İşlemci Zamanlaması

İşlemci Zamanlaması (Scheduling)

- Temel Kavramlar
- Zamanlama Kriteri
- Zamanlama Algoritmaları
- İş Parçacığı Zamanlaması
- Çok-İşlemci Zamanlaması
- İşletim Sistemi Örnekleri
- Algoritmaların Değerlendirilmesi

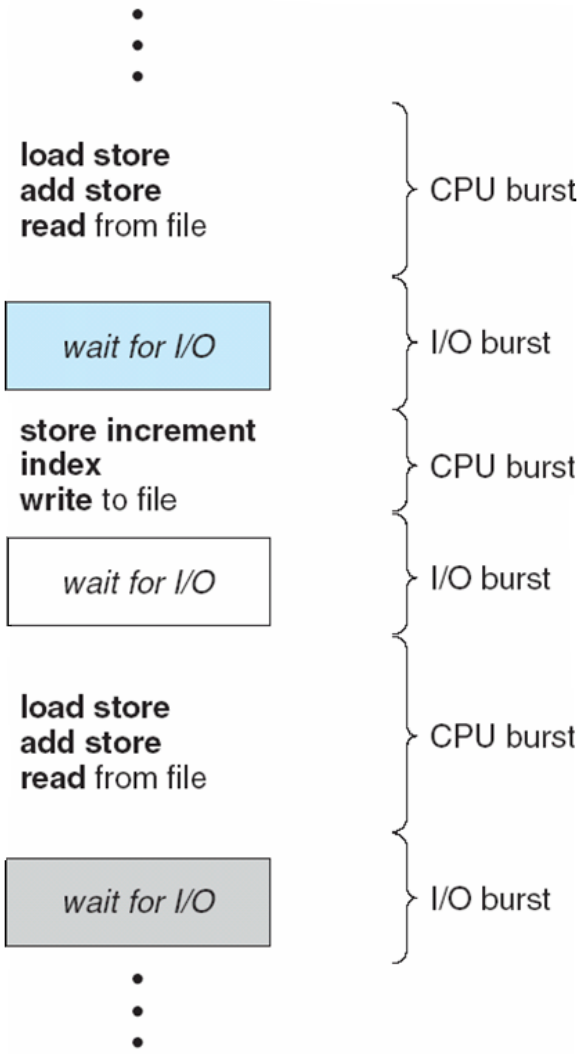
Hedefler

- Multi-programming işletim sistemlerinin temelini oluşturan işlemci zamanlamasının tanıtılması
- Pek çok işlemci zamanlama algoritmasının açıklanması
- Belirli bir sistem için işlemci zamanlama algoritması seçerken kullanılacak değerlendirme kriterlerinin irdelenmesi

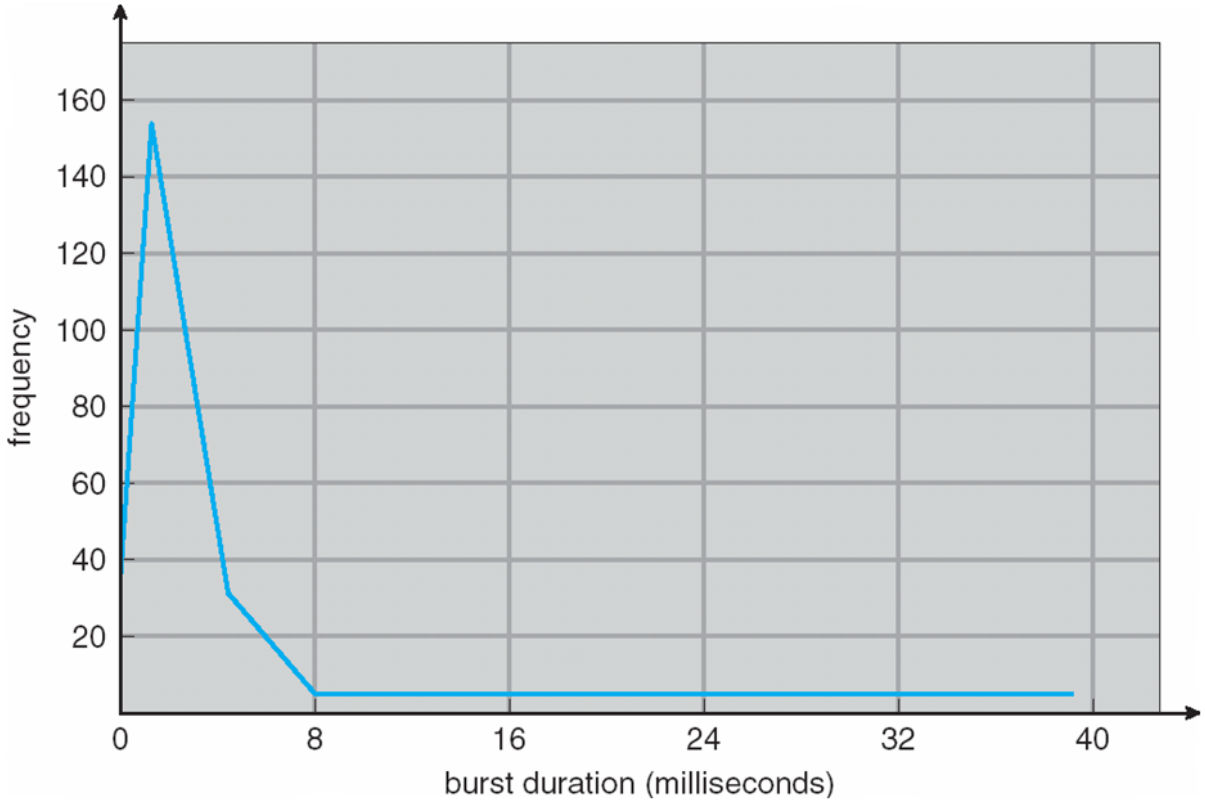
Temel Kavramlar

- Multi-programming sayesinde CPU kullanımının optimize edilmesi
- **CPU–I/O İşlem Döngüsü** – Bir işlemin çalıştırılması birbirlerini takip eden (1) CPU kullanımı ve (2) I/O beklemesi döngüsüden oluşur
- **CPU işleme süresi** dağılımı

Tekrar Eden CPU ve I/O İşlem Döngüsü



CPU İşleme Süresi Dağılım Eğrisi



Histogram: Dağılım Eğrisi

İşlemci Zamanlayıcısı

- Çalışmaya hazır şekilde hafızada bekleyen işlemlerden birini seçerek işlemciyi ona ayırır

- Scheduling şu durumlarda gerçekleşebilir:
  1. Çalışma durumundan (running state) bekleme durumuna (waiting state) geçişte
  2. Çalışma durumundan hazır durumuna (ready state) geçişte
  3. Bekleme durumundan hazır durumuna geçişte
  4. İşlem sonlandığında

- 1 ve 4 durumlarında zamanlama: **kesmeyen (nonpreemptive)** – eski işletim sistemleri (windows 3.x gibi)

- Diğer durumlarda: **kesen (preemptive)**

Dispatcher

- Dispatcher modül CPU kontrolünü kısa dönem zamanlayıcı (short-term scheduler) tarafından seçilen işleme devreder:
  - Kapsam değiştirme (switching context)
  - User moduna geçme
  - Usera ait program yeniden başlatıldığında programdaki uygun konuma atlama
- **Dispatcher gecikme süresi (dispatch latency)** – Dispatcherın bir programı sonlandırıp diğerini başlatması için gereken süre

Zamanlama Kriterleri

- **İşlemci kullanımı (CPU utilization)** – İşlemciyi olabildiğince meşgul tutmak
- **Üretilen iş (throughput)** – birim zamanda çalışması sonlanan processlerin sayısı
- **Devir zamanı (turnaround time)** – bir işlem sonlanana kadar geçen toplam zaman
- **Bekleme zamanı (waiting time)** – bir işlemin hazır kuyruğunda (ready queue) toplam bekleme zamanı
- **Yanıt süresi (response time)** – bir isteğin gönderilmesi ile bu isteğin yanıtının verilmesi arasında geçen zaman

Zamanlama Algoritması Optimizasyon Kriterleri

- Maksimum işlemci kullanımı (Max CPU utilization)
- Maksimum üretilen iş (Max throughput)
- Minimum devir zamanı (Min turnaround time)
- Minimum bekleme zamanı (Min waiting time)
- Minimum yanıt süresi (Min response time)

1. First-Come, First-Served (FCFS) Scheduling

İşlem	İşlem Süresi
P_1	24
P_2	3
P_3	3

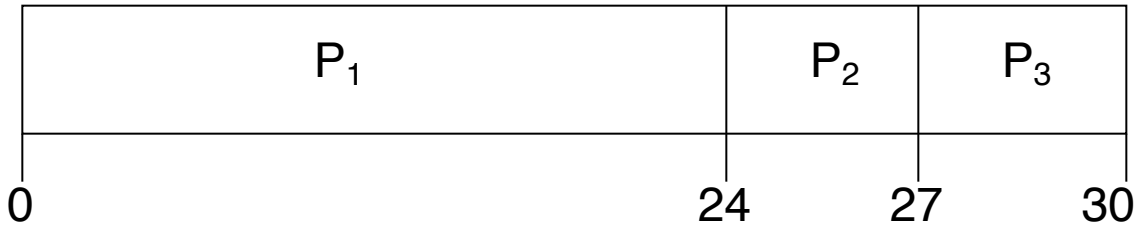
İlk Gelen Önce – İşlemci Zamanlama Algoritması

First-Come, First-Served (FCFS) Scheduling

- İşlemleri şu sırada geldiğini varsayalım: P_1 , P_2 , P_3
  Zamanlama için Gantt şeması:

- Bekleme zamanları: P_1 = 0; P_2 = 24; P_3 = 27

- Ortalama bekleme zamanı: (0 + 24 + 27)/3 = 17

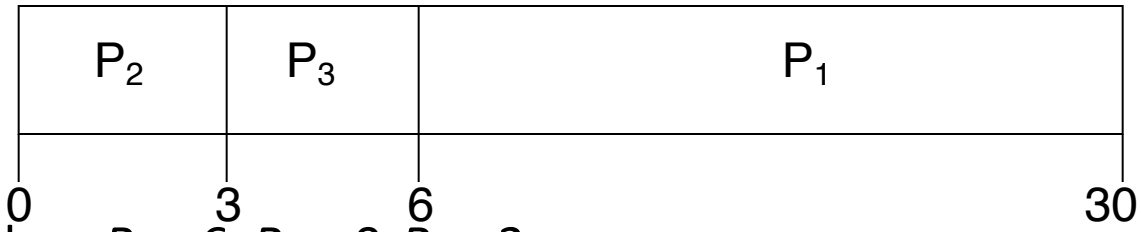


1. FCFS (Devam)

İşlemleri şu sırada geldiğini varsayalım:

$$P_2 , P_3 , P_1$$

- Zamanlama için Gantt şeması:



- Bekleme zamanları: $P_1 = 6; P_2 = 0, P_3 = 3$
- Ortalama bekleme zamanı: (6 + 0 + 3)/3 = 3
- Önceki durumdan çok daha iyi
- **Konvoy etkisi (convoy effect):** kısa işlemler uzun işlemlerin arkasında

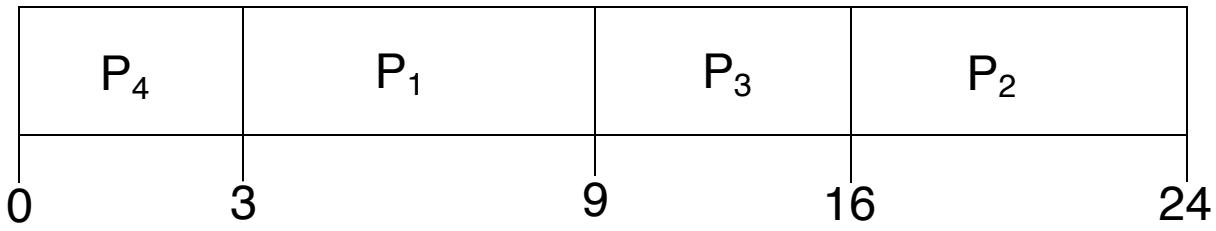
2. En Kısa İş Önce - SJF İşlemci Zamanlama Algoritması

- Shortest-Job-First (SJF)
- Her işlemi sonraki işlemci kullanım süresi ile ilişkilendirilmeli
- Bu kullanım sürelerini kullanarak en kısa sürecek iş önce seçilmeli

- SJF optimal zamanlama algoritmasıdır – verilen bir iş kümesi için minimum ortalama bekleme süresini sağlar
  - Zorluk, işlemci kullanım sürelerini tahmin etmektir

SJF Örneği

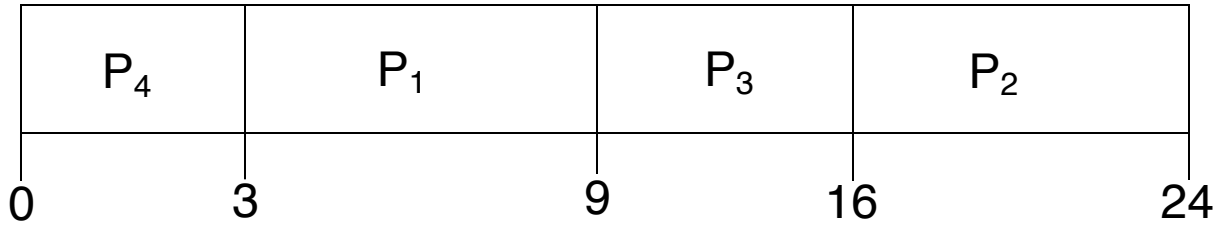
İşlem	İşlem Süresi
P_1	6
P_2	8
P_3	7
P_4	3



SJF Örneği

İşlem	İşlem Süresi
P_1	6
P_2	8
P_3	7
P_4	3

- SJF zamanlama şeması



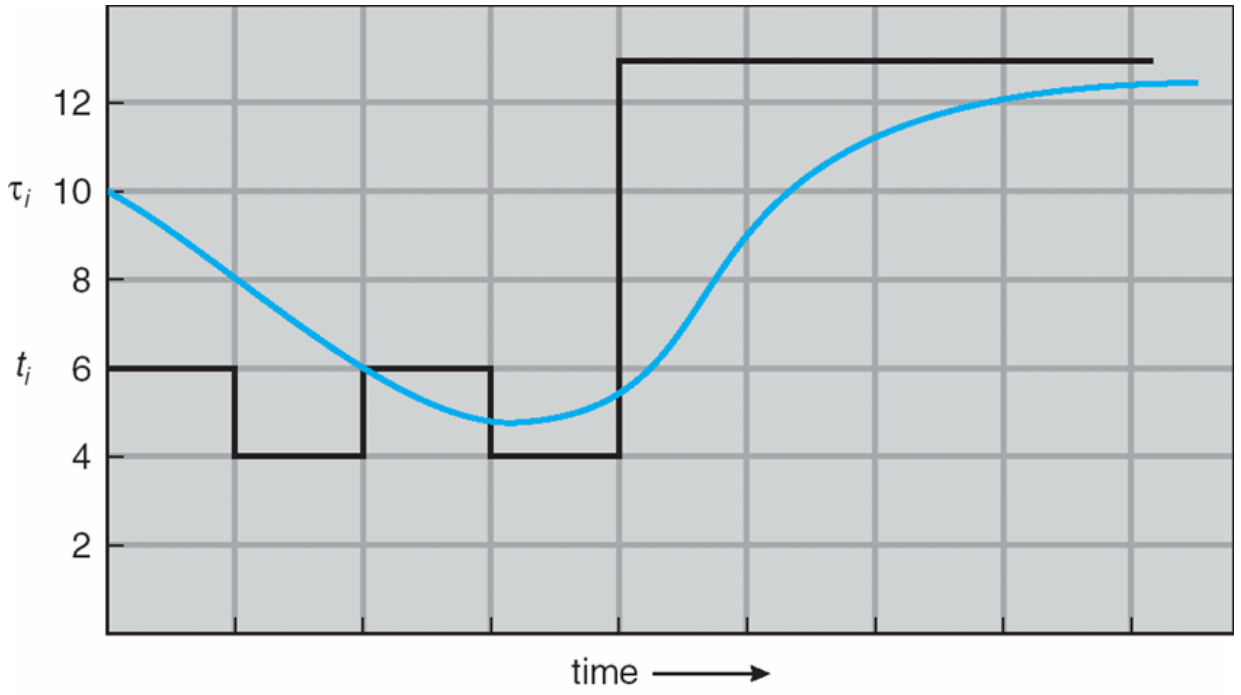
- Ortalama bekleme zamanı = (3 + 16 + 9 + 0) / 4 = 7

İşlemci Kullanım Süresinin Belirlenmesi

- Sadece tahmin edilebilir
- Daha önceki işlemci kullanım süreleri kullanılarak **üssel ortalama (exponential averaging)** yöntemiyle tahmin edilebilir

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n$.

Üssel Ortalama ile Kullanım Süresi Belirleme



CPU burst (t_i)		6	4	6	4	13	13	13 ...
"guess" (τ_i)	10	8	6	6	5	9	11	12 ...

$$\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n = (t_n + \tau_n)/2$$

Üssel Ortalama Örnekleri

- α =0
  - $\tau_{n+1} = \tau_n$
  - Yakın zaman bilgisi kullanılmıyor
- α =1
  - $\tau_{n+1} = \alpha\, t_n$
  - Sadece son CPU kullanım bilgisi hesaba katılıyor
- Formülü genişletirsek:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, \tau_{n-1} + \ldots$$
$$+(1-\alpha)^j \alpha\, \tau_{n-j} + \ldots$$
$$+(1-\alpha)^{n+1} \tau_0$$

- Hem α hem de (1 - α) 1 veya birden küçük olduğundan, her bir takip eden terim, kendinden önce gelen terimden daha az ağırlığa sahip

3. Priority Tabanlı – Scheduling Algoritması

- Priority Scheduling
- Her bir işleme bir öncelik sayısı (tamsayı) atanır

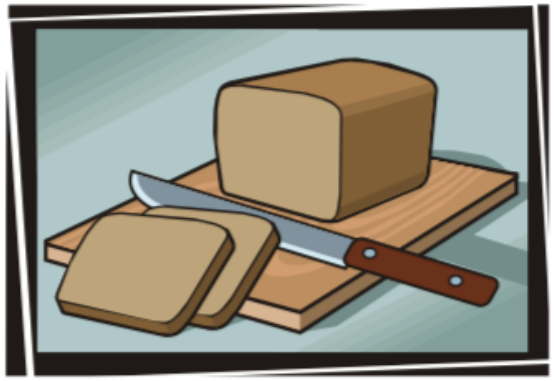
- İşlemci en öncelikli işleme verilir (en küçük tamsayı ≡ en yüksek öncelik)
  - Kesen (preemptive)
  - Kesmeyen (nonpreemptive)

- SJF, öncelik tahmini kullanım süresi olmak kaydıyla, öncelik tabanlı bir işlemci zamanlaması algoritmasıdır

- Problem ≡ **açlık (starvation)** – düşük öncelikli işlemler hiçbir zaman çalışmayabilir

- Çözüm ≡ **yaşlandırma (aging)** – taman geçtikçe bekleyen işlemlerin önceliğini arttırma

4. Round Robin Scheduling



- Her bir işlem işlemci zamanının küçük bir birimini alır: zaman kuantumu (time quantum)
- Genellikle 10-100 millisaniye
- Bu zaman dolduğunda işlem kesilir ve hazır kuyruğunun sonuna eklenir
- Eğer hazır kuyruğunda *n* tane işlem varsa ve zaman kuantumu *q* ise, her bir işlem CPU zamanının 1/*n* kadarını (en fazla *q* birimlik zamanlar halinde) ve hiç bir işlem (*n*-1)*q* zaman biriminden fazla beklemez
- Performans
  - *q* büyükse⇒ ilk gelen önce (FIFO)
  - *q* küçükse ⇒ *q* context switch süresine oranla daha büyük olmalıdır. Aksi halde sistem verimsiz çalışır

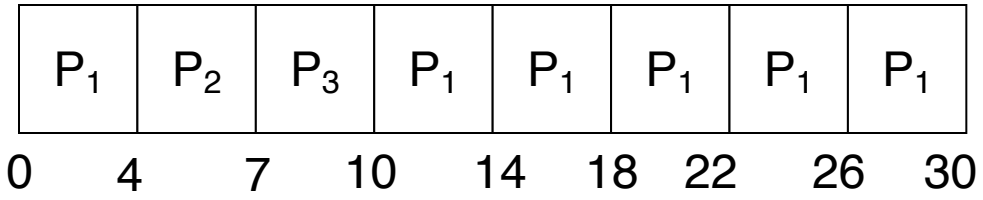
Zaman Kuantum = 4 olduğunda RR

İşlem	İşlem Süresi
P_1	24
P_2	3
P_3	3

Zaman Kuantum = 4 olduğunda RR

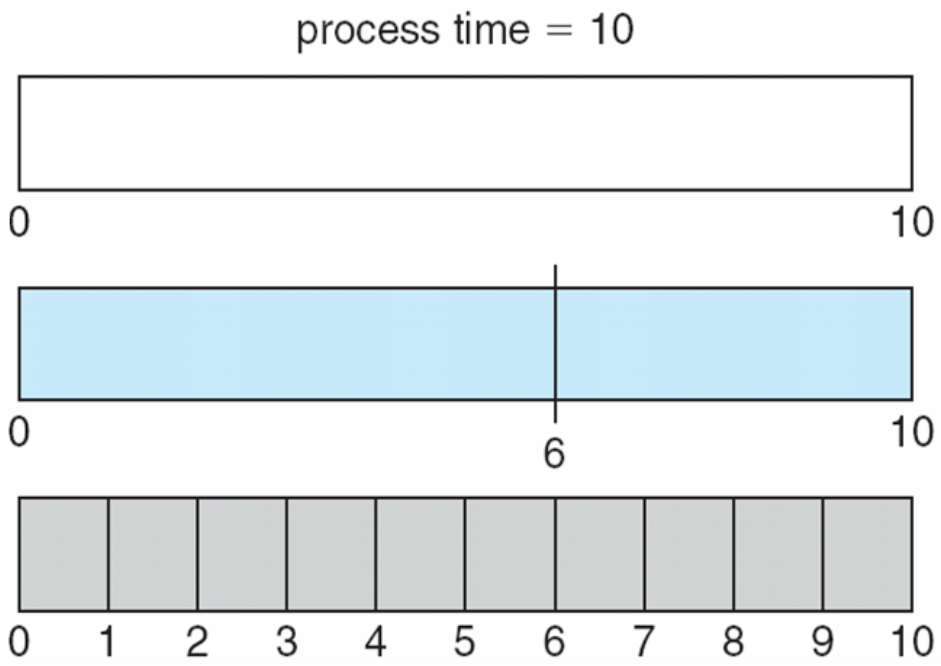
İşlem	İşlem Süresi
P_1	24
P_2	3
P_3	3

- Gantt şeması:



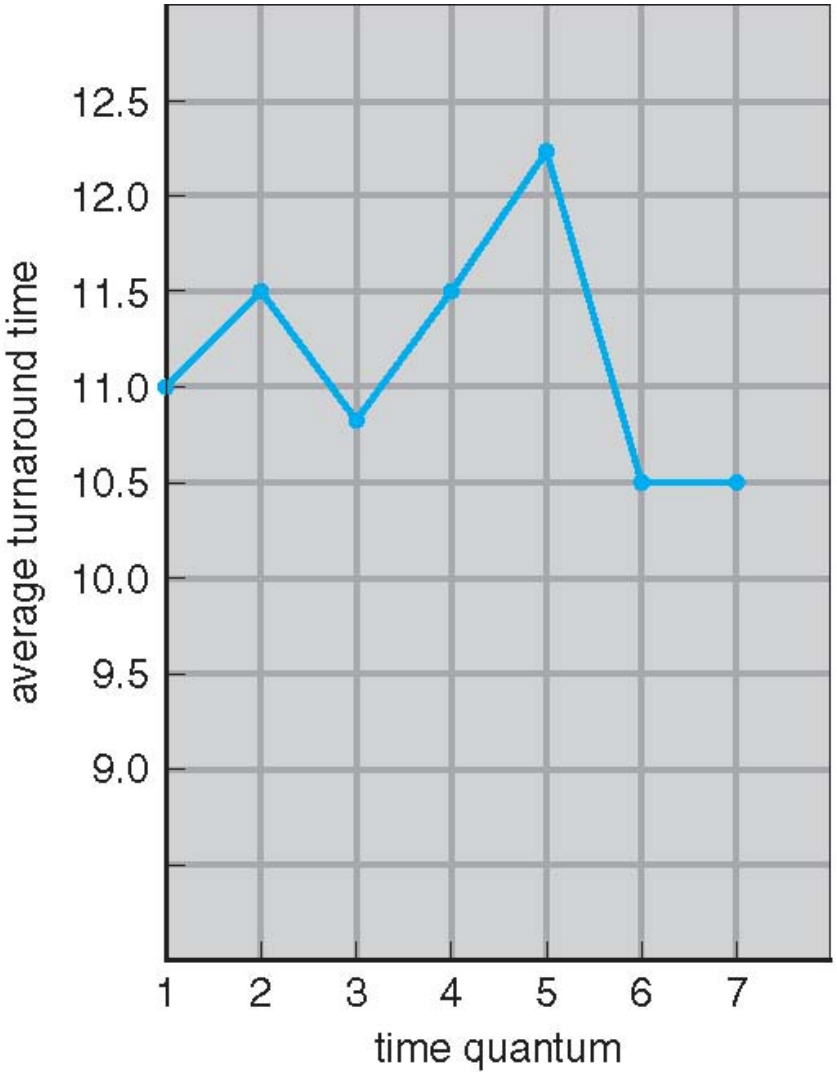
- Tipik olarak, SJF'den daha yüksek turnaround süresi, ama daha iyi *response time.*

Zaman Kuantumu ve Ortam Değiştirme Süresi



quantum	context switches
12	0
6	1
1	9

Turnaround Time Zaman Kuantumu ile Değişir



process	time
P_1	6
P_2	3
P_3	1
P_4	7

5. Çok kuyruklu - İşlemci Zamanlama Algoritması
Multilevel Queue

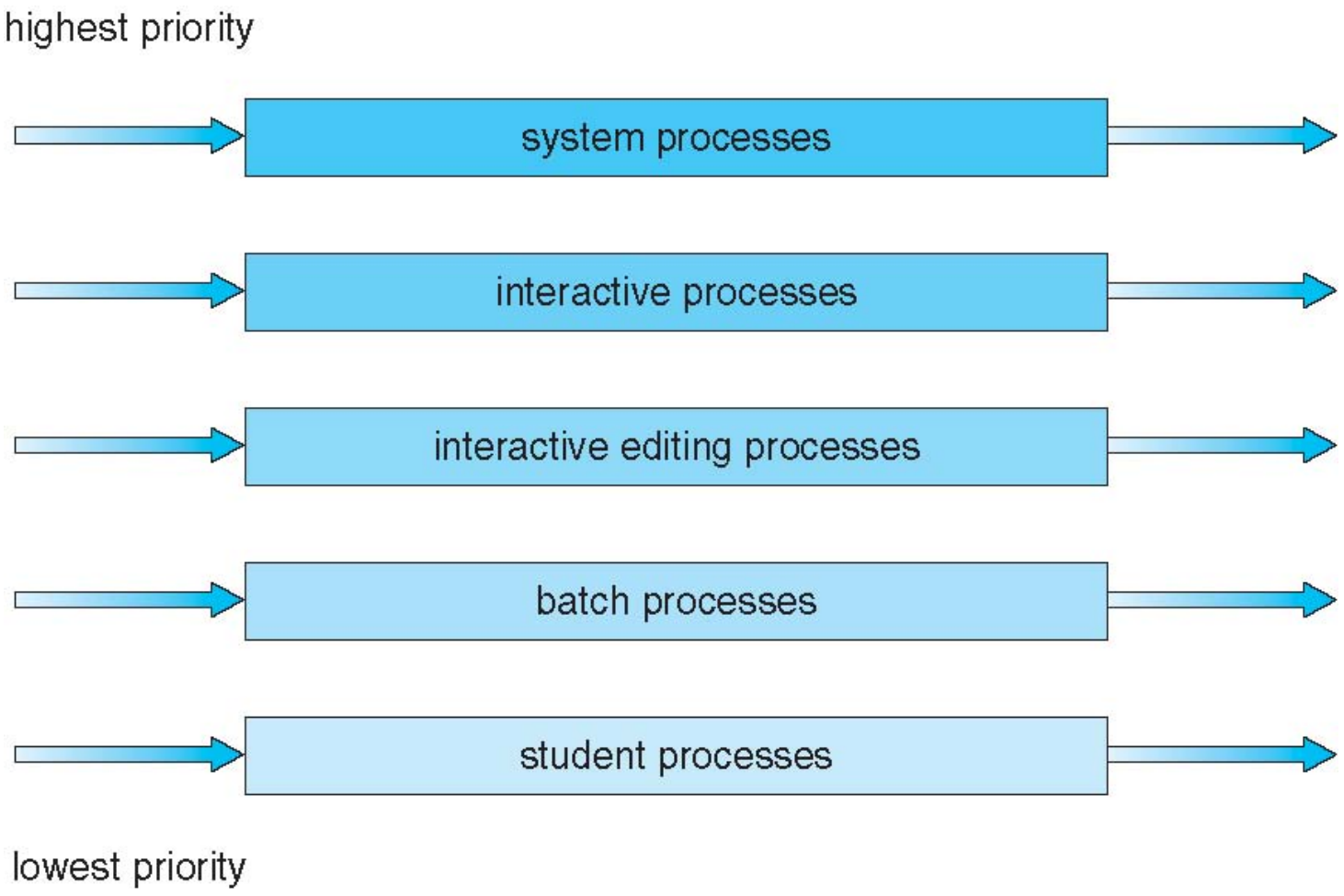
- Multilevel Queue
- Hazır kuyruğu: Ready birden fazla kuyruğa bölünür:
  ön plan (foreground – interaktif (interactive)
  arka plan (background) – toplu iş (batch)

- Her kuyruk kendine ait işlemci zamanlama algoritmasına sahiptir
  - **ön plan** – RR
  - **arka plan** – FCFS

Multilevel queue (devam)

- Planlama kuyruklar arasında yapılmalıdır.
  - Sabit öncelikli zamanlama (fixed priority scheduling)
  - Örn: önce tüm ön plan işlerini çalıştırıp ardından arka plan işlerini çalıştırmak. Olası açlık (starvation)
  - Zaman dilimi – her kuyruk CPU'nun belirli bir zamanını kendine ayırabilir.
  - Örn: CPU'nun %80 zamanı RR ile ön plan işlerine %20'si ise FCFS ile arka plan işlerine ayrılabilir.

Çok Kuyruklu Zamanlama



Çok-seviye Geri Besleme Kuyruğu

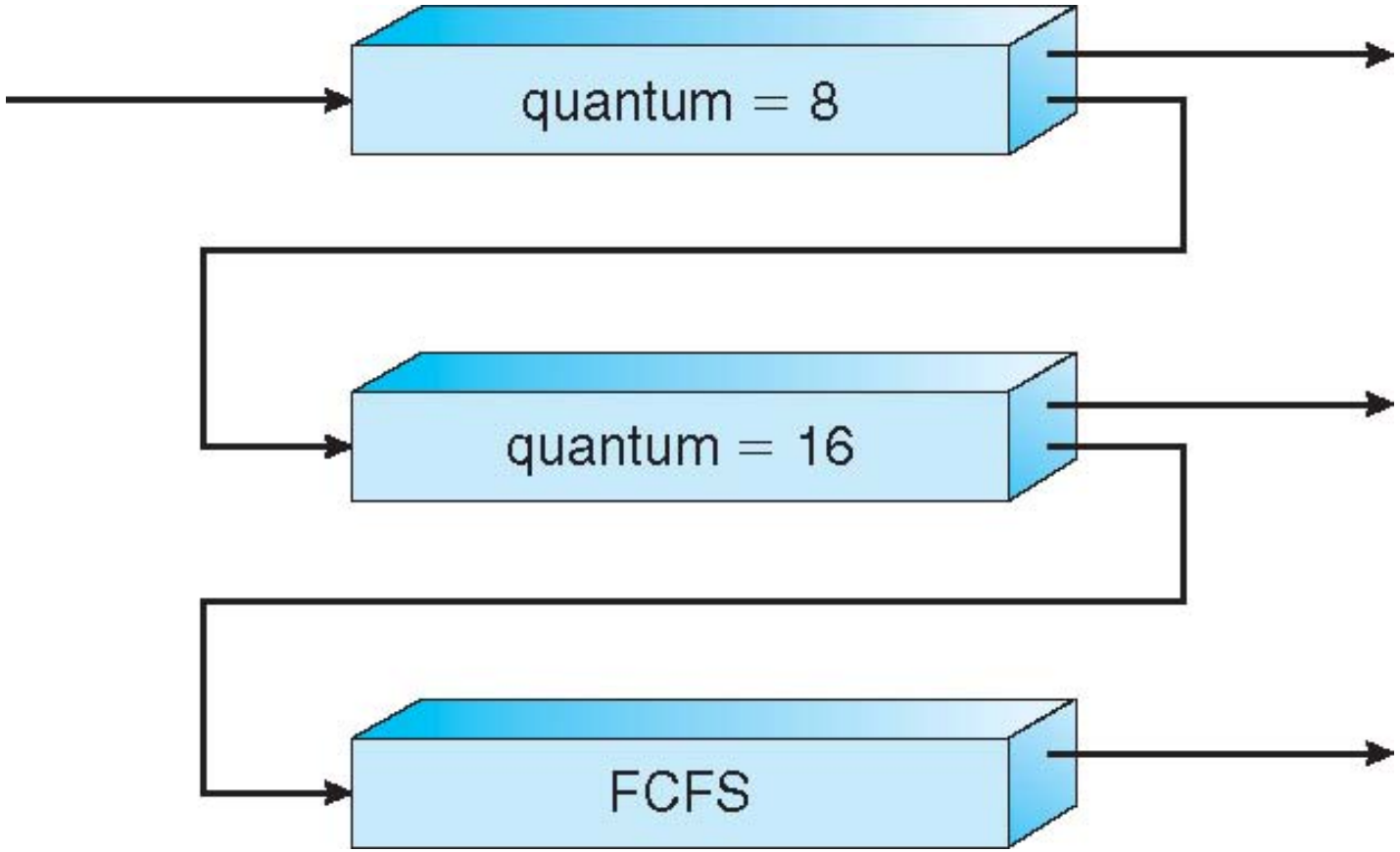
- Multilevel Feedback Queue
- İşlemler kuyruklar arasında yer değiştirebilir; yaşlanma (aging) bu şekilde gerçekleştirilebilir

- Çok-seviye geri besleme kuyruğu zamanlayıcısı aşağıdaki parametrelerle tanımlanır:
  - Kuyrukların sayısı
  - Her bir kuyruk için scheduling algoritması
  - Bir işlemin üst kuyruğa ne zaman alınacağını belirleme yöntemi
  - Bir işlemin alt kuyruğa ne zaman alınacağını belirleme yöntemi
  - Bir işlem çalıştırılmak için seçildiğinde hangi kuyruğa alınacağını belirleyen yöntem

Çok-seviye Geri Besleme Kuyruğu Örneği

- Üç kuyruk:
  - Q_0 – zaman kuantumu 8 milisaniye olan RR
  - Q_1 – zaman kuantumu 16 milisaniye olan RR
  - Q_2 – FCFS
- Zamanlama
  - Yeni işler Q_0 kuyruğuna eklenir ve FCFS ile yönetilir
  - CPU'yu elde ettiğinde bu işe 8 milisaniye verilir. Eğer 8 milisaniyede sonlanmazsa Q_1 kuyruğuna alınır
  - İş Q_1 kuyruğunda yeniden zamanlanır. Sıra ona geldiğinde 16 milisaniye ek süre verilir. Hala sonlanmazsa, çalışması kesilir ve Q_2 kuyruğuna alınır ve burada FCFS yöntemiyle zamanlanır

Çok-seviye Geri Besleme Kuyruğu



Thread Scheduling

- Kullanıcı-seviyesi ve çekirdek-seviyesi iş parçacıkları arasında ayrım
- Çoktan-teke ve çoktan-çoka modellerde, threadkütüphanesi kullanıcı seviyesi iş parçacıklarını LWP üzerinde sırayla çalışacak şekilde zamanlar
  - İşlem içerisinde zamanlama rekabeti olduğundan **işlem-çekişme alanı** olarak adlandırılır
    - **process-contention scope (PCS)**
- Çekirdek seviyesi iş parçacıkları ise sistemdeki tüm diğer iş parçacıkları ile zamanlama rekabeti içinde olduğundan **sistem çekişme alanı** olarak adlandırılır
  - **system-contention scope (SCS)**

Pthread Zamanlaması

- Pthread API'si, iş parçacığı oluşturulurken, zamanlamanın PCS veya SCS olarak ayarlanmasına izin verir
  - PTHREAD_SCOPE_PROCESS, iş parçacıklarını PCS ile zamanlar
  - PTHREAD_SCOPE_SYSTEM ise iş parçacıklarını SCS ile zamanlar

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
  int i;
  pthread_t tid[NUM THREADS];
  pthread_attr_t attr;

  /* get the default attributes */
  pthread_attr_init(&attr);

  /* set the scheduling algorithm to PROCESS or SYSTEM */
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* set the scheduling policy - FIFO, RT, or OTHER */
  pthread_attr_setschedpolicy(&attr, SCHED_OTHER);

  /* create the threads */
  for (i = 0; i < NUM THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
```

Pthread Scheduling API

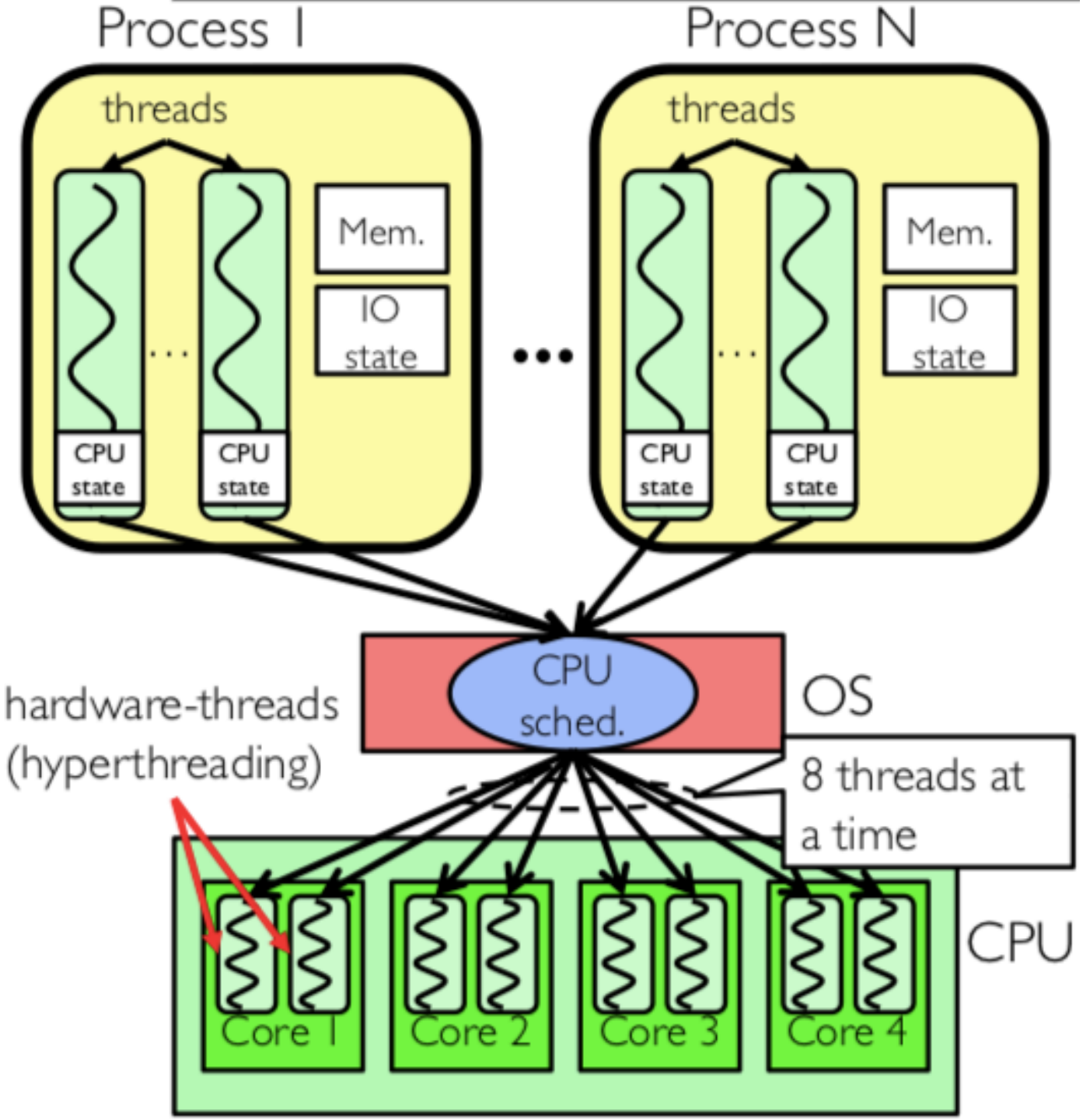
```
  /* now join on each thread */
  for (i = 0; i < NUM THREADS; i++)
      pthread_join(tid[i], NULL);
}
 /* Each thread will begin control in this
  function */
void *runner(void *param)
{
  printf("I am a thread\n");
  pthread exit(0);
}
```

HyperTreading

```
Loop {
    RunThread();
    newTCB = ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

Context Switch

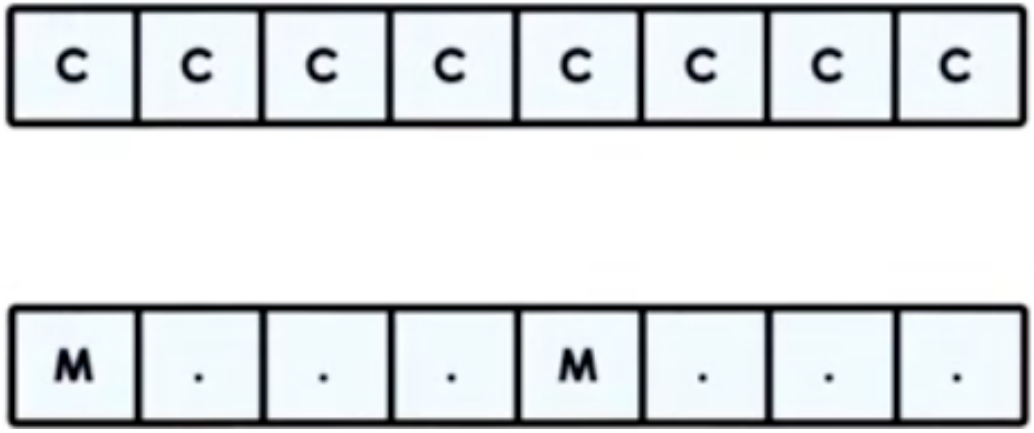
```
switch(tCur,tNew) {
   /* Unload old thread */
   TCB[tCur].regs.r7 = CPU.r7;
            …
   TCB[tCur].regs.r0 = CPU.r0;
   TCB[tCur].regs.sp = CPU.sp;
   TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

   /* Load and execute new thread */
   CPU.r7 = TCB[tNew].regs.r7;
            …
   CPU.r0 = TCB[tNew].regs.r0;
   CPU.sp = TCB[tNew].regs.sp;
   CPU.retpc = TCB[tNew].regs.retpc;
   return; /* Return to CPU.retpc */
}
```


Process 1
Process N
threads
threads
Mem.
IO state
Mem.
IO state
CPU state
CPU state
CPU state
CPU state
CPU sched.
OS
hardware-threads (hyperthreading)
8 threads at a time
CPU
Core 1
Core 2
Core 3
Core 4

HyperTreading

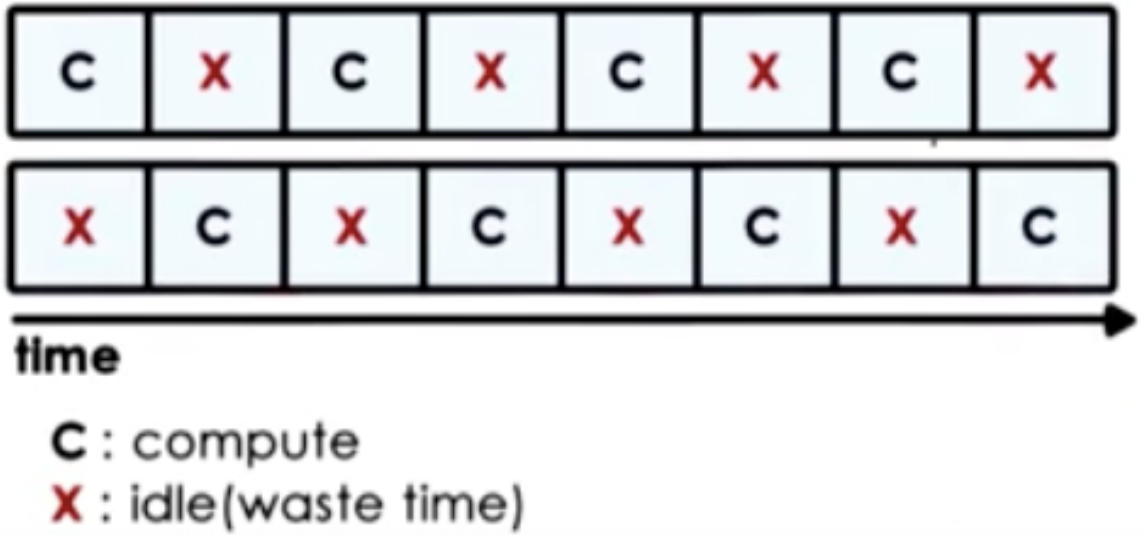
- Intel’in Symmetric Multithreading (SMT) implement etmesi (1999’da Compaq firması).
- İki thread eş zamanlı tek bir CPU’da çalışıyor.
- Çok hızlı context switch
- Diğer isimleri;
  - Hardware multithreading
  - Chip multithreading
  - Simultaneous Multithreading

HyperTreading



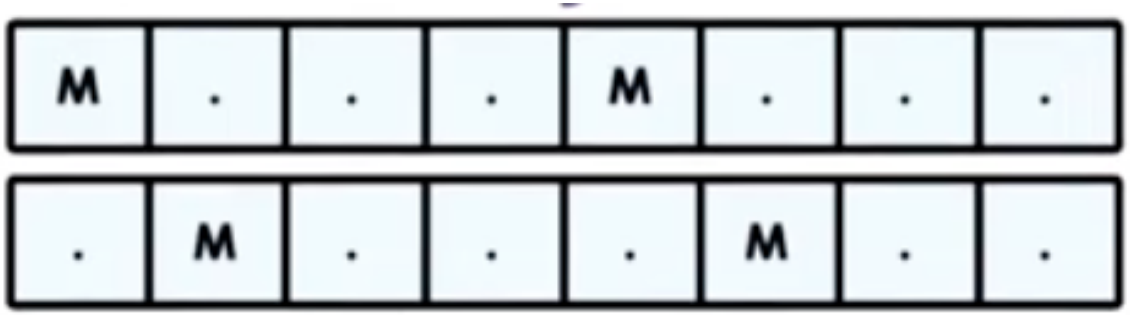
- Varsayımlar
- 1 cycle da maksimum çalışan komut sayısı
- IPC=1
- Memory erişimi: 4 cycle

HyperTreading (iki CPU-bound thread için)



- Sonuç
- Thread başına kötüleşme 2x

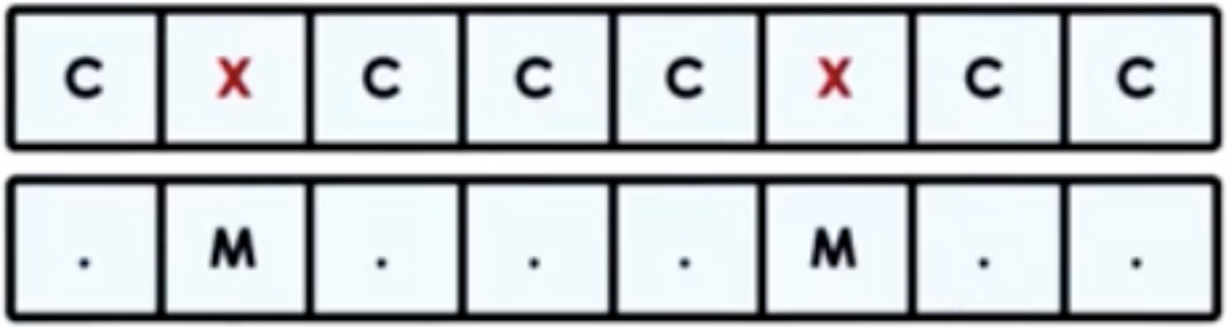
HyperTreading (iki memory-bound thread için)



M: compute
. : idle(waste time)

- Sonuç
- Boşa giden CPU cycleları

HyperTreading (bir memory-bound, bir CPU-bound thread için)



- Sonuç
- CPU ve Memory etkin bir şekilde kullanılıyor

İşletim Sistemi Örnekleri

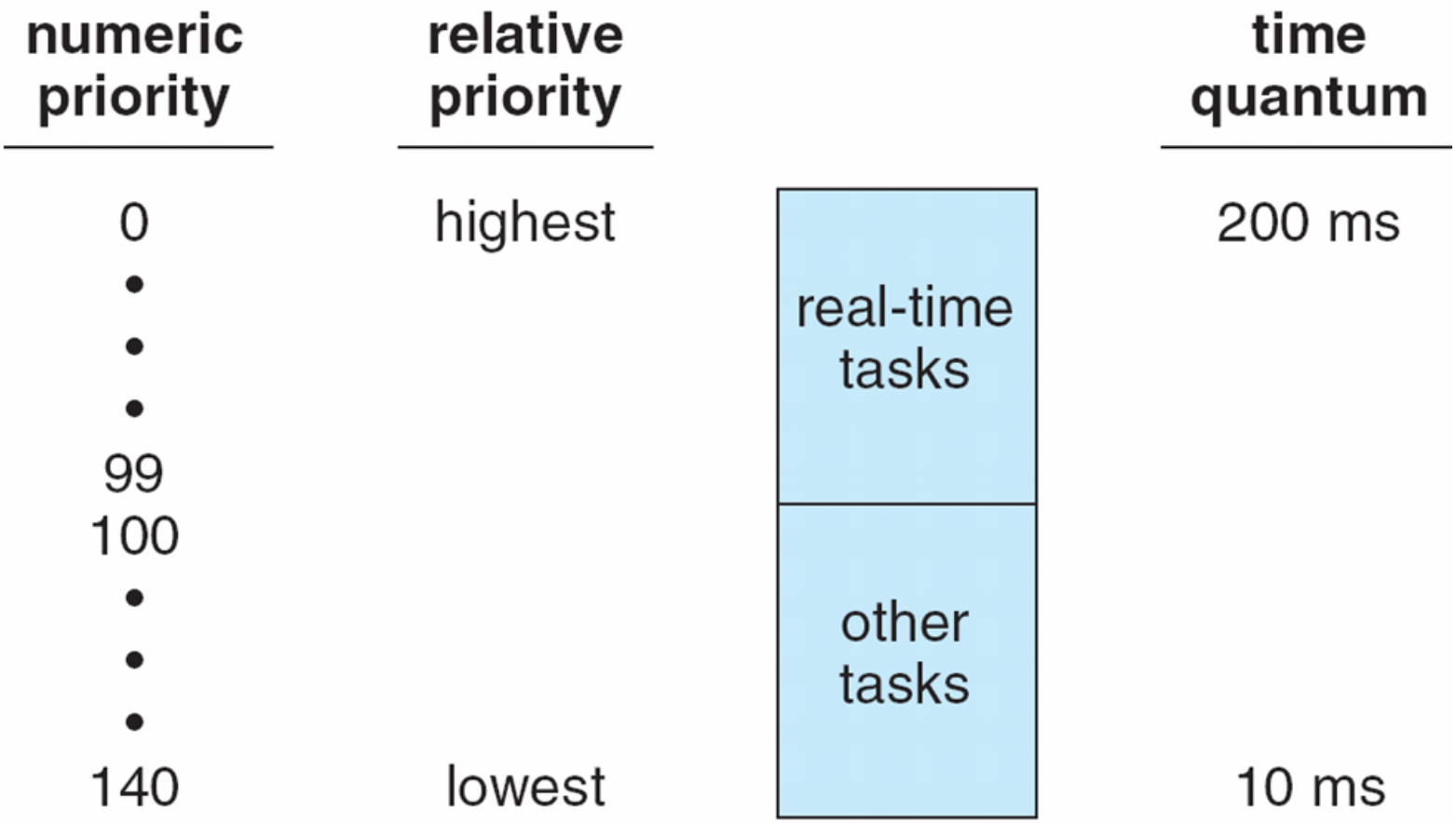
- Solaris zamanlaması
- Windows zamanlaması
- Linux zamanlaması

Linux Zamanlaması

- Linux çekirdeği 2.5 versiyonla birlikte gelen zamanlayıcı, sistemdeki iş parçacığı sayısından bağımsız olarak sabit zamanda çalışıyor - *O*(1) zamanlama süresi,
- Öncelik (priority) tabanlı ve kesen (preemptive) zamanlama algoritması,
- İki öncelik aralığı: zaman **paylaşımlı** (time-sharing) ve **gerçek zamanlı** (real-time),

- Gerçek zamanlı öncelik aralığı: 0'dan 99'a
- Zaman paylaşımlı (nice) öncelik aralığı: 100'den 140'a
- Diğer işletim sistemlerindeki (örn: solaris ve windows) zamanlayıcılardan farklı olarak
  - Düşük öncelikli thread daha az ve
  - Yüksek öncelikli thread daha çok zaman kuantumu veriyor.

Öncelikler ve Zaman Kuantumu

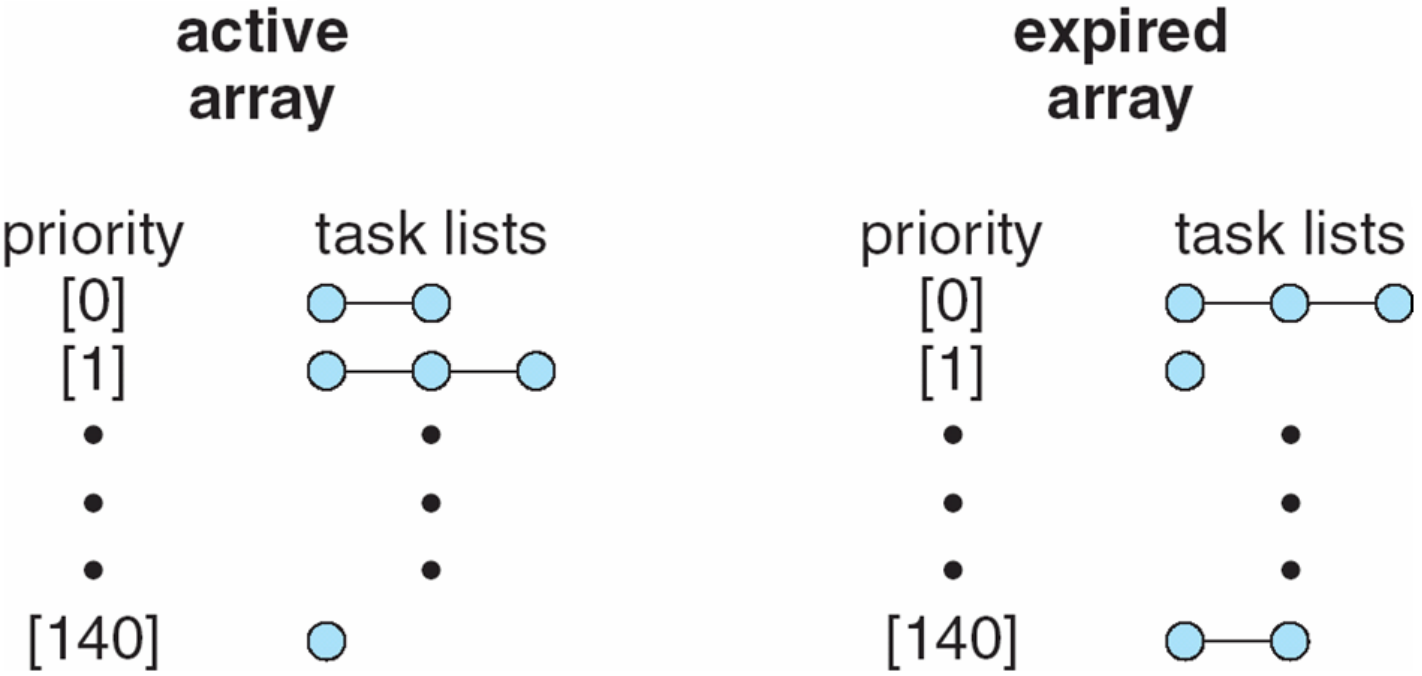


Aktif ve Süresi Dolmuş Dizileri

- Linux'de çalıştırılabilir işlerden zaman kuantumu dolmamış olanlara öncelik verilir
- Bu amaçla iki öncelik dizisi kullanılır:
  - **Aktif** (active)
  - **Süresi dolmuş** (expired)

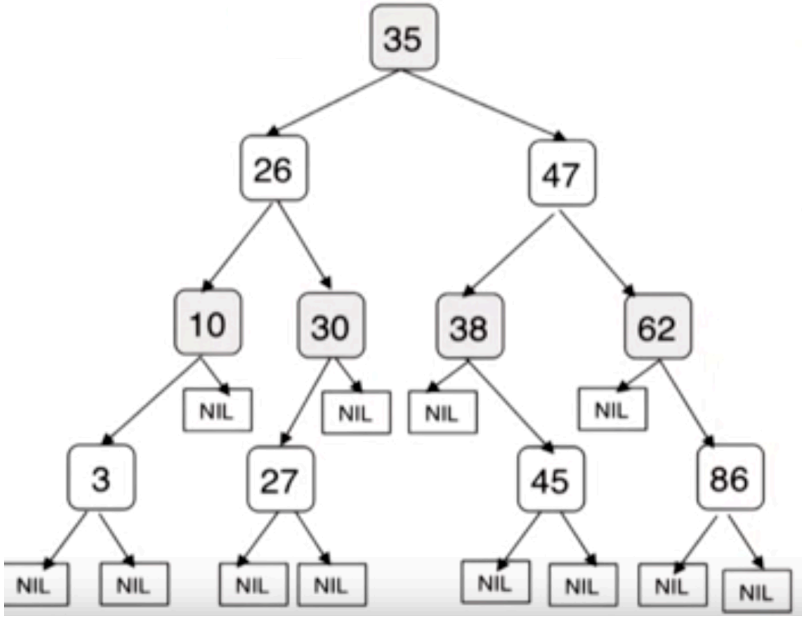
- Aktif dizisinde, zaman kuantumu henüz dolmamış ve çalışmaya hazır işler tutulmaktadır.
- Zamanlayıcı öncelikli işlerden başlayarak bu dizideki işleri çalıştırır.
- Zaman kuantumu dolan işler süresi dolmuş dizisine aktarılır.
- Aktif dizide çalışmaya hazır iş parçacığı kalmadığında, aktif ve süresi dolmuş dizileri yer değiştirir

Önceliklere Göre Listelenmiş Görevler



Linux 2.6.23 Kernel CFS Schedular (Ingo Molnar)

- O(1) schedular'ın performansı interaktif processler için iyi değil.



- Run Queue= Red- Black Tree
- Vruntime
- Vruntime düşük öncelikli processler için daha hızlı değişirken, büyük öncelikli processlerde daha yavaş değişir.

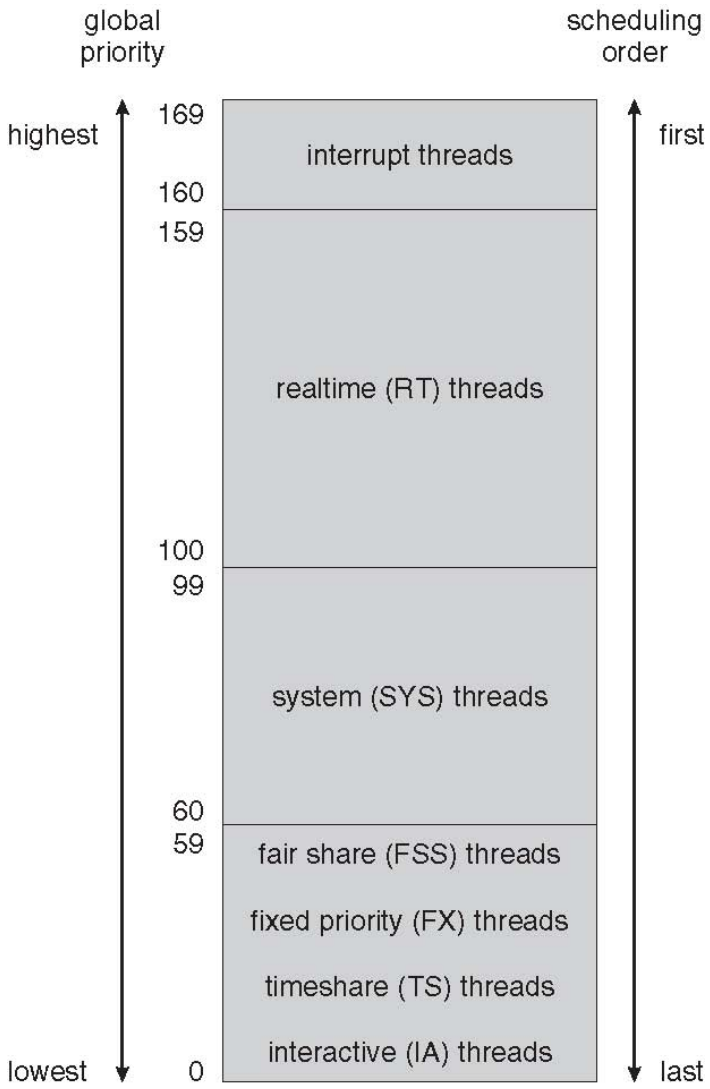
Node ekleme O(logn)

Seçme O(1)

Solaris Zamanlaması

- Öncelik (priority) tabanlı zamanlama kullanılmakta
- Altı zamanlama sınıfı var
  - Zaman paylaşımlı – time sharing (TS)
  - İnteraktif – interactive (IA)
  - Gerçek zamanlı – real time (RT)
  - Sistem – system (SYS)
  - Adil paylaşım – fair share (FSS)
  - Sabit öncelik – fixed priority (FP)

Solaris Zamanlama Sınıfları



Görev Tablosu: Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris'teki interaktif ve zaman paylaşımlı iş parçacıkları için

Windows Zamanlaması

- Öncelik (priority) tabanlı ve kesen (preemptive) zamanlama algoritması
- En yüksek öncelikli iş parçası her zaman çalışıyor

- Çalışan bir iş parçacığı aşağıdaki şartlardan biri oluşmadığı sürece çalışıyor
  - Normal şekilde sonlanma
  - Zaman kuantumunun bitmesi
  - Bloklayan bir sistem çağrısı yapılması (örneğin I/O)
  - Yüksek öncelikli bir iş parçacığının çalışmaya hazır olması

Windows Zamanlaması

- Çok seviyeli öncelik mekanizması – Yüksek seviye daha fazla öncelikli
- İki temel sınıfa ayrılıyor
  - Değişken öncelik sınıfı (1-15)
  - Gerçek-zaman sınıfı (16-31)
- Değişken öncelik sınıfındaki iş parçacıklarının öncelikleri çalışırken değiştirilebiliyor
- WIN32'de tanımlı öncelik sınıfları:
  - REALTIME_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - NORMAL_PRIORITY_CLASS
  - BELOW_NORMAL_PRIORITY_CLASS
  - IDLE_PRIORITY_CLASS

Windows Zamanlaması

- Belli bir öncelik sınıfındaki iş parçacığı aynı zamanda bu sınıf içerisinde tanımlı bir bağıl önceliğe sahip
- Bağıl öncelik değerleri
  - TIME_CRITICAL
  - HIGHEST
  - ABOVE_NORMAL
  - NORMAL
  - BELOW_NORMAL
  - LOWEST
  - IDLE

Windows Öncelikleri

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Java İş Parçacığı Zamanlaması

- Java sanal makinası (JVM) da öncelik tabanlı zamanlama algoritması kullanır

- Spesifikasyon kesen (preemptive) bir algoritma kullanmayı zorunlu kılmamıştır.
  - JVM gerçekleştirimine bırakılmıştır
  - O nedenle, bazı JVM'lerde az öncelikli bir iş parçacığı, yeni bir öncelikli iş parçacığı gelse dahi, çalışmaya devam edebilir
- Aynı önceliğe sahip iş parçacıkları, FIFO kuyrukları aracılığı ile yönetilir

Zaman Dilimleme

- Spesifikasyon zaman dilimlemeyi zorunlu kılmamıştır
- Bu yüzden çalışan iş parçacıklarının CPU kontrolünü devredebilmesi için yield() metodu sağlanmıştır

```
while (true) {
        // perform CPU-intensive task
        . . .
        Thread.yield();
}
```

- Bu metot, CPU kontrolünü aynı önceliğe sahip başka bir iş parçacığına bırakır

İş Parçacığı Öncelikleri

Öncelik	**Yorum**
Thread.MIN_PRIORITY	Minimum İş Parçacığı Önceliği
Thread.MAX_PRIORITY	Maksimum İş Parçacığı Önceliği
Thread.NORM_PRIORITY	Varsayılan İş Parçacığı Önceliği

Öncelikler setPriority() metodu kullanılarak değiştirilebilir:

```
Thread.currentThread().setPriority(Thread.NORM_PRIORITY + 2);
```

Algoritma Değerlendirmesi

- Belli bir sistem için hangi işlemci zamanlama algoritmasını seçmeliyiz?
- Pek çok algoritma ve bu algoritmaların pek çok parametresi var
- Öncelikle değerlendirme kriteri belirlenmeli – işlemci kullanımı, bekleme zamanı …
  - Örnek: İşlemci kullanımını maksimize ederken, cevap zamanını bir saniyenin altında tutmak
- **Deterministik modelleme** (deterministic modeling) – ön tanımlı bir iş yükünü alıp her bir algoritmanın performansını bu iş yükü açısından değerlendirmek

- **Kuyruklama Modelleri** (queueing models)

Simülasyon ile Zamanlama Algoritmalarının Değerlendirilmesi