

## İşletim Sistemleri

\* Elimizdeki donanımsal kaynakları kullanıcının hizmetine sunan kaynak işletim sistemidir. Amacı makine ile kullanıcı arasında arayüz oluşturmaktır.

\* Bir işletim sistemi kullanıcıya kolaylıkla program geliştirme sağlayan bir yazılımdır.

Donanım → bilgisayarın kendisidir.

Bir bilgisayar 2 kaynak oluşturur.

- Yazılımsal kaynak
- Donanımsal kaynak

Donanımsal kaynakları anakart bir araya getirir.

BIOS → Bir makinedeki kullanılan işletim sisteminin başlatılmasını sağlar.

CPU → Bilgisayar üzerindeki aritmetik ve lojik işlemleri gerçekleştiren birimdir.

RAM → Rastgele erişimli bellek entegre devrelerden (çiplerden) oluşur.

İşlemcide cache bellek var ancak küçüktür. Bu nedenle RAM'e ihtiyaç duyulur.

Cache neden küçük?

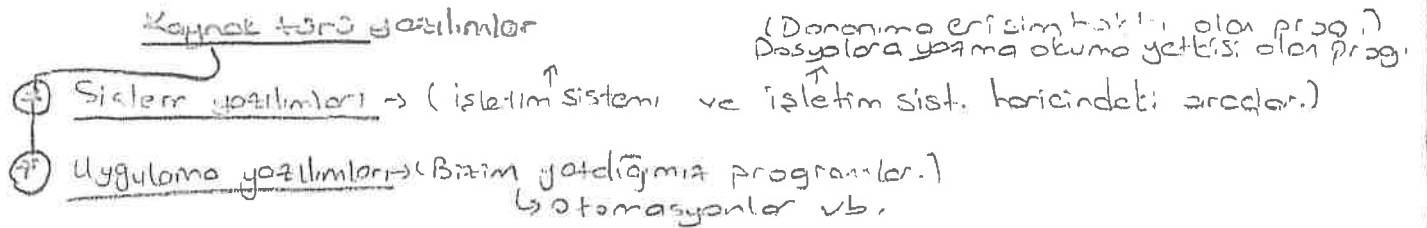
→ Büyük olsaydı adresleme gerekirdi ve işlem yavaşlardı. Bunden dolayı işlemler için cache hız açısından kullanılır.

CPU'nun ihtiyacı duyduğu hafızayı ilk olarak RAM'ler sağlar.

RAM'de sıralı erişim istenmez boş bulunan yer kullanılır. RAM'deki boş yer takibini işletim sistemi yapar.

\* İlk olarak işletim sistemi hafıza yönetiminde karşımıza çıkar.

\* Floppydisklerin üzerinde karmaşık kodlar bulunduran yazılımlar vardır.



# Donanım ile kullanıcı arasında bir arayüzdür.

# Hafıza yönetiminde karşımıza çıkar.

# Kullanıcıya kolaylıkla program geliştirme imkanı sunar.

\* İşletim Sistemi 2 kısımdan oluşur.

① Çekirdek (Kernel) → Bilgisayara ve donanıma direkt erişilebilen tek yazılım.

② Shell (Kabuk) → İşletim sisteminin çekirdeğe erişim, yetkisi olan kabuk kısmı

\* Unix açık kaynaklıdır. Yazılımlarımızı üzerinde her türlü gerçekleştirebiliriz.

Kabuk vasıtasıyla erişilebilirler.

\* İşletim Sistemi bilgisayar performansını direkt etkiler.

Performans için dikkat edilecek özellikler var.

① Tek kullanıcı işletim sisteminde çok yüksek performans istenir.

② İşletim sistemi donanımsal kaynakların kullanılabilirliğini sağlar

ya işletim/sistem soyutlama katmanı oluşturarak bu işi yapar.  
(Cihazın marka modeline bakmadan haberleşmeyi sağlar)

\* İşletim sistemi hem çekirdek hemde kullanıcı modunda çalışır.  
Diğer programlar kullanıcı modunda çalışır temel fark budur.

\* Programlar kernelle haberleşerek kullanılır. Shell (Kabuk) doğrudan kernell (çekirdek)'le haberleşebilir.

Shell + Kernel = İşletim Sistemi

Eniac vakum tüplerle yapılmış ilk bilgisayardır. (Onlu sayı sisteminde işlem yapıyordu)

## İşletim Sistemleri Servisleri

\* Arayüzlere sahiptir

- 1- Command Line Interface (MS-Dos, Shell progr. vs.)
- 2- GUI (Windows tarafında kullanılır.)

\* File Exception

İşletim sistemi aranan dosya olmadığı zaman hata vermelidir.

\* İletişim

İşlemler arasında iletişimi sağlar. (Process'ler arası iletişim, hafıza yönetimi, mesajlaşma yöntemi vs.)

\* Giriş çıkış (I/O) İşlemlerini yürütür.

ii Hata tespiti yapar. (İşletim sistemi ya programı sendendir ya da kullanıcıya seror.)

→ İşletim sistemi ; kendi çalışmasının düzenli olması için ;

\* Kaynakların tahsis : Bir server'ı, birden fazla kişi kullanıyorsa her kişiye hafıza vs. tahsis işletim sistemi tarafından yapılmalıdır.

Windows'ta birçok process aynı anda çalışabilir. Bunlar arasında hız, hafıza, CPU'da paylaştırma işlemi doğru yapılmalıdır.

\* Accounting (muhasebe) : Hangi process ne kadar CPU, hafıza vs. harcar?

\* Koruma ve güvenlik (Protection and Security) : Sistem kaynakları kullanılırken, sadece yettiği olan programların kullanılabilmesidir.

Önemli :

→ Donanımsal kaynaklar sadece işletim sistemi tarafından kullanılır.

→ Arayüzleri kullanabilmek için sistem yapıları yapılır.

Command-Line (Komut satırı) = Command Interpreter

① Microsoft yaklaşımı : Komutlar bir yerde tutulur.

② Unix yaklaşımı : Komut satırları ayrı ayrı dosyalarda tutulur.

→ Daha avantajlı. Çünkü kendi programını yazıp kullanabileceği ortam sunar. Alt program şeklinde ayrı bir dosyada tutulabilir. Microsoftta bu yok.

Sistem çağırısı kullanıcı programları ile işletim sistemleri arasındaki arayüz. İşletim sistemleri tarafından desteklenen genişletilmiş komut setiyle sağlanır. Bu komut setine sistem çağırılar denir.

Sistem çağırılarını Direkt erişemeyiz, uygulama programlarını arayüzle erişebiliriz.

Win32 API : windows için geçerli

POSIX API : Unix ve türevleri için geçerli

Java API : Platformdan bağımsız çalışır.

→ Neden API'ye gerek duyarız, sistem çağırılara direkt ulaşamıyoruz?

\* Windows7 ve windows vista'da program yazdığımızı fark ederiz. Bu programların sistem çağırılarını farklıdır, farklı API'leri vardır. API'nin aynı olması taşınabilirliği sağlar.

\* Sistem çağırılarını karmaktır. API bize daha sade, düzenlenmiş bilgi verir. Bu yüzden API kullanırız.

\* Sistem çağırılarında hafızaların yerlerini de belirtmemiz gerekir. Bu da assembly seviyesinde kod demektir. Bu yüzden karmaktır.

\* Sistem çağırılarının sınırlı sayıdadır.

Programlama dilinin derleyicisi tüm bu karmaşık sistem çağırılarını bize göstermez. Sistem çağırılarını kim gösterir derleyici.

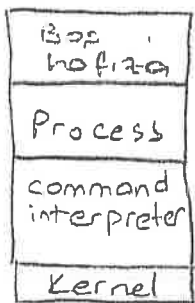
\* Sistem çağırılarının parametrelerinin geçişleri 3 yolla yapılır.

1- Register'larla

2- Hafızada bir yere yazarak (Linux ve Solaris bunu kullanır.)

3- Stack hafızasını kullanarak

MS-DOS çalışması



İşletim sistemi başlarken kernel yüklenir.

Kernel komut yorumlayıcısına hafızada yer ayırıp yükler

Komut bir iş yapıyorsa, process'le hafızada yer ayırılır.

Definot : API : Application Programming Interface = Yazılım geliştiricilere sağlanan bir uygulama geliştirme arayüzüdür.

Linux → Sistem çağırılarını, windows → API'lerle sağlanır.

## Processes (görev)

\* Process, bir programın aktif olarak çalışan kısmı demektir.

\* Bir simgeye çift tıkladığımızda; çalıştırılabilir dosyanın kodlarını ana hafızaya yükler. Yüklerken kendisine gerekli yerler için kaynak tahsis ediyor. En son CPU kendisine gelenleri işler.

\* Temel amaç;

Tüm programı hafızaya yüklemek yerine işimize yarayan yerleri hafızaya eklersek başka programlarda çalıştırabiliriz aynı anda.

Çoklu programlama; aynı anda birden fazla programın çalıştırılmasıdır.

Bir process'in çalışması için gerekli kaynaklar;

gerekli kaynaklar  
\* Ardışıl program kodu  
\* Program Counter.  
\* Yığın  
\* Veri alanı  
\* bellek kasesi (heap)

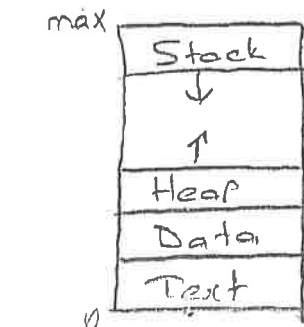
→ Hangi satırın işlediğini tutan bir sayıcı (PC)

→ Hafıza ihtiyacı (Stack adında geçici hafıza sağlanır.)

Heap = Dinamik olarak büyüyüp küçülebilen hafıza.

Multiprogram; Aynı anda birden fazla program aktif.

Multiprocessing; Bir programda birden fazla process aktif.



\* Stack hafıza geçici değerleri tutar (register, local değişkenler.)

Büyüyüp küçülebiliyor.

\* Heap: Dinamik olarak büyüyüp küçülebiliyor.

\* Data: Global değişkenlerin tutulduğu yer.

\* Text: Kodların tutulduğu yer.

Basit bir exe dosyası  
(Processin temsilci)

\* Processing çalışma listesi vardır. Yeni gelen process listeye eklenir.

Process ilk başladığında new konumundadır.

2. adım ready (kayıtlılık) aşamasındadır.

Kodlar işlenmeye başladığında running;

Bir işlemci belli bir nedenden dolayı bekletilirse waiting;  
(interrupt vs.)

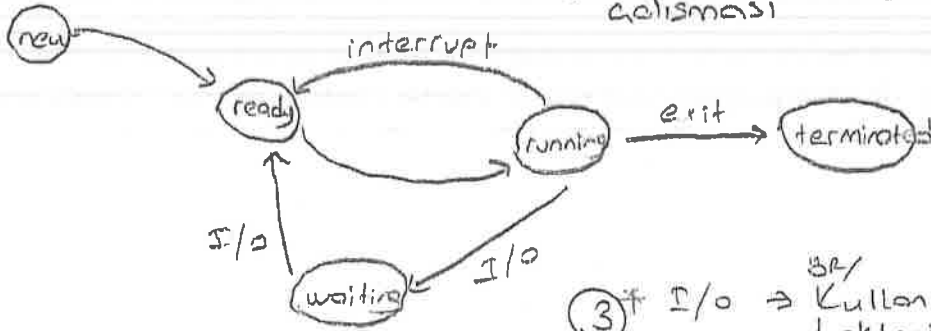
A.S Tanenbaum'un process örneği: Kek yapmayı seven bir bilgisayarci tarife göre kek yaptığını varsayalım. Process bilgisayarının kek tarifini okuyup malzemeleri kullanıp istenilen sonuç için işlem yapmasıdır.

① CPU üstünde belli bir çalışma süresi vardır.

\* Dolunca proses acılıktan çıkar. (Time.)

② \* Interrupt geldiğinde waiting durumuna geçer.

veya Bir saldırı olduğunda aniden virüs prog. çalışması



③ \* I/O → kullanıcıdan bir şey girme bekleniyor. (waiting durumuna geçilir.)

## İşletim Sistemi

Process'in durumunu (bekliyor mu, devam mı ediyor?) bilmek istiyor.

CPU üstündeki durumun kaydedilmesini ister. Bunun için Process Control Block (PCB) vardır.

Ana hafızada CPU register vardır. Peki neden?

PCB aktif olarak çalışan process'in resmini alır.

Interrupt geldiğinde başka process işleyebilir istedikten sonra diğer process CPU'da bulabilmeli (ataca adresine ihtiyacı var.)

Process Control Block içinde;

\* Process state

\* PC

\* CPU register

\* Memory-management information

\* Accounting information

\* I/O status information.

→ Prosesler arası anahtarlama, geçiş denir.

Context Switch işlemi çoklu programlamayı sağlar. (Bağlam değiştirme.)

→ Interrupt geldiğinde müdahale edilmesi durumunda yeni process çalışır.

→ Process işlem yapmadığı sırada kayıt işlemi yapılır.

\* \* Bir işletim sistemi üzerinde context switch işlemi varsa performans düşer.

Process'in artması performansı düşürür.

Hazır processlerin arasında birisini seçip çalıştırma Scheduling'tir. CPU'yu max kullanmayı sağlar.

İşlemler Kuyruk mantığıyla işletilir.

- Job Queue
- Ready Queue
- Device Queue

Kuyruklar bağlı listelerle çalışır. Çünkü daha kolay işletilir.

- Short-term scheduler: Hazır process kuyruğunda bekleyen programı seçen program. Hızlı olması gerekir, sıkca kullanılır.
- Long-term scheduler: Hazır process kuyruğuna gelecek olanları seçen programdır. Hızlı olmasına gerek yoktur. Artık long term scheduler'la gerek yoktur. Çünkü günümüzde daha fazla hafıza kullanılabiliyor.
- Medium-term scheduler: Short ve long term arasında ciddi bir zaman farkı vardır. Hazır process kuyruğu arasında bazıları harddisk'e götürüp yatar. Kalan yere öncelikli processler yatırılır. Bu işleme swap in/out denir.

Harddisk'i ana hafıza gibi kullanır - İşletim sistemini kullanıyor. Swap alanı deniyor.

Çocuk processler ana proseslerin Sistem kaynaklarını (hafıza bölgesini) kullanır.

Çağrı yapıldığında çocuk process '0' ; ana process '1' döndürür. Bu şekilde ayırt edebiliriz.

Çalışma Notu:



Bağlam değiştirme: Processler arası anahtarlamalı geçiş denir.

**PCB** (Process control block): (Process kimlik bilgileri)

Durum bilgisi, Program sayısı ve yazmaç değerleri, bellek tahsis bilgileri, iş sıralama bilgileri, yığın işaretçisi, işlemci kullanım bilgileri, GIC bilgileri gibi prosesle dair her bir bilginin tutulduğu yerdir.

Yazmaç değeri: Aklıya alma işleminden sonra kaldığı yerden devam edebilmeyi sağlar. CPU yazmaç değerleridir. Bağlam değiştirmeyi mümkün kılar.

Process üzerinde gerçekleştirilen işlemler: POSIX standardını destekleyen işletim sistemleri için process yönetimi sisten çağrılarla yapılmaktadır.

Windows işletim sisteminde ise sistem çağrısı NT sistem servislere karşılık gelir. Kullanıcının NT sistem servislere erişimi API ile sağlanır.

\* Prosesler arası hiyerarşide oluşturan procese parent proses;  
Olusturulan procese child process denir.

Bir proses olusturulurken su islemler yapılır:

- Proses tablasunda yer yoksa proses olusturulmaz.
- Proses tablasunda yer varsa, procese sistemde tek bir kimlik numaresi verilip baslangic öncelik degeri atılır. PCB olusturularak baslangic kaynakları sağlanır ve ilk degerler yüklenip hazır proses kuyruğuna eklenir.

Başlangıçta anne prosesin birebir kopyası olan çocuk proses kimlik numaralarından (PID) ayırt edebilir. Çocuk proses'e 0 döner. anne  $\rightarrow$  1

Yetim (orphan) proses: Anne prosesini sonlandıktan sonra çalışmaya devam eden prosesler. Zombinin aksine sistem kaynaklarını kullanmaya devam ederler.

\* Bir proses 4 farklı şekilde sonlanabilir.

- $\rightarrow$  İşlevini bitirip kendisi sonlanmış olabilir.
  - $\rightarrow$  Başka bir proses tarafından sonlandırılmış olabilir.
  - $\rightarrow$  Kullanıcı tarafından sinyal ile sonlandırılmış olabilir.
  - $\rightarrow$  Bir hata oluşup sonlandırılmış olabilir.
- (Oluşan hatanın kodu errno isimli euzensel bir değişkende tutulur.)

Zombi proses: POSIX standartlı işletim sistemlerinde, sonlanmış ancak proses tablasından bilgileri silinmemiş proseslerdir. Zombi prosesler kullandıkları tüm kaynakları iade etmişlerdir. Zombi prosesler ancak anne prosesleri tarafından gerçek olarak silinebilirler.

\*\*\* Herhangi bir proses: kaldığı noktadan yeniden başlatmak için o procese ait PCB bilgilerinin CPU'ya yüklenmesi beklenir.

Proses sinyalleri: Linux ortamında windows ortamında çok daha fazla sinyal bulunur. Bunun temel nedeni; windows ortamında sinyallerin yerini kullanım amacıyla işletilmiş API'lerin doldurmuş olmasıdır.

İş paylaştımı  
ve Paylaşımı



## Processler Arası Haberleşme

- \* Bağımsız iş yapan processler = çalışma süresince kendi işini yapar.
- \* Diğerleriyle ortak iş yapan processler = öz/paralel hesaplama için diğer process'in sonucuna ihtiyacı olur.

Öz Word programında yazarken 3 tane işlem aynı anda devam ediyor. Bu processler birbirlerinde haberdar olmak ister. Bu yüzden haberleşme işlemi yapılır.

Haberleşmenin 2 türü vardır.

→ Paylaşımlı hafıza modeli

→ mesajlaşma modeli

Bir process'in diğerine mesaj göndermesi için mesaj kernel'a teslim eder çünkü diğerinin hafıza bölgesine giremez. Orya sadece işletim sistemi girer.

\* İki process devamlı aralarında mesajlaşıyorlarsa ortak bir hafıza alanı ilan ediyorlar. Bunu işletim sistemi bildiriyor.

Paylaşımlı bölge ilan edilirken 1 tane sistem çağırısı yapılır. O bölgeyi kullanabilmesine dair.

Ama mesajlaşmada her mesaj için ayrı bir sistem çağırısı yapılır. (2 tane sistem çağırısı yapılıyor) Bu yüzden mesajlaşma daha yavaş.

## Üretici - Tüketici

Üreticimiz bir fırın olsun. Belli aralıklarla üretim yapıyor.

Ne kadar üretim yapar? Deponun büyüklüğü kadar.

Tüketicide, olduğu sürece tüketebilir.

Üretici paylaşımlı hafıza bölgesi kadar üretim yapabilir.

Tüketicide hafıza bölgesi olduğu kadar kullanılır.

İşletim sistemi aşamalı yolu üretir.

Sınırlı hafıza

Sınırsız hafıza → Pratikte gerçekte değil.

typedef struct {

üretici ile ilgili bilgi

? item++

item buffer [BUFFER-SIZE] → Buffer size kadar üretilir.

Üretim alanı dolduğunda basılması için bekleme şartım.

while (((in = (in + 1) % BUFFER-SIZE COUNT == out)

bu satır sağlar.

Tüketici açısından

in ve out aynı yeri işaret ediyorsa birşey yok demektir.

in++ / out++ diyorsa firında ekme var demektir. Alma işlemini yapar

Mesajlaşma yönteminde

send ve receive metodları aktif edilmelidir.



Bu mesaj nereye

gidecek (proses id)

mesajlaşmadaki problemler

Senkron asenkron mu

Direkt vasıla olacak mı

Diğerinde (Diğer proses) kaydedilecek mi (buffer var mı)

Tüm bunları işlemelidir.

Direkt communication: Giden ve gelen prosesin yeri belli.  
Anne ve anne 2 proses kullanılabiliyor.

$p \rightarrow q$  (gift yolla olabilir.)

## Indirect Communication

→ Kullanılan yöntem posta yöntemiyle çalışır.

→ Posta kutusuna atıyorsun postacı diğer kişinin posta kutusuna atar.

Buna direkt olmayan haberleşme denir.

(Bir tane posta kutusu ve bir postacı vardır.)  
İşletim sist. toyn eder.



### Avantajı

→ Bir posta kutusunu birden fazla process kullanabilir.  
(Aynı mesajı birden fazla kişiye yollayabiliriz.)

Hepsiyi teker teker boş kurmasına gerek yok.

→ Birden fazla mesajda (fazla posta kutusu) gönderilebilir.

### • Direkt Senkron haberleşme

→ Processlerin birbirinden haberi var.

Senkronda bir ürün tükenmede üretilmiyorsa bloking haberleşmedir.

### • Asenkron haberleşme

Mesaj teslim ediyor okundu mu okunmadı mı işletim sisteminin ilgilenirdiği.

Asenkronda non-bloking haberleşmedir (üreticiyi bekletmeyen)

\* Bazen üretilen mesajlar kaybolabilir bunlar ram buffering kullanılır. (Depoları)

1) Hafıza olmayan = Direkt ulaşır. Depoya gerek yok.

2) Sınırlı hafıza yöntemi = <sup>sr</sup> LFS dolana kadar mesaj gönderilebilir.

3) Sınırsız hafıza → Pratikte yok.

Solaris işletim sist. bunu kullanır.

⇒ Posta kutusu yöntemi kullanılır.

(Posta kutusu dolunca bekler yeni mesaj (letter) boşaltılmasını bekler.)

\* 4 tane makinede paralel programlama yapıldığını varsayalım.  
Bunlar paylaşımlı hafıza yönteminin kullandıkları mesajlaşma kullanılır.  
Bunun için sockets kullanılır.

\* Ağlar arası haberleşmeyi sağlayan port : sockets.

C'de başlamış Java'da yaygınlaşmış

Java'da Socket kütüphanesi kullanılır.

İp adresi > Haberleşme için gerekli.  
Port numarası

→ Tüm browser'ler 80 portuyla haberleşir. Bunu ayrıca haberleşmeye gerek yok.

---

Threadler ve prosesler arasındaki farklar:

Proses: Bir programın aktif olarak çalışan kısmıdır.

Thread: Senkronize olmuş, prosesden daha küçük yapıda, yetenekli iş parçacıklarıdır.

#### Proses

\* Kendilerine ait hafıza bölgesi vardır.

\* masraflıdır.

#### Thread

\* Proseslerin hafıza bölgesini kullanır.

\* Prosesler kadar masraflı değildir.  
(Aynı kaynakları paylaşırlar.)

\* Programın cevap verme yeteneği daha fazladır.

\* Öbeklendirme yapmamızı sağlar.

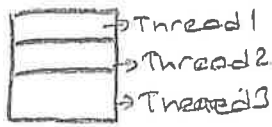
\* Kaynak israfı sst konusu değildir.

## THREADING

\* Bazı işler için process oluşturmak gereksizdir. Onun yerine alt processlar oluşturmak daha mantıklıdır. Bunun için thread'ler kullanılır.

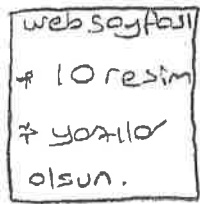
\* Thread'ler senkronize olmuş process'den daha hızlıdır.

\* Thread'ler ait oldukları process'in kaynaklarını kullanır. (Hafıza bölümleri ortak kullanılır.)



Threadler;  
 → Bağımsız code, data, file oluşturmuş oluruz  
 → Processler kadar masraflı değil.  
 → Yeterince iş parçacığıdır.

Web sayfası açan programı düşünelim;



Önceki zamanlarda;  
 Çalışan bir process varsa resim download edilirken aynı anda yazı download edilemezdi.

Ama artık threadler var ve threadlerle hem yazı hem resim vs. aynı anda isteniyor.

\* Threadler programın programı yazmasını kolaylaştırır.

multithreading : İşler parçalara bölünür bunlarda thread'ler yüklenir.

Örnek:

Dinleme threadimiz olsun. Bu thread http isteklerini görsün, ip numarasını alsın ona cevap versin.

Dinleyen threadimiz cevap veren thread'i başlatır ama kendi işini yapmaya devam eder.

Cevap veren thread ise cevapları istemciye yollar.

Tüm threadler devamlı işini yaparlar çalışma hızı artar.

Avantajları:

- 1) Programın cevap verme yeteneği artar. input/output sonucu bekletmez. Bir thread input/output beklerken diğer thread işini yapmaya devam eder.
- 2) Kaynakların paylaşımı; (Kaynak israfı olmaz.)
- 3) Ekonomi : Aynı kaynakları paylaştıkları için ekonomiktir.
- 4) Önceliklendirme;

1 müşteriye 10sn'de cevap veriliyorsa  
10 müşteriye de 10sn'de cevap verir.

İşletim sistemi bunu sağlar.

\* Eğer bir sistem ölçeklendirilirse müşteri sayısı artsa da kalitede düşme olmaz.

10 process çalıştırılabilen bir sistem olsun. Her process için 10 thread çalıştırılıyorsa 100 işlem yapılabilir deriz.

### Multicore Programming

Çok çekirdekli sistemler paralel hesaplamalarda çokça vardır.

Çok çekirdekli programlardaki bazı problemler

- 1) İşleri Bölme: Çekirdekler arası işlerin doğru bir şekilde paylaşılması gerekir. Günümüzdeki bilgisayarlarda bu problem tam olarak çözülmemiştir.
- 2) Yüklerin dengelenmesi: İşler mantıklı bir şekilde dengelenmiş mi? Sıra sılla atılacak hep aynı CPU'ya daha ağır işler yüklenabilir.
- 3) Verilerin paylaşımı: Aynı dosyayı kullanan birçok process var. Her CPU'da aynı dosya bulunabilir.
- 4) Verilerin bağımlılığı: Bir process işlenirken diğer processa ihtiyacı varsa farklı CPU'larda nasıl iletişimi sağlanacak.

→ CPU'yu en çok tüketen oyunlardır. Artık oyunlarda grafikler ilgili hesaplamalar GPU ile yapılmaya başlandı. (Paralel işlem)

Single Core

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>
1	2	3	4

T: Thread (4 zaman diliminde yapıldı.)

multi core

Çekirdek 1  
Çekirdek 2

T <sub>1</sub>	T <sub>3</sub>
T <sub>2</sub>	T <sub>4</sub>
1	2

(2 zaman diliminde yapıldı.)

→ Zaman tasarrufu edilir

## Kullanıcı ve Kernel Threadleri

### 1) Kullanıcı threadleri:

Kullanıcı threadleri kullanıcı seviyesinde; Kernel threadleri kernel seviyesinde çalışır.

Thread kütüphaneleriyle gerçekleşir. İşletim sisteminden sistem çağırısına ihtiyaç duymazlar. İşletim sistemi müdahalesi olmadan daha hızlı çalışırlar.

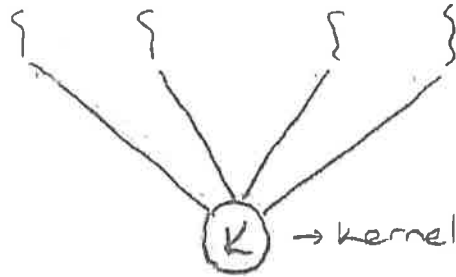
### 2) Kernel threadleri:

İşletim sistemi müdahalesine ihtiyaç duyar.

Kullanıcı düzeyde, kernel düzeye ulaşması için modeller:

#### Multi-threading modelleri:

##### 2) Many-to-one model:



Eğer threadlerden biri bloke olursa, kernel işlemin sadece bir thread tarafından yapıldığını düşündüğü için threadler arası geçiş yapamaz.

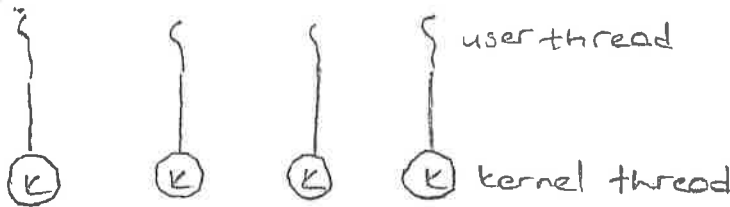
Kullanıcı hafızayı vs. kullanmak isterse kernel'a ulaşması gerekir.

Kernel thread → Kernel seviyesininin boş olması isterin.

→ Bu modelde herhangi bir user thread servis çağırısı yaptığında diğer thread'ler o çağırının sonucu gelene kadar bekler. (Multi-programming yapmayı engeller)

→ Senkronizasyonu etkiler

##### 3) One-to-one model

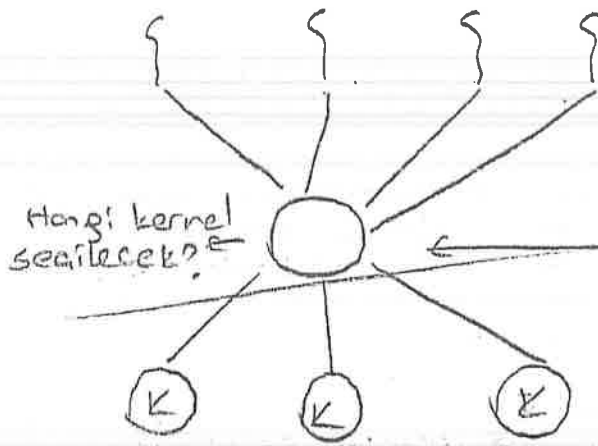


⇒ many-to-one modelde ki gibi bloke durumu söz konusu olmaz.

Threadlerden biri sistem çağırısı geldiğinde, diğer threadlerin kullanımı için cevap beklemeye gerek kalmaz.

Dezavantajı işletim sisteminin kernel'ı karmasık olur. Bu da kaynak israfı ve yönetim problemlerine neden olur. Maliyetlidir.

#### 4) Many-to-many model



many to one'la karşılaştırırsak;  
Bu modelde

- Herhangi biri klavyeyi kullandıkça isterse bekleme yapmaz.

Problem:

Seçme işlemi zaman kaybına neden olur. (Seçim algoritması)

One-to-one'la karşılaştırırsak;

- Bu modelde
- Kernel'da karmaşık bir yapı oluşmaz.

En çok one-to-one modeli kullanılır.

→ Bu modelleri kullanmak için thread kütüphaneleri olmalıdır.

- 1) Pthread kütüphanesi : POSIX threadleri UNIX destekler.
- 2) win32 API kütüphanesi : Windows'u destekler.
- 3) Java Threads kütüphanesi : Sanal makine oluşturup threadleri onun üstünde çalıştırılır. Bu yüzden işletim sisteminde bağımsız blok çalışırlar.

(POCIS)  
Thread havuzu:

Programcı threadleri bir yerde (havuz) toplar. İhtiyacı olduğunda thread'i alır, kullanır ve havuza geri atar.

(Bir prosesin istediği anda istediği kadar thread yaratması sistemde bir problem oluşturabilir bu yüzden thread havuzu vardır.)



## CPU scheduling :

Multiprogramlamada CPU'yu iyi kullanmak amaçlanır. Bunun için scheduler kullanılır.

→ Long term scheduler

→ Short term scheduler

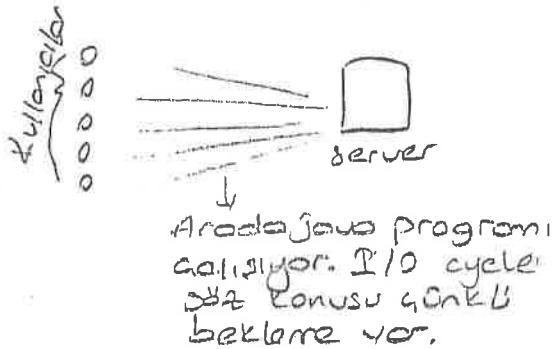
→ Medium term scheduler

bunların en önemlisi, kullanılan short-term schedulerdir.

\* Proseslerin 2 tane yaşam döngüsü vardır.

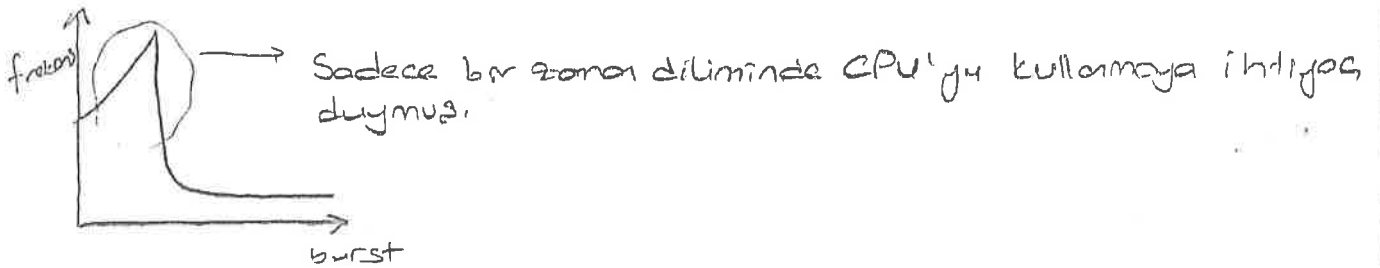
1.) CPU cycle → CPU'ya bağlı prosesler

2.) I/O cycle → Hayatlarını beklemeyle geçiren prosesler.



\* Bir proses ya CPU'da çalışıyor ya da I/O bekleme durumundadır.

Genelde bir prosesin ihtiyacı duyduğu CPU -burst '0' dir.



\* Hazır proses kuyruğundan prosesi seçip yürütmeyi sağlayan CPU scheduler'dır.

Prosesler bir kuyruk, bağlı liste, FIFO vs. şeklinde tutulabilir.

CPU zamanlama (Çıkarılma durumları)

I.) Proses running durumundan waiting durumuna geçiyorsa

II.) Proses running durumundan ready durumuna geçiyorsa (interrupt)

III.) Proses waiting durumundan ready durumuna geçiyorsa (I/O bekliyordur, çalışır vaziyette gelmiştir.)

IV.) Herhangi bir proses işini bitirmiş çıkıyordur.

I ve IV'de CPU scheduler açısından yapılacak bir işlem yoktu.

**nonpreemptive:** I ve IV durumlarında herhangi bir prosesin çıkarılmasına gerek yok.

**preemptive:** Birseyi haberi olmadan oradan çıkarmak demektir.

Bu yapı Windows 95, 2003'de kullanıldı.

Donanımsal olarak interrupt oluşturup prosesi çıkarmaya ihtiyacımız olabilir.

Çalışan prosesi çıkarmadaki problemler

1- Veri paylaşımı: Veri üzerinde işlem yapılırken proses çıkarılmak zorunda kaldı, veriler güncelleme yapılmadığı için başka proses veriyi kullandığında güncel olmayan veriyi kullanır.

2- Kernel modda problem: Önemli bir sistem dosyası üstünde işlem yapılırken çıkarma işlemi yapılırsa kernel'da işlem yapıldığı için işletim sistemi tüm süreçlerin çalışmasını engeller.) etkiler.



**Dispatcher:** Bir şeyi alıp bir yere ileten demektir.

→ öncelikle context switch yapmalı  
User moda geçmeli  
PC kullanarak upload ediyor

Dispatcher bir işletim sisteminin hızını etkiler (doğrudan değil)

CPU'nun (scheduler) zamanı ayarlanırken gerekli kriterler:

\* **CPU utilization**: CPU'yu mümkün olduğunca meşgul tutmak. CPU'yu çok kullanmaya çalışmak. Mümkün olduğunca yüksek olmasını istiyoruz, CPU'nun.

\* **Throughput**: (İş miktarı) CPU ne kadar çalışıyorsa o kadar iş yapıyordur. Birim zamanda yapılan iş miktarıdır. Yüksek olmasını istiyoruz.

\* **Turnaround time**: (İşin tamamlanma süresi) Bir proses başladığı andan işini bitirdiği ana kadar ki süreye denir.

## - Utilization :

\* **Waiting time** : (Bekleme süresi) : Prosesin hazır proses kuyruğunda bekleme süresidir. Bir prosesin defalarca hazır proses kuyruğunda beklediği süre.

\* **Response time** : Bir istek geldiğinde değerlendirilip cevap vermesine kadar olan süredir.

\* Örneğin word'de bir yazı yazıyoruz, word'de yazının görünmesine kadar geçen süre.

\* **Waiting time, response time ve turnaround time'in kısa olması** istenir.

Tüm bu kriterler ölçülmüş bunları kullanan çeşitli algoritmalar türetilmiştir. Bu algoritmalar;

→ **FCFS scheduling** : Prosesler aynı anda gelecek işletim sistemi önceliğini kendi tayin eder. (Biri id'ine bakar)

\* Uzun süren proseslerin kısa süren proseslerden önce gelmesine konvey etkisi denir. Bu durum istenmez çünkü kısa süren proses basu başına bekler.

Tek kullanıcı sistemlerde kullanılmasının pek fazla dezavantajı yok. Çok kullanıcı sistemlerde verimsizdir.

→ **SJF scheduling** : (En kısa süren ilk çalışsın.)

Proses ne kadar zaman alacak?

Proses CPU'da ne kadar sürede çalışacak. Bu sorulara cevap verilmesi <sup>güç.</sup>

Genellikle long-term scheduling'te kullanılır. Çünkü ortalama zamanı daha önceden muhasebesini yapmış oluruz.

Avantaj → Bekleme zamanını düşürür.

CPU çalışma süresi için tahmini süre hesaplaması  
(Sonraki prosesin çalışma süresi)

$$T_{n+1} = \alpha t_n + (1 - \alpha) T_n \text{ 'dir.}$$

Short term scheduler'da bu hesaplama yapılmaz. Çünkü işi yavaşlatır. Bu yüzden long term scheduler'da kullanılır.

→ **Shortest-remaining time first**

SJF ile preemptive (çıkarma işlemi) birleştirilince ortaya çıkar.

Dezavantaj → Uzun süreli proseslerin bekleme süresini çok artırır.

→ Priority Scheduling: Çıkarma işlemi yapar. Yüksek öncelikli prosesler önce işlenir. Öncelikli olan alır diğerini çıkarır.

Problem → Önceliği olmayan ağırlıktan ölür.

• Ağırlıktan ölme: Hiçbir zaman ihtiyacı olan CPU'yu kullanamaz

• Yaşlandırma: Belirli aralıklarla düşük öncelikli prosesin önceliği artırılarak bu problem çözülebilir.

→ Round Robin (RR): Proseslere time quantum verilir. ÖR/50ms olsun. Prosesi dairesel kuyruğa tutuluyor. Zamanı göre kuyruğa getirir. Zamanı dolduğunda proses geçer. Bekleme süresi belli, bir ortalama da olur.

Çok uzun çalışması gereken proses işini daha uzun sürede yapar.

→ Sürenin az seçilmesi context switch'i arttırır. Performans düşer, maliyet artar.

→ Çok uzun süre seçilirse FCFS gibi olur.

Yani bu sürenin seçilmesi çok önemlidir.

Multilevel Queue: Birden fazla kuyruk kullanmak.

Kuyruğu 2'ye bölebiliriz. Öndekiler → yüksek öncelikli zıt kullanılır.

↑ Sistem prosesler  
öncelik interactive (iletişim)  
↓ interactive editing

} Her kuyruk için ayrı bir algoritma kullanılabiliriz.



## Senkronizasyon:

- \* Hafıza bölgesinde bir tane veri yapısı var. Örneğin; 1. işlem sonucu bitmeden aynı hafıza bölgesini kullanan başka proses ortadan okuma yapabilir.
- \* Kernel modunda çalışan proseslerde işletim sistemi tutarlık önemlidir.
- \* Doğru bir buffer'ın olması. (Üretici-tüketici)

```
while (count == buffer size) { hata mesajı yazma
    küçülse üretim yap
}
```

Consumer'i içinde üretilecek birşey varsa tüketilecek.

Problem ne?

Tek satırda yapılan count ++ assembly dilinde 3 satırda yapılır. CPU 3 satırlık işlem yapacak demektir.

Senkronizasyon: Aynı anda çalışan 2 tane proses aynı paylaşımlı bölgede çalışırken tutarlı olmasına senkronizasyon denir.

Tutarlılığın sağlanması için;

1 → Kritik bölge tanımı yapılmalıdır.

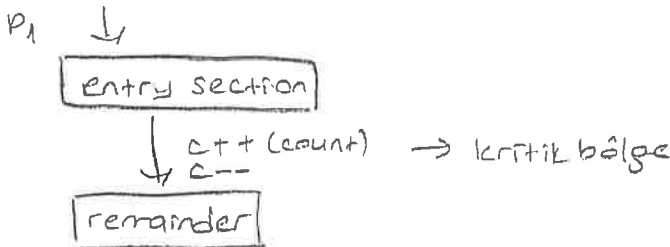
"Kernel modda çalışan proses işlem yaptığı bölgeye kritik bölge tanımlaması yapılıyor." (Değişiklik yapılan kod bölgesi, Bu bölgedeki değişiklik diğer sonuçları etkiler.)

İşletim sistemi, buradaki çalışmasını kontrol altına alır.

Bunun için karşılıklı dışlama yapılır.

• Karşılıklı dışlama = Eğer proseslerden bir tanesi kritik bölgede çalışıyorsa başka proses o kritik bölgeye giremez.

2 → İşlem: Bir proses işleme başladıktan sonra;



• Kritik bölgeye gelmek istediğinde proses ilk olarak entry section'a gelir.  
İşletim sistemi entry section'da bulunan proseslerden birini seçer kritik bölgeye girer.  
→ Bu karar sonsuza kadar bekletilemez.

Sınırlı bekleme ilkesi = Bir proses kritik bölgeye girmek istediğinde belli bir süre bekletilir. Sonra girme hakkı verilir. Çünkü prosesin asıllıkta olmasına izin verilemez.

Kritik bölge kullanımı; Dosyalama sistemlerinde kullanılır. Açık dosya listesine ulaşmak için kritik bölge kurallarını sapılması lazım. Örneğin; 0 ana kadar açık olan dosyaların listesi tutulur. Açık olan dosyayı başka bir proses açamaz sistem uyarı verir.

Bazı işletim sistemlerinde bu durum farklıdır.

Unix'te; Bir dosya açıksa başka süreçte bu dosyaya ulaşabiliriz ancak dosya üzerinde değişiklik yapamaz.

Kernel modda çalışan proseslerin adresleri kritik bölgede tutulur.

→ Yazıcıyı 2 proses aynı anda kullanmak istem. Kritik bölgesi bulunur. Burdaki prosesler paylaşılan bir kaynak olduğu için bir yazıcıyı kullanırken, diğerinin kullanmasına izin verilmez.

Kernel modda çalışan proseslerin kritik bölge şekilleri :

Prosesi dışarı çıkarma : Çalışma esnasında prosesin sonlandırılmasına izin veriyor. Bu yüzden yapılan işlem sona eriyor. Kritik bölgede yapılan işlemler geri alınır. İşletim sisteminin cevap verilebilirliğini artırır.

Prosesin bitmesini bekleme : Eski tür kernel'larda vardır.

Çözüm yolları ;

Yazılımsal olarak Paterson çözümü sık konusudur.

Ç # Paterson çözümü : Şuanda kullanılmamaktadır. İlk kullanılanlardan.

Sistemde aktif 2 proses 1 CPU olsun. LOAD ve STORE işlemleri tek bir komutla gerçekleştirilir.

\* turn = bir prosesin hangisinin sırası olduğunu tutuyor (P1 mi; P2 mi)

\* flag = kritik bölgeye girip girmeme isteği.

flag = true ise hangi prosesin sırası geldiyse gerçekleştirerek.

$\left. \begin{array}{l} \text{flag}[i] = \text{true;} \\ \text{turn} = j; \end{array} \right\}$  flag[i] = true ise P<sub>i</sub>'ye öncelik vermiştir diğer proses.

\* Sadece 2 proses olduğunu kabul ediyor

# Anahtarlama tekniği ile de çözülebilir. Ancak bu donanımsal bir çözümdür.

Kesme gelmesini engellenerek için ;

- Çalışan prosesin işi bitmeden çıkarılmasını önleyebiliriz.

- Yüksek öncelikli bir proses geldiğinde onu isteyip geri döndürebilir.

Donanımsal olarak ; Bazı komutların kesilmesini engelleriz.

Automatic komutu kullanılır. İşlemin bitene kadar interrupt bölünemez.

→ Hafızaya yazma  
→ Hafızada değişiklik yapma isteği

Donanımsal olarak ;

Test and set gözetimi i instruction gerçekleşirken işlem bölünmez.

lock olan proses kilitleme yapar. lock = true

Boşta proses o alana girer ve çıkışta değer false yapılır.

Lock'ın değerini true yapmak için test and set kullanılır.

Kontrol ve değiştirme işlemlerinden kurtulmak için bunu bu şekilde yaparız. Test and set komutu 2 işlemi tek komutla yapar. Fakat donanımın bu yapıyı desteklemesi lazım.

α Pointer'larda hem hafıza bölgesini hem de son değeri kullanabiliriz

```
boolean rv = *target;
target = true;
return rv;
```

lock = true

swap instruction komutu i

a      b  
3      5

bunları değiştirmek istediğimizde, bu işlemi yaparken tutarsızlık olmasın, bölünmem diye işletim sistemi bunu otomatik yapar.

Semaphore:

S sayısı

wait() → int değerini azaltır.

signal() → int " artırır

lock = 0 veya 1 olmasına göre ayarlanır.

1) Sayan semafor

2) Binary semafor.

→ 0 ve 1 kullanır. Genelde kritik bölge adresleri için kullanılır.

wait() fonksiyonu atomik olarak çağrılır. Başka bir interrupt işlemini bölmemek.

mutex() i karşılıklı dışlamayı sağlar.

Deadlock : S=0 olursa wait işlemi yapmadan 0'dan büyük olduğu için başına while döner. Sistemin kaynakları başına tüketilmiş olur.

Hazır proses kuyruğundan çıkar, semafor kuyruğuna konulur.

Prosesin while döngüsü içinde beklemesi busy waiting'dir.

Prosesi beklerken block() ederiz. Signal() geldiğinde semafor kuyruğundaki procese wakeup() yapılır.

Semaphore: Kritik bölge kontrolü ve proses senkronizasyonu için kullanılan ve iki adet kesilemeyen işleme ( $\text{up}(V)$ ,  $\text{down}(P)$ ) sahip olan tamsayı bir değişkendir.  $P$  işlemi ile semaforun değeri sıfırdan büyükse bu değer 1 azaltılır ve işleme devam edilir; semaforun değeri 0 ise proses bu semafor üzerinde bloke olur. (sleep)

$V$  işlemi ile semaforun değeri 0'dan farklı ise (veya 0 olupta bu semafor üzerinde bloke olmuş proses yoksa) bu değer 1 arttırılır. semaforun değeri 0 ise ve üzerinde bloke olmuş en azından bir proses varsa, bu proseslerden biri uyandırılır. (wakeup), semaforun değeri arttırılmaz. Bu iki işlem atomiktir, kesilemez.

$\text{down } P$  işlemi ile ;

$$S > 0 \rightarrow S - 1$$

$$S = 0 \rightarrow \text{Proses bu semafor üzerinde bloke olur. (sleep)}$$

$\text{up } V$  işlemi ile ;

$$S \neq 0 \parallel \text{semafor üzerinde } \cancel{S \neq 0} \text{ bloke olmuş proses yoksa} \Rightarrow S + 1$$

$$S = 0 \parallel \text{en az 1 tane bloke olmuş proses varsa} \Rightarrow \text{Proses wakeup semaforun değeri arttırılmaz}$$

DMA : Bellek ile diğer aygıtlar arasında veri iletişimi için kullanılan bir yapıdır.

Avantajı : Veri iletişimi sağlarken CPU'yu kısmen devere dışı bırakarak veri akışını hızlandırmak, CPU'yu daha az meşgul etmek.

Simetrik Çoklu İşleme

Asimetrik Çoklu İşleme

\* Her işlemci işletim sisteminin ayrı bir kopyası üzerinde çalışır. İhtiyaç duydukları anda birbirleriyle çalışırlar.

\* Her işlemci ayrı bir iş için tahsis edilmiştir. Master olan bir işlemci, görev ve iş dağılımı yapar. Slave olan ise işleri sıraya sokar ve tamamlar.



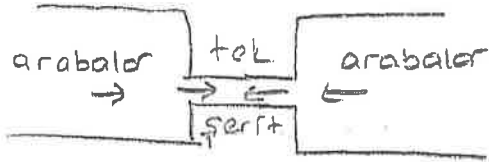
Sistem kaynaklarını ortak olarak kullanan veya birbirleriyle haberleşen bir grup prosesin zorunlu olarak block olması durumudur.

8. hafta

DeadLock = Kilitlenme

Deadlock konusu : Bir kavşakta 2 tren birbirine yaklaşıyor bir durmalı geçmek için. Birbirlerini beklemelidir.

İşletim sistemi sınırlı sayıda kaynakları sınırsız sayıda istenilen için ayırmaya çalışır.



Bu sistemde deadlock vardır. Görmek için ?  
→ Eldeki kaynağı bırakmayı sağlar. Proseslerden biri sonlandırılır.

Dezavantaj : Elimizde kaynağın gereksiz yere tutulmasına neden olur.

\* Sistem modeli = Herbir prosesin izlediği adımlar

- 1) Request : Kaynağa ihtiyaç duyduğunu söyler ve onu ister
- 2) Use : Proses kaynağı kullanır.
- 3) Release : Proses sistem kaynaklarını geri bırakır. Kullanılan hafıza bölgeünden 0 adresleri siler.

Release işlemini Java'da Garbage Collector yapar.

Deadlock için gerekli koşullar :

- \* Mutual exclusion : Kısıklı dışlama : Bir proses paylaşılan bir kaynağı kullanıyorsa diğer prosesin bu kaynağı kullanmasının önlenmesidir.
- 1. Hold and wait : Tut ve Bekle : Proseslerin eline geçirdikleri kaynakları, diğer istedikleri kaynakları da ele geçirene kadar bırakmaları
- 2. No preemption : Prosesin par bitmeden dışarı çıkarılmasına izin verilmiyorsa deadlock oluşur. (Proseslerin elinde tuttukları kaynakları işletim sistemi zorla geri alamıyorsa.)
- 3. Delayed release : Circuit wait

\* Graf algoritması = Grafı döngü oluştursa deadlock oluşur. Bir döngü varsa deadlock olabilir. Bir döngü varsa kaynak varsa deadlock olma olasılığı vardır.

Deadlock'ları nasıl önleriz ?

- 1 - Sistemin hiçbir zaman ölmeli kilitlenme durumuna girmemesini sağlamak
- 2 - Sistemin ölmeli kilitlenme durumuna girdikten sonra tekrar çalıştırılmasını sağlamak.
- 3 - Problemi gözardı edilip sistemde ölmeli kilitlenme olmayacağına inanmak.

Ölme algoritması

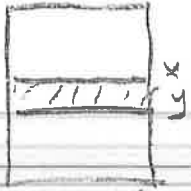
Tut ve Bekle'yi engellemek için;  
Benim çalışma süremde bana su su lazım der proses.  
Fakat kaynakları hepsini aynı anda bulmamız zor.  
Yada aklıktan ölmeye olabilir.

Safe state: Bir proses kaynakları istediğinde o kaynağın verilmesi ile  
güvensiz durum oluşacaksa o süreç o kaynak verilmez.

ana hafıza: Kodlar hafızaya yüklenir. PC'de sırasıyla çalıştırılır. Kodun  
çalıştırıldığı yer geçicidir.  
Ana hafızanın dolu mu boş mu vs. nasıl yönetileceğine memory  
management karar verir.

Ana hafıza çok büyük. Proses ana hafızaya ulaşamaz. Bu  
zahmetli bir iş.  
Ana hafıza sayfa denilen kısımlara ayrılır.

Base and limit registerlerle bir prosesin hangi kısımları kullanılabileceği  
tahsis edilir.



ana hafıza

Proseslerin sadece kendi bölgelerine ulaşmasını sağlıyor

CPU kendisi adresleme yapmakla zorlanır. 100 bin byte'lık bir  
adresleme kapasitesine sahiptir.

CPU'da mantıksal adresler oluşturulur, fiziksel adresleme yapılır.

CPU'da base & adres & base + limit kontrolü yapılır. Bu işlem  
donanım ile yapılır.

lojik & gerçek adres dönüşümü yapar.

Yazılım - seviyesinde yaparsa performans düşer. Donanım ile yapılır.

Adresleme 3 aşamada yapılabilir.

1) compile time : Binary koda dönüştürme.

2) Load time

3) execution time (run time)

\* Link işlemi : Bazı kütüphaneler devamlı kullanılır. Bunları bağlama  
işlemiyle tekrar kullanımı sağlanır. (Dinamik)

## Adress Binding İşlemi :

\* **Compile time :** Compile time anında bağlama yapılırsa, hangi bölgelerin neye tahsis edileceği bellidir. Eskiden kullanılan bir yöntemdir. MS-Dos'da kullanılır. A değişkeni 10 numaralı hafıza bölgesindedir. Artık mutlak kod olur. Başka makinede çalıştırırsak çalışmaz.

\* **Load time :** Yükleme zamanında yer değiştirilebilir kod olur. Birden fazla base adres kullanılıyorsa, bu yöntem çalışmaz.

\* **Execution time :** Dinamik bir şekilde bağlama işlemidir. Adres bağlama işlemi run time sırasında hafıza bölgesi seçiliyor.

Mantıksal ve fiziksel adres alanları : A değişkeni için 0 numaralı yeri veriyorsa  $A + 4$  'der. Bunu deme nedeni sanal bir dünyada yaşamayı sağlar. Dönüşüm işini donanım gerçekleştirir.

Run time'da binding yapılırsa fiziksel adreste mantıksal adres devamlı farklı olur.

memory management unit işletim sistemi üzerinden önemli bir yükü alır. Lojik adresi gerçek adrese dönüştürür.

Dinamik Loading = Dinamik yükleme

makinenin üzerinde 8/ 1GB'lık hafıza var ben 5GB'lık bir program çalıştırmak istiyorum. Execute edilen kısmı bile benim hafızamdan fazla diyelim.

Bir programın tüm kısımlarını kullanmadan, dinamik yüklemede o sırada lazım olanlar getirilip yüklenir sonra silinir. Toplama işlemine ihtiyacı varsa onu yükler diğerini çıkarır. Bu olayı işletim sistemi destek vermez. Donanım ve kullanıcı tarafından sağlanır.

Dinamik Linking = Bağlama işlemine kadar programın ihtiyacı duyduğuunu entegre etmeyi sağlar.

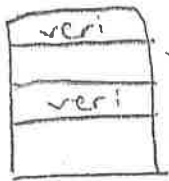
İhtiyacı duyulduğunda derlenmiş olanları otomatik olarak otomatik olarak kullanmayı sağlar.

**Swapping :** Proseslerin ihtiyacı duyulanlarını hafızada tutarız.

**Backing store** diye depolama bölgesi kullanılır. Hafızanın devamı gibi kullanılabilir.

Yükleyemeyeceğimiz kadar büyük olan parçaların tutulduğu yerdir.

\* Hafıza belli büyüklükteki sayfalardan oluşur. Sıra sıla prosesleri baslatıp, hafıza bölgelerini tahsis ettik diyoruz.



Bazı alanlar var. İşletim sistemi bu boş bölgeleri birleştirir.

Hafızaya yerleştirmede 3 yöntem:

1) First-fit: İlk bulunduğu boşluğa prosesi yerleştirir. Prosesin bir kısmı dörderde kalabilir.

2) Best-fit: En iyi yer nerediyle oraya yerleştirilir. Hafızanın tamamını taraması lazım.

3) Worst-fit: En büyük yere, prosesi yerleştirilir. Büyük boşluklar doldurulmuş olur.

Hız ve depolama açısından worst-fit daha iyidir.

Proses 1	10KB
Proses 2	11KB
Başkaldı	15KB

\* Fragmentation: Boş yer

\* External: Prosesi parçalayarak hafızaya yerleştirir.

\* Internal: Yerleştirilen yerin içindeki boşluk, küçük bir boşluk. Hiçbir prosede kullanılmaz.

Compaction: Hafızadaki küçük boşlukları birleştirip büyük boşluk elde etmeye çalışır, proseslerin yerleri değiştirilerek yapılır.

Fragmentation ve compaction artık kullanılmıyor.

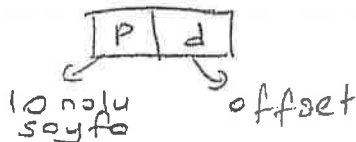
Paging (Sayfalama)

Fiziksel hafızada → çerçevelere

Mantıksal hafızada → sayfalara bölünür.

Page table = Mantıksal

Page number offset = Frame'in mantıksal alanda nerede çıkacağını söyler.



Paging hardware ile page number'ın büyük olmasından kurtulur.

Sarat Bellek (Virtual memory): İkinci bellek üzerinde ana belleğin parçalarını gibi adreslenebilir, fiziksel bellekten çok büyük bir depolama alanı tahsisidir.

Sayfalama: Toplam bellek alanının eşit büyüklükte sayfalara bölünmesine denir.  
mantıksal bellek üzerindeki aynı uzunluktaki bloklara 'sayfa' denir.

Sayfa Tablosu / İletim sistemi: her proses için sayfa tablosu tutar.  
- Prosesin her sayfasının hangi çerçevede olduğu  
- bellek adresi - Sayfa no ve sayfa içi offset adresini tutar.

Çok seviyeli sayfa tablosuz Bm sayfa tablosunun yeterli olmadığı durumlarda kullanılır.

Translation Lookaside Buffer (TLB):

En yakın zamanda kullanılan olan sayfa tablosu kayıtlarını tutar.  
Ana bellek için kullanılan cep bellek yapısına benzer bir işlev görür.

Deadlock örneği:

200K'lık bir bellek bölgesi proseslere atanabilir durumda ve aşağıdaki istekler oluşuyor:

Her iki süreçte ilk isteklerini aldıktan sonra ikinci isteklerini alamıyorsa kilitlenme oluşur.

P<sub>1</sub> Prosesi  
80K ister

P<sub>2</sub> Prosesi  
70K ister

60K ister

80K ister.

$\frac{1}{\sqrt{1+2x^2}}$

## Dosya Sistemleri

Dosya: Bilginin tutulduğu yapılardır.

Dosyaları işletim sistemi adı ve uzantısına göre tanır.  
doc, docx ...

1. exe dosyaları
2. com dosyaları
3. bat dosyaları



Çalışması için 2. bir yapıya gereksinim duymaz.  
Doğrudan çalıştırılabilen dosyalar.

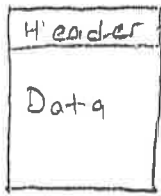
Diğerleri extra bir programa ihtiyaç duyar.



→ operating system en altta bulunur.

Bir işletim sistemi bir dosyanın bit (resim) dosyası olduğunu nasıl anlıyor?

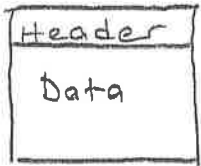
\* Öncelikle dosyayı okuması lazım.  
Dosya okuma!



a.bmp

Dosya'nın bmp dosyası olduğunu, şu pikselde vs. bilgisini verir.  
Boşluk kısmına bakıldıktan sonra point'i açar.  
Bu işlemleri CPU'nun kontrolüyle yapar.

\* Doğrudan Çalıştırılabilir Dosyalar \*



a.exe

Header'i okunduğunda magic number'ı görür. (26 H)  
Exe dosyalarının magic number'ı vardır. Bu no  
dosyanın doğrudan çalışmasını sağlar.

bat dosyası i satır satır çalıştırma yapar. (Sequation çalışır.)  
com dosyaları kendr kendine çalışır.

\* Yapı olarak dosya türleri nelerdir?

- 1) Byte yapılı dosyalar = Hard diskteki dosyalar.
- 2) Record yapılı dosyalar = Veritabanı dosyaları
- 3) Bit yapılı dosyalar

\* Erişim Türleri

1.) Sequeation (ardışıl) 2.) Rastgele erişim 3.) Ağac yapısı



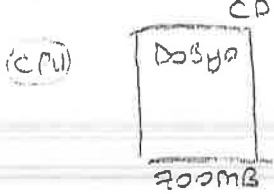
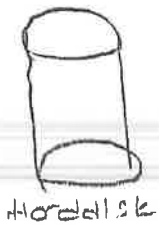
Herhangi bir dosyaya direkt ulaşmak için.  
CP, DVA, FID

Ör/ 20 tane dosyamız olsun. Sonraki dosyaya ulaşmak için sırasıyla  
çalıştırılır. Teyp tabanlı yapılardır. Sadeceleme amaçlıdır.



→ Dosyanın içindeki bir yere erişmek için bu iki yol kullanılır.

Pointer ya da referansın o satıra gitmesini istersin. Erişim hızı programın hızını etkiler. Random şekilde çalışır. Random ağaç yapısındaki erişimi destekler. Tüm bunlar yazılım kısmı. Donanım kısmı nasıl olur peki?



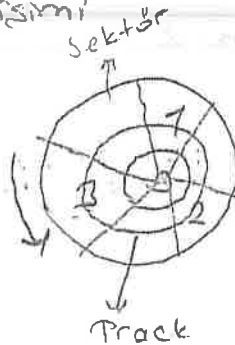
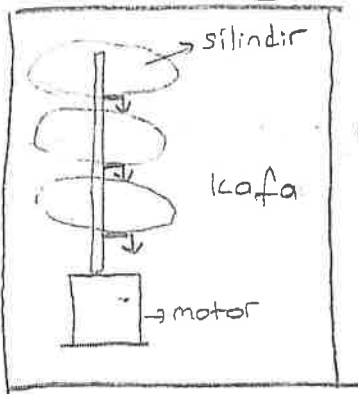
→ Bu dosyayı harddisk'e yazdırır. Çok hızlı şekilde gerçekleşmez çünkü harddisk fiziksel bir elemandır, yavaş çalışır.

CPU ne kadar hızlı olursa olsun, harddisk'e bağımlıyız.

Harddisk'e gönderdiğimiz bilgi doğru gönderildi mi?

- ① Bunun için eşlik biti kullanılır. Karşı taraf eşlik bitiyle karşılaştırılır. Yoksa kayıt olmaz.
- ② CRC + LRC kodları eklenerek yolların.

Harddisk'in doğrudan erişimi



2 sektör = 1 klasör

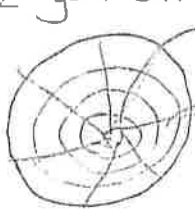
Sektör numaraları 1 atlatılarak verilir. Interleaving

Dönme hızı çok yüksek. 5400 rpm vs.

\* 1. yöntem

Hızlı olduğundan sektör yazılmadan kayabilir. Bunu engellemek için işletim sisteminde interleaving yapılır. Sektör numaraları 1'er atlanarak verilir.

\* 2. yöntem

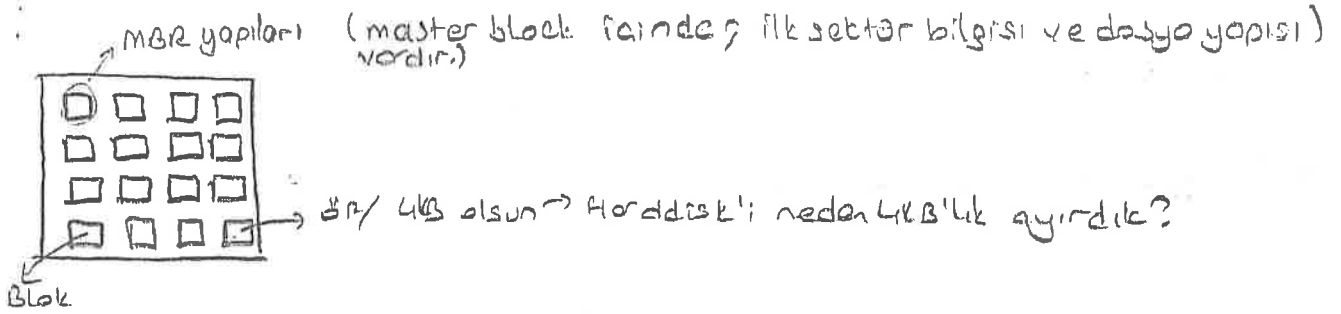


→ İç kısım 3'e bölünürse

→ Dış kısım 5'e bölünür.

Çünkü dış kısım daha çok bilgi yazılabilir.





Harddisk formatlama işleminden sonra, blocking olur. Windows tabanlı işletim sisteminde işletim sisteminin harddisk tanıması için yapılır.

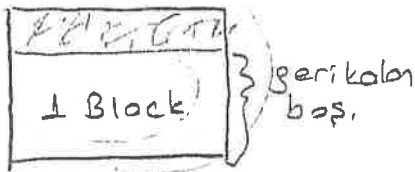
Diğer işletim sistemlerinde mapping dir.

blocking  
waiting

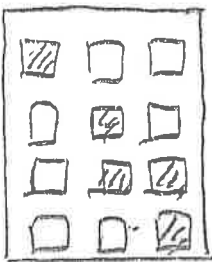
ör: Elimizde 100MB'lık bir harddisk var.

4KB'lık bölgeye  
8KB'lık "

Block sayısının az olması istenir. Optimumu bulmak için kullanıcılardan feedback alınır.

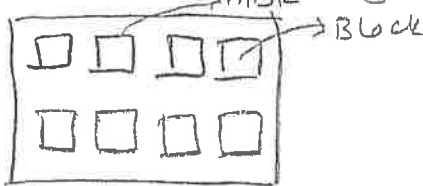


(Baş olarak kalması problemidir. Bu probleme internal fragmentation error (dahili parçalanma) denir.)



örneğin 4 block baş 5 block'la veriyor. Veri aradında yazılmak istendiğinde yazılıyor. Bu hataya harici parçalanma denir.

Disk Yerleştirme Algoritmaları



1-2 sektör → FAT (MBR)  
3-4 sektör → FAT'ın kopyası (MBlock)  
5-6 sektör → Dizin yapısı  
7- → Dosya tutar.

İşletim sistemi bir sektöre ulaşamıyorsa bir sonrakiye bakar.

1 sorun: Harddiskteki boş olan bloklar nasıl tutulacak?  
Bunu "Boş yer yönetimi" algoritması sağlar.

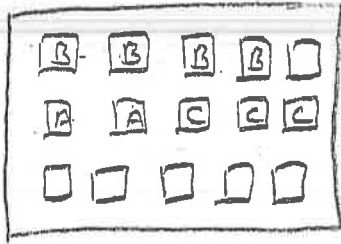
\* Boş yer yönetimi algoritması

10 block'luk bir harddisk' olsun. Her block için 1 bit tutulur.

bit seviyesinde bir dizi  $\rightarrow 10001101$   $\Rightarrow$  1 olanlar dolu, 0 boş.

Data sayısı arttıkça bilg. için sıkıntı alın, 2 tane pointer tutulur.  
Biri dizinin başını diğeri sonunu tutar. Bu sayede boş olan yerler doldururuz.

① Bitistik yerleştirme algoritması (Contiguous algoritması)



ör/ elimizde A, B, C, D dosyaları olsun

$\frac{A}{2}$   $\frac{B}{4}$   $\frac{C}{3}$   $\frac{D}{5}$   $\rightarrow$  Bayut (kaplanılan alan)


Herhangi bir t anında?

Yanyana yüklenir. Dosya oradaki bloğa sığacak mı?  
Sığsa bile boş olan kalıyor.

Harici parçalanmada etkilendiğini gösterir. Dahili parçalanma algoritmasında da etkilendir.

Peki nasıl yerleştireceğiz? Bunun için 3 tane yöntem vardır.

- First-fit yaklaşımı: En uygun ilk yere yerleştirir.
- worst-fit yaklaşımı: Boşlukta en büyük yere yerleştirir.  
(En fazla boşluk olana)
- Best-fit yaklaşımı: Kendisine en uygun olan yere yerleştirir.

  $\rightarrow$  Blok 4KB'lık olsun.

Veri 4.1KB ise internal fragmentation



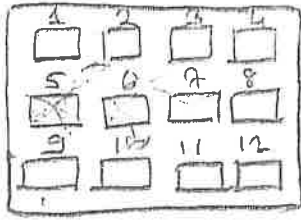
$\rightarrow$  Ard arda yerleştiremiyoruz  $\rightarrow$  Harici fragmentation

Harici parçalanmayı önlemek için compaction (öteleme) yapılır. Dosyaları aşağıya doğru mu yukarı doğru mu taşımak mantıklı? Bu alanın doluluğuna göre değişir.

Mümkün olduğu kadar az blok taşımak önemlidir.

Bitistik yerleştirme.  
indexlenmiş algoritma  
Bağı yerleştirme.

## ② Indexlenmiş Algoritma



Dizayn yapısı

7  
index Bloğu = 5

(Verisi yanyana kaydedilmesine gerek yok.)

10' dosyanın Harddisk yüzünde işaret ettiğini tutar.

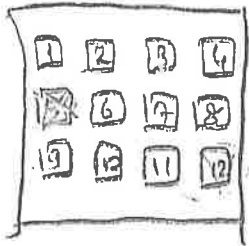
External fragmentation yok.  
Internal " " vardır.

Dezavantajı ; 1 Block'luk dosya için 2 block'luk alan tutulur.  
(1'inde index tutma bloğu)

Dosyalar çok büyük olduğunda index tutma bloğunda 1 bloğa sığ-  
mayabilir, 2 bloğa ihtiyaç duyulabilir.  
2 index arasındaki bağlantı için pointer kullanılır.

Dosya boyutu büyük olduğunda avantajlıdır.

## ③ Linked Allocation Method (Bağı yerleştirme Algoritması)



Dizayn yapısı

7

5 -> Dosyanın tutulduğu başlangıç yeri

6 -> Boyutu -> Neden tutulur? Harddiskte ne kadar boş yer olduğunu anlarak izin.

5 nolu blokta başlar 6 bloklu dosya yerleştirir.

FAT burada gerçekleştirir, Windows tabanlı yapılardır.

Bir blokta diğerine geçmek için pointer gerekir. Pointer dinamik-  
tir. Daha kompakttır. Kontrolü zordur.

Harddiske yüklerken elektrik giderse sonraki yere ulaşamayabiliriz.  
Öy sen 3 blok yazılmadı. Bunun için çift pointer kullanılır.

External fragmentation olmaz. Internal fragmentation hatası olabilir.

# 8 bitlik bir yapı kullanıyorsa, işletim sistemi 2<sup>8</sup> kadar alan-  
lara dosya adı verebiliriz.

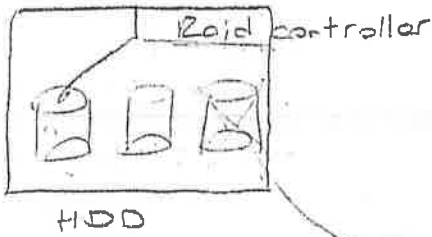


İşletim sistemi kısmına müdahale edemeyiz. İnterruptları kullanarak müdahale söz konusu olabilir.

Bu ailesner  
ama çok esnemez.

İşletim sistemi çalışırken veri yolu kullanır. CPU tarafı hızlıdır. ama harddisk tarafı yavaştır. Hızın ayarlanması için DMA kullanılır. Diğer aygıtlarla bağlantıyı sağlar.

### RAID (konusu)



1986'da (Paterson) Raid algoritmayı geliştirmiş.

Raid controller diğer parçalarla (harddisk) iletişime geçer. Buna göre veriyi yerleştirir.

Bir tane harddisk'te problem olsa veride fazla kayıp olmaz.

- \* Bir harddiskten bilgi diğerine kopyalanıpta yapılır.
- \* Birden fazla harddiskte bilgileri yaymaktır.
- \* Ya kopyalarını hepsine yayarız ya da bir tanesinde tutarız.

## Scheduling algoritmaları

### 1) FCFS

En basit zamanlama algoritmasıdır. Bu algoritmada proseler istek sıralarına göre işlemeide alınırlar.

- Gerçekleştirilmesi kolay
- Kesintisiz

Örnek!

Proses	İşleme süresi
P <sub>1</sub>	17ms
P <sub>2</sub>	5ms
P <sub>3</sub>	5ms

İşlemler yukarıdaki sırada hazır kuyruğunda bekletildiği kabul edilirse,



P<sub>1</sub> işlemi için bekleme süresi = 0  
P<sub>2</sub> " " " " = 17  
P<sub>3</sub> " " " " = 22

$$\frac{39}{3} = 13 \text{ ms} = \text{Ortalama bekleme süresi}$$

### 2) Round Robin (RR)

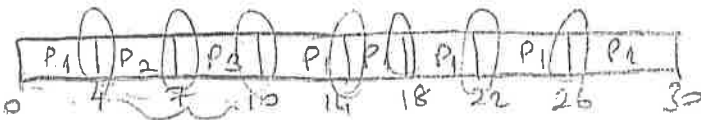
FIFO'ya göre proseler kuyruktadır, fakat her işlem için kuantum süresi denilen sabit bir işlem süresi ayrılır. Ve CPU'ya gelen her işlem bu sürede işlenir.

- Kesintili: (Preemptive)
- Performans kuantum süresinin büyüklüğüne bağlı.
- Özellikle zaman paylaşımlı sistemler için tasarlanmıştır.

Örnek!

Proses	İşleme süresi
P <sub>1</sub>	24ms
P <sub>2</sub>	3ms
P <sub>3</sub>	3ms

Kuantum süresinin 4ms seçildiğini farzedelim.



P<sub>1</sub> işlemi için bekleme süresi = 10 - 4 = 6

P<sub>2</sub> işlemi için " " = 4

P<sub>3</sub> işlemi " " = 7

$$\frac{17}{3} = 5.66 \text{ ms'dir.}$$

### 3.) Shortest-Job-First (SJF) → En kısa iş önce

Zamanlama, kuyrukta bulunan süreçlerden en kısa sürede tamamlanacak olan süreci seçer.

- Non-preemptive (kesintisiz)
- Ortalama bekleme süresi az
- Zaman paylaşımlı sistemler için kullanışlı değil.

Örnek:

Proses	İşleme süresi
P1	16ms
P2	3ms
P3	5ms



$$\begin{array}{l}
 P_1 \text{ için bekleme süresi} = 8 \\
 P_2 \text{ için } " = 3 \\
 P_3 \text{ için } " = 5 \\
 \hline
 \end{array}$$

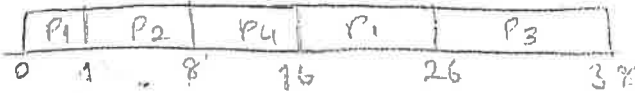
$$11/3 = 3.66ms$$

### 4.) Shortest-Remaining-Time (SRT) → En kısa işletim süresi kalan.

Bu algoritma, var olan zamanlama algoritmaları içinde, kuramsal olarak, ortalama bekleme süresi yönünden en iyi sonucu veren algoritmadır.

- Preemptive
- SJF'ye göre daha fazla yük.

Proses	Varış zamanı	İşleme süresi
P1	0	14
P2	1	7
P3	2	12
P4	3	8



$$\begin{array}{l}
 P_1 \text{ için bekleme süresi } 16 - 1 = 15 \\
 P_2 \text{ için } " = 0 \\
 P_3 \text{ için } " = (26 - 2) = 24 \\
 P_4 \text{ için } " = (8 - 3) = 5 \\
 \hline
 \end{array}$$

$$44/4 = 11ms$$

### 5) Priority (Öncelik)

Bu algoritmada her görevin bir önceliği bulunur. Yani bir işlem kısımlarını zaman en yüksek önceliğe sahip görev seçilir.

Örnek:

Proses	İşlem süresi	Öncelik
P <sub>1</sub>	7	3
P <sub>2</sub>	10	1
P <sub>3</sub>	6	4
P <sub>4</sub>	15	2
P <sub>5</sub>	6	0

\* Yüksek öncelikli prosesler çok uzun süre katabilir. Bundan dolayı diğer bazı proseslerin kısımları aşırı geciktirilir. Bu nedenle, belirli zaman aralıklarında bu proseslerin öncelikleri düşürülmelidir.

P <sub>5</sub>	P <sub>2</sub>	P <sub>4</sub>	P <sub>1</sub>	P <sub>3</sub>
0	6	16	31	38
				44

\* Kısa süren işlerde ve I/O tipi işlerde kullanılır.

P<sub>1</sub> için bekleme süresi = 31 ms

P<sub>2</sub> " " " = 6 ms

P<sub>3</sub> " " " = 38

P<sub>4</sub> " " " = 16

P<sub>5</sub> " " " = 0

+

$$91/5 = 18.2 \text{ ms}$$

Deadlock = "ölümcül kilitlenme". Sistem kaynaklarını ortak olarak kullanan veya birbirini ile haberleşen bir grup prosesin kalıcı olarak bloke olması durumudur.

Ölümcül kilitlenme olması için gereken koşullar

→ Kısıtlı diploma

→ Proseslerin eline geçirdikleri kaynakları diğer istedikleri kaynakları da ele geçirene kadar bırakmamaları

→ Proseslerin ellerinde tuttukları kaynaklar işletim sistemi tarafından zorla geri alınmıyorsa. ("pre-emption" yok.)

→ Çevrel bekleme durumu oluşuyorsa

\* Ölümcül kilitlenme durumunda kullanılan yaklaşımlar

→ Sistemin hiçbir zaman ölümcül kilitlenme durumuna girmemesini sağlamak

→ Sistemin ölümcül kilitlenme durumuna girdikten sonra bu durumdan kurtulmasını sağlamak.

→ Problemi gözardı edip sistemde ölümcül kilitlenme olmayacağını varsaymak

→ Sistemin hiçbir zaman ölümcül kilitlenme durumuna girmemesini sağlamak.

\* Bir proses başka bir proses tarafından elinde tutulan bir kaynağı isterse, ikinci prosesin kaynaklarını bırakması işletim sist. tarafından sağlanabilir.

## Ölümçül Kilitlenme Sızıldıktan Sonra Yapılabilecekler

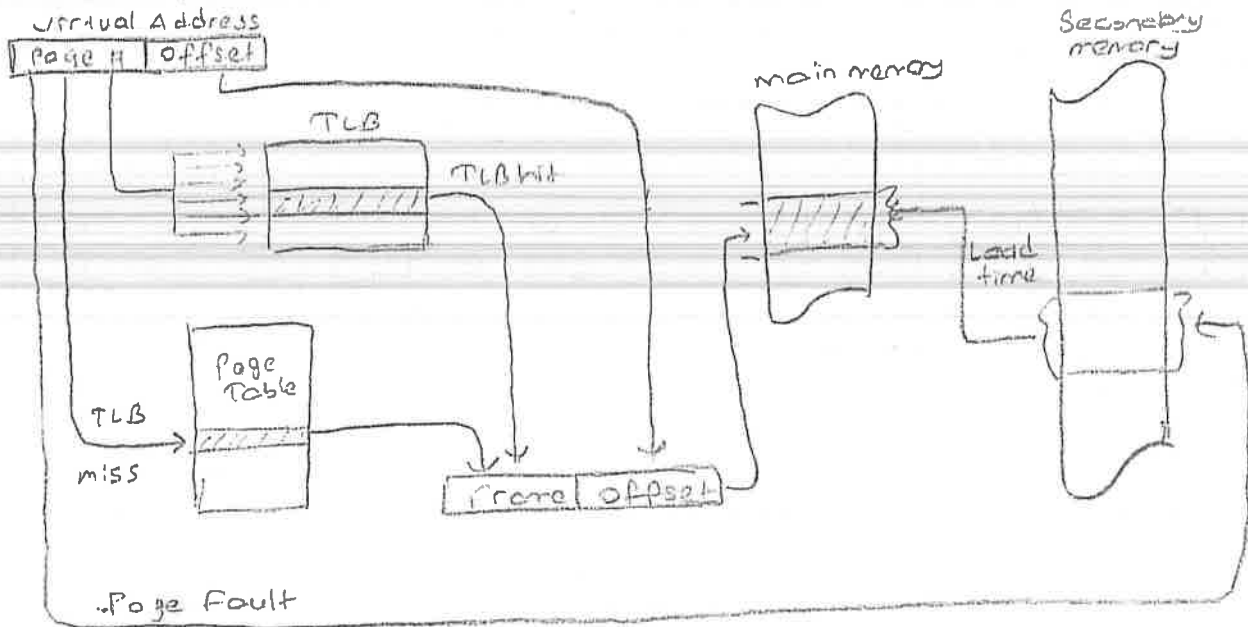
- Tüm kilitlenmiş süreçleri sonlandırır.
- Tüm kilitlenmiş süreçlerin eski bir kontrol noktasına kadar kopyasını al. ve tüm süreçleri bu noktadan yeniden başlat.
- Aynı ölümçül kilitlenme yeniden oluşabilir.
- Kilitlenme ortadan kalkana kadar sırayla kilitlenmiş süreçleri sonlandırır.
- Kilitlenme ortadan kalkana kadar, sırayla atamış kaynakları geri al.

## Kilitlenmiş Süreçler İçin Seçim Kriterleri

- O ana kadar en az işlemci zamanı kullanmış olan.
- O ana kadar en az sayıda çıktı satırı oluşturmuş olan.
- Beklenen çalışma süresi en uzun olan
- En düşük öncelikli olan

## Translation Lookaside Buffer (TLB)

- En yakın zamanda kullanılmış olan sayfa tablosu kayıtlarını tutar.
- Ana bellek için kullanılan cep bellek yapısına benzer bir işlev görür.

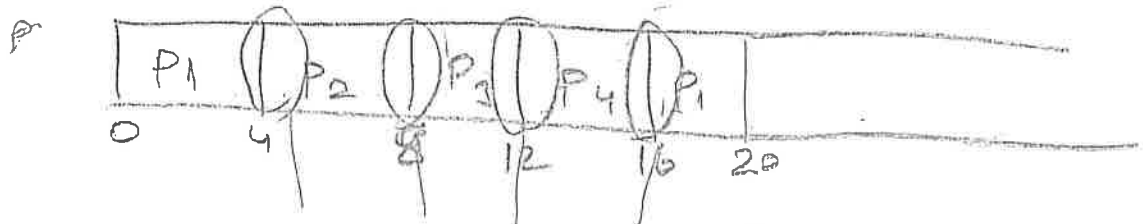


**Sayfa Boyu:** Sayfa boyu küçük olunca bellekteki sayfa sayısı artar. Zaman içinde süreçlerin yakın zamanda eriştikleri sayfaların büyük kısmı bellekte olur. Sayfa hatası düşük olur.

Sayfa boyu büyük olunca sayfalarda yakın zamanlı erişimlere uzak kısımlar da olur. Sayfa hataları artar.



Kuantum = 4ms



context switch sayısı (Aradıkları zamanları say)

