sırasıyla karşılanır. Kafa sürekli disk üzerinde ileri ve geri yönde hareketine devam eder.

Örneğimize geçmeden önce, 53 nolu silindirde bulunan kafanın hareket yönünü bilmemiz gerekir. Şayet disk kolu 0 nolu silindire doğru gidiyorsa, ilk olarak yol üzerindeki 37 ve 14 nolu silindirlere servis yapılacak, 0 nolu silindirde disk kolu yön değiştirecek ve sırasıyla 65, 67, 98, 122, 124, 183 nolu silindirlere servis verilecektir.

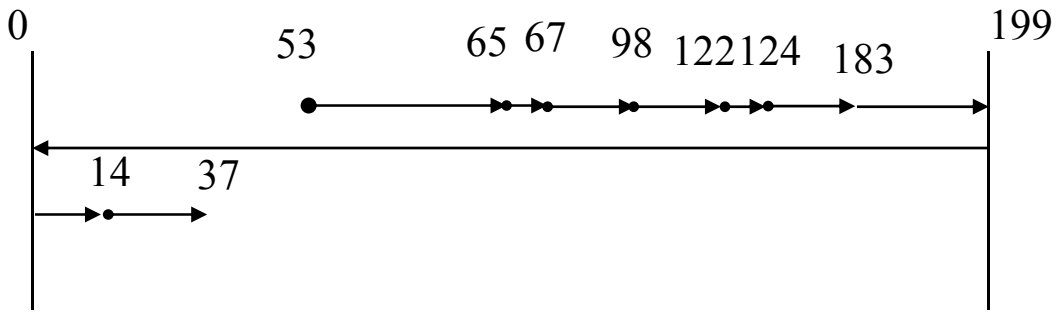
Talep edilen silindirlerin sırası: 98, 183, 37, 122, 14, 124, 65, 67



Disk Planlaması İçin Kullanılabilecek Yöntemler

4. C-Scan (circular scan) planlaması: Scan planlamasının bir varyantıdır. Kuyruktaki bekleme sürelerini daha makul düzeye getirmeye çalışır. Scan planlamasıyla arasındaki fark; kafa, yol üzerindeki son silindire geldiğinde hızlı bir şekilde yoldaki taleplere cevap vermeden ilk silindire konumlanmasıdır. Aynı örnek için disk kolunun hareketi aşağıdaki gibidir.

Talep edilen silindirlerin sırası: 98, 183, 37, 122, 14, 124, 65, 67

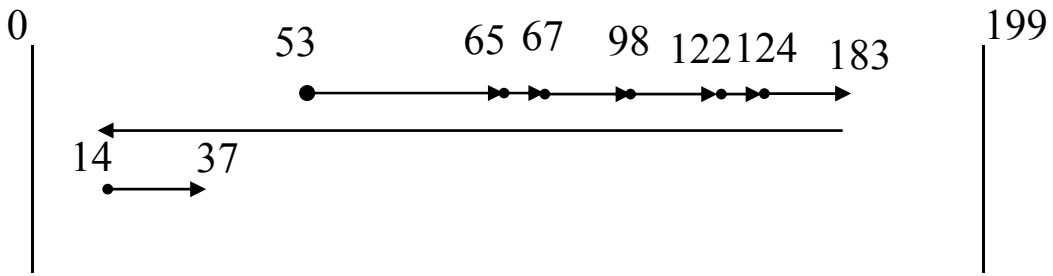


Disk Planlaması İçin Kullanılabilecek Yöntemler

5. Look planlaması: Scan ve c-scan planlamaları, disk kolunu tüm disk boyunca hareket ettirir. Pratikte bu algoritmalar bu şekilde tasarlanmazlar. Disk kolu yolu üzerindeki son talep ne ise oraya kadar gider ve derhal yön değiştirir fakat diskin en başına gitmez, bu yol üzerindeki ilk talebe konumlanır.

Scan ve c-scan planlamalarının bu versiyonu,
look ve c-look olarak anılır.

Talep edilen silindirlerin sırası: 98, 183, 37, 122, 14, 124, 65, 67



c-look planlaması

Algoritma Seçimi

Performansın, büyük ölçüde talep tipine ve sayısına bağlı olduğu unutulmamalıdır.

Yoğun disk işlemlerinin yapıldığı bir sistemde, scan ve c-scan planlamaları, SSTF ve FCFS planlamalarına göre daha az açlık problemine neden olacaktır.

Bahsi geçen planlama şekillerinde dikkate alınan ölçüt diskin arama zamanıdır. Oysa döngüsel gecikme, ortalama arama süresine yakın bir değerdir. İşletim sistemi açısından bu süreyi de hesaba katmak zordur. Son zamanlarda disk üreticileri disk planlama algoritmalarını kontrolörün içine gömmektedirler. İşletim sistemi, disk kontrolörüne bir talepler kümesi gönderdiğinde, kontrolör bu talepleri kuyruklayabilecek ve hem arama hem de döngüsel gecikme sürelerini optimize edecek şekilde planlama yapabilecek kabiliyete sahiptir. Pratikte ise, işletim sisteminin bazı talepleri öncelikli olarak ele alması gerekebileceğinden, algoritma seçimini genellikle işletim sisteminin kendisi yapar.

DOSYA SİSTEMLERİ

Dosya Tipleri

Dosyalara Erişim

Dosya İşlemleri

Dosya Özellikleri

Dizinler

Yol (Path) Kavramı

Dosyalara Diskte Yer Atama Yöntemleri

Dosya Sistemi Niçin Gereklidir?

1. Çok miktardaki bilginin yüklenme gereksinimi,

2. Proses sonlandığında bilginin kaybolmaması,

3. Çok sayıda prosesin eş zamanlı olarak erişebilmesi.

Tüm bu gereksinimleri karşılayacak çözüm de bilgiyi disk veya diğer medya aygıtlarında dosya olarak saklanmasıdır. Dosyalardaki bilgiler proseslerin oluşturulması ya da sonlandırılmasından etkilenmemelidir. Sadece dosyanın sahibi olan kullanıcının isteği doğrultusunda sistemden kaldırılabilmelidir.

Dosyaların İsimlendirmesi ve Uzantı verilmesi

Dosya isimlendirmesindeki kurallar sistemden sisteme değişiklik göstermekle birlikte, tüm işletim sistemleri en azından, 8 karakterden oluşan harf ve özel karakterlerin kullanımına izin verir. Bazı işletim sistemlerinde dosya isimlendirmesinde büyük ve küçük harf duyarlılığı vardır, bazısında yoktur (Unix duyarlı, DOS duyarsızdır). Ayrıca bazı işletim sistemleri dosya isimlerinin iki kısımdan oluşmasını destekler; uzantı olarak bilinir ve nokta karakteriyle ayırt edilir.

Bazı dosya uzantıları şunlardır;

.c veya .cc: C kaynak kodu

.o: Obje dosyası (derleyici çıktısı)

.txt: Genel text dosyası

.hlp: Yardım dosyası

.zip: Sıkıştırılmış arşiv dosyası

.gif: resim dosyası (Graphical Interchange Format)

.ps:PostScript dosya

.html: html dosyası (Hyper Text Markup Language)

Dosya yapılandırması

Üç şekilde olabilir:

- Dosya, Yapılandırılmamış byte dizisidir.

- Dosya, sabit uzunluktaki kayıtlardan (record) oluşan bir dizidir.

- Dosya, kayıtlardan oluşan bir ağaç içerir. Kayıtlar eşit uzunlukta olmak zorunda değildir ve kayıtların her biri kayıtın belli bir yerinde anahtar bir alana sahiptir. Ağaç, bu anahtar alana göre sıralanabilir ve hızlı bir şekilde arama yapılabilir.

Dosya Tipleri

İşletim sistemleri birkaç dosya tipini desteklerler. Örneğin Unix ve DOS'ta sıradan dosyalar ve dizinler vardır. Unix ayrıca karakter tabanlı ve blok tabanlı özel dosyalara da sahiptir.

Sıradan dosyalar genellikle ya ASCII ya da ikili (binary) dosyalardır. ASCII dosyalar yazdırılabilir, bir text editör programı tarafından görüntülenebilir yapıda text satırlardan oluşur. Her bir satır bir enter ve satır besleme karakterleriyle sonlandırılır.

Linux'ta ikili dosya formatı;

Dosyalara Erişim

Dosyalara erişim, **Sıralı** ya da **Rastgele** olabilmektedir.

İlk işletim sistemleri sadece sıralı dosya erişimlerine izin vermiştir. Böyle bir sistemde bir proses, dosyadaki byte'ları ya da kayıtları sırayla okumak durumundadır. Bu tip erişim diskten ziyade manyetik teypler için uygundur.

Disk teknolojisi ile birlikte dosyalara erişim (byte, kayıt ya da anahtar alana göre) rastgele yapılabilir hale gelmiştir. Dosyadan okuma yapılacak yer belirtilir ve bilgi okunur. Eskiden dosyalara erişimin sıralı mı yoksa rastgele mi olacağı dosyanın oluşturulması aşamasında belirtiliyordu. Günümüz işletim sistemlerinde dosyalar otomatik olarak rastgeledir.

Dosya Özellikleri

Her dosyanın bir ismi ve datası vardır. Ayrıca işletim sistemi, her dosyayla ilgili olarak örneğin dosyanın oluşturulma zamanını ve dosya boyutu gibi bilgiler de içerir. Dosya özellikleri sistemden sisteme değişiklik gösterir.

Bu özellikler şunlar olabilir;

- Dosyaya kimlerin erişip erişemeyeceği ile ilgili özellikler (koruma, şifre, oluşturanın bilgileri).
- Dosyanın ne tip bir dosya olduğu ve erişim biçimi ile ilgili bayraklar (Read-only flag, hidden flag, system flag, ascii/binary flag, random acces flag, temporary flag, archive flag gibi)
- Dosyanın oluşturulma zamanı, en son erişim zamanı, mevcut boyutu, olabilecek maksimum boyutu (günümüz PC'lerinde bu özellik başka şekillerde çözülmüştür)
- Kayıt (record) uzunluğu, anahtar alan pozisyonu ve uzunluğu gibi özelikler içerebilir.

Dosya İşlemleri

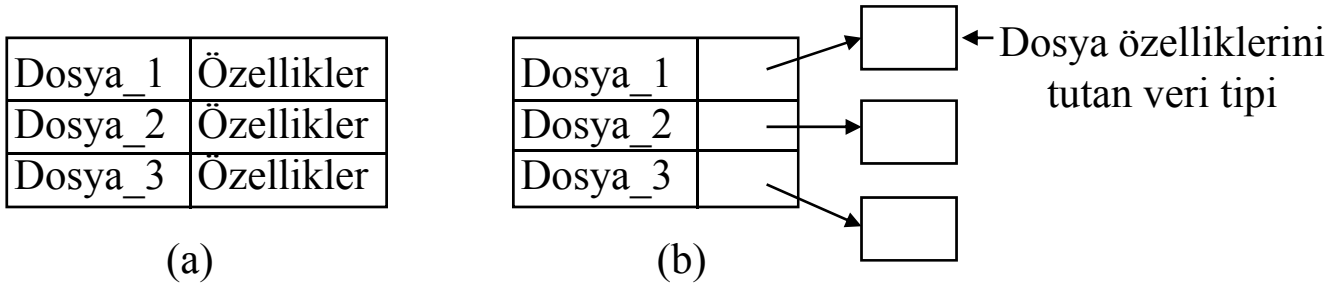
Dosyalarla ilgili olarak en yaygın sistem çağrıları;

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attributes
10. Set attributes
11. Rename

DİZİNLER (Directories)

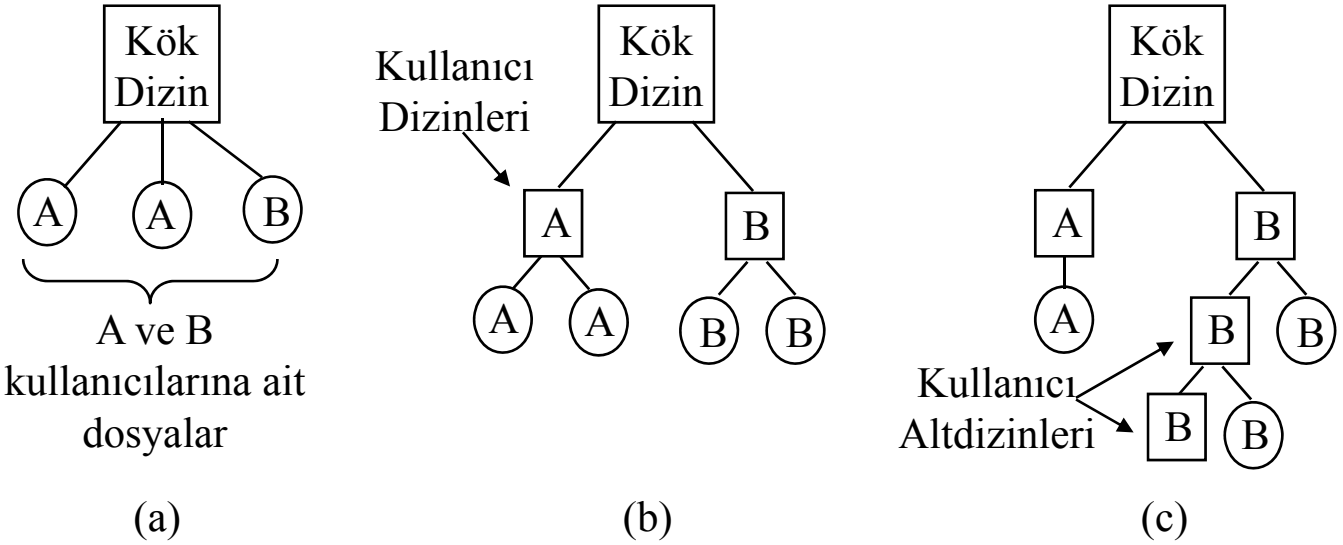
Dosyaların izlerini tutmak için dosya sistemleri dizinlere sahiptir. Birçok sistemde dizinlerin kendileri de birer dosyadır.

Bir dizin, sahip olduğu her dosya için bir kayda sahiptir. Bu kayıtlar iki farklı şekilde düzenlenebilir:



Bir dosya açılacağı zaman işletim sistemi, açılacak dosyanın ismini buluncaya kadar dizinini araştırır. Dosya özelliklerini ve disk adreslerini ya direkt olarak dizin kaydından ya da işaret edilen veri tipinden alarak ana bellekte bir tabloya koyar.

Dizin Hiyerarşisinin Gelişimi



Yol (Path) İsimlendirmesi

Dosya sistemi dizin ağacı şeklinde olduğundan dosya isimlerini belirtmek için bir yöntem gereklidir. 2 metot kullanılır;

İlki **mutlak yol ismidir (absolute path name):** Kök dizinden başlayıp dosyayı da içeren yoldur.

Örneğin, */dizin1/dizin2/dokument* yolu, kök dizinin *dizin1* isminde bir altdizine, bu altdizinin */dizin2* altdizinine ve onun da *dokument* isimli dosyaya sahip olduğunu gösterir.

Unix'te adresteki ifadeler / sembolü ile, DOS'ta ise \ sembolü ile birbirinden ayrılır. Kök dizininden başlayan mutlak yol isimleri sistemde tektir.

Yol (Path) İsimlendirmesi

Diğer isimlendirme şekli de **bağıl yol ismidir (relative path name):** Çalışma dizini ile bütünleşik bir yapısı vardır. Kullanıcı, bir dizini çalışma dizini alarak tasarlar.

Şayet */kullanici* dizini çalışma dizinimiz ise,

bu dizin altındaki *dosya_1* isimli dosyanın bağıl adresi *dosya_1* ile belirtilir. Mutlak adresi ise */kullanici/dosya_1* olur.

Ayrıca hiyerarşik dosya sistemini destekleyen işletim sistemleri, her dizin için '.' ve '..' ile belirtilen mevcut dizini ve bir üst dizini gösteren özel kayıtlar içerir.

Örneğin, çalışma dizinimiz */kullanici/dizin1* olsun.

Şayet */kullanici/dizin2/dosya_1* dosyasını çalışma dizinimize kopyalamak istiyorsak,

Çalışma dizini: */kullanici/dizin1*

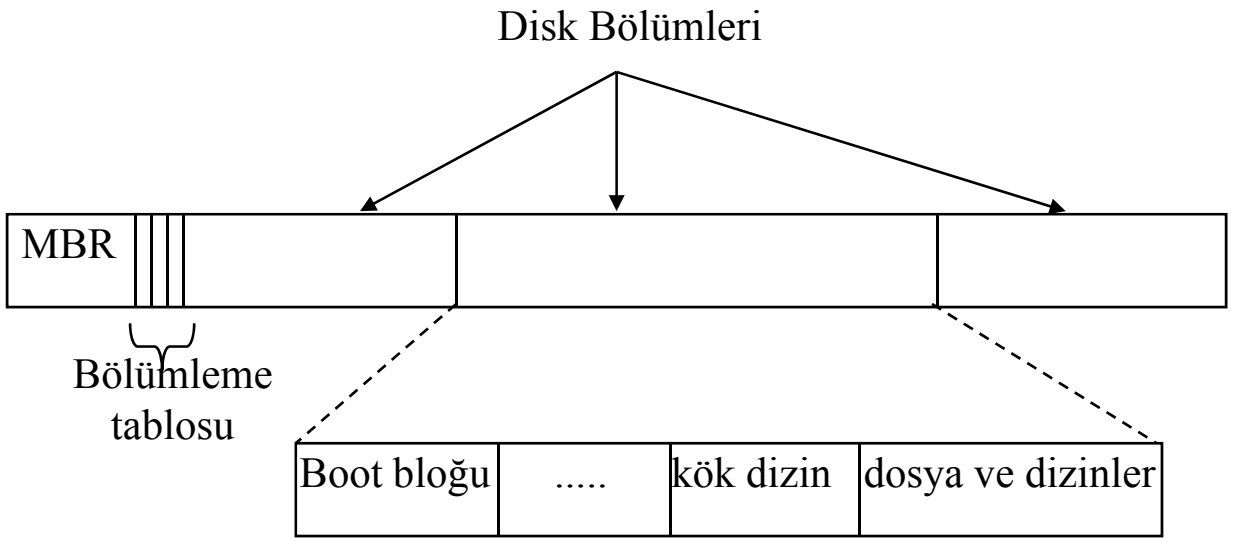
Unix'te *cp ../dizin2/dosya_1 .* komutunu kullanabiliriz.

Çalışma dizininin bir üst dizini: */kullanici*

Dosya Sistemi Tasarımı

Dosya sistemleri disk üzerine yüklenir. Diskler bir ya da daha çok kısma (partition) bölünebilir. Her bir kısım ayrı bir dosya sistemine sahiptir. Diskin sıfırıncı sektörü MBR (Master Boot Record) olarak bilinir ve sistemi boot etmek için kullanılır.

Bilgisayar boot edildiğinde BIOS, MBR'yi okur ve işletir. MBR programının yaptığı ilk şey aktif bölümü tespit edip, bu bölümün ilk bloğunu (boot block) okumak ve işletime almaktır. Boot bloğundaki program da işletim sistemini yükler.



Dosyalara Diskte Yer Atama Yöntemleri

Bitişik (Contiguous) Yer Atama: En basit yöntem, dosyaları bitişik disk bloklarına yerleştirmektir. Bloklar genellikle birkaç sektörlük bilgiyi içerir ve bir defada sürücüden belleğe aktarılırlar.

Blok boyutu 1KB olan bir diskte (2 sektör, 1 disk bloğu düşünülmüştür), 50KB'lık bir dosyayı tutmak için 50 bitişik bloğu kullanmak gerekecektir.

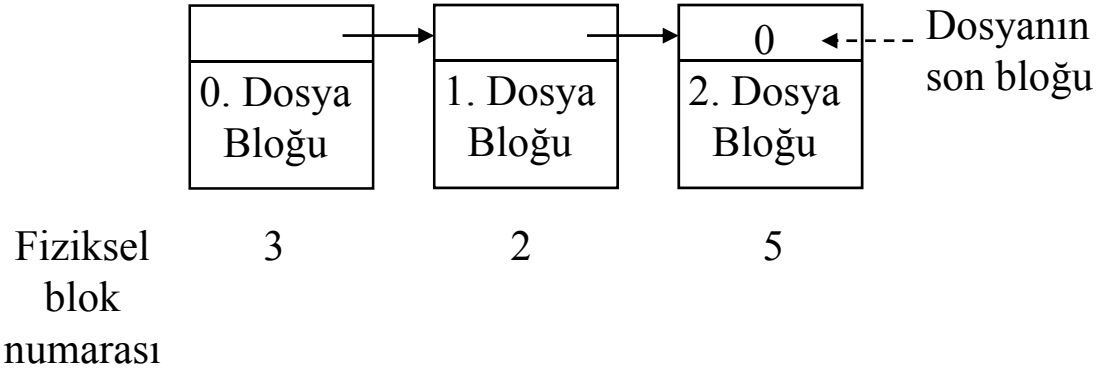
İki avantajı vardır; dosyanın bloklarından ilkini (ayrıca dosyanın kaç blok içerdiği de bilinmelidir) bilmek yeterli olacaktır. Diğer bir avantajı da performansın çok iyi olmasıdır. Basit bir işlemle tüm dosyayı diskten okumak mümkün olacaktır.

İki dezavantajı vardır; dosyanın oluşturulacağı zamanda dosyanın maksimum boyutu bilinmezse uygulanamaz. Yani bu bilgi olmaksızın işletim sistemi ne kadar disk alanı rezerve etmesi gerektiğini bilemez. Diğer dezavantajı da parçalanmadır. Boş alanlar ya kullanılacak ya da atıl kalacaktır. Birleştirme işlemi yapılabilir ancak zaman alıcı bir prosestir.

Dosyalara Diskte Yer Atama Yöntemleri

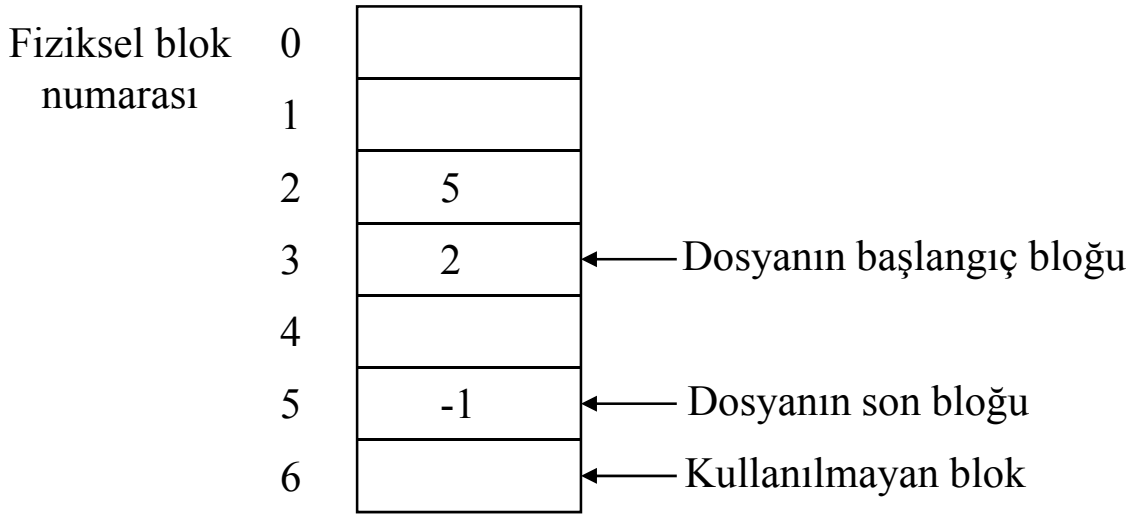
Bağlı Liste (linked list) Yapısıyla Yer Atama: Dosyaların disk bloklarında bağlı liste biçiminde tutulmasıdır. Her bir bloğun ilk verisi diğer bloğu göstermek için kullanılır, geri kalanı dosya verisinin kendisidir. Bitişik yöntemin aksine her disk bloğunun kullanımına imkan tanır. Disk parçalanması söz konusu değildir. Dosyanın ilk bloğunun disk adresi yeterlidir.

Dezavantajları; dosyaya sıradan erişim hızlıdır ancak rasgele erişim oldukça yavaştır.



Dosyalara Diskte Yer Atama Yöntemleri

Index Kullanarak Bağlı Liste Şeklinde Yer Atama: Göstericilerin, disk bloklarından alınıp bir tabloya konulması ve indekslenmesi suretiyle olur. Bu yöntem sayesinde bir disk bloğunun tamamı veriye tahsis edilmiş olur. Ayrıca dosyalara ait disk blokları hafızada tutulduğundan rasgele erişim daha hızlı ve kolaydır (yine zinciri takip etmek gereklidir ama diske herhangi bir referans yapılmaz).



Dosyalara Diskte Yer Atama Yöntemleri

I-nodes: Hangi disk bloklarının hangi dosyayla ilişkili olduğu bilgisi i-node (index node) olarak bilinen küçük bir tabloda tutulur. Her bir dosyanın birer i-node tablosu mevcuttur. Dosyanın ilk birkaç disk adresi i-node'un kendisinde tutulur ve dosya açıldığı zaman diskten hafızaya alınır.