İşletim Sistemleri
CPU Scheduling

Dr. Öğr. Üyesi Ertan Bütün

Bu dersin içeriği hazırlanırken Operating System Concepts (Silberschatz, Galvin and Gagne) kitabı ve
Prof. Dr. M. Ali Akcayol'un (Gazi Üniversitesi Bilgisayar Mühendisliği Bölümü) ders sunumlarından faydalanılmıştır.

https://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/index.html

http://w3.gazi.edu.tr/~akcayol/BMOS.htm

Konular

- Temel kavramlar
- Scheduling kriterleri
- Scheduling algoritmaları
- Thread Scheduling
- Multiple-Processor Scheduling
- Gerçek zamanlı (Real-Time) CPU scheduling
- İşletim Sistemlerinde Scheduling Örnekleri
- Belirli Bir Sistem İçin Bir Scheluling Algoritması Nasıl Seçilir?

Thread Scheduling

- Thread'lerin desteklendiği OS'lerde process'ler yerine thread'ler schedule edilir.
- User-level thread'ler bir thread kütüphanesi tarafından yönetilir, kernel bunların farkında değildir.
- User-level ve kernel-level thread'ler farklı şekilde schedule edilir.

User-Level Thread Scheduling

- user-level düzeyindeki thread'lerin bir CPU'da çalıştırılması için kernel düzeyinde ilgili bir thread'le eşleşmelidir, ancak bu eşleşme dolaylı olabilir ve bir lightweight process (LWP) bunun için kullanabilir.
- many-to-one ve many-to-many modelleri uygulayan sistemlerde, thread kütüphanesi, user-level thread'leri kullanılabilir bir LWP'de çalışacak şekilde schedule eder.
  - Bu şema, **process-contention scope (PCS)** olarak bilinir, çünkü CPU için rekabet, aynı process'e ait olan thread 'ler arasında gerçekleşir.
  - Tipik olarak, **PCS** önceliğe (priority) göre yapılır — scheduler, çalıştırılması için en yüksek önceliğe sahip thread'i seçer. User-level thread öncelikleri, programcı tarafından belirlenir.

Kernel-Level Thread Scheduling

- Kernel, bir CPU'da hangi kernel-level thread'in schedule edileceğine karar vermek için **system-contention scope (SCS)** kullanır. SCS scheduling ile CPU için rekabet, sistemdeki tüm thread'ler arasında gerçekleşir.
- Windows, Linux ve Solaris gibi one-to-one modeli kullanan sistemler, yalnızca SCS kullanarak thread'leri schedule eder.

Pthread Scheduling

- POSIX Pthread API thread oluşturma sırasında PCS veya SCS'nin belirlenmesine izin verir.
  - PTHREAD_SCOPE_PROCESS, PCS scheduling kullanarak thread'leri schedule eder.
  - PTHREAD_SCOPE_SYSTEM , SCS scheduling kullanarak thread'leri schedule eder.
- OS'leri tarafından sınırlandırılabilir, örneğin Linux and macOS sadece PTHREAD_SCOPE_SYSTEM'e izin verir.

Pthread Scheduling

```
#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

```
   /* 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);
}
```

Konular

- Temel kavramlar
- Scheduling kriterleri
- Scheduling algoritmaları
- Thread Scheduling
- Multiple-Processor Scheduling
- Gerçek zamanlı (Real-Time) CPU scheduling
- İşletim Sistemlerinde Scheduling Örnekleri
- Belirli Bir Sistem İçin Bir Scheluling Algoritması Nasıl Seçilir?

Multi-processor scheduling

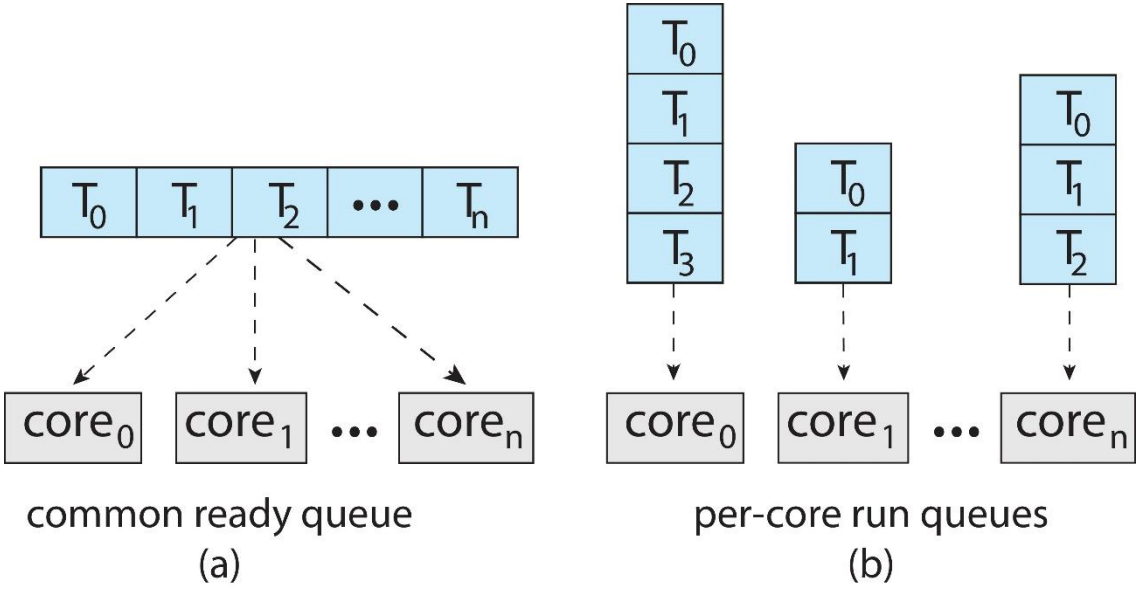
- Multi-processor kullanılan sistemlerde scheduling yaparken yük paylaşımı (load sharing) kullanılabilir, ancak scheduling çok daha karmaşık hale gelir.
- Geleneksel olarak, multiprocessor terimi, her işlemcinin bir tek çekirdekli CPU içerdiği birden çok fiziksel işlemci sağlayan sistemlere atıfta bulunur.
  - Bununla birlikte, multiprocessor tanımı önemli ölçüde gelişmiştir ve modern bilgi işlem sistemlerinde, multiprocessor artık aşağıdaki sistem mimarileri için geçerlidir:
    - Multicore CPUs
    - Multithreaded cores
    - NUMA systems
    - Heterogeneous multiprocessing
- Burada, bu farklı mimariler bağlamında multiprocessor scheduling ele alınacaktır.

Multi-processor scheduling

- Multiprocessor bir sistemde scheduling'e yönelik bir yaklaşım: tüm scheduling kararlarının, I/O işlemlerinin ve diğer sistem işlerinin tek bir işlemci (ana sunucu, master CPU) tarafından gerçekleştirilmesidir. (asymmetric multiprocessing)
    - Diğer CPU'lar kullanıcı kodlarını çalıştırır.
- Bu şekilde bir asymmetric multiprocessing basittir, çünkü sistem veri yapılarına yalnızca bir çekirdek erişir ve veri paylaşımına olan ihtiyacı azaltır.
- Bu yaklaşımın olası bir dezavantajı, ana sunucunun genel sistem performansını düşürecek şekilde bir darboğaz haline gelebilmesidir.

Multi-processor scheduling

- Multiprocessor sistemleri desteklemek için standart bir yaklaşım ise: her işlemcinin kendi scheduling algoritmasına sahip olduğu symmetric multiprocessing yaklaşımdır(SMP).
- Bu yaklaşımda master CPU yoktur.
- Scheduling için iki farklı strateji vardır:
  - Tüm thread'ler ortak hazır kuyruğuna sahip olabilir.
  - Ya da her işlemcinin kendi özel thread kuyruğu olabilir.



Multi-processor scheduling

- Birinci yaklaşımda paylaşılan hazır kuyruğu için race condition olabilir, bu nedenle, iki ayrı işlemcinin aynı thread'i schedule etmemesi ve thread'lerin kuyruktan kaybolmaması sağlanmalıdır.
    - Bunun için kilitleme (locking) tabanlı bir senkronizasyon yöntemi kullanılmalıdır. Ancak kuyruğa tüm erişimler kilidi almayı gerektirdiğinden ve paylaşılan kuyruğa erişim büyük olasılıkla bir darboğaz oluşturacağından performans olumsuz etkilenebilir.

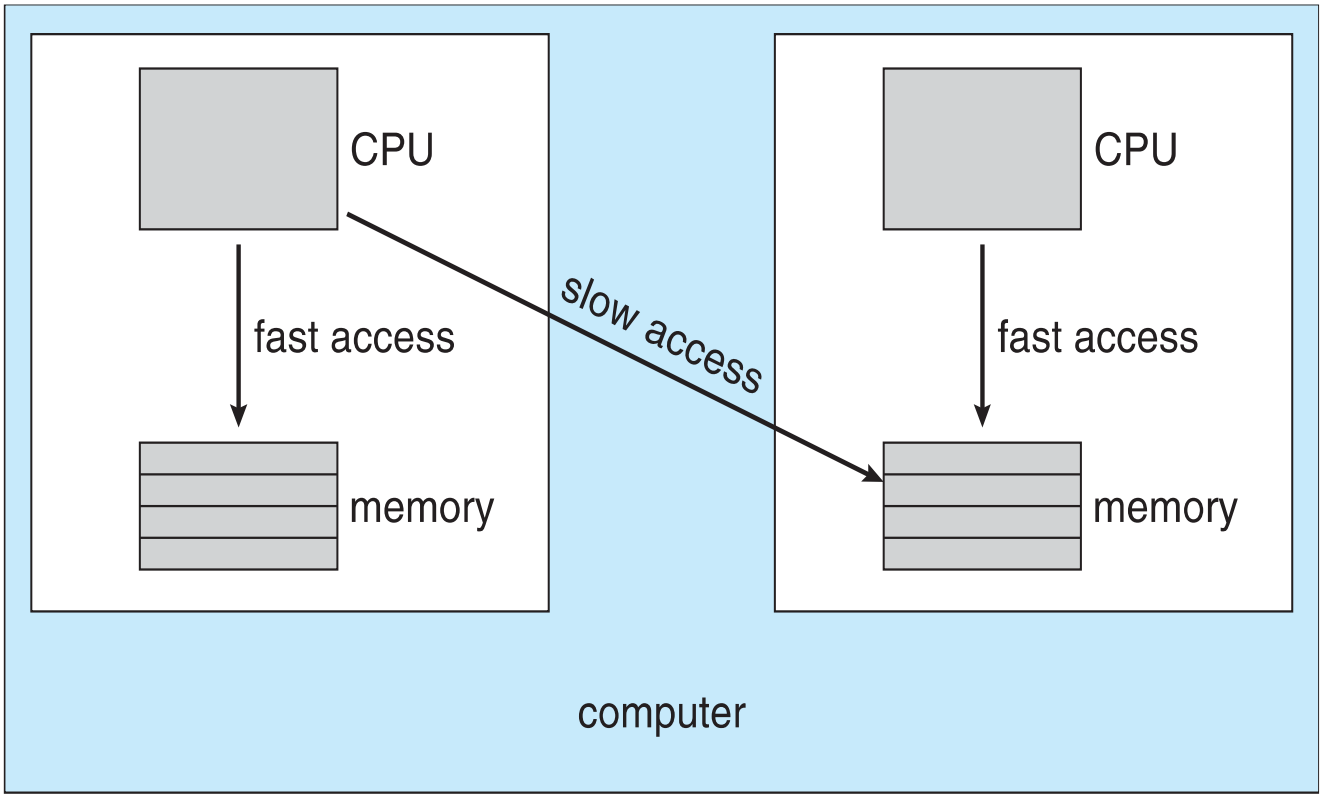
Multi-processor scheduling

- İkinci yaklaşımda her işlemcinin kendisine özle hazır (ready) kuyruğu olacağından böyle bir performans kaybı söz konusu değildir. Bu yüzden SMP sistemlerde en yaygın yaklaşımdır.
- Ancak bu yaklaşımda işlemci başına düşen hazır kuyruklarıyla ilgili **sorunlar** olabilmektedir, bunlardan en önemlisi işlemciler arasında değişken boyutlardaki iş yüklerinin olmasıdır.
    - Bunun için tüm işlemciler arasında iş yüklerini dengelemek için **dengeleme algoritmaları** kullanılabilir.
- Hemen hemen tüm modern işletim sistemleri, Windows, Linux ve macOS'un yanı sıra Android ve iOS mobil işletim sistemleri de SMP'yi destekler.
- Bu bölümün geri kalanında, multi-processor scheduling konusu SMP sistemler için ele alınacaktır.

Multi-processor scheduling

- Bir process **başka bir işlemciye** aktarıldığında, process'in eski işlemcideki cache bellek bilgilerinin geçersiz kılınması gerekir, yeni aktarılan işlemcide cache bellek bilgilerinin yeniden yüklenmesi gerekir.
- Bu işlemlerin yüksek maliyeti nedeniyle çoğu SMP sistem, process'lerin bir işlemciden diğerine taşınmasını önlemeye çalışır.
- Bunun yerine bir process'i **aynı işlemcide** yürütmeye çalışır. Bu, işlemci yakınlığı (processor affinity) olarak bilinir; yani bir process'in üzerinde çalıştığı işlemciye yakınlığı vardır.
  - Bazı sistemlerde process bir işlemciye atanır, ancak aynı işlemcide çalışması garanti edilmez (soft affinity).
  - Bazı sistemlerde process bir işlemciye atanır ve her zaman aynı işlemcide çalışması garanti edilir (hard affinity).
  - Linux işletim sistemi soft affinity ve hard affinity desteğine sahiptir.

NUMA and CPU Scheduling



Multi-processor scheduling– Load Balancing

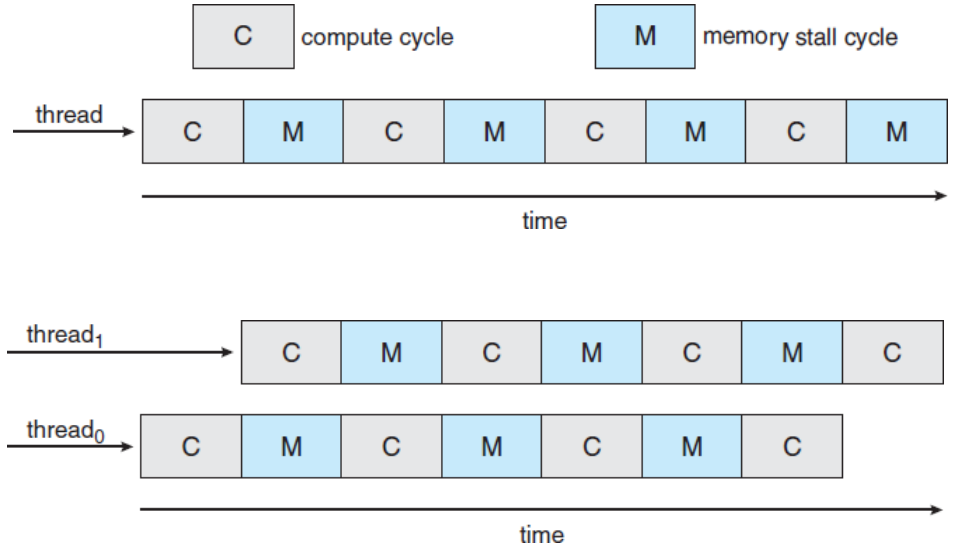
- **Yük dengeleme** (load balancing), SMP sistemlerde tüm işlemciler üzerinde iş yükünü dağıtarak verimi artırmayı amaçlar.
- Ortak kuyruk kullanan sistemlerde **yük dengelemeye** ihtiyaç olmaz.
- Her işlemcinin kendi kuyruğuna sahip olduğu sistemlerde, **yük dengeleme gereklidir ve** iyi yapılmazsa bazı işlemciler boşta beklerken diğer işlemciler yoğun çalışabilir.
- Yük dengeleme için iki yöntem kullanılır: push migration ve pull migration.
- Push migration yönteminde periyodik olarak işlemcilerin iş yükü kontrol edilir, bir dengesizlik bulunursa aşırı yüklü olan işlemcilerdeki process'ler boş olan veya daha az meşgul olan işlemcilere aktarılır. (itilir-pushing)
- Pull migration yönteminde boş kalan işlemci, dolu olan diğer işlemcilerde bekleyen bir process'i kendi üzerine alır.(çeker-pull)
- push migration ve pull migration işlemlerinin birbirini dışlamasına gerek yoktur ve aslında genellikle yük dengeleme sistemlerinde paralel olarak uygulanır.

Multicore Processors

- Aynı fiziksel yonga üzerine birden çok işlemci çekirdeği yerleştirerek çok çekirdekli işlemci (**multicore** processor) oluşturma son zamanlarda işlemci üretiminde daha yaygın bir uygulamadır.
- **Multicore** processor sistemde her çekirdek işletim sistemine ayrı bir fiziksel işlemci olarak görünür. **Multicore** processor kullanan SMP sistemleri, her işlemcinin kendi fiziksel yongasına sahip olduğu sistemlerden **daha hızlıdır ve daha az güç tüketir**.

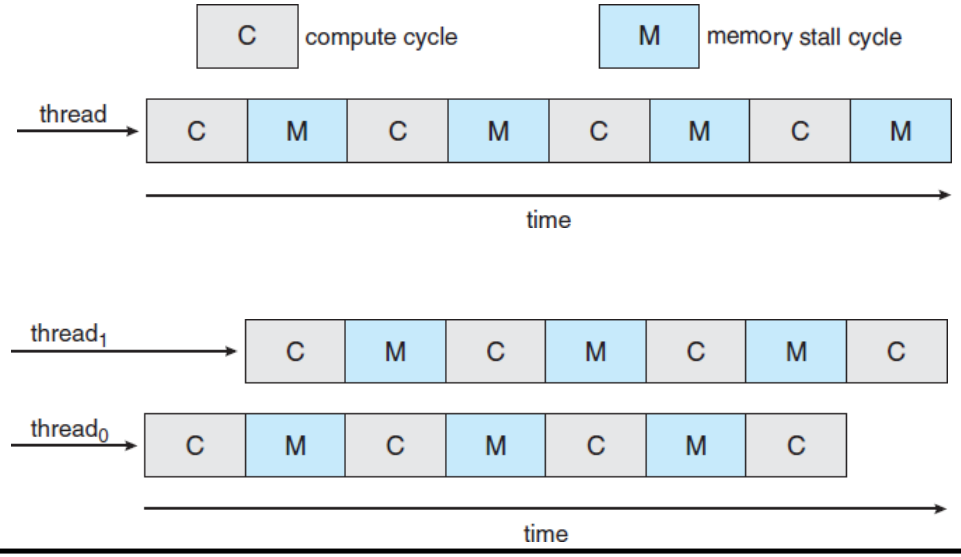
Multi-processor scheduling

- Çok çekirdekli (multi-core) işlemciler Scheduling konusunu karmaşık hale getirebilir. Araştırmacılar, bir işlemcinin belleğe eriştiğinde, verilerin erişilebilir olmasını beklerken önemli miktarda zaman harcadığını keşfetmişlerdir.
- Bellek durması (memory stall) olarak bilinen bu durum, önbellekte olmayan verilere erişim gibi çeşitli nedenlerle ortaya çıkabilir. Birinci şekilde bir memory stall gösterilmektedir. Bu senaryoda, işlemci zamanının yüzde 50'si kadarını verilerin bellekten kullanılabilir hale gelmesini bekleyerek geçirebilir.



Multi-processor scheduling

- Bu durumu gidermek için **her bir çekirdeğe iki veya daha fazla donanım thread'inin atandığı multithreaded işlemci çekirdekleri** uygulamıştır.
- Bu şekilde, bir thread bellek için beklerken durursa, çekirdek başka bir thread'e geçebilir. şekilde thread0 ve thread1'in aralarında geçiş yapılarak yürütüldüğü bir dual-threaded işlemci çekirdeği gösterilmektedir.
- Bir işletim sistemi açısından her donanım thread'i, bir yazılım thread'ini çalıştırmak için kullanılabilen mantıksal bir işlemci olarak görünür. Örneğin dual-threaded, dual-core bir sistem, işletim sistemi açısından dört mantıksal işlemci olarak kabul edilir.



Konular

- Temel kavramlar
- Scheduling kriterleri
- Scheduling algoritmaları
- Thread Scheduling
- Multiple-Processor Scheduling
- Gerçek zamanlı (Real-Time) CPU scheduling
- İşletim Sistemlerinde Scheduling Örnekleri
- Belirli Bir Sistem İçin Bir Scheluling Algoritması Nasıl Seçilir?

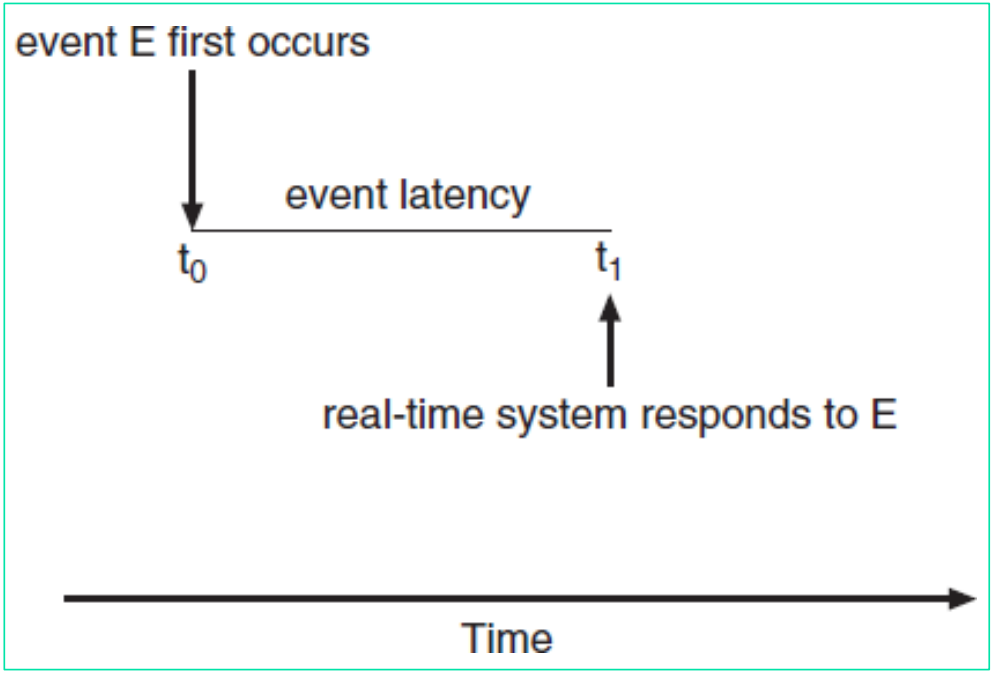
Real-Time CPU scheduling

- Gerçek zamanlı (Real-Time) sistemler iki gruba ayrılır:
  - Soft real-time sistemler
  - Hard real-time sistemler
- Soft real-time sistemlerde, kritik bir gerçek zamanlı process'in ne zaman yürütüleceğine dair hiçbir garanti verilmez, ancak kritik bir real-time process'in diğerlerine göre öncelikli yürütülmesine garanti verilir.
- Hard real-time sistemlerin daha katı gereksinimleri vardır. Bir görev için en son tamamlanması gereken zamana (deadline) kadar mutlaka hizmet verilmelidir; son teslim zamanının sona ermesinden sonraki verilen hizmet, hiç hizmet verilmemesiyle aynıdır.

Real-Time CPU scheduling

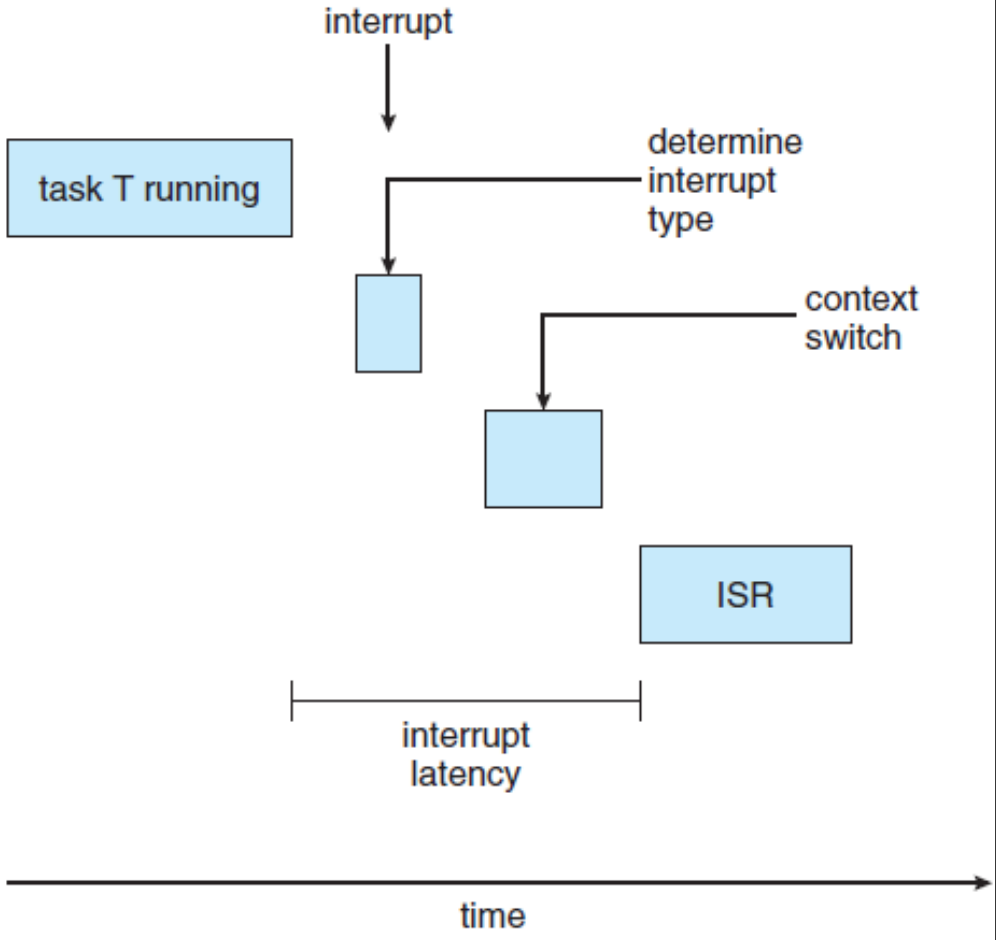
Minimizing latency

- Bir olay oluştuktan sonra ona cevap verilinceye kadar bir süre geçer (event latency).
- Sistemde bir olay gerçekleştiğinde, olabildiği kadar kısa sürede gerekli işlemin yapılması zorunludur.



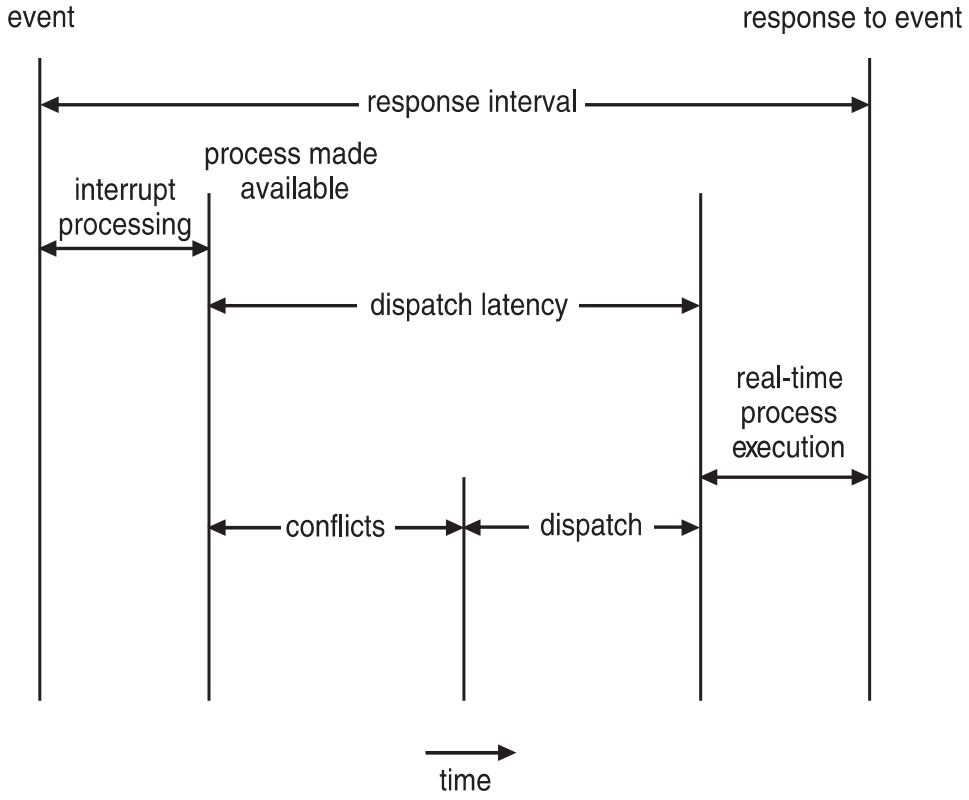
Real-Time CPU scheduling

- Gerçek zamanlı sistemlerin performansını iki tür gecikme etkiler:
  - interrupt latency
  - dispatch latency
- Interrupt latency, CPU'ya interrupt gelmesi ile CPU'nun istenen işleme başlaması için geçen süredir.



Real-Time CPU scheduling

- Bir process'in durdurulması ve diğer process'in başlatılması için geçen süreye dispatch latency (sevk gecikmesi) denir. Dispatch latency iki aşamadan oluşur: conflicts ve dispatch.
- Conflict aşaması iki kısımdan oluşur:
  1. Çekirdekte çalışan herhangi bir process'in durdurulması
  2. Yüksek öncelikli bir process'in ihtiyaç duyduğu kaynakların düşük öncelikli process'ler tarafından serbest bırakılması
- Conflict aşamasından sonra dispatch aşamasında, yüksek öncelikli process uygun bir CPU'ya schedule edilir.



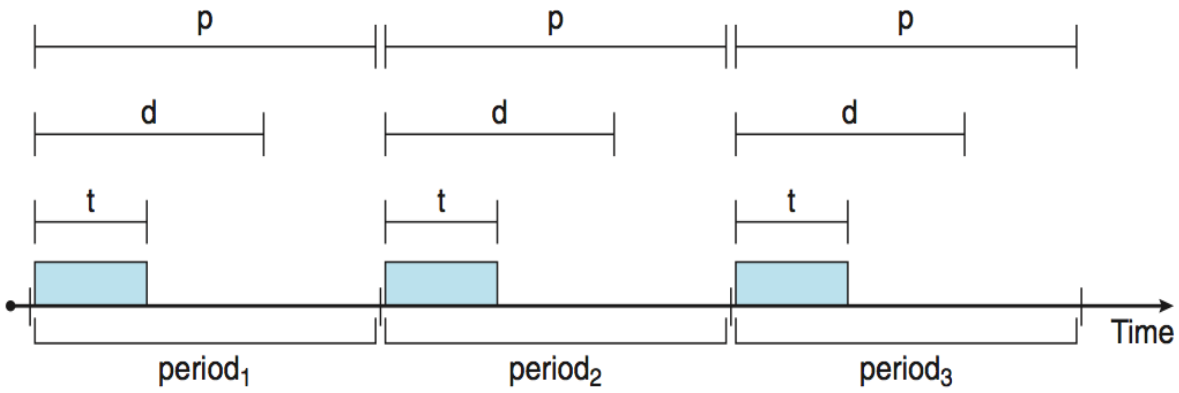
dispatch latency şeması

Real-Time Priority-based Scheduling

- Gerçek zamanlı bir işletim sisteminin en önemli özelliği, gerçek zamanlı process CPU'ya ihtiyaç duyduğu anda **anında yanıt vermesidir**.
- Bu yüzden gerçek zamanlı bir işletim sistemi için scheduler, preemption ile birlikte önceliğe dayalı (priority-based) bir algoritmayı desteklemelidir.
  - Preemptive systemde daha yüksek öncelikli bir process çalıştırılabilir hale geldiğinde CPU'da halihazırda çalışan bir process'in yürütülmesi askıya alınır.
- Bu şekildeki çalışma ile (preemptive, priority-based scheduler) **soft real-time** sistem özelliği garanti edilir.
    - Ancak bir process için deadline'dan önce çalışma garantisi yoktur.
- **Hard real-time** sistemler, gerçek zamanlı görevlerin deadline gereksinimlerine göre hizmet verileceğini garanti etmelidir ve bu tür garantilerin sağlanması ek schedule özellikleri gerektirir.

Real-Time Priority-based Scheduling

- **Hard real-time** sistemlerin schedule özelliklerine geçmeden önce process'lerin bazı karakteristikleri tanımlanmalıdır.
- Process'ler periyodik olarak dikkate alınır. Yani, CPU'ya sabit periyotlarla ihtiyaç duyarlar.
- Periyodik bir process CPU'yu edindikten sonra:
  - *t* sabit bir işlem süresine, *d* deadline'ye, *p* periyoduna sahiptir.
    - Her process için deadline, bir sonraki peryod başlayıncaya kadar CPU burst süresinin tamamlamasını gerektirir.
  - $0 \leq t \leq d \leq p$
  - Peryodik bir görevin rate değeri $1/p$ dir.



Real-Time Scheduling

- Sonraki kısımlarda anlatılacak olan
  - Rate-Monotonic Scheduling
  - Earliest-Deadline-First Scheduling
  - Proportional Share Scheduling

  algoritmaları **hard real-time sistemler** için uyumlu algoritmalardır.

Rate-Monotonic Scheduling

- Rate-Monotonic Scheduling preemptive priority-based bir algoritmadır, her process sisteme geldiğinde **periyot süresiyle ters orantılı öncelik seviyesi alır**.
- Periyot süresi kısaldıkça öncelik seviyesi artar. Bu algoritmada CPU'yu daha sık gerektiren görevlere daha yüksek bir öncelik atanır.
- Algoritmada periyodik bir process'in işlem süresinin her CPU burst için aynı olduğu varsayılır. Yani, bir process CPU'yu her aldığında, CPU burst süresi aynıdır.

Rate-Monotonic Scheduling

- Örneğin P1 ve P2 processleri için periyot süreleri ve işlem süreleri:
  - P1 için p1=50, t1=20 (p1:peryot t1:CPU burst time)
  - P2 için p2=100, t2=35 ms olsun.
- Her process için deadline, bir sonraki periyodun başlama zamanı olsun. Örneğin P1, 0.ms'de yürütülmeye başladığında sonraki periyodu 50.ms'de başlayacaktır, P1 için ilk deadline 50.ms'dir, 50.ms'ye kadar P1'in CPU burst süresinin tamamlanması gerekir
  - Yani 50.ms'ye kadar P1'in t1=20ms süre kadar CPU'da yürütülmesi gerekir.

Rate-Monotonic Scheduling

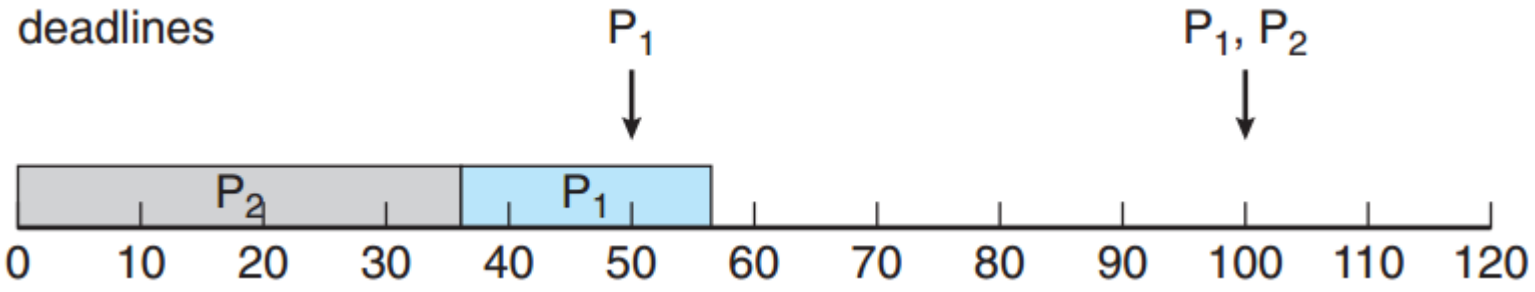
- Örnek için öncelikle bu görevleri her birinin son teslim tarihlerini (deadline) karşılayacak şekilde schedule etmenin mümkün olup olmadığına bakılmalıdır.
- Bir Pi process'inin CPU kullanımını, işlem süresinin periyoda oranı olarak ölçersek ti / pi:
  - P1'in CPU kullanımı 20/50 = 0.40
  - P2'nin CPU kullanımı 35/100 = 0.35'tir.
  - toplam kullanım yüzde 75 olur, bu nedenle bu görevler hem deadline'leri karşılayacak şekilde hem de CPU'da başka process'lerin işlerini yürütmek için kullanılabilir cycle'ler bırakacak şekilde schedule edilebilir.

Rate-Monotonic Scheduling

- P2'ye P1'den daha yüksek öncelik verilsin.(varsayılan öncelik kriteri periyot ile ters orantılıdır, normalde P1 önceliklidir)
  - P1 için p1=50, t1=20 (p1:peryot t1:CPU burst time)
  - P2 için p2=100, t2=35 ms olsun.
- Bu iki process Rate-Monotoic Scheduling ile schedule edildiğinde processlerin uymaları gereken deadline'ler sağlanır mı, gant şeması ile gösteriniz?

Rate-Monotonic Scheduling

- P2'ye P1'den daha yüksek öncelik verilsin, P1 ve P2'nin yürütülmesi şekildeki gibi olur. (varsayılan öncelik kriteri periyot ile ters orantılıdır, normalde P1 önceliklidir)
- P2 ilk olarak yürütülmeye başlar ve 35. ms'de tamamlanır.
- Bu noktada P1 başlar; 55. ms'de CPU burst süresini tamamlar. Ancak P1 için ilk deadline 50'ydi, bu nedenle scheduler, P1'in deadline'ı kaçırmasına neden olur.

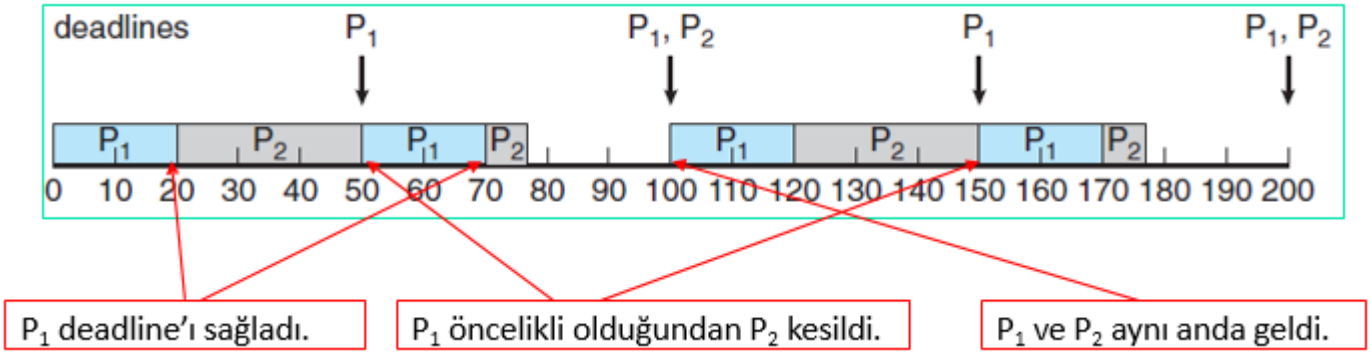


Rate-Monotonic Scheduling

- Process'lere ayrıca öncelik verilmediğinde P1'in periyodu (50ms) P2'ninkinden (100ms) daha kısa olduğundan P1 öncelikli çalışır. Bu duruma göre bu iki process Rate-Monotoic Scheduling ile schedule edildiğinde processlerin uymaları gereken deadline'ler sağlanır mı, gant şeması ile gösteriniz?
  - P1 için p1=50, t1=20 ms
  - P2 için p2=100, t2=35 ms

Rate-Monotonic Scheduling

- Process'lere ayrıca öncelik verilmediğinde P1'in periyodu (50ms) P2'ninkinden (100ms) daha kısa olduğundan P1 öncelikli çalışır. Bu durumda bu işlemlerin yürütülmesi şekilde gösterilmiştir.
  - P1 için p1=50, t1=20 ms
  - P2 için p2=100, t2=35 ms

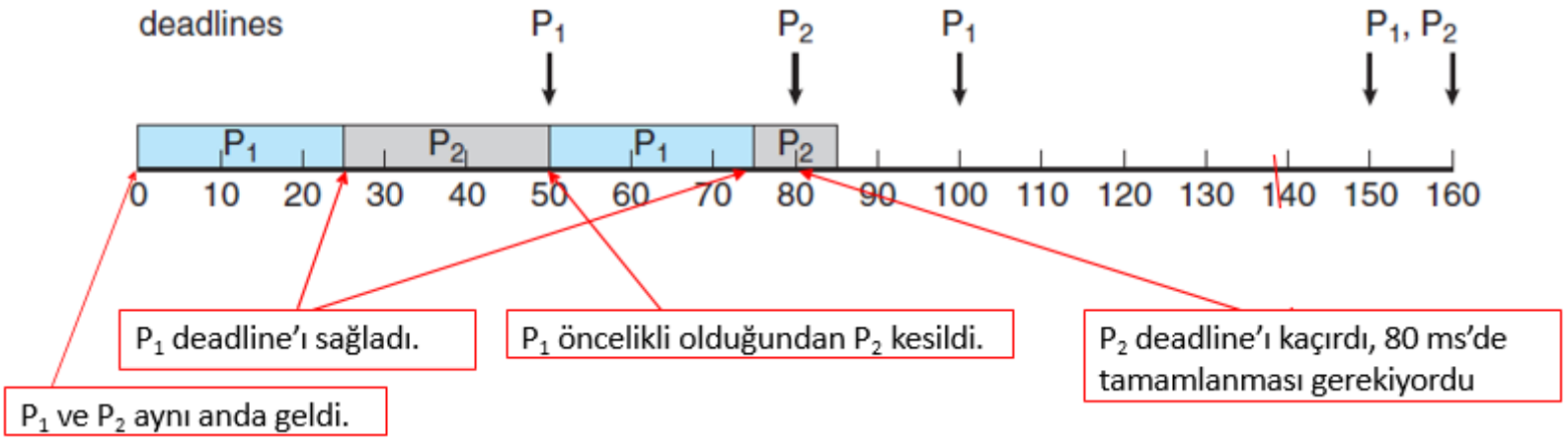


Rate-Monotonic Scheduling – Missing Deadline

- Bir başka örnek, P1 ve P2 process'leri için periyot süreleri ve işlem süreleri:
  - P1 için p1=50, t1=25
  - P2 için p2=80, t2=35 ms olsun.
- Önceki örnekteki gibi process'lere ayrıca öncelik verilmediğini ve her process için deadline'ın bir sonraki periyodun başlama zamanı olduğunu varsayalım.
- Yukarıdaki peryot süreleri ve CPU burst sürelerine göre bu iki process Rate-Monotoic Scheduling ile schedule edildiğinde processlerin uymaları gereken deadline'ler sağlanır mı?

Rate-Monotonic Scheduling – Missing Deadline

- Bir başka örnek, P1 ve P2 process'leri için periyot süreleri ve işlem süreleri:
  - P1 için p1=50, t1=25
  - P2 için p2=80, t2=35 ms olsun.
- P1 daha kısa periyoda sahip olduğundan öncelikli yürütülecektir.
- İki process için toplam CPU kullanım oranı (25/50)+(35/80)=0.94, bu durumda iki process schedule edilebilir ve CPU %6 boşta kalıyor görünüyor.
- Şekilde P1 ve P2 process'leri için schedule grafiği verilmiştir.

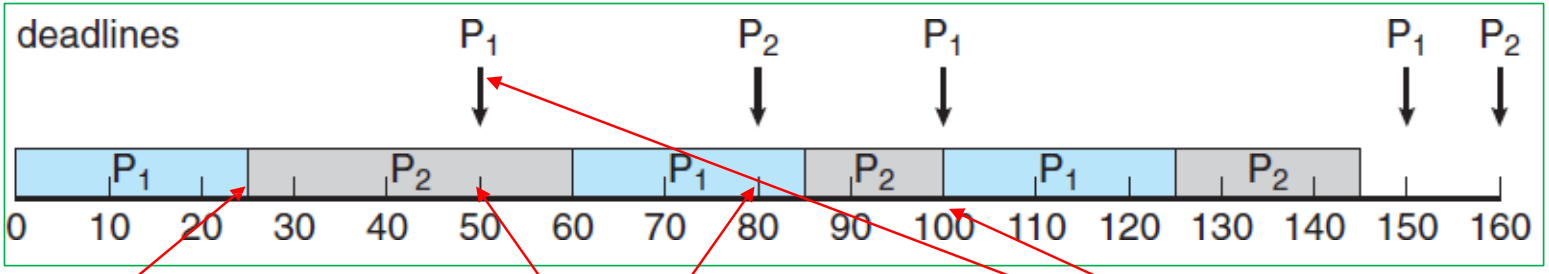


Earliest-Deadline-First Scheduling

- Earliest-Deadline-First Scheduling algoritmasında, **dinamik** olarak her process'e deadline'a göre öncelik seviyesi atanır.
- Deadline'ı kısa olanın öncelik seviyesi yüksektir.
- Aşağıdaki iki process Earliest-Deadline-First scheduling ile schedule edildiğinde processlerin uymaları gereken deadline'ler sağlanır mı, gant şeması ile gösteriniz?
- p1 = 50 t1 = 25, p2 = 80 t2 = 35.

Earliest-Deadline-First Scheduling

- Earliest-Deadline-First Scheduling algoritmasında, **dinamik** olarak her process'e deadline'a göre öncelik seviyesi atanır.
- Deadline'ı kısa olanın öncelik seviyesi yüksektir.
- Bir önceki örnekte Rate-Monotonic Scheduling algoritması ile P2 için deadline karşılanmamıştı. Aynı örnek için Earliest-Deadline-First scheduling şeması şekilde verilmiştir.
- p1 = 50 t1 = 25, p2 = 80 t2 = 35.



P_1 deadline'ı sağladı.

deadline'ı daha kısa process devam etti.

P_1 'in deadline'ı

Proportional Share Scheduling

- Proportional Share Scheduling, tüm uygulamalar arasında toplam T payı tahsis ederek çalışır. Bir uygulama N zaman payı alabilir, böylece uygulamanın toplam işlemci süresinin N / T'sine sahip olmasını sağlar.
- Örnek olarak, toplam T = 100 payının A, B ve C olmak üzere üç process'e bölüneceğini varsayalım. A'ya 50 pay, B'ye 15 pay ve C'ye 20 pay tahsis edilsin. Bu şema, A'nın toplam işlemci süresinin yüzde 50'sine, B'nin yüzde 15'e ve C'nin yüzde 20'ye sahip olmasını sağlar.
- Proportional Share Scheduling, bir uygulamanın tahsis edilen zaman paylarını almasını garanti etmek için bir kabul-kontrol politikası (admission control policy) ile birlikte çalışmalıdır. Kabul-kontrol politikası, bir müşterinin belirli miktar pay talep etmesini, ancak yeterli pay varsa kabul eder.

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:

1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority

- Defines two functions for getting and setting scheduling policy:

1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```

POSIX Real-Time Scheduling API (Cont.)

```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* 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);
}
```

Konular

- Temel kavramlar
- Scheduling kriterleri
- Scheduling algoritmaları
- Thread Scheduling
- Multiple-Processor Scheduling
- Gerçek zamanlı (Real-Time) CPU scheduling
- İşletim Sistemlerinde Scheduling Örnekleri
- Belirli Bir Sistem İçin Bir Scheluling Algoritması Nasıl Seçilir?

Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling

Linux Scheduling Through Version 2.5

- Linux'un sürüm 2.5'ten öncesinde, Linux çekirdeği, geleneksel UNIX schedule algoritmasının bir varyasyonunu çalıştırıyordu.
- Linux'un 2.5 sürümü ile, sistemdeki görev sayısından bağımsız olarak sabit zamanda çalışan ve O(1) olarak isimlendirilen bir scheduler algoritmasına geçildi.
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - Map into global priority with numerically lower values indicating higher priority
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged

Linux Scheduling in Version 2.6.23 +

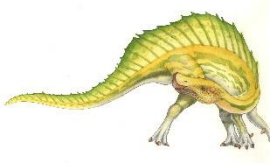
- Linux'un 2.6.23 sürümünde ***Completely Fair Scheduler*** (CFS) varsayılan Linux scheduling algoritması olmuştur.
- Linux sisteminde scheduling, scheduling sınıflara **(scheduling classes)** dayalıdır.
  - Her sınıfa belirli bir öncelik (priority) atanır.
  - Çekirdek, farklı scheduling sınıfları kullanarak, sistemin ve process'lerin ihtiyaçlarına göre farklı scheduling algoritmalarını barındırabilir.
  - Örneğin, bir Linux sunucusu için scheduling kriterleri, Linux çalıştıran bir mobil cihazdan farklı olabilir.
  - Bir sonraki hangi görevin çalıştırılacağına karar vermek için scheduler, en yüksek öncelikli scheduling sınıfına ait olan en yüksek öncelikli görevi seçer.
  - Standart Linux çekirdekleri iki scheduling class uygular:
    1. Default scheduling class – CFS schedling algoritması kullanır
    2. Real-time scheduling class

Linux Scheduling in Version 2.6.23 +

- Sabit zamanlı quantum yerine, CFS scheduler her göreve CPU zamanının bir oranını atar. Bu oran, her göreve atanan **nice value**'ye göre hesaplanır.
  - Quantum -20 ile +19 arasında değişen **nice value**'ye göre hesaplanır;
  - Düşük bir **nice value** daha yüksek bir önceliği gösterir.
  - Daha düşük **nice value**'ye sahip görevler, daha yüksek oranda CPU zamanına sahip olur.
- CFS, time slice'lerin ayrık değerlerini kullanmaz ve bunun yerine, her çalıştırılabilir görevin en az bir kez çalışması gereken bir zaman aralığı olan bir **target latency** tanımlar.
  - CPU zamanının oranları, **target latency** değerine göre belirlenir.
  - Sistemdeki etkin görevlerin sayısı belirli bir eşiğin üzerine çıkarsa **target latency** artabilir.

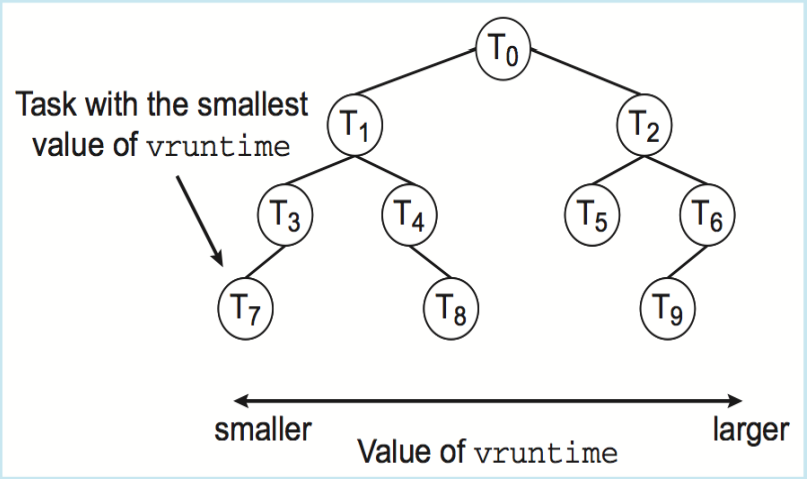
Linux Scheduling in Version 2.6.23 +

- CFS scheduler, öncelikleri doğrudan atamaz. Bunun yerine, her bir görevin sanal çalışma süresini **`vruntime i`**sminde bir değişkeninde tutar.
- Sanal çalışma süresi, bir görevin önceliğine dayalı bir decay factor ile ilişkilidir: düşük öncelikli görevler, yüksek öncelikli görevlerden daha yüksek decay oranlarına sahiptir.
- Normal önceliğe sahip görevler için sanal çalıştırma süresi gerçek fiziksel çalıştırma süresiyle aynıdır.
- Örneğin, varsayılan önceliğe sahip bir görev 200 ms çalışırsa, vruntime da 200 ms olacaktır.
  - Ancak, daha düşük öncelikli bir görev 200 ms boyunca çalışırsa, vruntime 200 ms 'den daha yüksek olacaktır.
  - Benzer şekilde, daha yüksek öncelikli bir görev 200 ms boyunca çalışırsa, vruntime 200 ms'den az olacaktır.
- Sonraki hangi görevin çalıştırılacağına karar vermek için scheduler, en küçük vruntime değerine sahip görevi seçer.
- Ayrıca, çalıştırılabilir hale gelen daha yüksek öncelikli bir görev, daha düşük öncelikli bir görevi preempted edebilir.

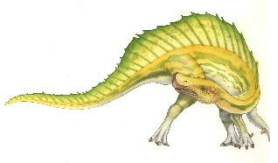


CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

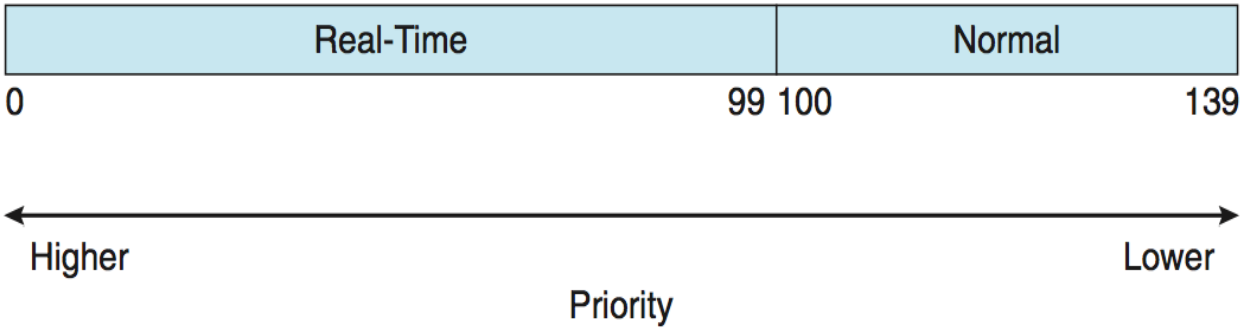


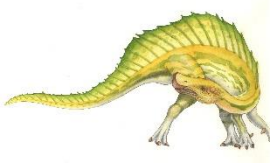
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.



Linux Scheduling (Cont.)

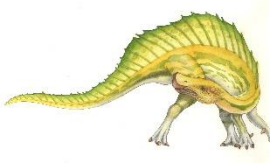
- Linux ayrıca POSIX standardını kullanarak real-time scheduling uygular.
- Real-time görevlere 0 ila 99 aralığında statik öncelikler atanır ve normal görevlere 100 ila 139 arasında öncelikler atanır.
- Bu iki aralık, sayısal olarak daha düşük değerlerin daha yüksek göreceli öncelikleri gösterdiği bir global öncelik şemasına eşlenir.
- Normal görevlere nice değerlerine göre bir öncelik atanır; burada -20 nice değeri 100'e ve +19 nice değeri 139'a eşittir.





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**



Windows Priority Classes

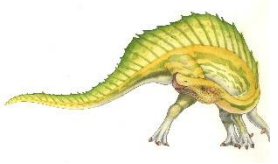
- Win32 API identifies several **priority classes** to which **a process** can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- **A thread** within a **given priority class** has a **relative priority**
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Priority Classes (Cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows Priorities

	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

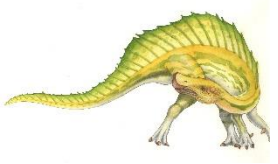


Solaris

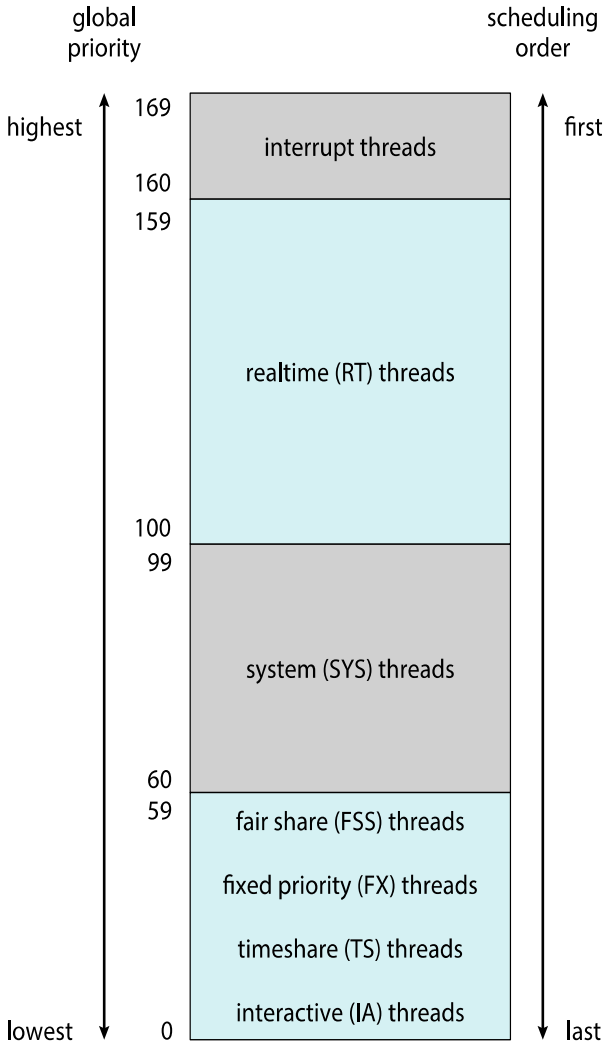
- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level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



Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Konular

- Temel kavramlar
- Scheduling kriterleri
- Scheduling algoritmaları
- Thread Scheduling
- Multiple-Processor Scheduling
- Gerçek zamanlı (Real-Time) CPU scheduling
- İşletim Sistemlerinde Scheduling Örnekleri
- Belirli Bir Sistem İçin Bir Scheluling Algoritması Nasıl Seçilir?

Belirli Bir Sistem İçin Bir Scheluling Algoritması Nasıl Seçilir?

- Belirli bir sistem için bir scheluling algoritması seçerken ilk problem, bir algoritma seçiminde kullanılacak kriterlerin tanımlanmasıdır.
- Bu bölümde gördüğümüz gibi, kriterler genellikle CPU utilization, response time, veya throughput açısından tanımlanır. Bir algoritma seçmek için önce bu kriterlerin göreceli önemi tanımlanmalıdır.
- Kriterlerimiz aşağıdakiler gibi birkaç tane olabilir:
  - Maksimum response time'ın 1 saniye olduğu kısıtlama altında CPU utilization'ı en üst düzeye çıkarma
  - Turnaround süresinin (ortalama olarak) doğrusal olarak toplam yürütme süresiyle orantılı olmasını sağlayacak şekilde throughput'u en üst düzeye çıkarma

Belirli Bir Sistem İçin Bir Scheluling Algoritması Nasıl Seçilir?

- Seçim kriterleri belirlendikten sonra algoritmaların belirlenen kriterlere göre değerlendirilmesi yapılır. Algoritmaların değerlendirilmesi için çeşitli yöntemler vardır:
  - Deterministic Modeling
  - Queueing Models
  - Simulations
  - Implementation

Deterministic Modeling

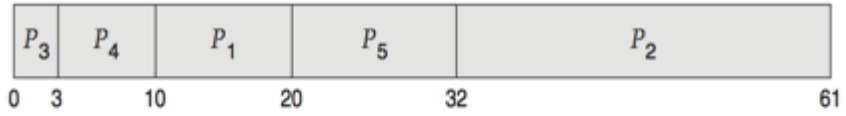
- Değerlendirme yöntemlerinin ana sınıflarından biri analitik değerlendirmedir **(analytic evaluation)**.
- Analitik değerlendirme, söz konusu iş yükü için algoritmanın performansını değerlendirmek üzere bir formül veya sayı üretmek için verilen algoritmayı ve sistem iş yükünü kullanır.
- Deterministik modelleme **(deterministic modeling)**, bir tür analitik değerlendirmedir. Bu yöntem, önceden belirlenmiş belirli bir iş yükünü alır ve bu iş yükü için her algoritmanın performansını tanımlar.
- Örneğin, aşağıda gösterilen iş yüküne sahip olduğumuzu varsayalım. Beş process'in tümü, verilen sırayla, milisaniye cinsinden CPU burst süreleriyle ile 0 zamanında ulaşsın:

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

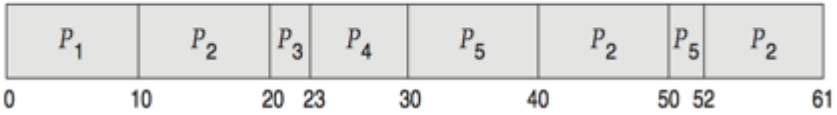
Deterministic Modeling

- Bu process kümesi için FCFS, SJF ve RR (kuantum = 10 milisaniye) scheduling algoritmalarından hangisi minimum ortalama **waiting time** değerini verir?
- Deterministik modelleme basit ve hızlıdır. Ancak, input için kesin sayılar gerektirir ve yanıtları yalnızca bu durumlar için geçerlidir.
- Deterministik modellemenin ana kullanım alanları, scheduling algoritmalarını tanımlamak ve örnekler sağlamak içindir.*
  - FCS is 28ms:

  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Queueing Models

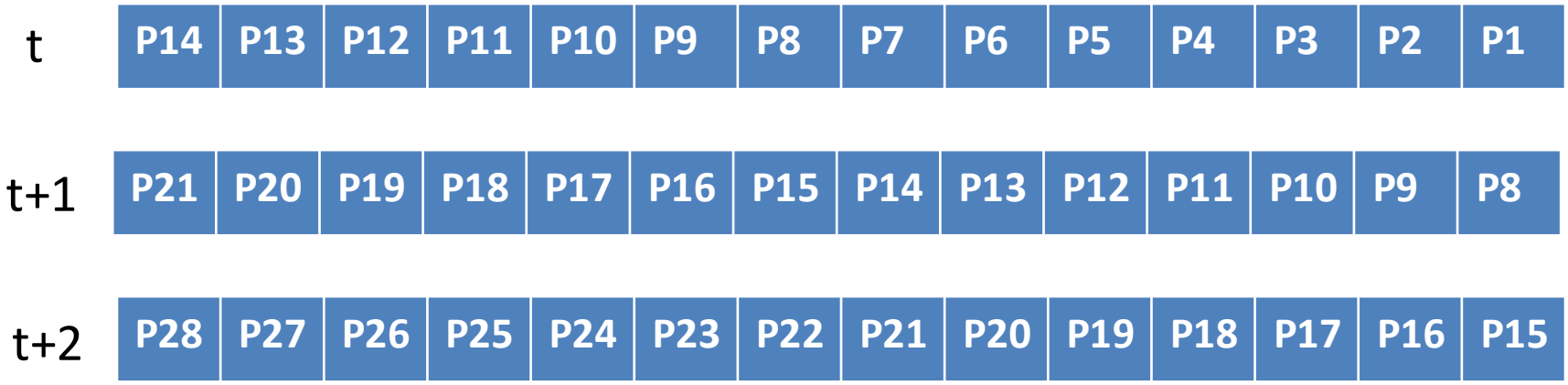
- Çoğu sistemde, çalıştırılan process'ler günden güne değişir, bu nedenle deterministik modelleme için kullanılacak statik süreçler (veya zamanlar) yoktur.
- Ancak belirlenebilecek şey, CPU ve I/O burst dağılımıdır. Bu dağılımlar ölçülebilir ve sonrası için tahminde bulunulabilir.
- Bunun için belirli bir CPU burst olasılığını açıklayan matematiksel bir formül kullanılabilir. Genellikle, bu dağılım eksponansiyel'dir ve ortalaması ile tanımlanır.
- Benzer şekilde, process'lerin sisteme ulaştığı zamanların dağılımını da tanımlayabiliriz (varış zamanı dağılımı).
- Bu iki dağılımdan, çoğu algoritma için ortalama throughput, utilization, waiting time vb. kriterleri hesaplamak mümkündür.

Queueing Models

- Örneğin,
  - n ortalama kuyruk uzunluğu,
  - W kuyruktaki ortalama bekleme süresi
  - λ yeni processlerin kuyruğa ortalama varış hızı olsun (örneğin, saniyede 3 process).
- Bir prosesin beklediği W süresi boyunca, λ × W tane yeni prosesin kuyruğa gelmesini bekliyoruz.
- Sistem sabit bir durumdaysa, kuyruktan çıkan proseslerin sayısı, gelen proseslerin sayısına eşit olmalıdır, yani:
  - n = λ x W
  - Bu denklem **Little's formula** olarak bilinir, herhangi bir scheduling algoritması ve varış dağılımında geçerli olduğu için özellikle kullanışlıdır.

Queueing Models

- Örneğin, her saniyede (ortalama olarak) 7 process geldiğini ve kuyrukta normalde 14 process olduğunu bilirsek, process başına ortalama waiting time 2 saniye olarak hesaplayabiliriz.



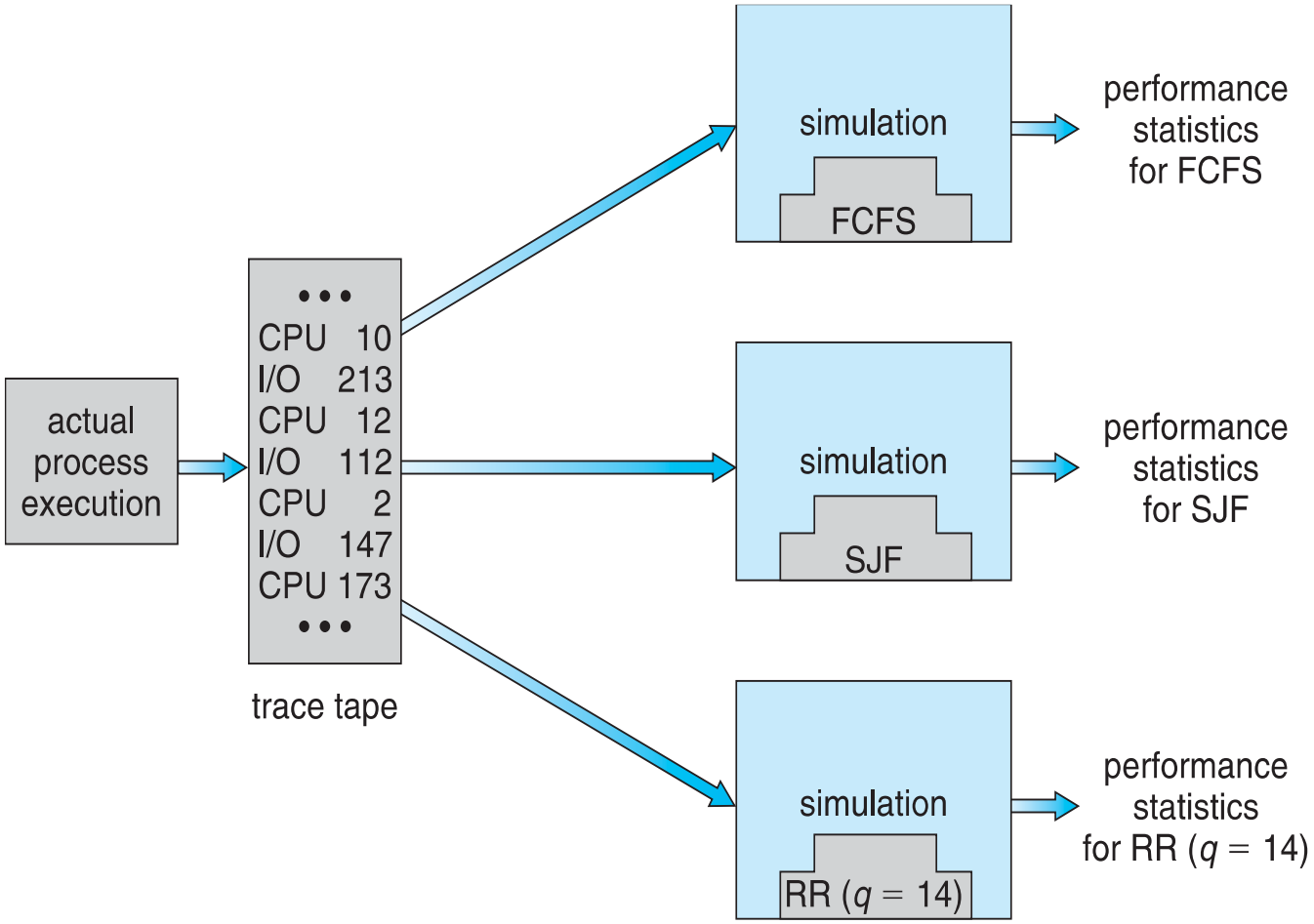
Queueing Models

- Kuyruk analizi, scheduling algoritmalarını karşılaştırmada yararlı olabilir, ancak aynı zamanda sınırlamaları da vardır:
  - Karmaşık algoritmaların ve dağılımların matematiği ile çalışmak zor olabilir. Bu nedenle, varış ve hizmet dağılımları genellikle matematiksel olarak izlenebilir - ancak gerçekçi olmayan - şekillerde tanımlanır.
  - Ayrıca, genellikle doğru olmayabilecek bir dizi bağımsız varsayım yapmak da gereklidir.
- Bu zorlukların bir sonucu olarak, kuyruk modelleri genellikle gerçek sistemlerin yalnızca yaklaşık tahminleridir ve hesaplanan sonuçların doğruluğu sorgulanabilirdir.

Simulations

- Scheduling algoritmalarının daha doğru bir değerlendirmesini elde etmek için simülasyonları kullanabiliriz.
- Simülasyon yazılımlarının veri yapıları, sistemin ana bileşenlerini temsil eder. Simülatörün, bir saati temsil eden bir değişkeni vardır. Bu değişkenin değeri arttıkça simülatör sistem durumunu, cihazların, process'lerin ve scheduler'ın etkinliklerini yansıtacak şekilde değiştirir.
- Simülasyon yürütülürken, algoritma performansını gösteren istatistikler toplanır ve yazdırılır.
- Simülasyonu yürütecek veriler birkaç yolla oluşturulabilir. En yaygın yöntem, olasılık dağılımlarına göre process'ler, CPU burst süreleri, varışlar, ayrılışlar vb. oluşturmak için programlanmış bir rastgele sayı üreteci kullanılır.
- Olasılık dağılımlar matematiksel (uniform, exponential, Poisson) veya deneysel olarak tanımlanabilir.

Evaluation of CPU Schedulers by Simulation



Implementation

- Simülasyon'da bile sınırlı doğruluk söz konusudur. Bir scheduling algoritmasını değerlendirmenin tamamen doğru tek yolu onu kodlamak, işletim sistemine koymak ve nasıl çalıştığını görmektir.
- Bu yaklaşım, gerçek çalışma koşullarında değerlendirme için gerçek algoritmayı gerçek sisteme yerleştirir.
- Bu yaklaşımdaki en büyük zorluk yüksek maliyettir. Maliyet, yalnızca algoritmanın kodlanması ve işletim sisteminin onu destekleyecek şekilde değiştirilmesi değil, aynı zamanda kullanıcıların sürekli değişen bir işletim sistemine tepkisi için de olur.
- Çoğu kullanıcı daha iyi bir işletim sistemi oluşturmakla ilgilenmez, sadece işlemlerini yürütmek ve sonuçlarını kullanmak isterler. Sürekli değişen bir işletim sistemi, kullanıcıların işlerini halletmelerine yardımcı olmaz.

Implementation

- Diğer bir zorluk, algoritmanın kullanıldığı ortamın değişmesidir. Yeni programlar yazıldıkça ve problem türleri değiştikçe scheduler performansı da değişecektir.
  - Örneğin kısa process'lere öncelik verilirse, kullanıcılar daha büyük process'leri daha küçük process'lere bölebilir.
- En esnek scheduling algoritmaları, sistem yöneticileri veya kullanıcılar tarafından, belirli bir uygulama veya bir dizi uygulama için ayarlanabilecek şekilde değiştirilebilen algoritmalardır.
  - Örneğin, üst düzey grafik uygulamaları gerçekleştiren bir workstation, bir web sunucusundan veya dosya sunucusundan farklı programlama gereksinimlerine sahip olabilir.
  - Bazı işletim sistemleri - özellikle birkaç UNIX sürümü - sistem yöneticisinin belirli bir sistem yapılandırması için scheduling parametrelerinde ince ayar yapmasına izin verir.