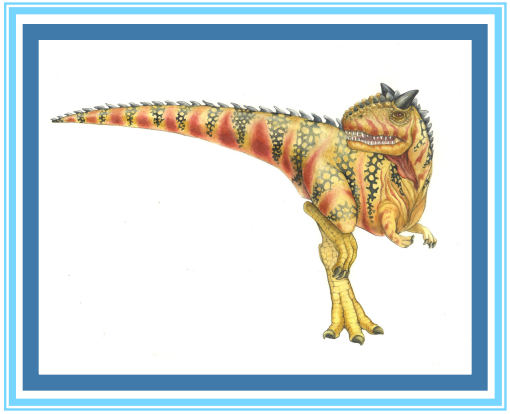
Chapter 6: CPU Scheduling

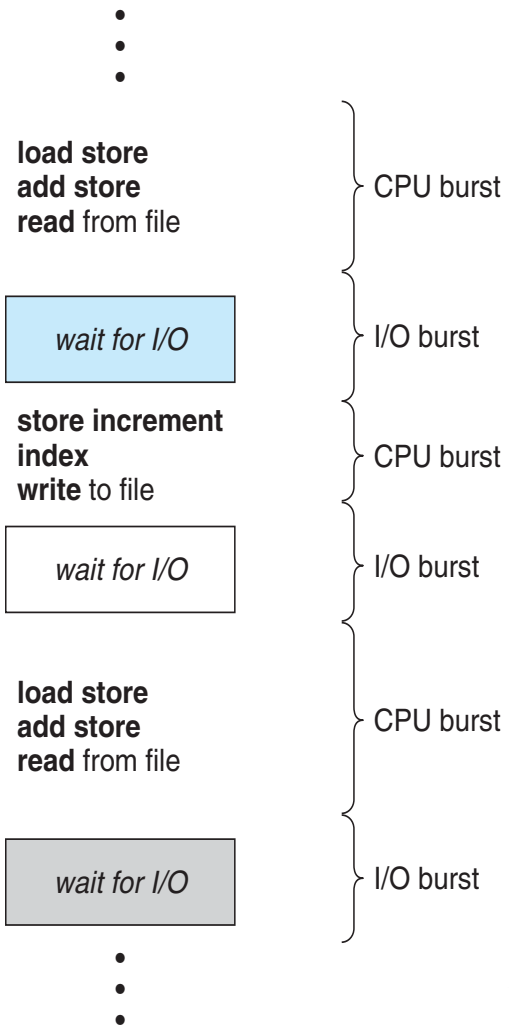


Chapter 6: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

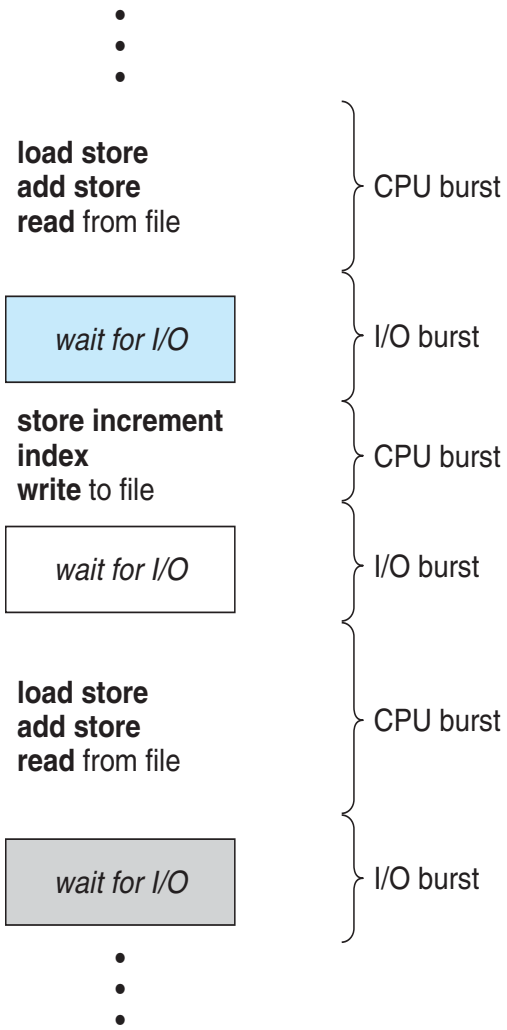
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

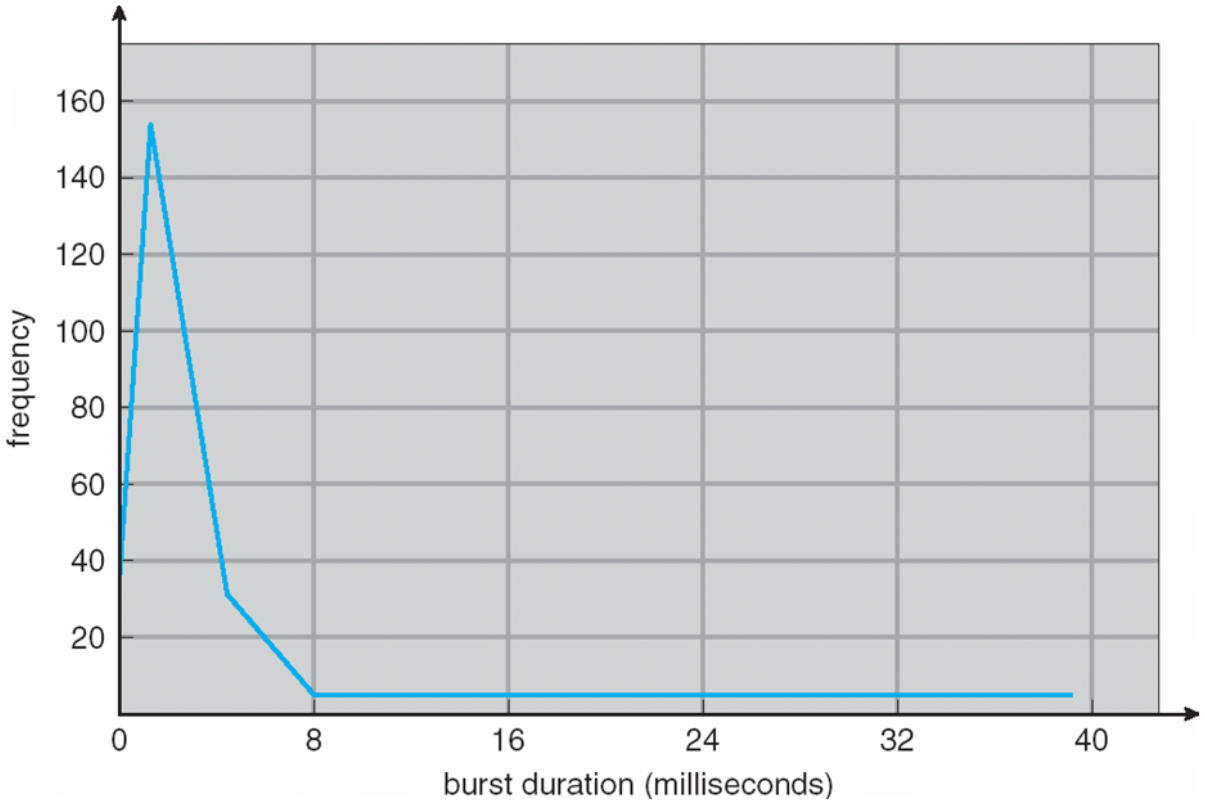


Basic Concepts

- Çoklu programlama kullanarak maksimum CPU kullanımı
- **CPU–I/O Burst Cycle**: CPU'nun bir prosesi işlemek için I/O Wait gelene kadar ihtiyacı olan zaman aralığı.
- CPU burst ile başlar ve sonra I/O burst gelir bunu başka bir CPU burst ve arkasından başka bir I/O burst bunu takip eder.
- **CPU scheduling-> amaç CPU nun çalışma döngüsünü yönetme**



Histogram of CPU-burst Times



CPU burst süresi, prosesten prosese ve bilgisayardan bilgisayara farklı olabilir CPU burst süresi kısa olanlar çok sık, CPU burst süresi uzun olanlar ise çok seyrek çalışmaktadır.

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them(**short term ready kuyruğundan Cpu ya gönderilme işlemini kontrol eder)**
  - Queue may be ordered in various ways**(kuyruk farklı yollarla sıralanabilir)**
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- **İşlemci zamanlayıcı 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**

- Scheduling under 1 and 4 is **nonpreemptive(kesmeyen) (process bitmeden diğer process çalıştırılmaz, I/O bloklayabilir)**
- All other scheduling **is preemptive(kesen)**
  - Consider access to shared data(paylaşılan verilere erişim)
  - Consider preemption while in kernel mode(kernel modda önleme)
  - Consider interrupts occurring during crucial OS activities
  - (Preemption işletim sistemi kernel dizaynını etkiler.Sistem çağrısı oluştuğunda yürütülen proses kernel datayı işleyen bir proses olabilir. Bu tip durumlar için OS sistem çağrısını yapılan iş tamamlanana kadar bekletir.

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
- Görevlendirici modül CPU kontrolünü kısa dönem zamanlayıcı (short-term scheduler) tarafından seçilen işleme devreder:
  - Ortam değiştirme (switching context)
  - Kullanıcı moduna geçme
  - Kullanıcya ait program yeniden başlatıldığındaprogramdaki uygun konuma atlama
- **Görevlendirme gecikme süresi (dispatch latency)** – Görevlendiricinin bir programı sonlandırıp diğerini başlatması için gereken süre

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible**(işlemciyi sürekli meşgul etme, boş beklemesin)**
- **Throughput** – # of processes that complete their execution per time unit (**birim zamanda çalışması sonlanan işlemlerin sayısı)**
- **Turnaround time** – amount of time to execute a particular process(**bir processin yaşam süresi, ne kadar sürede bitirilmiş:Hafızaya alınmak için bekleme süresi, Hazır kuyruğunda bekleme süresi,CPU'da çalıştırılması ve I/O işlemi yapması için geçen sürelerin toplamı**)
- **Waiting time** – amount of time a process has been waiting in the ready queue**(çalışma kuyruğunda toplam bekleme süresi)**
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment**) (bir isteğin gönderilmesi ile bu isteğin yanıtının verilmesi arasında geçen zaman)**

Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Konvoy etkisi (convoy effect): kısa işlemler uzun işlemlerin arkasında

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the use
- **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.**

  Sorun: işlemlerin çalışma uzunluğunu nereden bileceğiz?

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

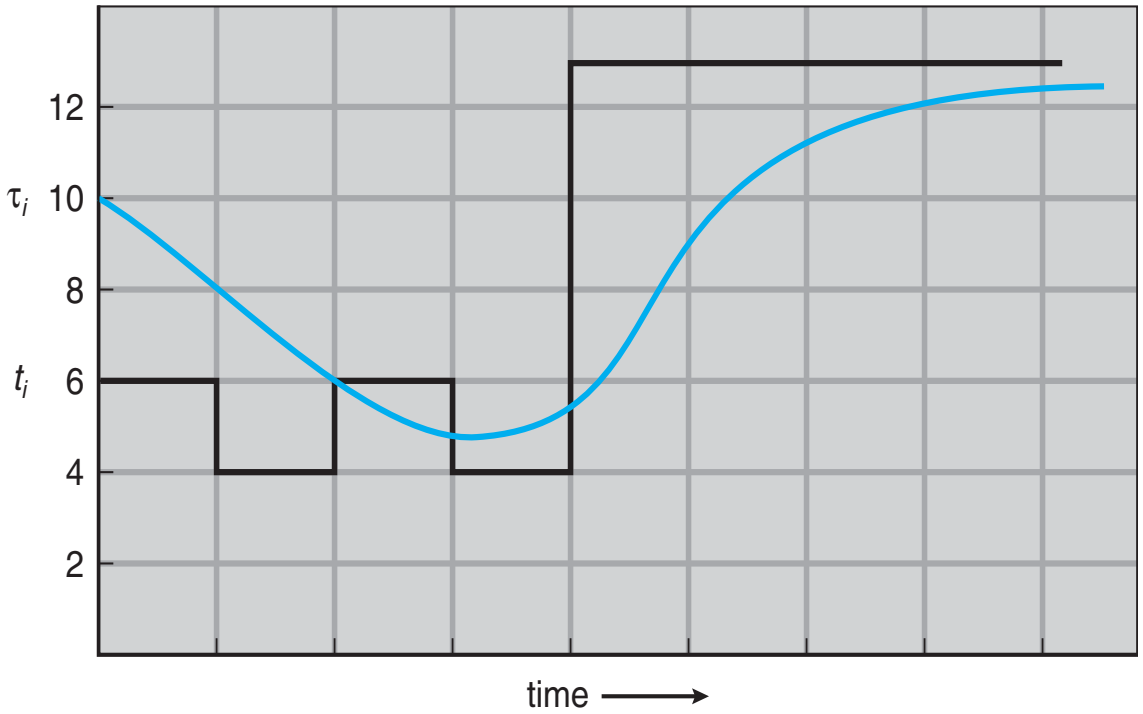
- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

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

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**
- **Sadece tahmin edilebilir :Daha önceki işlemci kullanım süreleri kullanılarak üssel ortalama (exponential averaging) yöntemiyle tahmin edilebilir**

Prediction of the Length of the Next CPU Burst



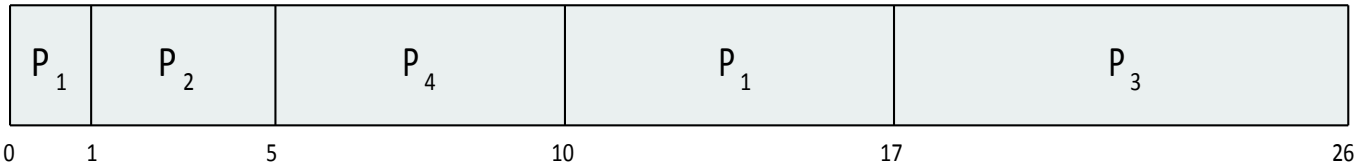
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

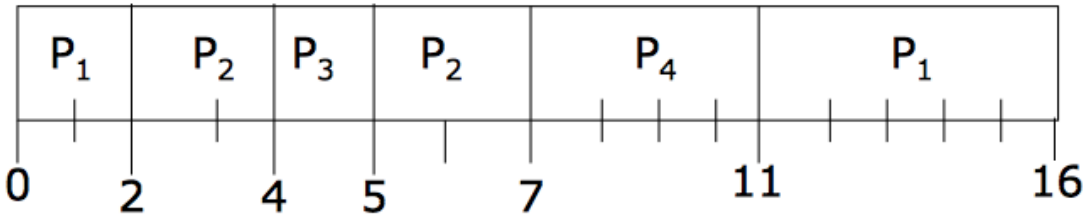
- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Proses	Geliş Zamanı	Burst Zamanı
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (**preemptive**)



- Ortalama bekleme süresi = ((11-2) + (5-4) + (4-4) +(7-5))/4
- Ortalama bekleme süresi = (9 + 1 + 0 +2)/4 = 3

Priority Scheduling

- A priority number (integer) is associated with each process
- **Her process için öncelik numarası verilir.**
- The CPU is allocated to the process with the highest priority **(smallest integer ≡ highest priority)**
  - Preemptive (Kesen)
  - Nonpreemptive(Kesmeyen)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute

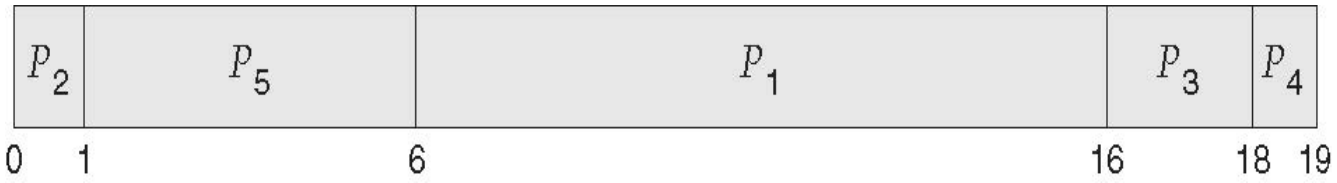
(Açlık: önemi düşük processler hiç bir zaman işlenemeyebilir-önüne gelen önceliği yüksek processler neden olur)

- Solution ≡ **Aging** – as time progresses increase the priority of the process(bekleyen processin önceliği zamanla yükselir).

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

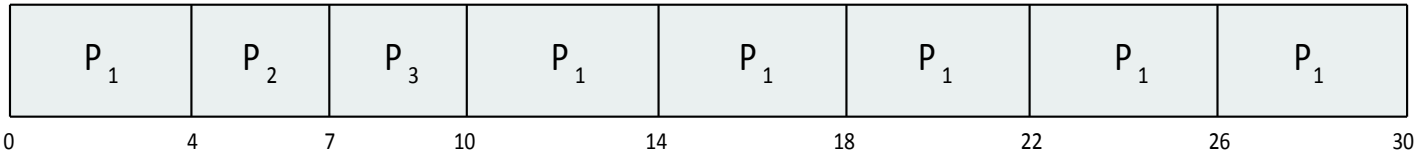
Zaman Dilimli - İşlemci Zamanlama Algoritması

- Round Robin (RR)
- 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* ortam değiştirme süresine oranla daha büyük olmalıdır. Aksi halde sistem verimsiz çalışır

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

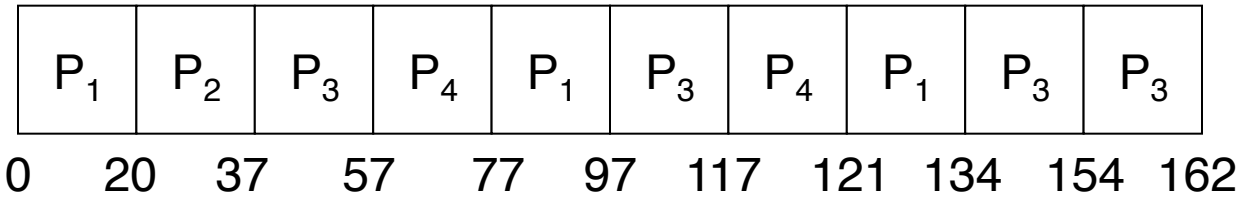


- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Quantum Time=20, RR

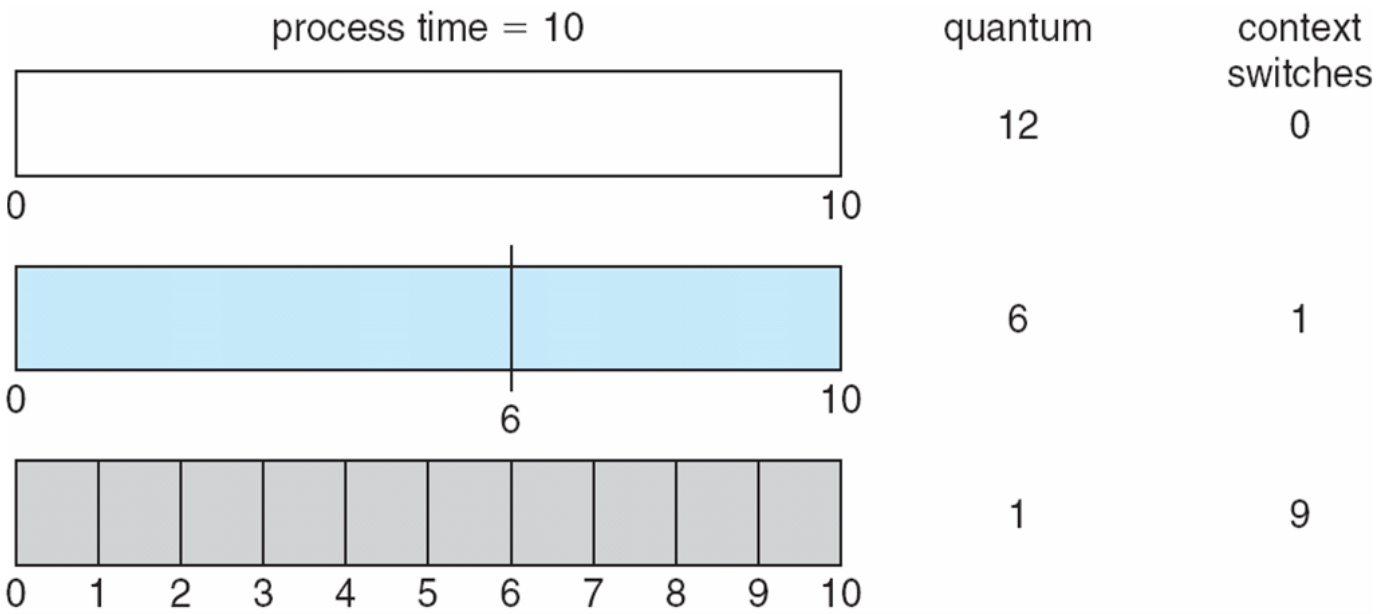
Process	Servis Zamanı
P_1	53
P_2	17
P_3	68
P_4	24

- Gantt chart:



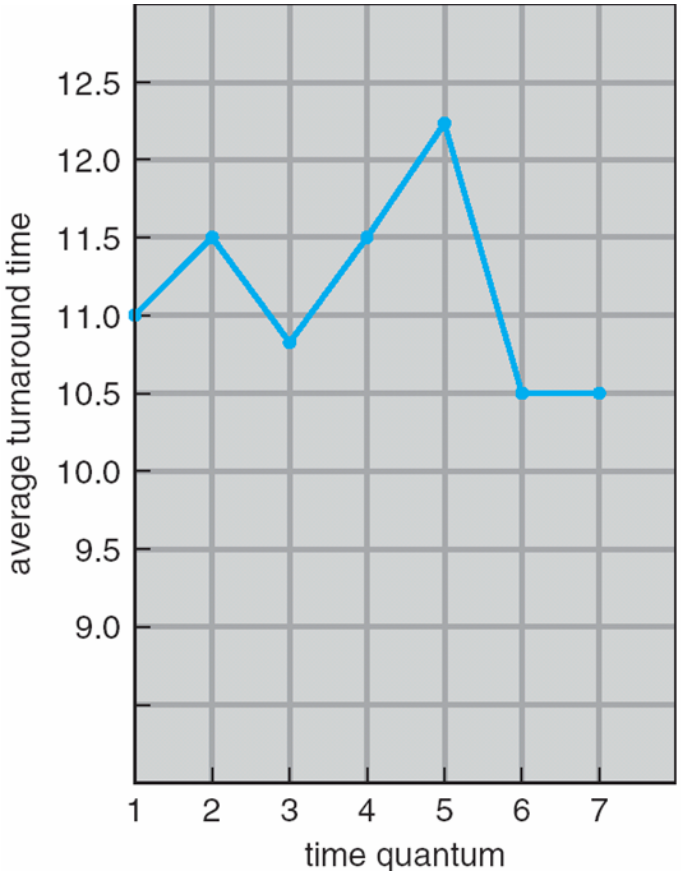
- Waiting Time: P1:57+24=81; P2:20; P3:37+40+17=94; P4:57+40=97

Time Quantum and Context Switch Time



Uygulamada, modern sistemler 10-100 milisecond arasında quantum time belirler.
context switch time <10 milisecond

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

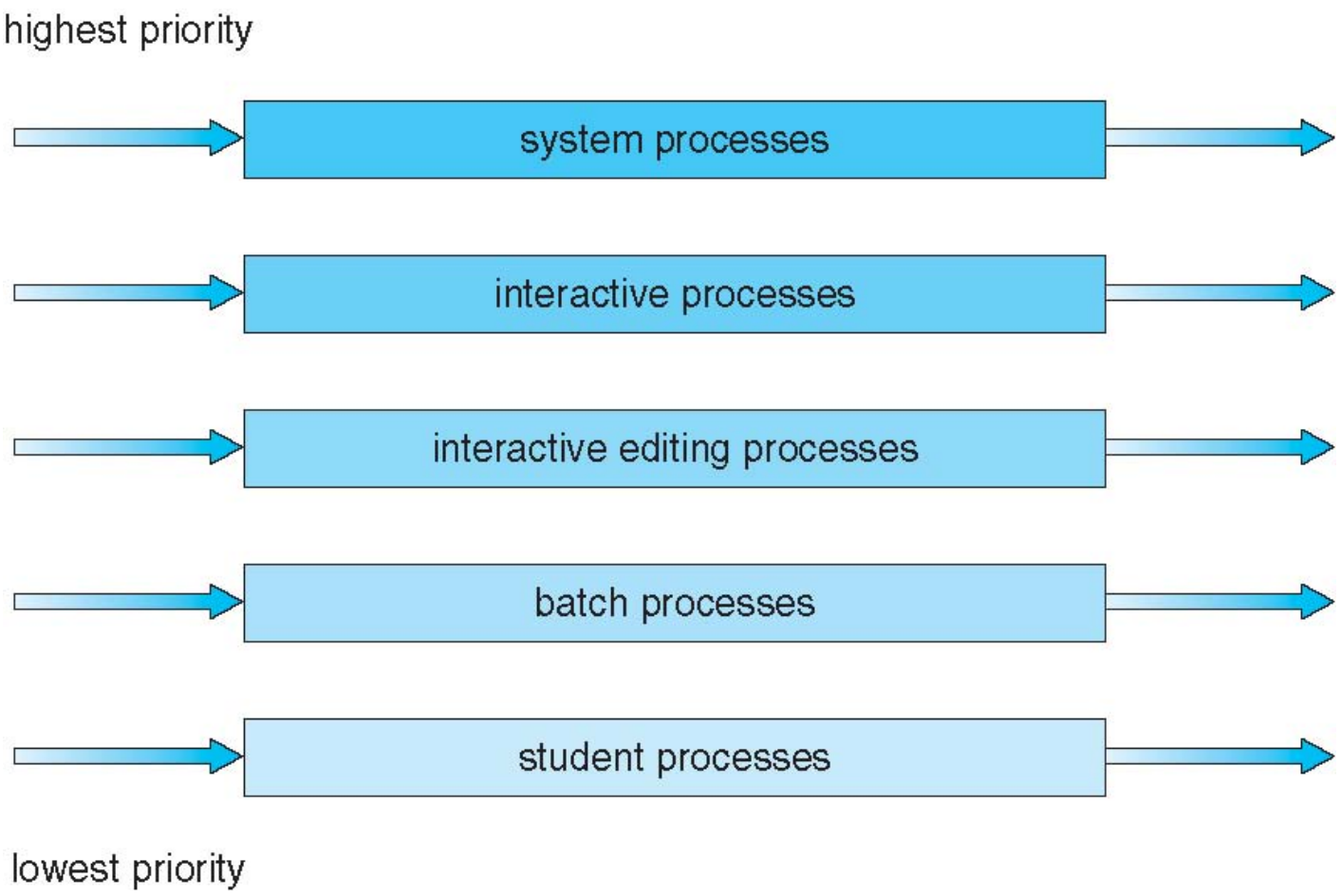
Multilevel Queue…

- Ready queue is partitioned into separate queues, eg: **çalışma kuyruğu alt kuyruklara bölünür**
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm: (her kuyruk kendi çalıştırma algoritmasını çalıştırır)
  - foreground – RR
  - background – FCFS

···Multilevel Queue

- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS
- **Zamanlama 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 kendi işlemlerine verebilir**
  - **Örn: CPU'nun %80 zamanı RR ile ön plan işlerine %20'si ise FCFS ile arka plan işlerine ayrılabilir**

Multilevel Queue Scheduling



Multilevel Feedback Queue

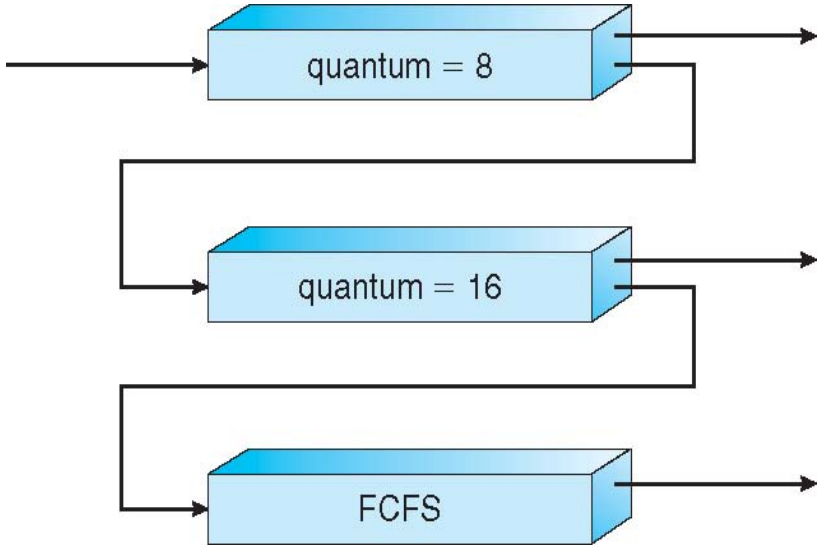
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process.
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queue

- **Processler kuyruklar arasında hareket edebilir, eğer aging işlemi yapılırsa.**
- Çok-seviye geri besleme kuyruğu zamanlayıcısı aşağıdaki parametrelerle tanımlanır:
  - Kuyrukların sayısı
  - Her bir kuyruk için zamanlama 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

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



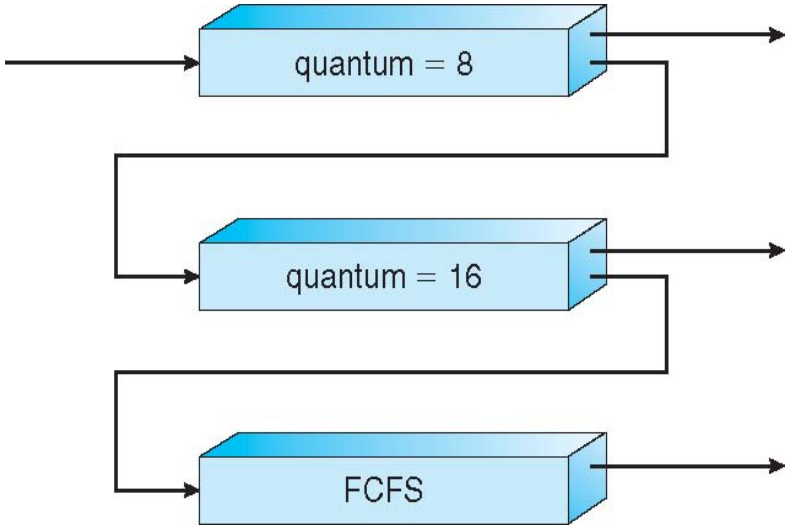
Example of Multilevel Feedback Queue

- 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



Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Thread Scheduling

- User-level thread' lerin Cpu' da çalıştırılabilmeleri için kernel-level thread' lere eşleştirilmeleri gerekir.
- **Contention Scope:** user-level ve kernel-level thread' ler arasındaki bir ayrım Cpu da nasıl düzenlendikleri ile ilgilidir.
- many-to-one şemasını uygulayan sistemlerde ve many-to-many modellerde, thread library, uygun bir LWP(light weight process) 'de koşmaları için user-level thread' leri düzenler. Bu şema **process contention scope (PCS) olarak bilinir.**
- Kernel, hangi kernel-level thread in CPU' da çalıştırılması gerektiğini bilmek için **system-contention scope (SCS)** kullanır.
- one-to-one modelini kullanan sistemler(Windows, Linux ve Solaris) thread leri sadece SCS kullanarak düzenler.
- Tipik olarak, **PCS önceliğe göre yapılır**—düzenleyici runnable thread' i en yüksek öncelik değerine göre seçer. User-level thread öncelikleri programcı tarafından kurulur.

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM
- **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, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* 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)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- Tüm düzenleme kararları (scheduling decisions), I/O processing, ve diğer sistem aktiviteleri tekil bir işlemci ile yürütülür (master server). Diğer işlemciler sadece user code çalıştırırlar. Bu **asymmetric multiprocessing yaklaşımında sadece tek bir işlemci sistemin veri yapılarına erişebilir**, bu da veri paylaşımını küçültür.

- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common

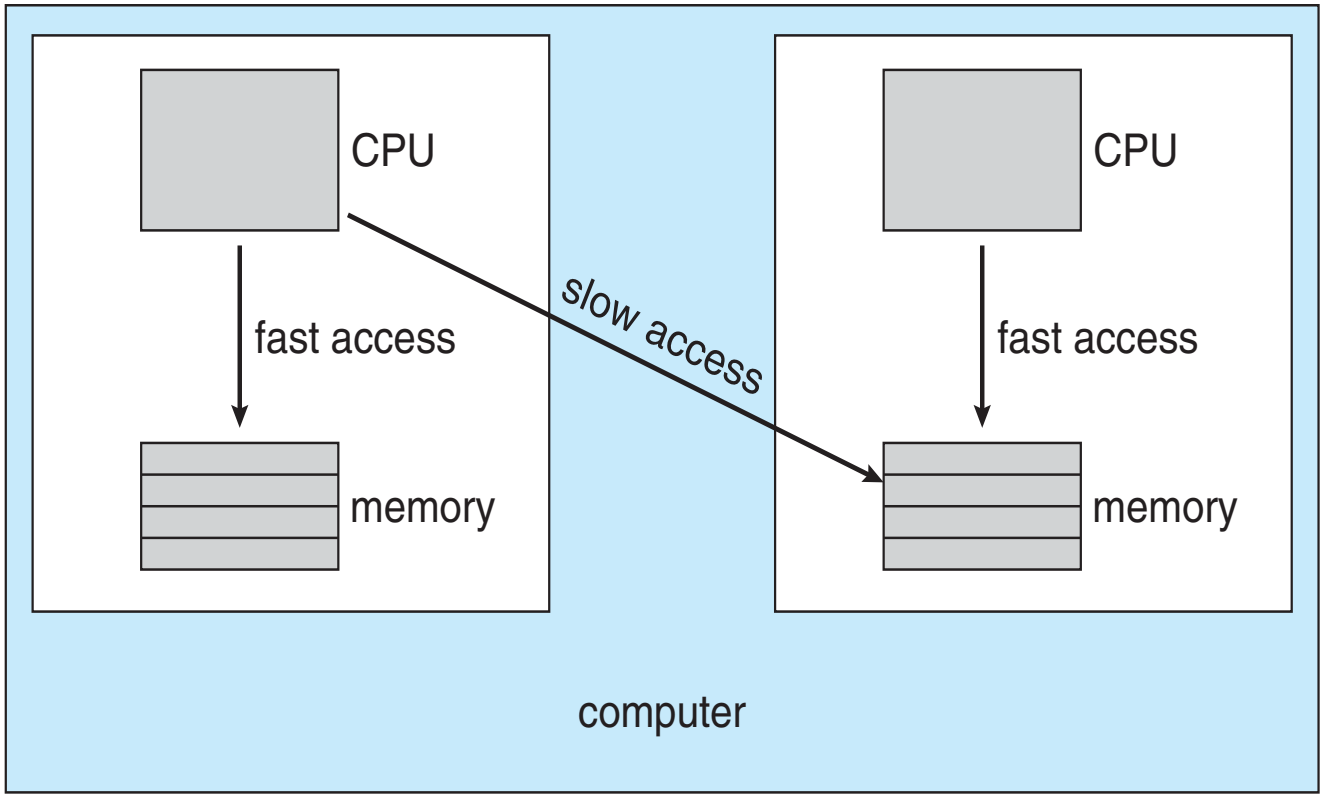
  ikinci yaklaşım **symmetric multiprocessing (SMP)** kullanır**. Her işlemci kendi düzenleme mekanizması ile yönetilir.**

  Tüm prosesler genel bir ready kuyruğundadır, yada her işlemci hazır prosesler için kendi özel kuyruğuna sahiptir. Düzenleme için her işlemci için olan düzenleyici, ready kuyruğunu kontrol eder ve işletilecek prosesi seçer. Modern işletim sistemleri SMP' yi destekler. (Windows, Linux, ve Mac OS X)

Birden fazla CPU'nun paylaşılan veri yapısına erişimi engellenmelidir. Birden fazla CPU'nun aynı prosesi çalıştırması engellenmelidir.

- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**
- Bir proses başka bir işlemciye aktarıldığında, eski işlemcideki cache bellek bilgileri aktarılmaz. Yeni aktarılan işlemcinin cache bellek bilgileri oluşana kadar hit rate oranı çok düşük kalır.
- Bir proses çalışmakta olduğu işlemci ile ilişkilendirilir (processor affinity) ve sonraki çalışacağı işlemci de aynı olur.
- Bazı sistemlerde, proses bir işlemciye atanır, ancak aynı işlemcide çalışmayı garanti etmez (soft affinity).
- Bazı sistemlerde, process bir işlemciye atanır ve her zaman aynı işlemcide çalışmayı garanti eder (hard affinity).
- Linux işletim sistemi soft affinity ve hard affinity desteğine
- sahiptir.

NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

- Yük dengeleme (load balancing), SMP sistemlerde tüm işlemciler üzerinde iş yükünü dağıtarak verimi artırmayı amaçlar. Her işlemcinin kendi kuyruğuna sahip olduğu sistemlerde, yük dengeleme iyi yapılmazsa bazı işlemciler boş beklerken diğer işlemciler yoğun çalışabilir. Ortak kuyruk kullanan sistemlerde yük dengelemeye ihtiyaç olmaz.

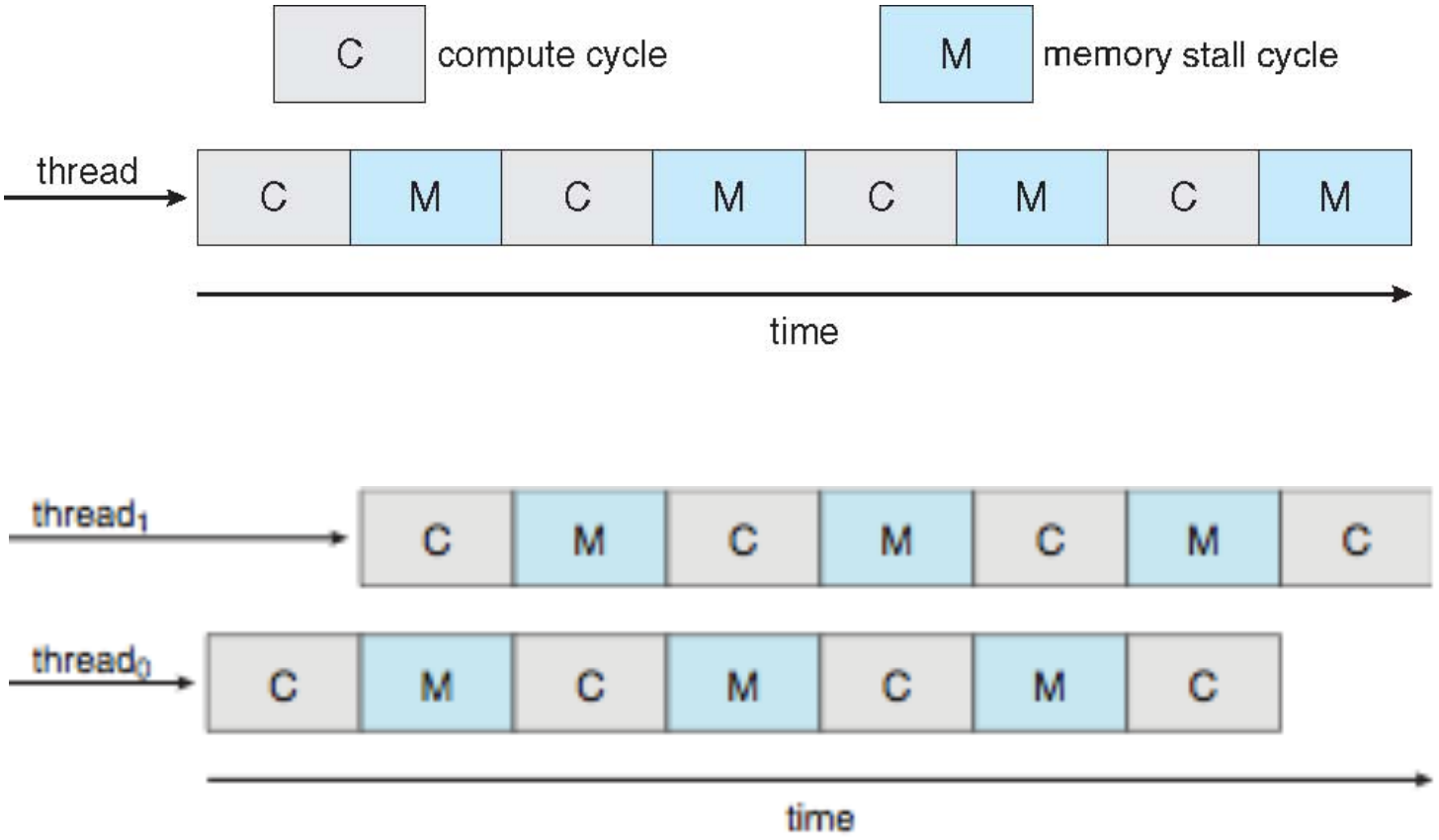
Yük dağılımı için iki yöntem kullanılır: push migration ve pull migration.

- Push migration yönteminde: bir görev işlemcilerin iş yükünü kontrol eder ve boş olanlara dolu olan diğer işlemcilerdeki prosesleri aktarır.
- Pull migration yönteminde: boş kalan işlemci dolu olan diğer işlemcilerde bekleyen bir prosesi kendi üzerine alır

Multicore Systems

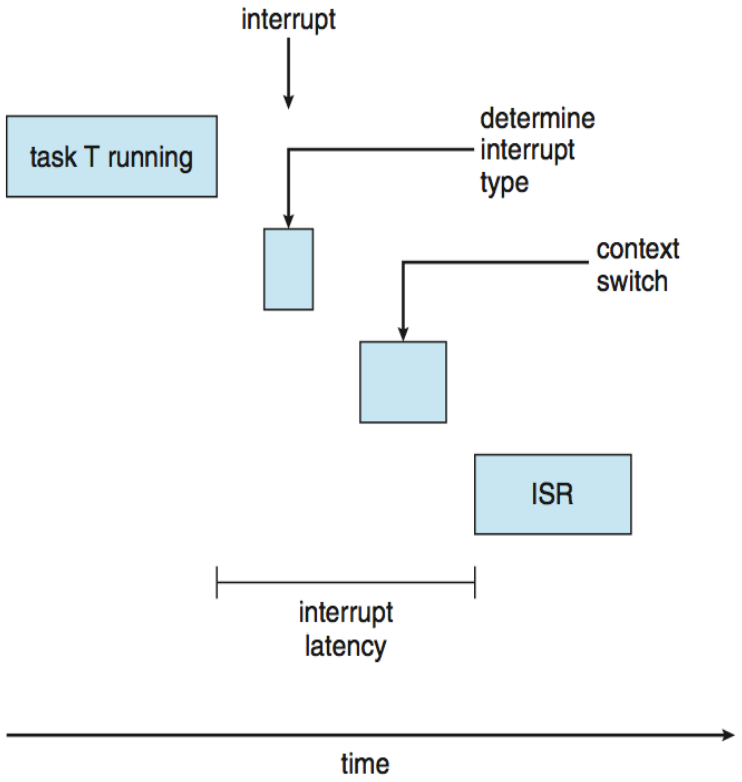
- Bir multithreaded multicore processor düzenleme için iki farklı seviyeye ihtiyaç duyar.
- **Birinci seviyede**, işletim sistemi tarafından yapılması gereken scheduling kararları: hangi software thread hangi hardware thread (logical processor) üzerinde koşacak. Bu aşama için işletim sistemi herhangi bir scheduling algorithm seçebilir.
  - her hardware thread, bir software thread çalıştırabilecek bir logical processor olarak görünür.
- **İkinci seviyede,** herbir core hangi hardware thread i çalıştıracağına nasıl karar verir? Bunun için farklı stratejiler vardır.
- Örn:UltraSPARC T3 CPU:
  - her çipte 16 core
  - her core da 8 hardware threads
  - UltraSPARC T3: her core'daki 8 hardware threads için round robin algoritması kullanır.

Multithreaded Multicore System



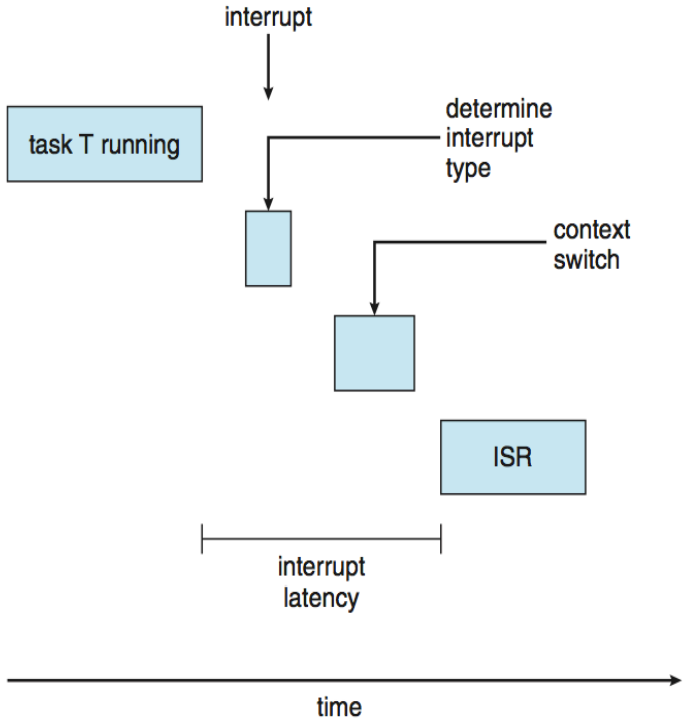
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



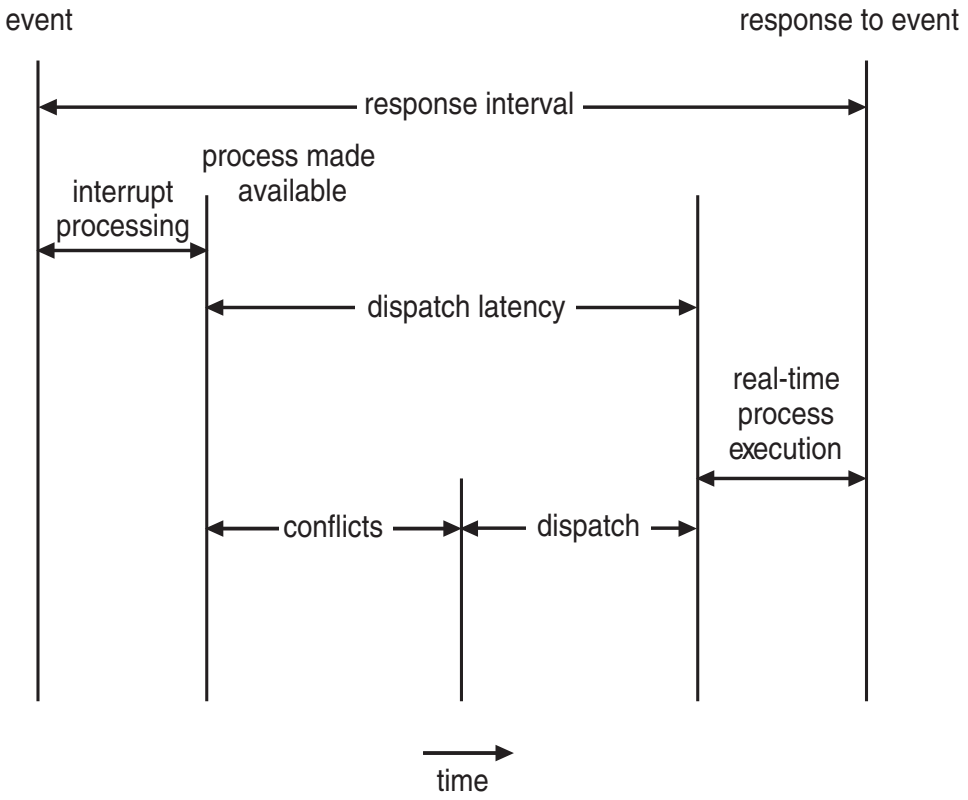
Real-Time CPU Scheduling

- Bariz zorluklar sunabilir:
- Yumuşak(soft) gerçek zamanlı sistemler - kritik gerçek zamanlı sürecin ne zaman programlanacağına dair hiçbir garanti yok
- Zor(hard) gerçek zamanlı sistemler - görev son teslim tarihine kadar yerine getirilmelidir
- İki tür gecikme süresi performansı etkiler
  - Kesinti gecikme süresi - kesintinin varışından itibaren hizmetin kesildiği rutinin başlangıcına kadar geçen süre
  - Gönderme gecikmesi - mevcut işlemi CPU'dan alma ve bir başkasına geçme zamanı



Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes

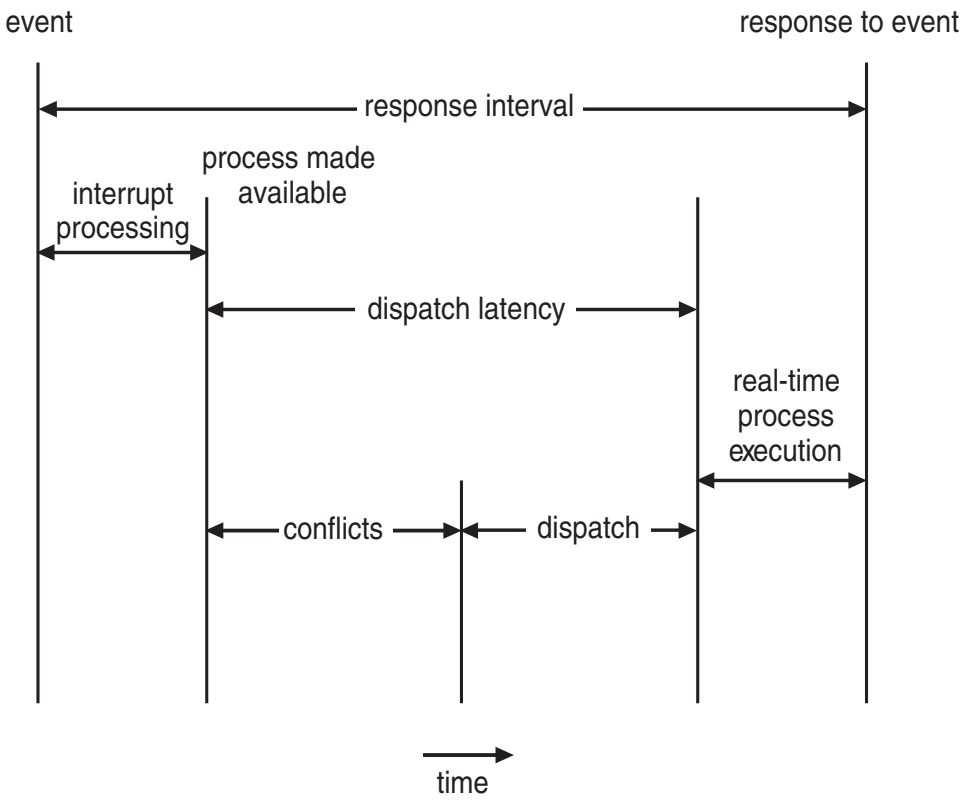


Real-Time CPU Scheduling (Cont.)

- Gönderim gecikmesinin çakışma aşaması:

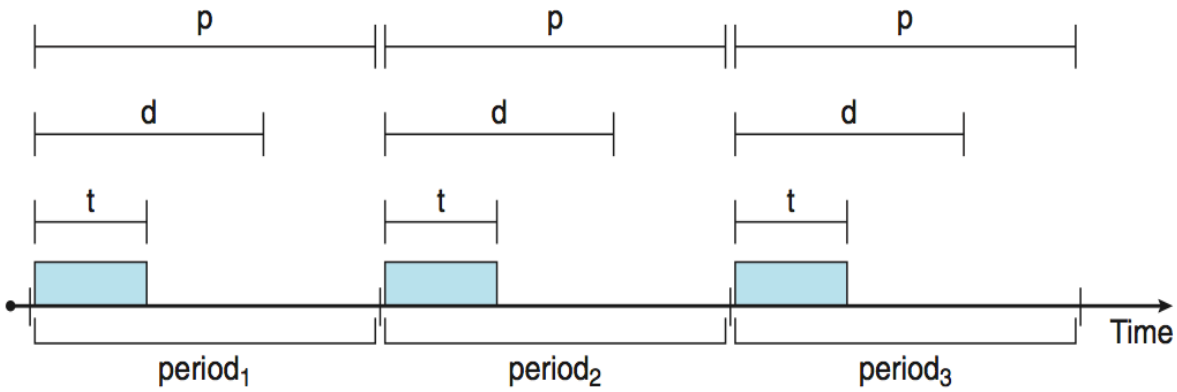
1.Çekirdek modunda çalışan herhangi bir işlemin önlenmesi

2.Yüksek öncelikli süreçler tarafından ihtiyaç duyulan düşük öncelikli kaynak işlemiyle serbest bırakma



Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$

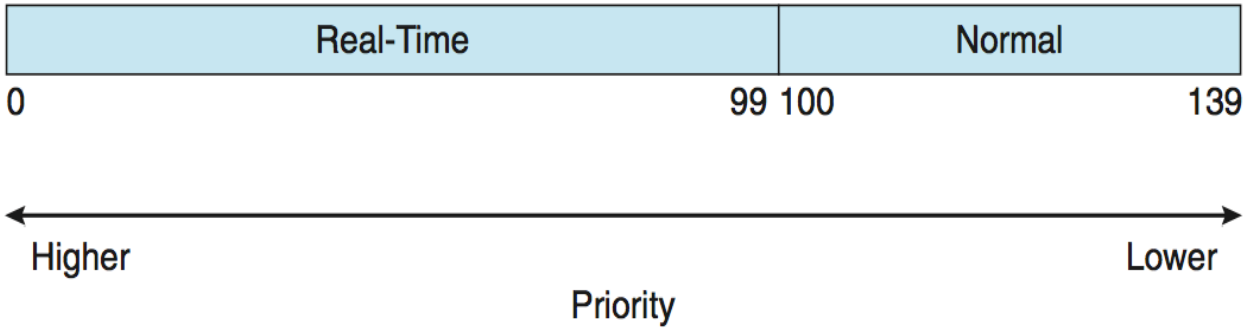


Linux Scheduling

- **Default Linux scheduling algorithm: *Completely Fair Scheduler* (CFS)**
- Linux sistemlerde CPU düzenleme **scheduling sınıflarına** dayanır. Her sınıf bir öncelik değerine sahiptir. Farklı düzenleme sınıflarını kullanarak, sistemin ihtiyaçlarına göre farklı düzenleme algoritmaları çalıştırılabilir.
- Bir Linux server için düzenleme kriteri, örneğin bir mobil aygıttan farklı olabilir. Bir sonra işletilecek göreve karar vermek için, düzenleyici, highest-priority düzenleme sınıfına ait bir highest-priority task seçer.
- Standard Linux kernel iki düzenleme sınıfı uygular:
  - (1) default scheduling class using the CFS scheduling algorithm
  - (2) a real-time scheduling class.

Linux Scheduling (Cont.)

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Windows Scheduling

- Windows scheduler highest-priority thread in her zaman ilk çalıştırılacağını garanti eder.
- Düzenlemeden sorumlu Windows kernel kısmı **dispatcher** olarak adlandırılır.
  - Dispatcher tarafından seçilen bir thread daha yüksek öncelikli iş ile bölünene kadar yada teminate olana kadar yada I/O call – Sys Call ile bölünene kadar çalışmaya devam eder.
- Düşük öncelikli iş çalıştırılırken yüksek öncelikli hazır duruma geçerse, düşük öncelikli olan ertelenir.
- Öncelikler iki sınıfa ayrılmıştır. **variable class:** priorities from 1 to 15,
- **real-time class:** priorities ranging from 16 to 31.
- priority 0: used for memory management.
- dispatcher her scheduling priority için bir kuyruk kullanır ve en yüksekten düşüğe doğru kuyrukları tarar yeni bir ready state thread bulmak için. Eğer bulamaz ise, dispatcher **idle thread** adlı özel bir thread işletir.

	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

Figure 6.22 Windows thread priorities.

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Solaris Scheduling

- Öncelik tabanlı düzenleme gerçekleştirir. Her thread aşağıdaki öncelik sınıflarından birine aittir.
- **1.** Time sharing (TS)
  2. Interactive (IA)
  3. Real time (RT)
  4. System (SYS)
  5. Fair share (FSS)
  6. Fixed priority (FP)
  Her sınıf için farklı düzenleme algoritması uygulanabilir.

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

Figure 6.23 Solaris dispatch table for time-sharing and interactive threads.

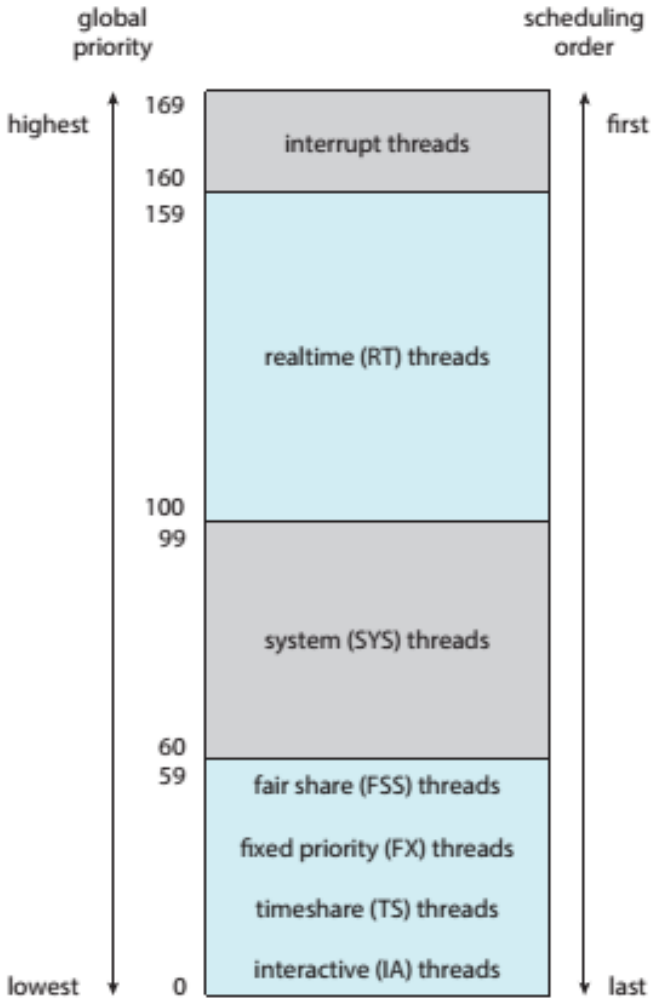


Figure 6.24 Solaris scheduling.

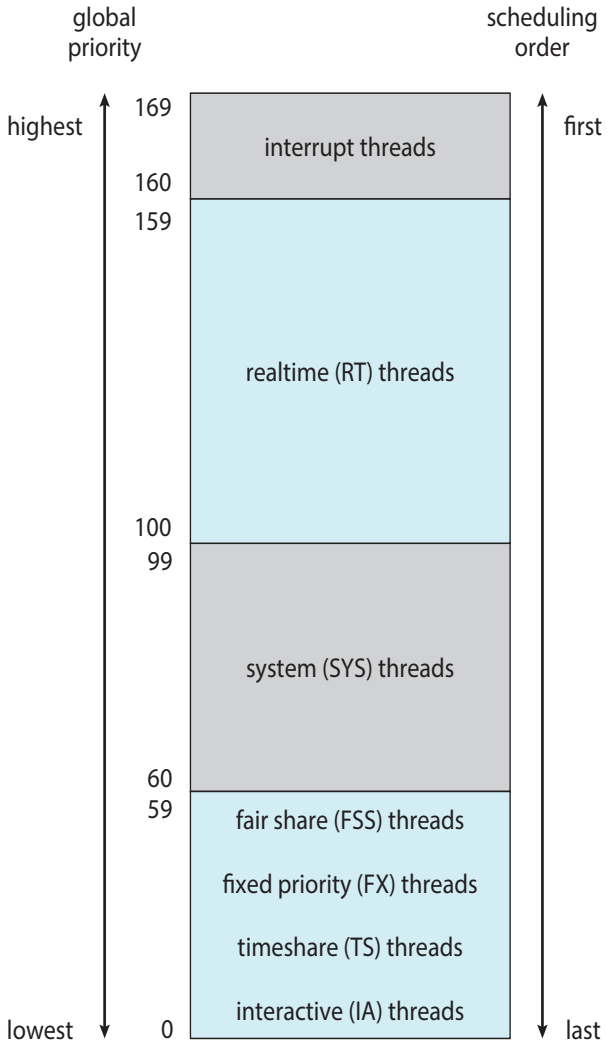
Solaris

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris 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 Scheduling



Algorithm Evaluation

- **Simulations:** en genel yöntem,proses yaratmak için bir random-number generator kullanmaktır. (process Id, CPU burst times, arrivals according to probability distributions). Dağılımlar uniform, exponential, Poisson dağılımlar olarak tanımlanabilir.
- Bir distribution-driven simulasyon gerçek sistemlerde geçerli olmayabilir. Bunun için **trace tapes** modeli kullanılır. Olayların sadece oluş frekansı değil aynı zamanda sequence'i kaydedilir. Ve bu sequence bilgisi simulasyon için kullanılır.

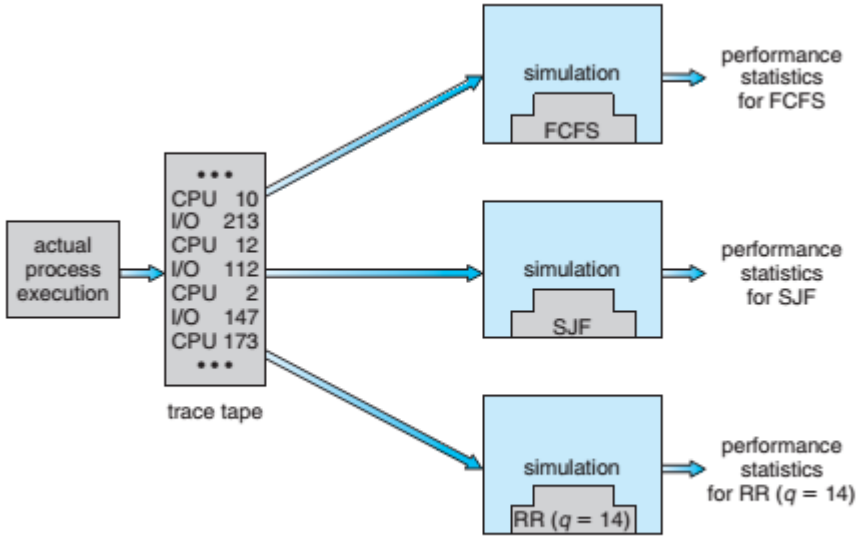


Figure 6.25 Evaluation of CPU schedulers by simulation.

Algorithm Evaluation

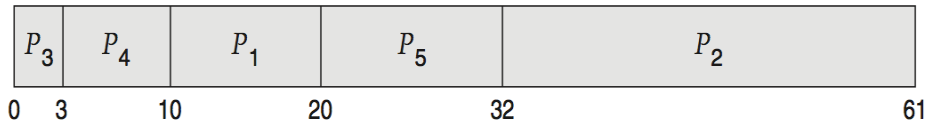
- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

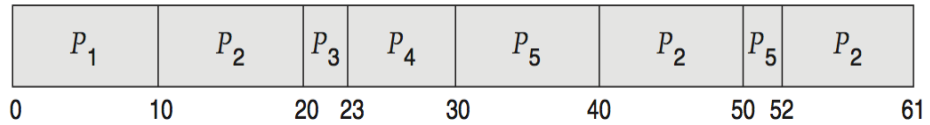
Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:

  - Non-preemptive SFJ is 13ms:



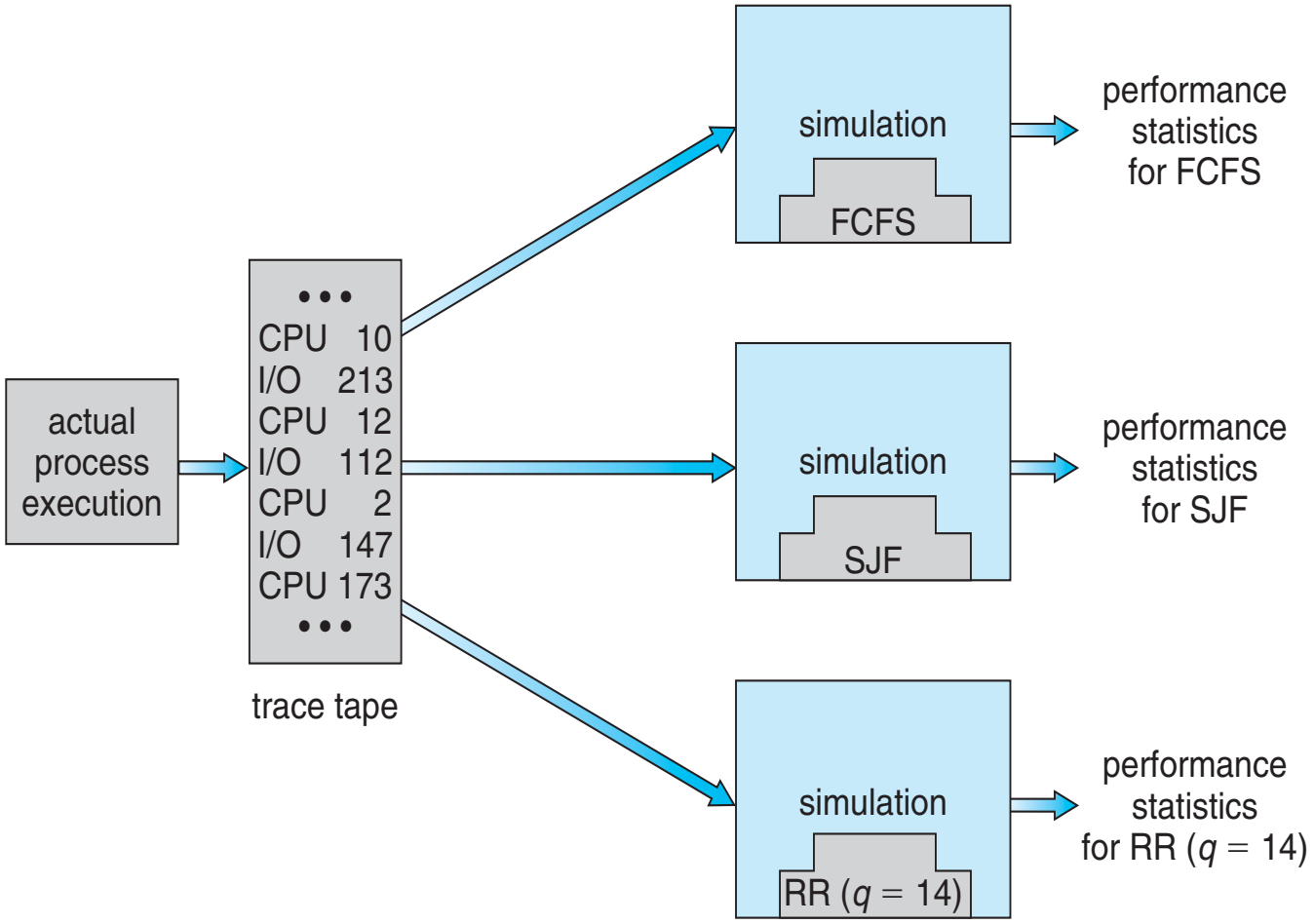
  - RR is 23ms:



Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



End of Chapter 6