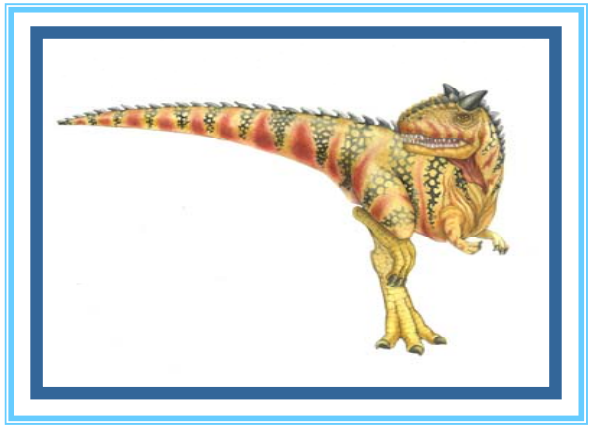
# 5. Bölüm: İş Sıralama (CPU Scheduling)



# 5. Bölüm: İş Sıralama (CPU Zamanlama / Planlama / Çizelgeleme)

- Temel Kavramlar
- Sıralama Kriterleri
- İş Sıralama Algoritmaları
- İş Parçacığı Sıralama
- Çoklu-işlemci Sıralama
- İşletim Sistemleri Örnekleri
- Algoritma Değerlendirme

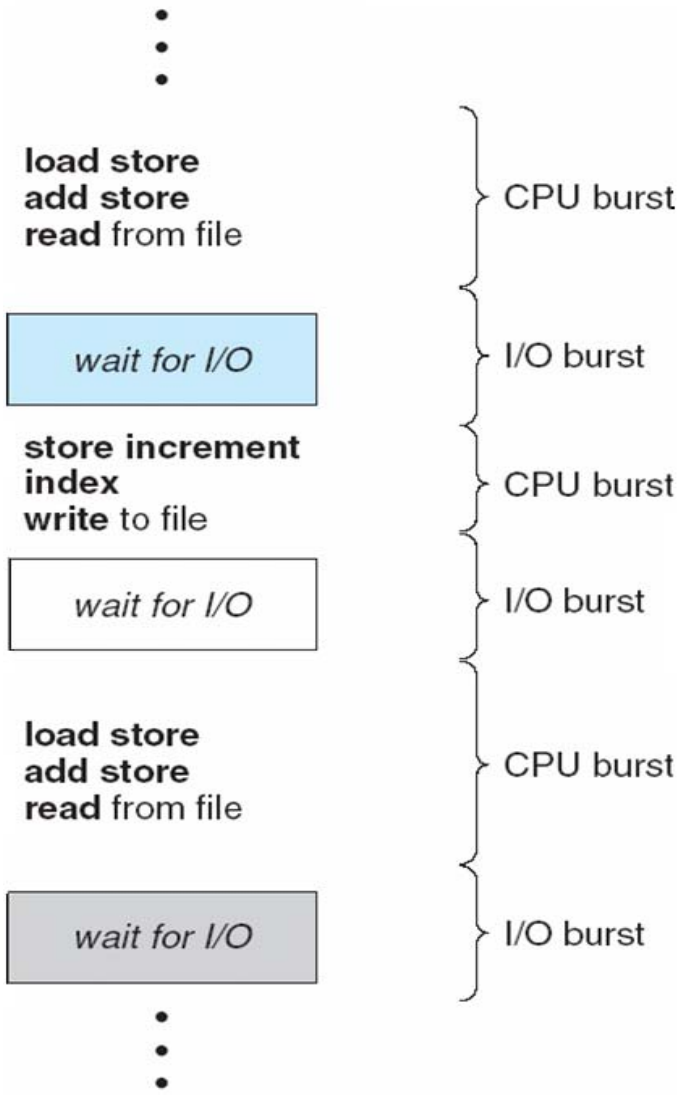
# Hedefler

- Çeşitli CPU iş sıralama algoritmalarını tanımlamak
- CPU iş sıralama algoritmalarını değerlendirmek
- Çok işlemcili ve çok çekirdekli iş sıralama ile ilgili sorunları açıklamak
- Çeşitli gerçek zamanlı iş sıralama algoritmalarını tanımlamak
- Windows, Linux ve Solaris işletim sistemlerinde kullanılan iş sıralama algoritmalarını açıklamak
- CPU iş sıralama algoritmalarını değerlendirmek için modelleme ve simülasyon yönteminden faydalanmak
- Belirli bir sistem için CPU İş sıralama algoritma seçim kriterlerini değerlendirmek

# Temel Kavramlar

- Çoklu programlama ile elde edilen **maksimum CPU kullanımı**
- CPU–I/O Patlama (Burst) Çevrimi – Proses çalışması CPU çalışması **çevrimi** ve G/Ç beklemesinden oluşur
- CPU patlamasını G/Ç patlaması takip eder
- **CPU patlama** dağılımı ana ilgi noktasıdır

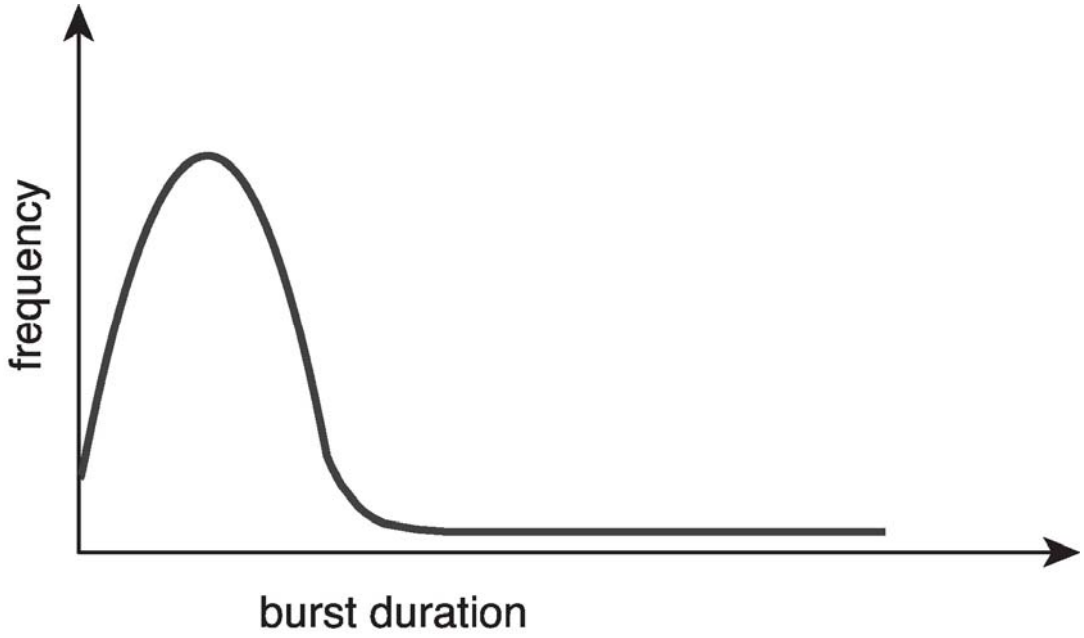
# CPU ve G/Ç Patlamaları Sırası



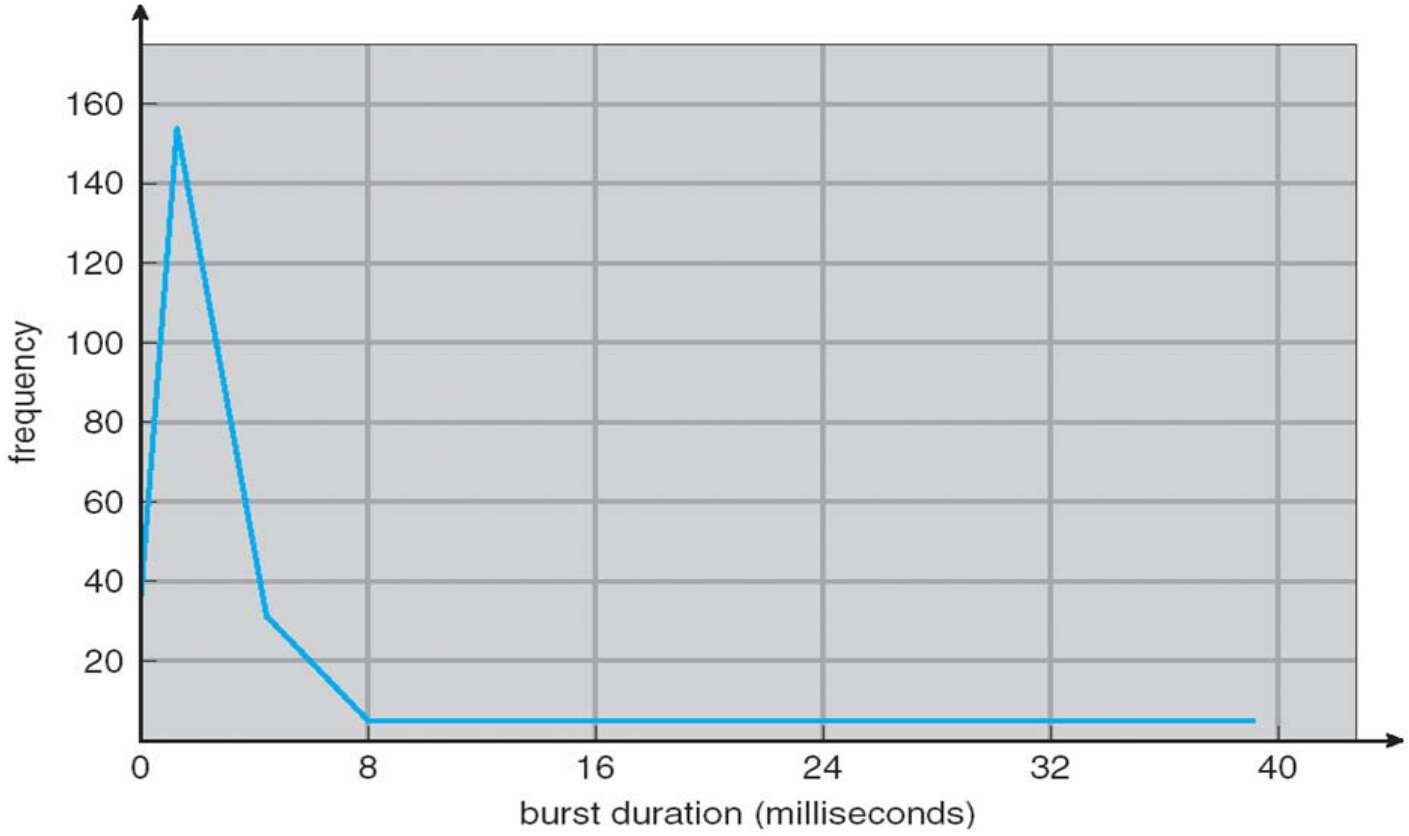

# CPU Patlama Zamanları Histogramı

Çok sayıda kısa patlama

Az sayıda daha uzun patlama




# CPU-patlama Zamanları




# CPU İş Sıralayıcı

- Hazır kuyruğunda bekleyen prosesler arasından seçim yapar ve CPU çekirdeğini bir tanesine tahsis eder.
  - Kuyruk çeşitli şekillerde sıralanabilir
- CPU iş sıralama kararları aşağıdaki durumlarda verilir:
  1. Çalışıyor durumundan bekleme durumuna geçerken
  2. Çalışıyor durumundan hazır durumuna geçerken
  3. Bekleme durumunda hazır durumuna geçerken
  4. Proses sonlanınca

- 1 ve 4 durumları için iş sıralama bakımından başka seçenek yoktur. Yeni bir proses ( eğer hazır kuyruğunda bulunuyorsa) çalışması için seçilir.
- 2 ve 3 nolu durumlarda bir seçenek vardır.

# Kesintili (Preemptive) ve Kesintisiz(Nonpreemptive) İş Sıralama

- 1 ve 4 durumları **kesintisizdir**
- Diğer tüm iş sıralama  işlemleri **kesintilidir**

- Kesintisiz iş sıralamada, CPU bir prosese tahsis edildikten sonra, proses CPU'yu sonlanana veya bekleme durumuna geçene kadar elinde tutar.

- Windows, MacOS, Linux ve UNIX dahil olmak üzere neredeyse tüm modern işletim sistemleri kesintili iş sıralama algoritmaları kullanır.

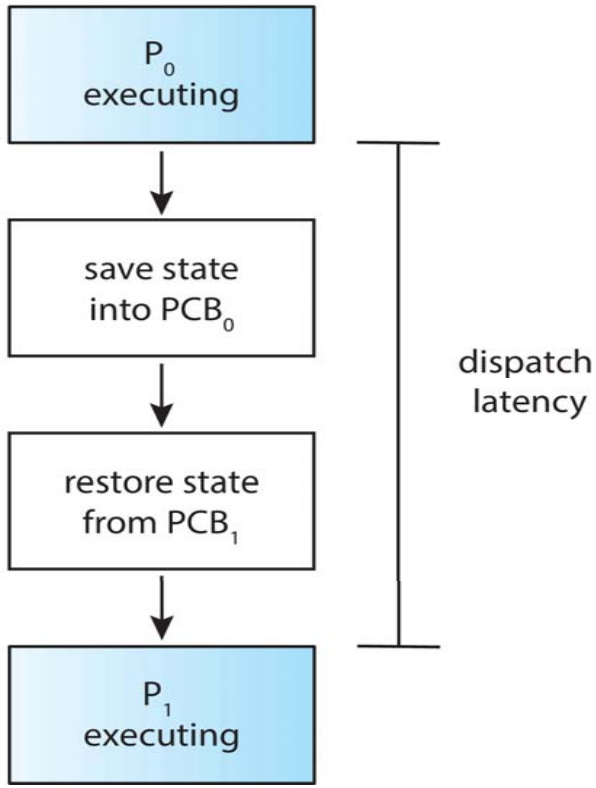
# Kesintili İş Sıralama ve Yarış Koşulları

- Kesintili iş sıralama, verilerin çeşitli prosesler arasında paylaşılması durumunda yarış koşullarına neden olabilir.

- Verileri paylaşan iki proses düşünün. Bir proses verileri güncellerken, diğer bekleyen proses çalışabilsin diye çalışması yarıda kesilir. İkinci proses daha sonra tutarsız bir durumda olan verileri okumaya çalışır.

# Görevlendirici - Dispatcher

- Görevlendirici modülü CPU'nun kontrolünü iş sıralayıcı tarafından seçilen prosese verir. Bu aşağıdaki işlemleri içerir:
  - Bağlam anahtarlama
  - Kullanıcı moduna değişim
  - Kullanıcı programını yeniden başlatmak için programdaki uygun bir konuma dallanma
- **Görevlendirme Gecikmesi**– bir prosesi sonlandırmak ve bir başkasını çalıştırmak için geçen süre




# İş Sıralama Kriterleri

- **CPU kullanım oranı (utilization)**– CPU'yu olabildiğince meşgul tut
- **Çıkış (throughput)** – birim zamanda çalışmasını tamamlayan proses sayısı
- **Tamamlanma (turnaround) zamanı** – belirli bir prosesin çalışması için gerekli zaman
- **Bekleme zamanı** – hazır kuyruğundaki beklemekte olan prosesin geçirdiği süre
- **Cevap zamanı** – bir istek gönderildikten ilk cevap alınana (çıkış değil) kadarki geçen süre

# İş Sıralama Algoritması Optimizasyon Kriterleri

- Max CPU kullanımı
- Max çıkış
- Min tamamlanma zamanı
- Min bekleme zamanı
- Min cevap zamanı

# İlk Gelen İlk Çalışır Algoritması (First-Come, First-Served - FCFS)

| Proses | Patlama zamanı |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Proseslerin $P_1$ , $P_2$ , $P_3$ *sırasında geldiğini varsayalım*
  Gantt Diyagramı :


- *Bekleme zamanları* $P_1$ = 0; $P_2$ = 24; $P_3$ = 27
- Ortalama bekleme zamanı: (0 + 24 + 27)/3 = 17

# FCFS İş Sıralama (Devam)

Proseslerin aşağıdaki sırada geldiğini varsayalım:

$$P_2 , P_3 , P_1$$

- Gantt diyagramı :


- Bekleme zamanı $P_1 = 6; P_2 = 0, P_3 = 3$
- Ortalama bekleme zamanı: (6 + 0 + 3)/3 = 3
- Az önceki durumdan daha iyi
- **Konvoy etkisi** – uzun proseslerin kısa proseslerden önce gelmesi
  - Bir adet CPU-bağımlı proses ve birden fazla I/O-bağımlı prosesler durumu gibi

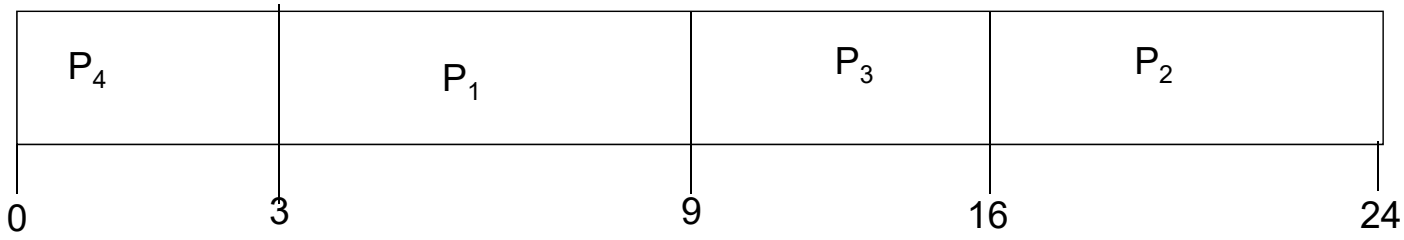
# En Kısa İş Önce (Shortest-Job-First - SJF) Algoritması

- Her bir proses bir sonraki CPU patlama süresiyle ilişkilendir
  - En kısa zamanlı prosesi tespit etmek için bu zaman değerlerini kullan
- SJF en uygundur – verilen bir grup proses için minimum ortalama bekleme zamanına neden olur
  - Zorluk bir sonraki CPU isteğinin patlama süresinin hesaplanmasındadır
  - Kullanıcıdan istenir
  - Veya hesaplanır
- Kesintili versiyonu **ek kısa kalan zaman önce** olarak adlandırılır

# SJF örneği

| Proses | Patlama Zamanı |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

- SJF iş sıralama diyagramı




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

# Bir Sonraki CPU Patlama Zamanının Belirlenmesi

- Süre sadece tahmin edilebilir – bir öncekine benzer olur
  - Bir sonraki CPU patlaması tahmin edilen en kısa süreli prosesi al

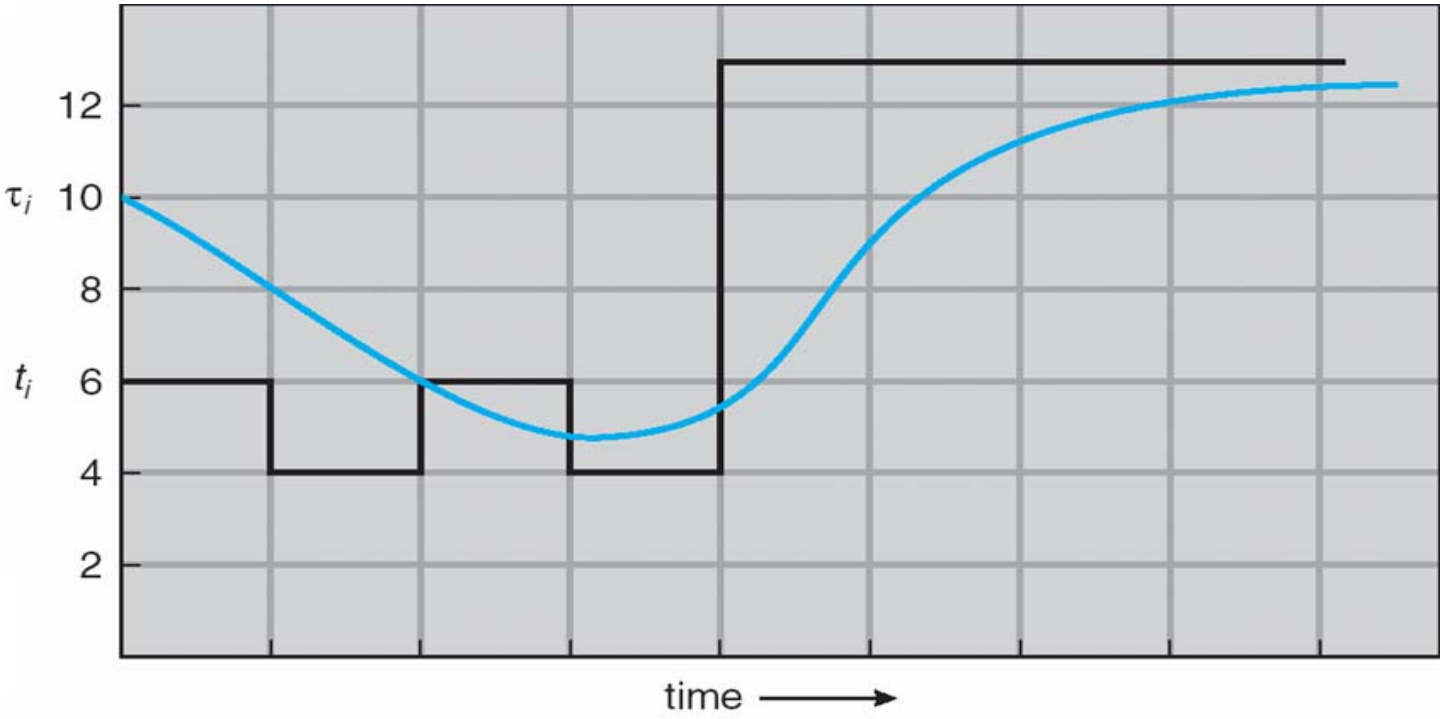
- Üstel ortalama ve bir önceki CPU patlama süresi kullanılarak hesaplanabilir

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

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

- Genelde , α ½ seçilir

# Bir Sonraki CPU Patlamasının Süresinin Tahmini




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| CPU burst ($t_i$) | | 6 | 4 | 6 | 4 | 13 | 13 | 13 | ... |
| "guess" ($\tau_i$) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 | ... |

# Üstel Ortalama Örnekleri

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Son kayıtlar hesaba katılmaz
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Sadece en son gerçek CPU patlaması hesaba katılır
- Eğer formülü genişletirsek:

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

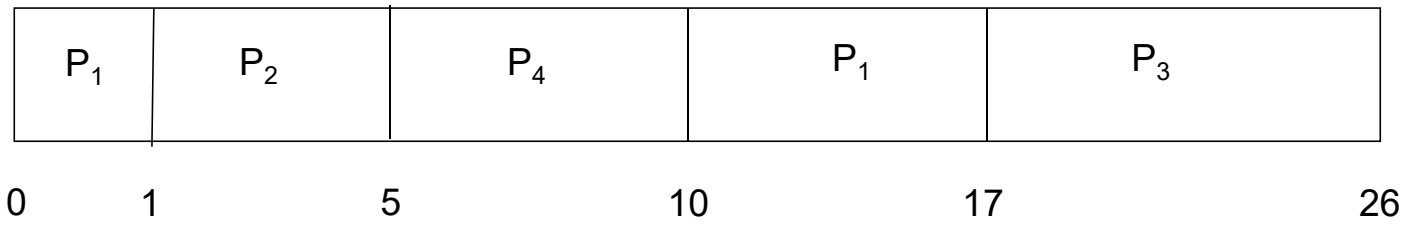
- $\alpha$ ve $(1 - \alpha)$ 1 e eşit veya daha küçük olduğu için, her bir terim kendisinden bir öncekine göre daha az ağırlığı vardır

# En Kısa Kalan Zaman Önce Örneği

- Bu örnekte farklı varış zamanı ve kesintili olan prosesleri analiz ederiz.

| Proses | *Varış Zamanı* | Patlama Zamanı |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

- *Kesintili* SJF Gantt Diyagramı




- Ortalama Bekleme zamanı= [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

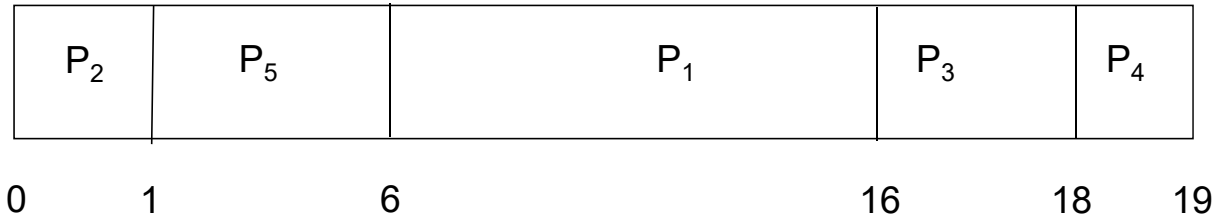
# Öncelikli İş Sıralama

- Bir öncelik sayısı (tamsayısı) her bir prosese atanır
- CPU en yüksek öncelik değerine sahip prosese tayin edilir (en küçük tamsayı ≡ en yüksek öncelik)
  - Kesintili
  - Kesintisiz
- SJF öncelik değerinin tahmini bir sonraki CPU patlama zamanının tersinin olduğu bir öncelikli iş sıralama yaklaşımıdır
- Problem ≡ **Açlıktan Ölme (Starvation)** – düşük öncelikli prosesler hiç çalışmayabilir
- Çözüm ≡ **Yaşlandırma** – zaman ilerlerken proses önceliğini artır

# Öncelikli İş Sıralama Örneği

| Proses | Patlama Zamanı | Öncelik |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

- Öncelikli İş Sıralama Gantt Diyagramı




- Ortalama bekleme zamanı = 8.2 msec

# Çevrimsel Sıralı Öncelikli Sıralama

| Process | Patlama Zamanı | Öncelik |
|---|---|---|
| $P_1$ | 4 | 3 |
| $P_2$ | 5 | 2 |
| $P_3$ | 8 | 2 |
| $P_4$ | 7 | 1 |
| $P_5$ | 3 | 3 |

- En yüksek öncelikli prosesi çalıştır. Öncelik aynı ise çevrimsel sıralı
- Gantt Diyagramı, quantum = 2

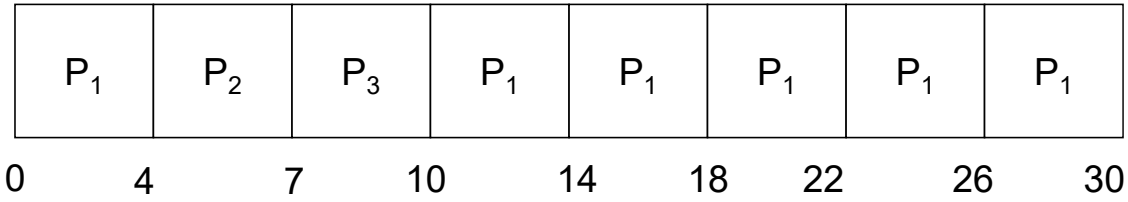

# Çevrimsel Sıralı (Round Robin - RR)

- Her bir proses genellikle 10-100 milisaniye arası küçük bir zaman süresince CPU'ya sahip olur .
- Bu zaman değerine **kuantum zamanı** denir ve q ile gösterilir.
- Bu süre tamamlandıktan sonra proses kesilir ve hazır kuyruğunun sonuna eklenir.
- Eğer hazır kuyruğunda n adet proses varsa ve kuantum zamanı değeri q ise her bir proses 1/n kadar CPU 'yu elde eder. Hiçbir proses $(n-1)q$ süresinden daha fazla beklemez
- Zamanlayıcı bir sonraki prosesi devreye almak için her bir kuantum süresi sonunda keser.
- Performans
  - $q$ büyük$\Rightarrow$ FIFO
  - $q$ küçük$\Rightarrow$ $q$ bağlam değişimine göre büyük olmalıdır aksi halde sistem kasılır

# RR Örneği (q = 4)

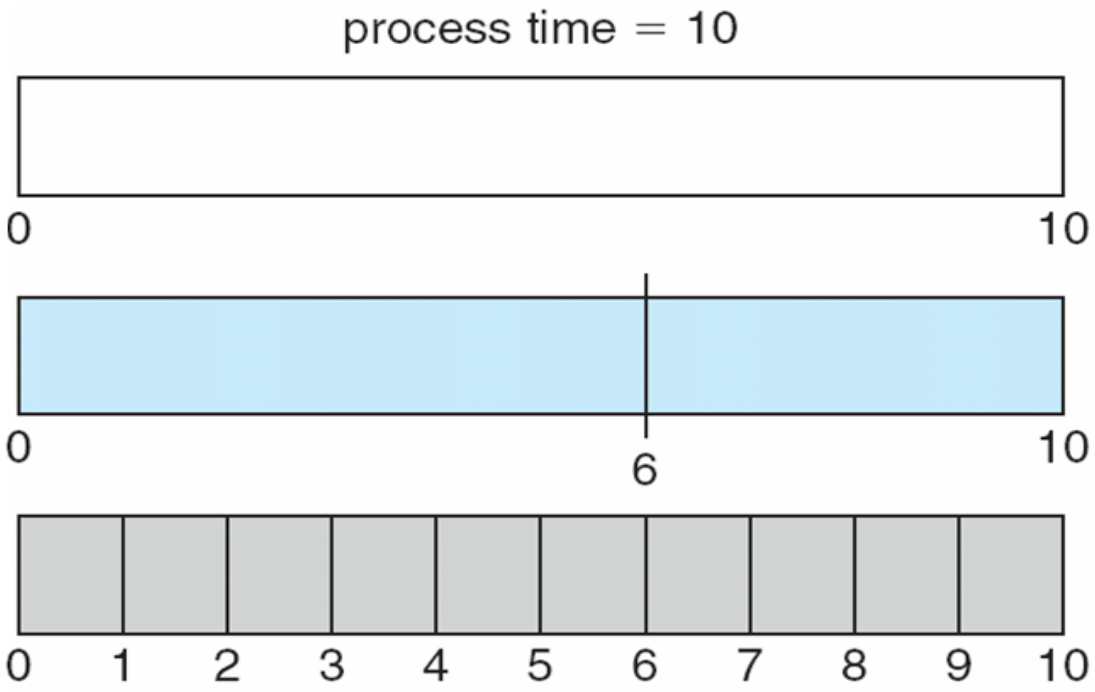
| Process | Patlama Zamanı |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Gantt diyagramı:



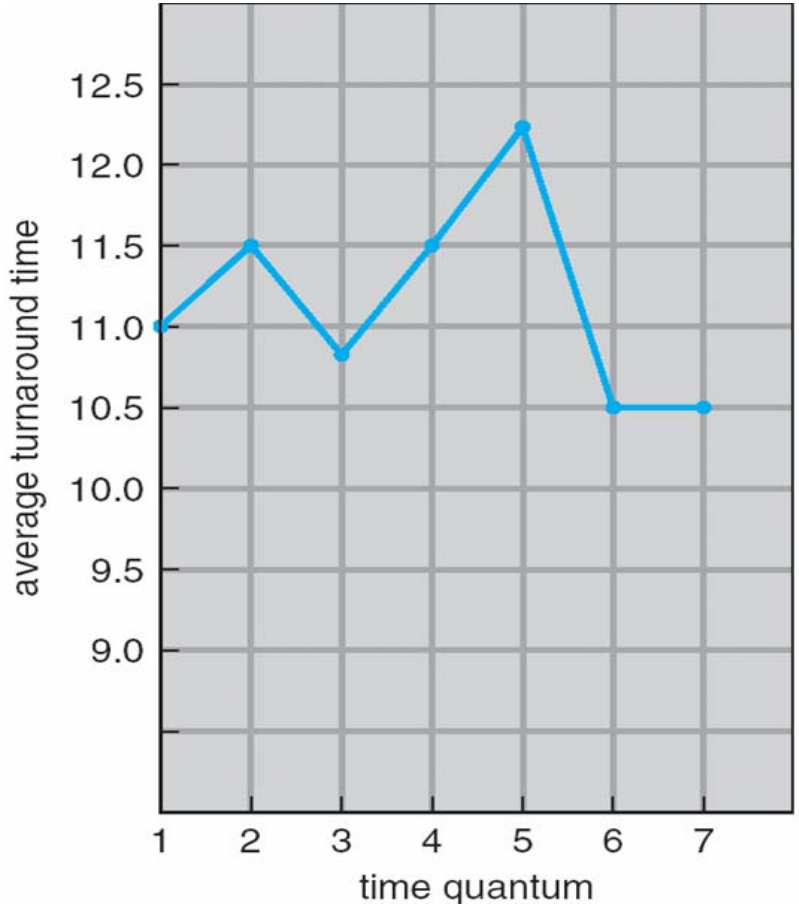

- Tipik olarak, SJF'den daha yüksek bir tamamlanma zamanı, ancak daha iyi bir cevap zamanına sahiptir.
- q bağlam değişimi zamanına göre büyük olmalıdır
- q genellikle 10ms ila 100ms,
- bağlam değişimi < 10 mikrosaniye

# Kuantum and Bağlam Değişim Zamanı




| quantum | context switches |
|---|---|
| 12 | 0 |
| 6 | 1 |
| 1 | 9 |

# Tamamlanma Zamanının Kuantum Zamanıyla Değişimi

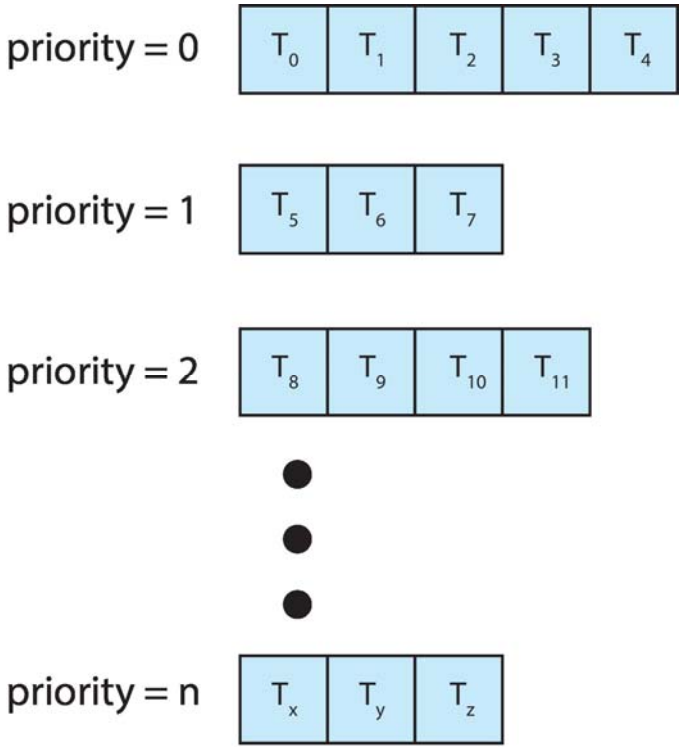



| process | time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

**CPU patlamalarının 80% i q dan daha küçük olmalıdır**

# Çok Seviyeli Kuyruk

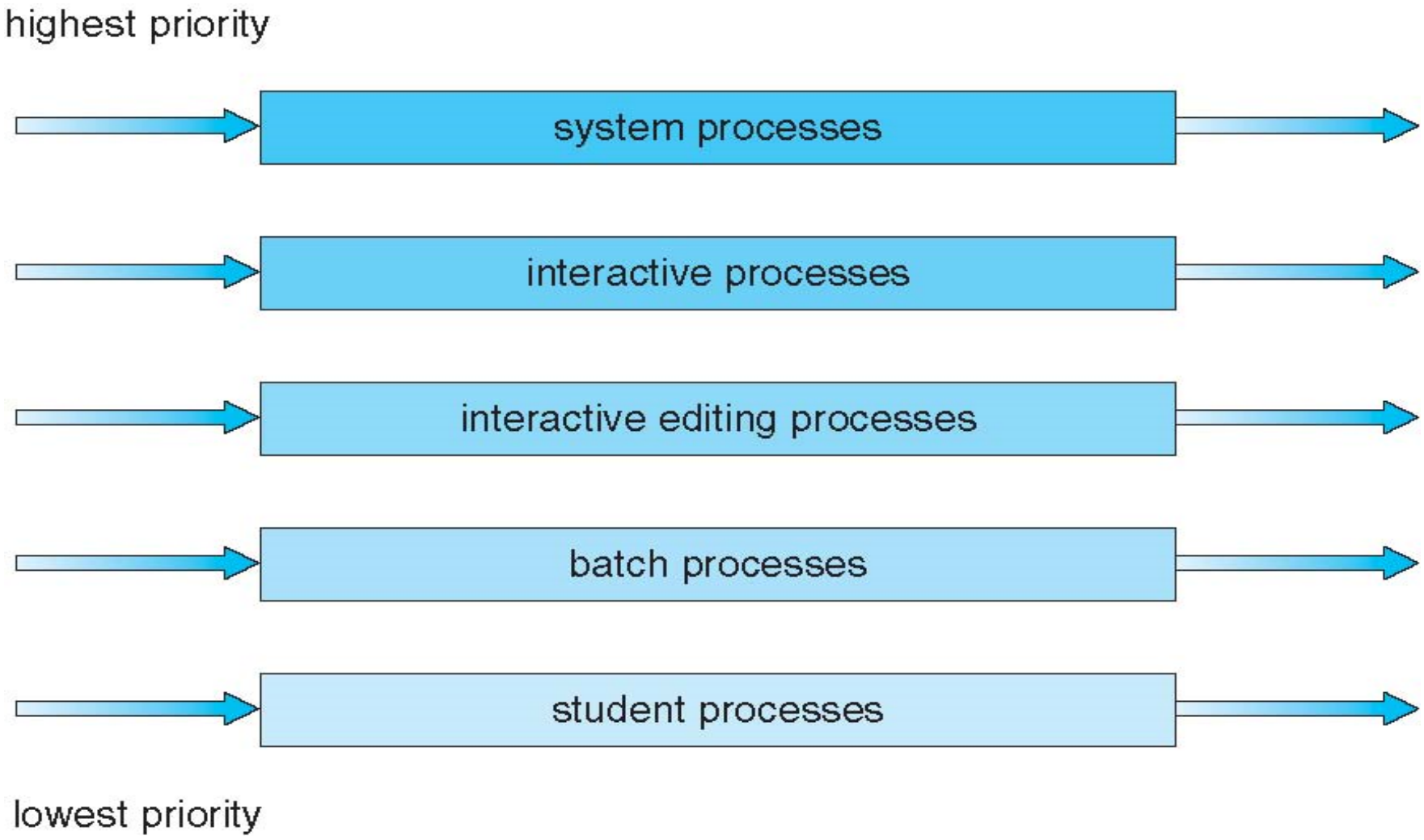
- Her öncelik için ayrı kuyruğun olduğu öncelikli sıralama
- En yüksek öncelikli kuyruktaki prosesler en önce sıralanır




# Çok Seviyeli Kuyruk

- Hazır kuyruğu ayrı kuyruklar halinde düzenlenir, ör:
  - Ön plan (interaktif)
  - Arkaplan (toplu iş - batch)
- Proses sürekli bir kuyruktadır
- Her kuyruk kendi iş sıralama algoritmasına sahiptir.
- ön plan – çevrimsel sıralı - RR
- arka plan – FCFS
- İş sıralama kuyruklar arasında yapılmalıdır:
  - Sabit öncelikli iş sıralama (ön plandakilerin tümü bittikten sonra arka plandakilere hizmet edilir ). Ölüm olasılığı.
  - Zaman dilimli iş sıralama– her kuyruk belirli bir CPU zamanını elde eder ve prosesleri arasında paylaştırır, RR' de ön plana kadar 80%
  - FCFS'de arka plan için 20%

# Çok Seviyeli Kuyruk




# Çok Seviyeli Geri Beslemeli Kuyruk

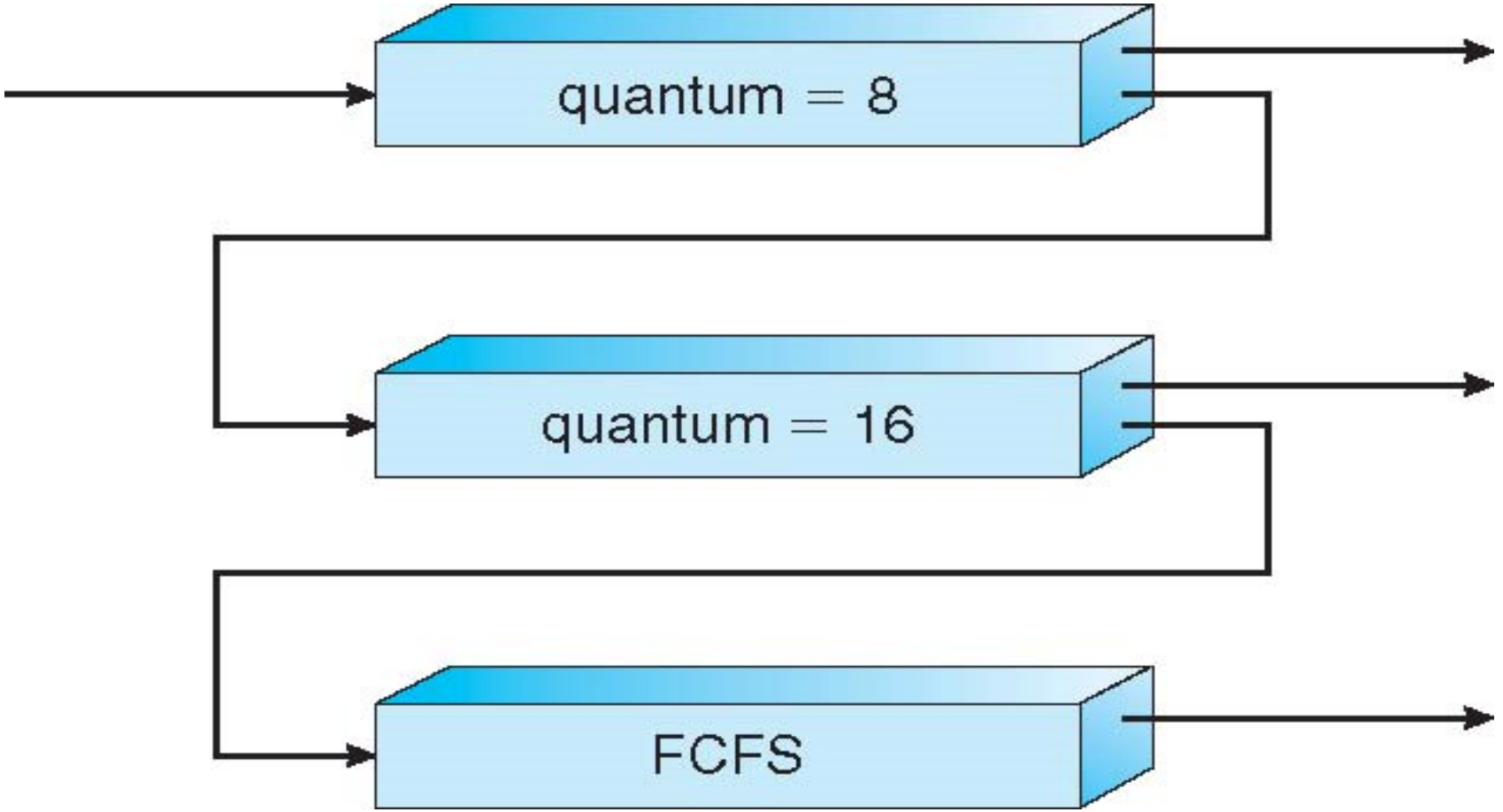
- Bir proses çeşitli kuyruklar arasında hareket edebilir;
- Bir tür yaşlandırma - aging uygulamasıdır
- Çok seviyeli-geri besleme kuyruğu iş sıralama işlemi aşağıdaki parametreler ile tanımlanır :
  - Kuyruk sayısı
  - Her kuyruğun iş sıralama algoritması
  - Bir prosesin ne zaman bir üst kuyruğa geçeceğini belirleme yöntemi
  - Bir prosesin ne zaman bir alt kuyruğa geçeceğini belirleme yöntemi
  - Bir prosesin çalışmaya ihtiyaç duyduğunda hangi kuyruğa ekleneceğini belirleme yöntemi

# Çok Seviyeli Geri Beslemeli Kuyruk Örneği

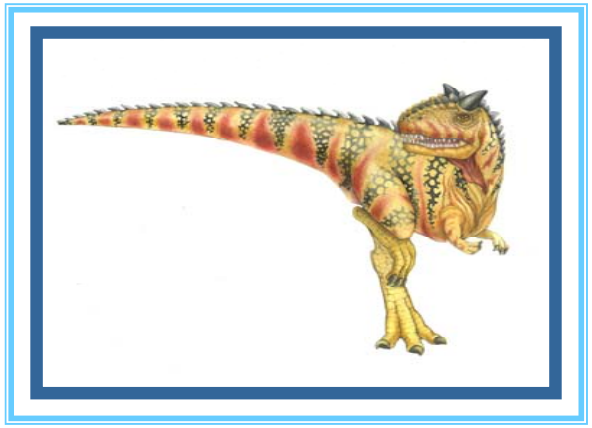
- Üç adet kuyruk:
  - $Q_0$ – 8 milisaniye kuantum değerine sahip RR
  - $Q_1$ – 16 milisaniye kuantumlu RR
  - $Q_2$ – FCFS

- İş Sıralama
  - Yeni bir görev $Q_0$ kuyruğuna girer ve FCFS olarak hizmet görür
    - CPU'yu ele geçirdiğinde 8 milisaniyesi vardır
    - 8 milisaniyede işini bitiremezse, $Q_1$ kuyruğuna geçer
  - $Q_1$ kuyruğunda görev yine FCFS gibi işlenir ve ilave 16 milisaniye alır
    - hala işini tamamlayamazsa kesilir ve $Q_2$ ye iletilir.

# Çok Seviyeli Geri Besleme Kuyruğu




# End of Chapter 5a



# İş Parçacığı Sıralama

- Kullanıcı-seviyeli ve çekirdek-seviyeli iş parçacıkları arasında ayrım
- İş parçacıkları desteği varsa prosesler değil iş parçacıkları sıralanır
- Çoktan- teke ve çoktan-çoka modelleri, iş parçacığı kütüphanesi LWP üzerinde çalışmak için kullanıcı seviyeli iş parçacıklarını sıralar
  - İş sıralama yarışı proses içinde olduğu için proses çekişme kapsamı (**process-contention scope-PCS) olarak bilinir**
  - Programcı tarafından öncelik kümesi yardımıyla yapılır
- Mevcut CPU üzerinde sıralanan çekirdek iş parçacığı sistem çekişme kapsamı (**system-contention scope-SCS)** dır– sistemdeki tüm iş parçacıkları arasında yarış

# Pthread İş Sıralama

- API iş parçacığının oluşturulması sırasında PCS veya SCS olup olmamasını belirtmeyi olanaklı kılar
  - PTHREAD_SCOPE_PROCESS PCS ile iş parçacıklarını sıralar
  - PTHREAD_SCOPE_SYSTEM SCS ile iş parçacıklarını sıralar
- OS tarafından sınırlandırılabilir – Linux ve Mac OS X sadece PTHREAD_SCOPE_SYSTEM i kullanır

# Pthread İş Sıralama 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;
      /* varsayılan nitelikleri al */
      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);
```

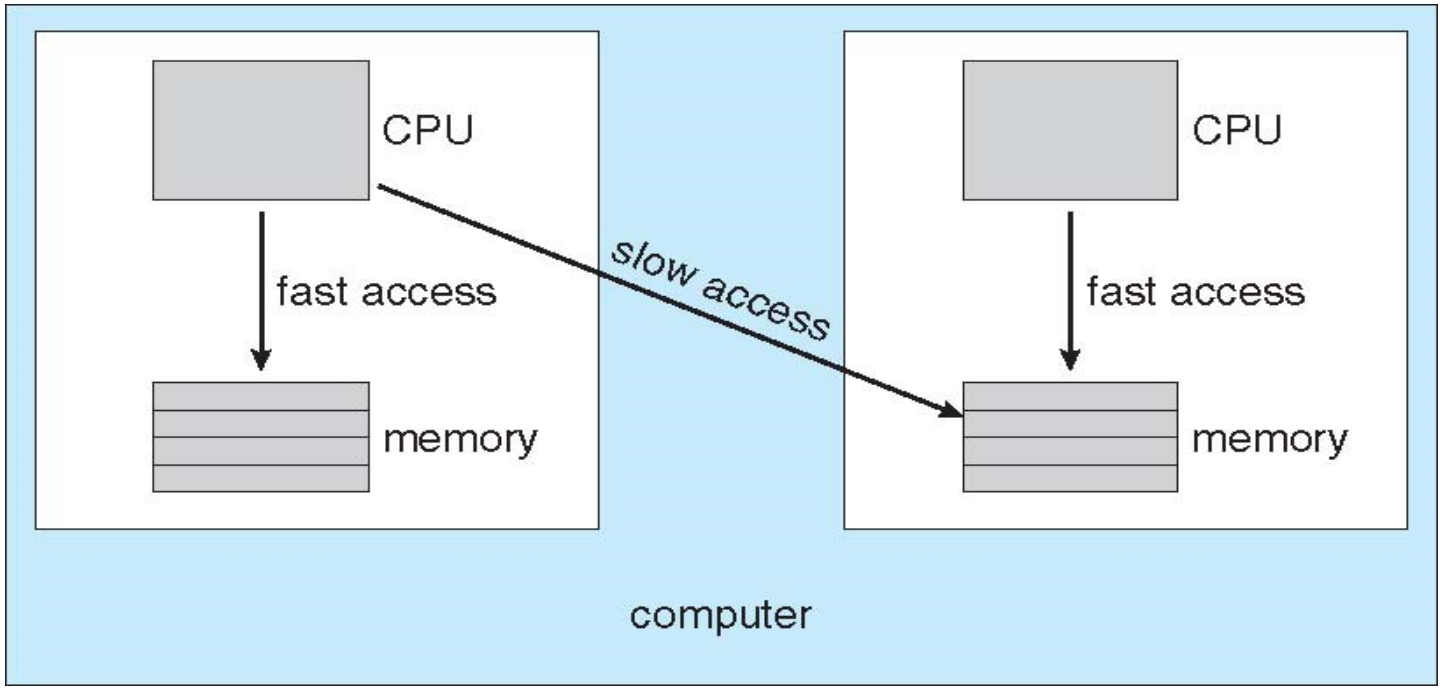
# Pthread İş SıralamaAPI

```
        /* 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);
}
```

# Çok İşlemcili İş Sıralama

- Birden fazla CPU varsa iş sıralama daha karmaşık olur.
- Tek bir blok içerisinde **homojen işlemciler**
- **Asimetrik çok işlemcili yapı** – veri paylaşımı ihtiyacını sağlamak için sadece bir adet işlemci sistem veri yapılarına erişir
- **Simetrik Çok İşlemcili Yapı (SMP)** – her işlemci kendi kendine iş sıralar, tüm prosesler ortak bir hazır kuyruğundadır veya her biri kendi özel hazır kuyruğuna sahiptir
  - Şuanda en yaygın
- **İşlemci İlişkisi**– proses üzerinde çalıştığı işlemci ile ilişkilidir
  - **Hafif ilişki**
  - **Sert ilişki**
  - **İşlemci kümelerini** içeren çeşitli versiyonlar

# NUMA ve CPU İş Sıralama

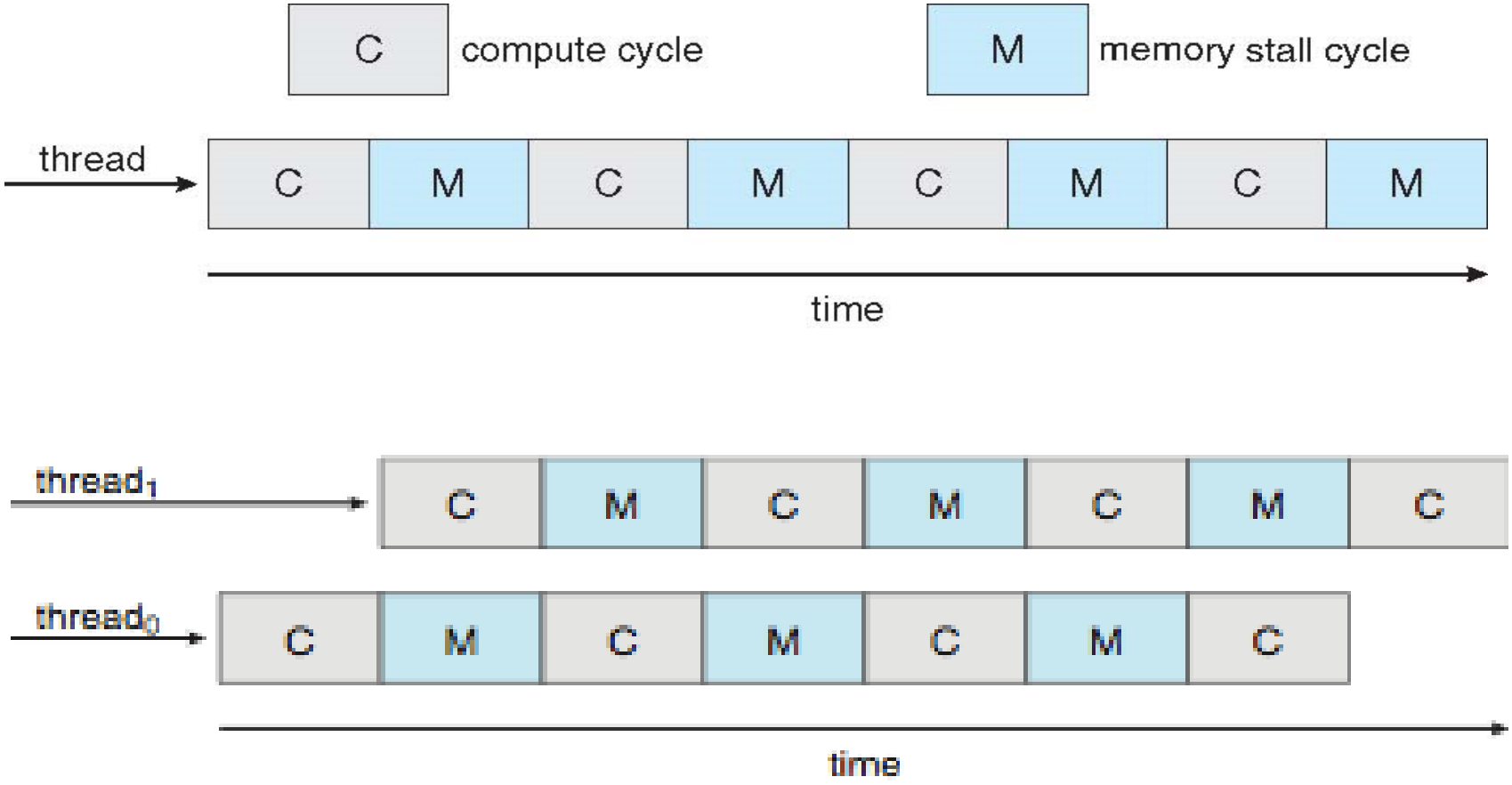



Bellek yerleşim algoritmalarının da CPU ilişkisini(affinity) göz önünde bulundurduğuna dikkat ediniz

# Çok Çekirdekli İşlemciler

- Son eğilim birden fazla işlemci çekirdeğini tek bir fiziksel çip üzerine yerleştirmektir
- Daha hızlı ve daha az güç harcar
- Çekirdek başına birden fazla iş parçacığı ortaya çıkıyor
  - Bellekten veri alınırken bir başka iş parçacığı üzerinde devam etmek için ara verilmesinin avantajını kullanır

# Çoklu İş Parçacıklı Çok Çekirdekli Sistem




# Sanallaştırma ve İş Sıralama

- Sanallaştırma yazılımı birden fazla konuğu CPU üzerine programlar
- Her bir konuk kendi iş sıralama işlemini yapar
  - CPU' ya sahip olup olmadığını bilmeyerek
  - Düşük bir cevap zamanına neden olabilir
  - Konukların güncel zamanları etkilenebilir
- Konukların iyi iş sıralama algoritması çabalarını telafi edebilir

# İşletim Sistemi Örnekleri

- Solaris iş sıralama
- Windows XP iş sıralama
- Linux iş sıralama

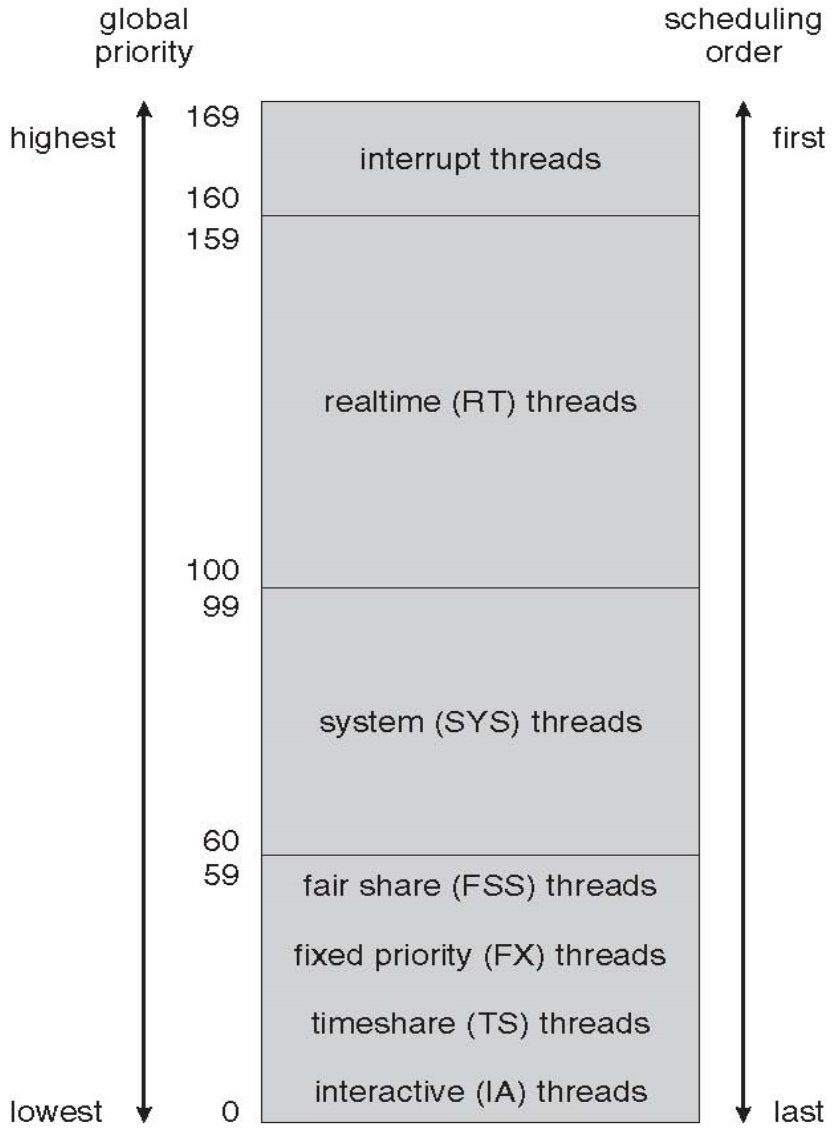
# Solaris

- Öncelik tabanlı iş sıralama
- Altı adet sınıf mevcuttur
  - Zaman paylaşımı (varsayılan)
  - İnteraktif
  - Gerçek zamanlı
  - Sistem
  - Açık paylaşım
  - Sabit öncelik
- Belirli bir iş parçacığı herhangi bir zamanda sadece bir sınıfta olabilir
- Her bir sınıf kendi iş sıralama yaklaşımına sahiptir
- Zaman paylaşımı çok seviyeli geri beslemeli kuyruktur
  - Sistem yöneticisi tarafından yüklenebilir tablo konfigüre edilebilir

# Solaris Görevlendirme Tablosu

| 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 İş Sıralama




# Solaris İş Sıralama(devam)

- İş sıralayıcı sınıfa özgü öncelikleri iş parçacığı global önceliğine dönüştürür
  - En yüksek öncelikli iş parçacığı bir sonra çalışır
  - (1) bloke olana kadar çalışır, (2) zaman periyodu kullanılır, (3) daha yüksek öncelikli iş parçacığı tarafından kesilir
  - Aynı öncelikli birden fazla iş parçacığı RR'ye göre çalışır

# Windows İş Sıralama

- Windows öncelik tabanlı kesintili bir iş sıralama yaklaşımını kullanır
- En yüksek öncelikli iş parçacığı bir sonra çalışır
- *Görevlendirici (Dispatcher) iş sıralayıcıdır*
- İş parçacıkları (1) bloke olana, (2) zaman periyodu bitimini, (3) daha yüksek öncelikli bir iş parçacığı tarafından kesilene dek çalışır
- Gerçek zamanlı iş parçacıkları olmayanları kesebilir
- 32-seviyeli öncelik düzeni
- **Değişken sınıf** 1-15, **gerçek zamanlı sınıf** 16-31
- 0 önceliği bellek yönetimi iş parçacığıdır
- Her öncelik için kuyruk vardır
- Eğer çalışabilir bir iş parçacığı yoksa, **boşta iş parçacığı** çalışır

# Windows Öncelik Sınıflar

- Win32 API prosesin ait olduğu öncelik sınıflarını tanımlar
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - REALTIME harici tümü değişkendir
- Belirli bir öncelik sınıfındaki bir iş parçacığı bağıl bir önceliğe sahiptir
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Öncelik sınıfı ve bağıl öncelik nümerik önceliği oluşturur
- Taban önceliği  NORMAL dir
- Eğer kuantum zamanı aşarsa, öncelik tabanın altına inmeyecek şekilde düşürülür
- Eğer bekleme olursa, ne için beklediğine göre öncelik artırılır
- En öndeki pencereye 3x öncelik verilir

# Windows XP 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 |

# Linux İş Sıralama

- Sabit dereceli *O*(1) iş sıralama zamanı
- Kesintili öncelik tabanlı
- İki adet öncelik aralığı: zaman paylaşımlı ve gerçek zamanlı
- **Gerçek zamanlı** aralık 0 dan 99 a dır
- Daha yüksek önceliği gösteren nümerik düşük değerlere sahip global öncelik
- Daha yüksek öncelik daha büyük q ya neden olur
- Zaman aralığında kalan zaman süresi kadar görev çalışabilir (**active**)
- Eğer zaman kalmamışsa (**expired**), diğer görevler kendi zaman aralıklarını kullanana kadar çalışamaz
- Tüm çalışabilir görevler CPU başına bir adet çalışır kuyruğunda tutulur
  - İki adet öncelikli dizi (active, expired)
  - Görevler önceliğe göre sıralanır
  - Aktif görev kalmadığında diziler yer değişir

# Linux İş Sıralama (devam)

- Gerçek zaman iş sıralama POSIX e göre yapılır
  - Gerçek zamanlı görevler statik önceliklere sahiptir
- Tüm diğer görevler nice değerine göre artı eksi 5 olarak dinamiktir.
  - Görevin interaktivitesi artı veya eksi olmasını belirler
    - Daha interaktif -> daha eksi
  - Görev zaman aşımına uğradığında öncelik yeniden hesaplanır
  - Bu karşılıklı değişilen diziler ayarlı öncelikleri uygular

# Öncelikler ve Zaman Süresi

| numeric priority | relative priority | | time quantum |
|---|---|---|---|
| 0 | highest | real-time tasks | 200 ms |
| • | | | |
| • | | | |
| • | | | |
| 99 | | | |
| 100 | | other tasks | |
| • | | | |
| • | | | |
| • | | | |
| 140 | lowest | | 10 ms |

# Önceliklere Göre Sıralanmış Görev Listesi


# Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority

# Java Thread Scheduling (Cont.)

JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not

# Time-Slicing

Since the JVM Doesn't Ensure Time-Slicing, the yield() Method
May Be Used:

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

This Yields Control to Another Thread of Equal Priority

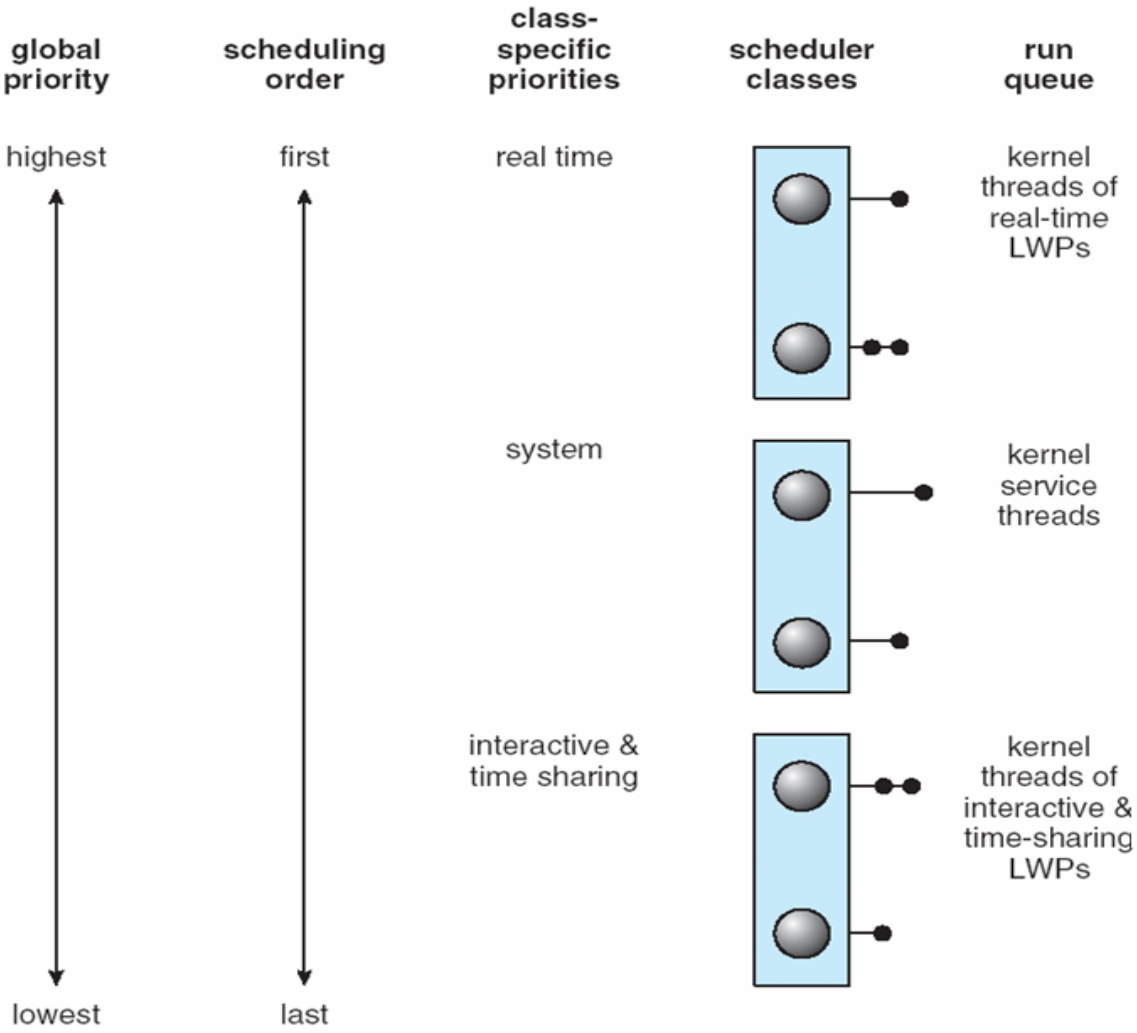
# Thread Priorities

| Priority | Comment |
|---|---|
| Thread.MIN_PRIORITY | Minimum Thread Priority |
| Thread.MAX_PRIORITY | Maximum Thread Priority |
| Thread.NORM_PRIORITY | Default Thread Priority |

Priorities May Be Set Using setPriority() method:

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

# Solaris 2 Scheduling




# Algoritma Değerlendirme

- Bir işletim sistemi için CPU iş sıralama algoritması nasıl seçilir ?
- Kriterleri belirle ve daha sonra algoritmaları değerlendir
- Deterministik modelleme
  - Bir tür analitik değerlendirme
  - Belirli bir iş yükünü alır ve o iş yükü için herbir algoritmanın performansını hesaplar

# Kuyruk Modelleri

- Proseslerin varışını, CPU ve I/O patlamalarını olasılıksal olarak tanımlar
  - Genelde üstel ve ortalama ile tanımlanır
  - Ortalama çıkış(throughput), kullanım oranı, bekleme zamanını hesaplar
- Bilgisayar sistemini herbiri bekleyen prosesler kuyruğuna sahip sunucular ağı olarak tanımlar
  - Varış oranı ve hizmet oranı bilinir
  - Kullanım oranını, ortalama kuyruk boyutunu ve ortalama bekleme zamanını hesaplar

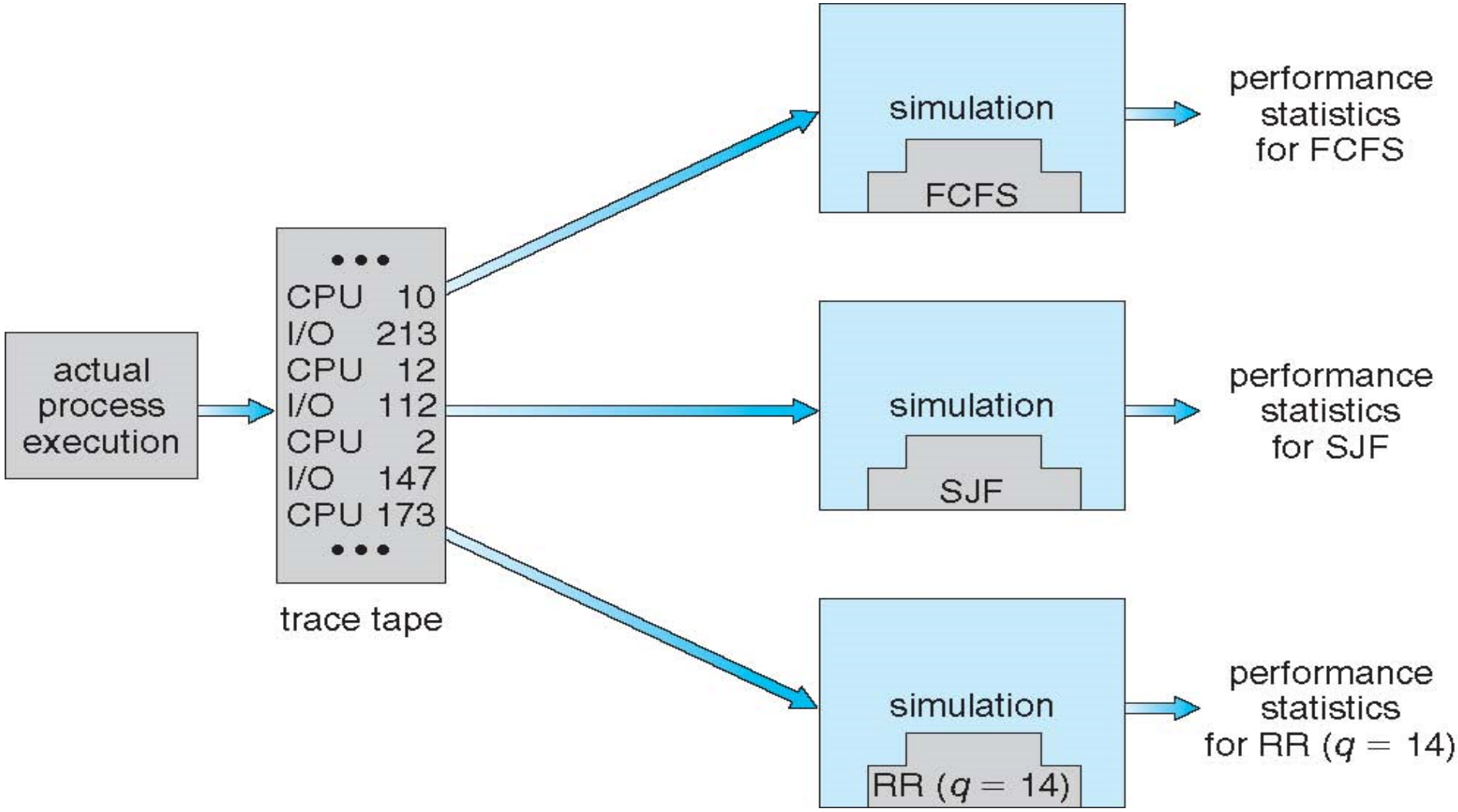
# Little Formülü

- $n$ = ortalama kuyruk boyutu
- $W$ = kuyruktaki ortalama bekleme zamanı
- $\lambda$ = ortalama kuyruğa varış oranı
- Little'ın kuralı – kararlı durumda, kuyruğu terk eden prosesler kuyruğa varanlarla eşit olmalıdır, bu nedenle
- $n = \lambda \times W$
  - Herhangi bir iş sıralama algoritması ve geliş dağılımı için geçerlidir
- Mesela, eğer saniyede ortalama 7 proses varıyorsa ve normalde kuyrukta 14 proses varsa prosesler için ortalama bekleme zamanı = 2 saniye olur.

# Simülasyonlar

- Kuyruk modelleri sınırlıdır
- **Simülasyonlar** daha doğru sonuç verir
  - Bilgisayar sistemi modeli
  - Saat değişkendir
  - Algoritma performansını gösteren istatistikler üret
  - Simülasyonu süren veri :
    - Rasgele sayı üreteci
    - Matematiksel dağılımlar
    - Gerçek sistemlerdeki gerçek olayların sırasını tutan kayıtlar

# CPU İş Sıralayıcıların Simülasyon ile Değerlendirilmesi




# Uygulama

- Simülasyonlar da sınırlı doğruluğa sahiptir
- Sadece yeni iş sıralayıcıyı uyarla ve gerçek sistemlerde test et
  - Yüksek maliyet, yüksek risk
  - Çevre değişebilir
- Çok esnek iş sıralayıcılar mekana veya sisteme göre değiştirilebilir
- Veya  API'ler öncelikleri değiştirmek maksadıyla geliştirilebilir
- Ancak çevre değişkendir

# 5. Bölümün Sonu