Bölüm 2: Süreçler

İşletim Sistemleri

Süreç

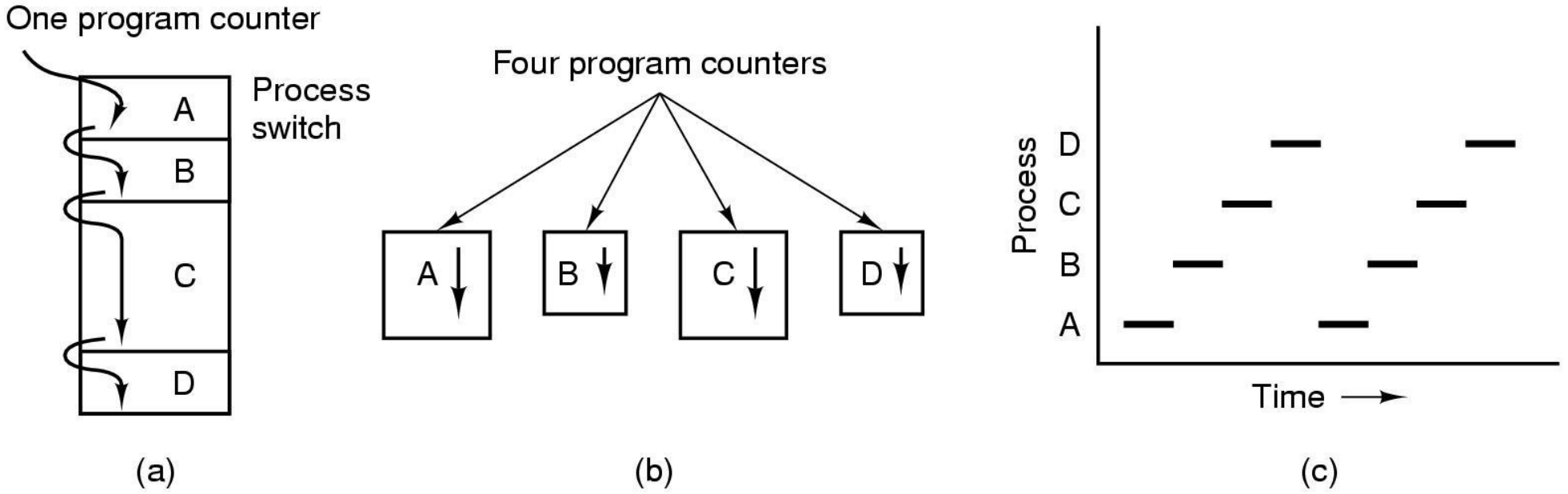
- Yazmaçlar, değişkenler ve bir program sayacına sahip bir program örneği
- Program, girdi, çıktı ve durumu vardır.
- Bu fikir neden gerekli?
  - Bir bilgisayar aynı anda birçok hesaplamayı yönetir - bunu nasıl yaptığını açıklamak için bir soyutlamaya ihtiyaç duyar

Sözde Paralellik

- Tüm modern bilgisayarlar aynı anda birçok iş yapar.
- Tek işlemcili bir sistemde, herhangi bir anda, işlemci sadece bir işlem yürütebilir.
- Ancak çoklu programlama sisteminde işlemci, her biri onlarca veya yüzlerce ms boyunca çalışan işlemler arasında hızlıca geçiş yapar.
- Sözde paralellik kullanıcılar için çok faydalıdır. Ancak; yönetimi bir o kadar zordur.

Çoklu Programlama Süreç Modeli

(a) Dört programın çoklu programlanması. (b) Birbirinden bağımsız dört ardışık sürecin kavramsal modeli. (c) Aynı anda bir program etkindir.



Tekrarlanamaz Yürütme

- Non-reproducible

Program 1: repeat n = n + 1;

Program 2: repeat print(n); n = 0;

Yürütme sırası farklı olabilir.

- n = n + 1; print(n); n = 0;
- print(n); n = 0; n = n + 1;
- print(n); n = n + 1; n = 0;

Süreç ve Program Arasındaki Farklar

- Program, bilgisayar kodlarının bir koleksiyonudur ve çalıştırılabilir bir dosya halindedir.
- Bir program, bir süreç oluşturulduğunda çalıştırılır.
- Süreçler bellekte yer kaplar.
- Bir program birden fazla süreç oluşturabilir ve her süreç ayrı sistem kaynakları kullanır.
- Süreçler arasında haberleşme, veri paylaşımı ve iş bölümü gerçekleşebilir.

Süreç Başlatma

Süreç oluşturmaya neden olan olaylar:

- Sistem başlatma.
- Çalışan bir süreç tarafından bir süreç oluşturma sistem çağrısının yürütülmesi.
- Yeni bir süreç oluşturmak için bir kullanıcı isteği.
- Toplu işin başlatılması. (batch)

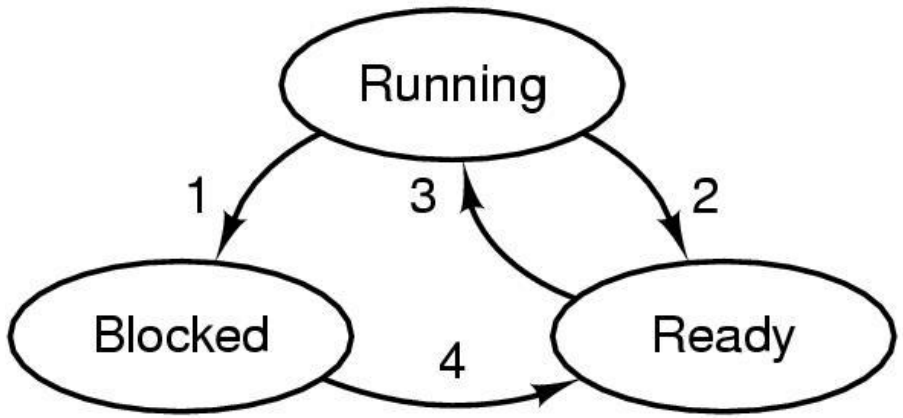
Süreç Sonlandırma

İşlemin sonlandırılmasına neden olan olaylar:

- Normal çıkış (gönüllü).
- Hata sonrası çıkış (gönüllü).
- Ölümcül hata sonrası çıkış (istem dışı).
- Başka bir süreç tarafından sonlandırılma (kill) (istemsiz).

Süreç Durumları

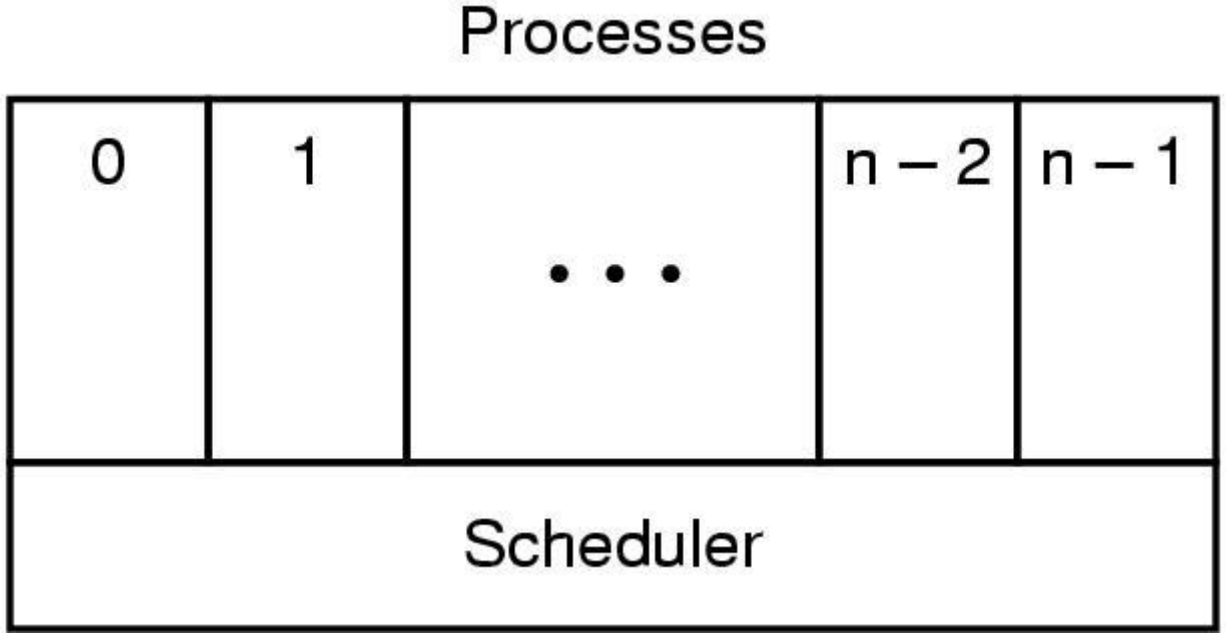
Bir süreç çalışıyor, engellenmiş veya hazır durumda olabilir.



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Süreçleri Gerçekleştirme

Süreç yapılı bir işletim sisteminin en alt katmanı kesilmeleri ve çizelgelemeyi yönetir. Bu katmanın üzerinde sıralı süreçler bulunur.



Süreçleri Gerçekleştirme

Süreç tablosunda bulunan bazı alanlar.

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

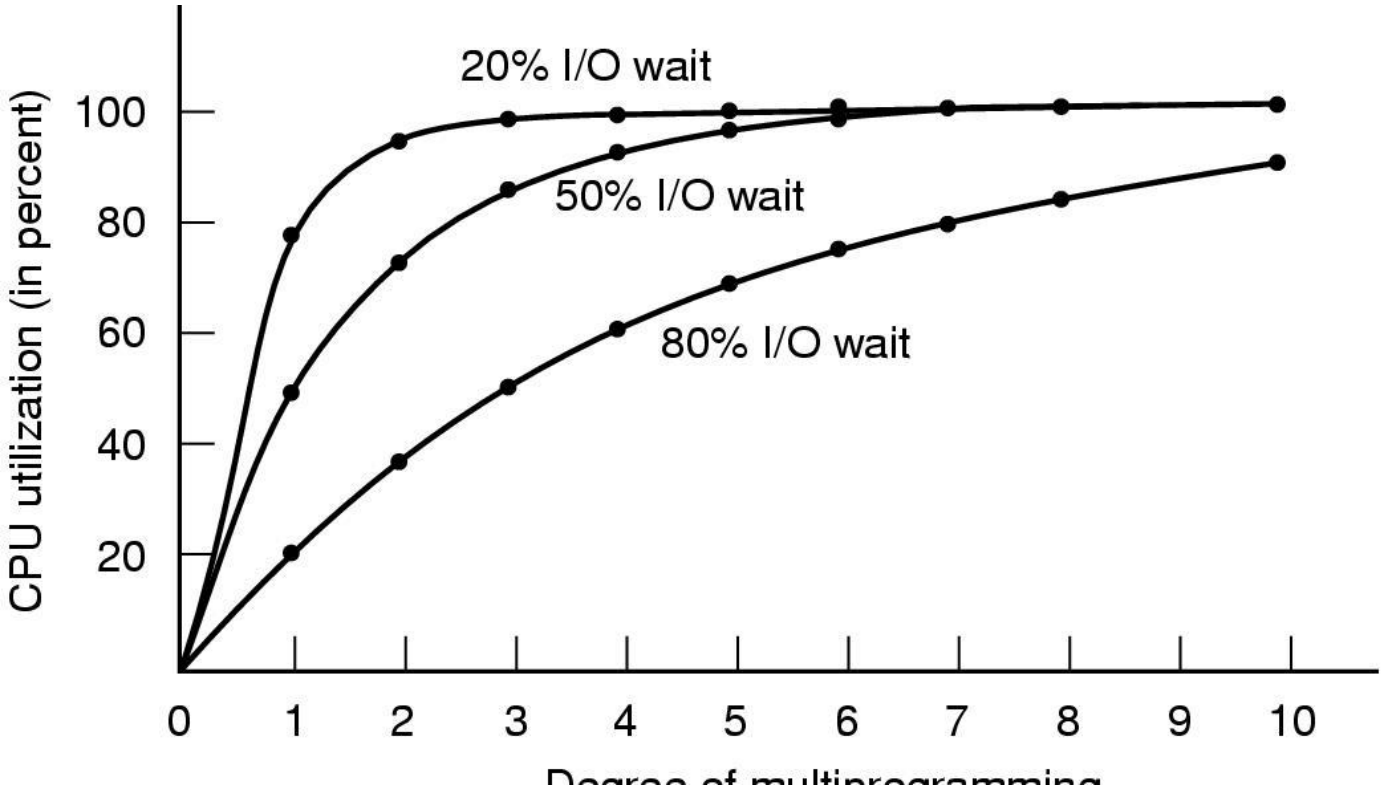
Süreçleri Gerçekleştirme

Bir kesilme oluştuğunda işletim sisteminin en düşük seviyesi ne yapar.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

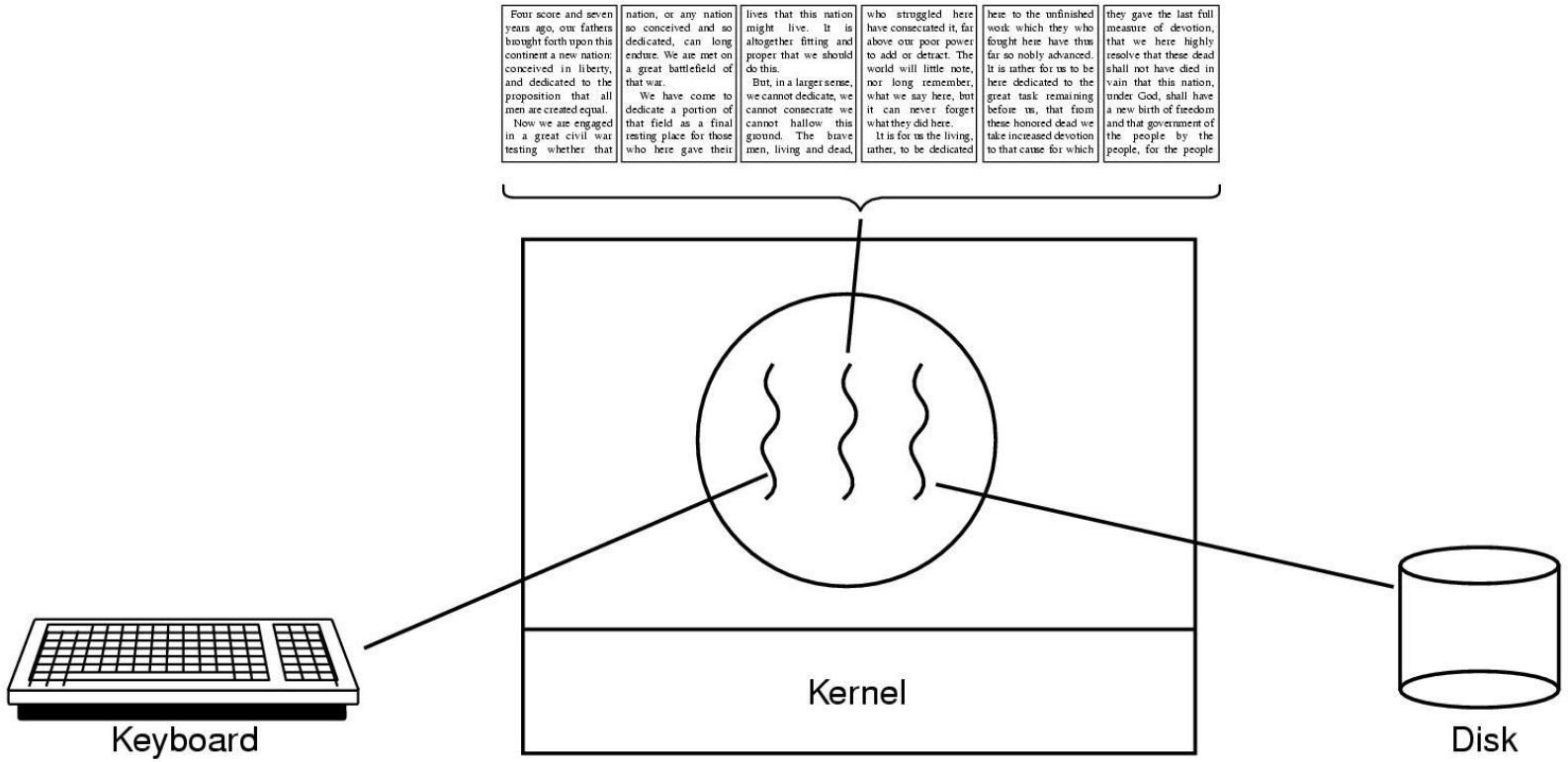
Çoklu Programlama Modellemesi

Bellekte bulunan süreç sayısının bir fonksiyonu olarak CPU kullanımı grafiği.



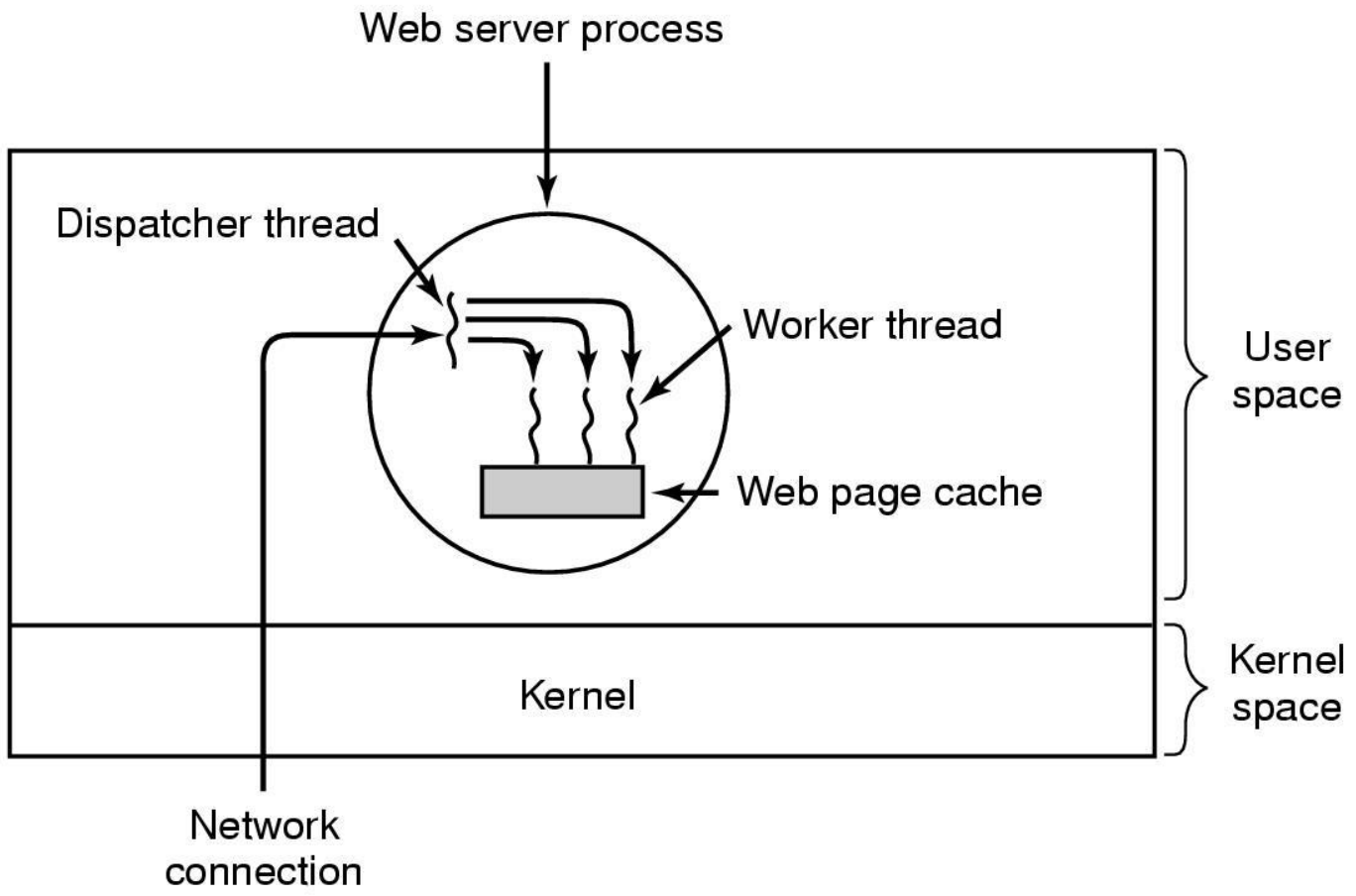
İş Parçacığı

3 iş parçacığına sahip bir uygulama



İş Parçacığı Kullanımı

Çoklu iş parçacığına sahip bir web sunucusu



İş Parçacığı Kullanımı

(a) İşlemci zamanlayıcı (dispatcher) iş parçacığı

(b) İşçi (worker) iş parçacığı

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

Web Sunucusu

- Eğer sayfa yoksa, iş parçacığı bloklar
- Sayfa beklenirken, işlemci hiçbir şey yapmaz
- İş parçacığı yapısı, sunucunun başka bir sayfayı başlatmasını ve bir şeyler yapmasını sağlar

Bloke Olmayan Çağrı

- Eğer sayfa yoksa, bloke olmayan bir çağrı oluştur
- İş parçacığı sayfayı döndürdüğünde CPU'ya kesme gönder, (sinyal)
- Süreç bağlamı anahtarla (pahalı)

Web Sunucusu için 3 Yol

- İş Parçacığı
  - Paralellik, sistem çağrılarını bloklama
- Tek iş parçacıklı süreç
  - Paralellik yok, sistem çağrılarını bloklama
- Sonlu durum makinesi
  - Paralellik, bloklanmayan sistem çağrıları, kesmeler

İş Parçacığı Modeli

- Süreç içerisindeki tüm iş parçacıkları ile paylaşılanlar
- Her bir iş parçacığına özel veriler

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

İş Parçacığı

- Kendi program sayacı, yazmaç kümesi ve yığını vardır
- Kod (text), global veri ve açık dosyaları paylaşır
  - Aynı süreci sonlandırmak için paralel çalıştığı iş parçacıkları ile
- Kendi süreç kontrol bloğuna (PCB) sahip olabilir
  - İşletim sistemine bağlıdır
  - Bağlam, iş parçacığı kimliğini, program sayacını, kayıt kümesini, yığın işaretçisini içerir
  - Aynı süreçteki diğer iş parçacıklarıyla bellek adres uzayı paylaşılır
    - bellek yönetimi bilgileri paylaşılır

İş Parçacıkları ve Süreçler

- Aynı durumlara sahiptirler
  - Koşma
  - Hazır
  - Engellendi
- Kendi yığınları var – süreçlerle aynı
- Yığınlar (döndürülmemiş) prosedür çağrıları için çerçeveler içerir
  - Yerel değişkenler
  - Prosedür tamamlandığında kullanılacak dönüş adresi

İş Parçacıkları Nasıl Çalışır

- Bir süreçte bir iş parçacığı ile başlar
- İş parçacığı içeriği (kimlik, yazmaçlar, nitelikler)
- Yeni iş parçacıkları oluşturmak ve kullanmak için kütüphane çağrıları kullanılır
  - Thread_create parametre olarak aldığı prosedürü başlatır
  - Thread_exit iş parçacığını sonlandırır
  - Thread_join başka bir iş parçacığının bitmesi beklenir
  - Thread_yield diğer iş parçacıklarına çalışma şansı verir

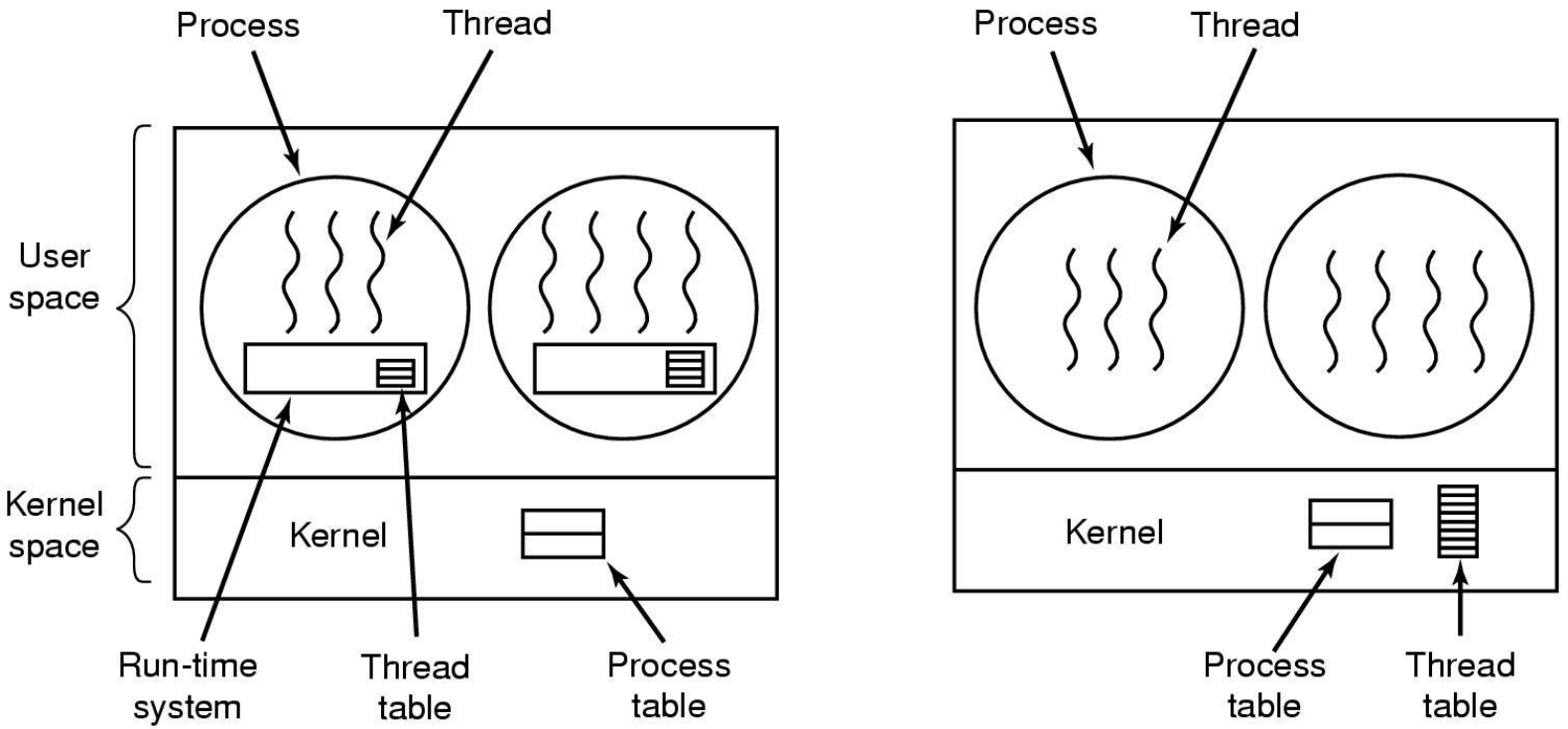
POSIX kütüphanesi (pthreads)

- IEEE Unix standart kütüphane çağrıları

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

İş Parçacıkları

(a) Kullanıcı düzeyinde iş parçacığı paketi. (b) Çekirdek tarafından yönetilen bir iş parçacığı paketi.



Kullanıcı Modu İş Parçacıkları (+)

- İş parçacığı tablosu, iş parçacığı hakkında bilgi içerir (program sayacı, yığın işaretçisi...), böylece çalışma zamanı sistemi bunları yönetebilir
- İş parçacığı bloke olursa, çalışma zamanı sistemi iş parçacığı bilgilerini tabloda saklar ve çalıştırılacak yeni iş parçacığını bulur
- Durum kaydetme ve çizelgeleme, çekirdek modundan daha hızlı çağrılır (gizli kapı yok, önbellek temizleme yok) (no trap, no cache flush)

Kullanıcı Modu İş Parçacıkları (-)

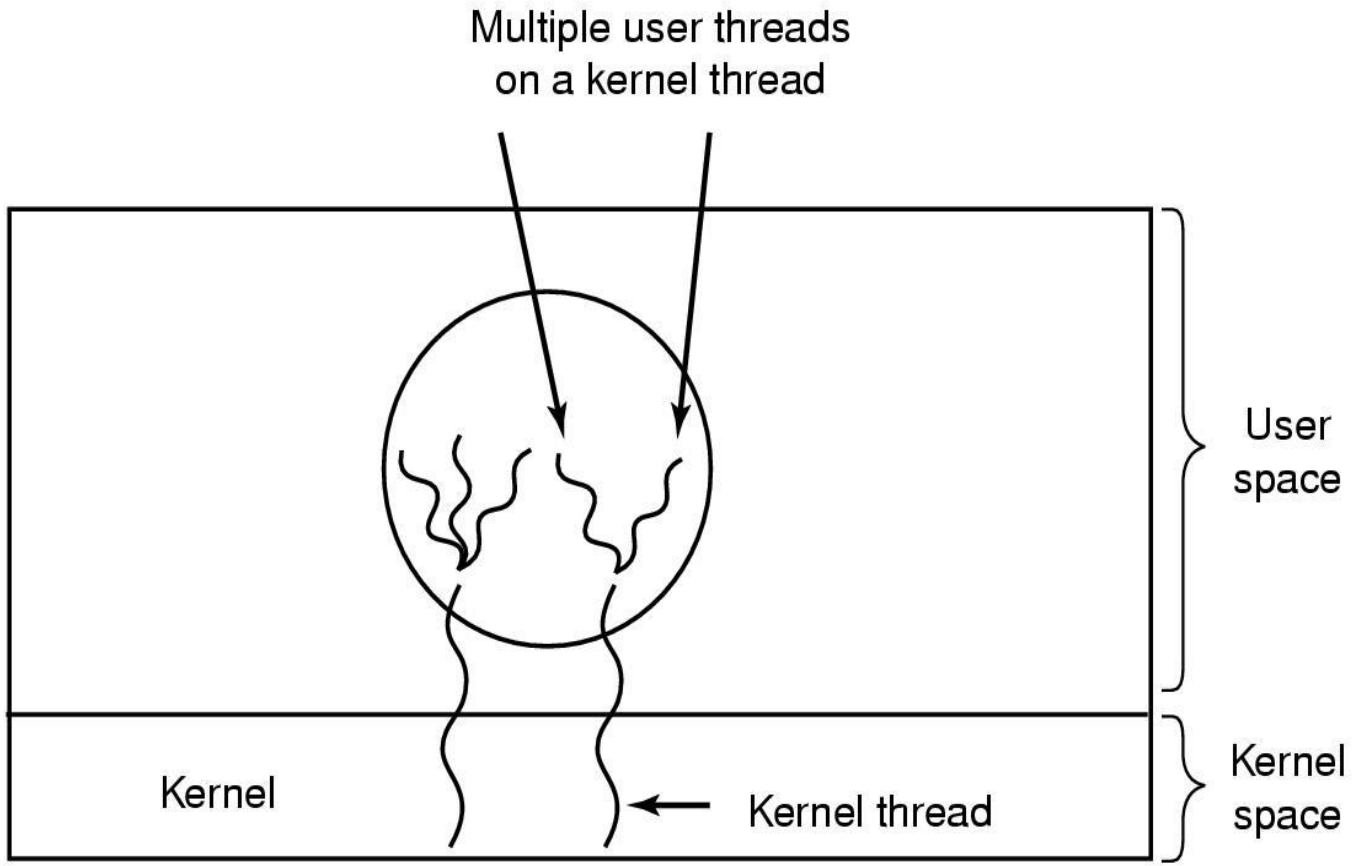
- İş parçacığının sistem çağrısını yürütmesine izin verilemez, çünkü diğer tüm iş parçacıklarını engelleyecektir.
- Zarif bir çözüm yok
  - Çağrıları engellememek için sistem kütüphanesi kırılabilir (hack)
  - Unix'in bazı sürümlerinde aynı işi yapan benzer sistem çağrıları kullanılabilir
- İş parçacıkları gönüllü olarak işlemciyi bırakmaz
  - Kontrolü sisteme vermek için periyodik olarak kesmeye uğrar
  - Bu çözümün maliyeti de bir sorundur

Çekirdek Modu İş Parçacıkları

- Çekirdek, kullanıcı modu ile aynı iş parçacığı tablosunu tutar
- İş parçacığı bloke olursa, çekirdek başka bir tanesini seçer
  - Aynı süreçten olması gerekmez!
- Çekirdekte iş parçacıklarını yönetmek çok maliyetli ve değerli olan çekirdek alanında yer kaplar

Hibrit Yaklaşım

Kullanıcı düzeyindeki iş parçacıklarını çekirdek düzeyindeki iş parçacıklarına çoklama



Hibrit Yaklaşım

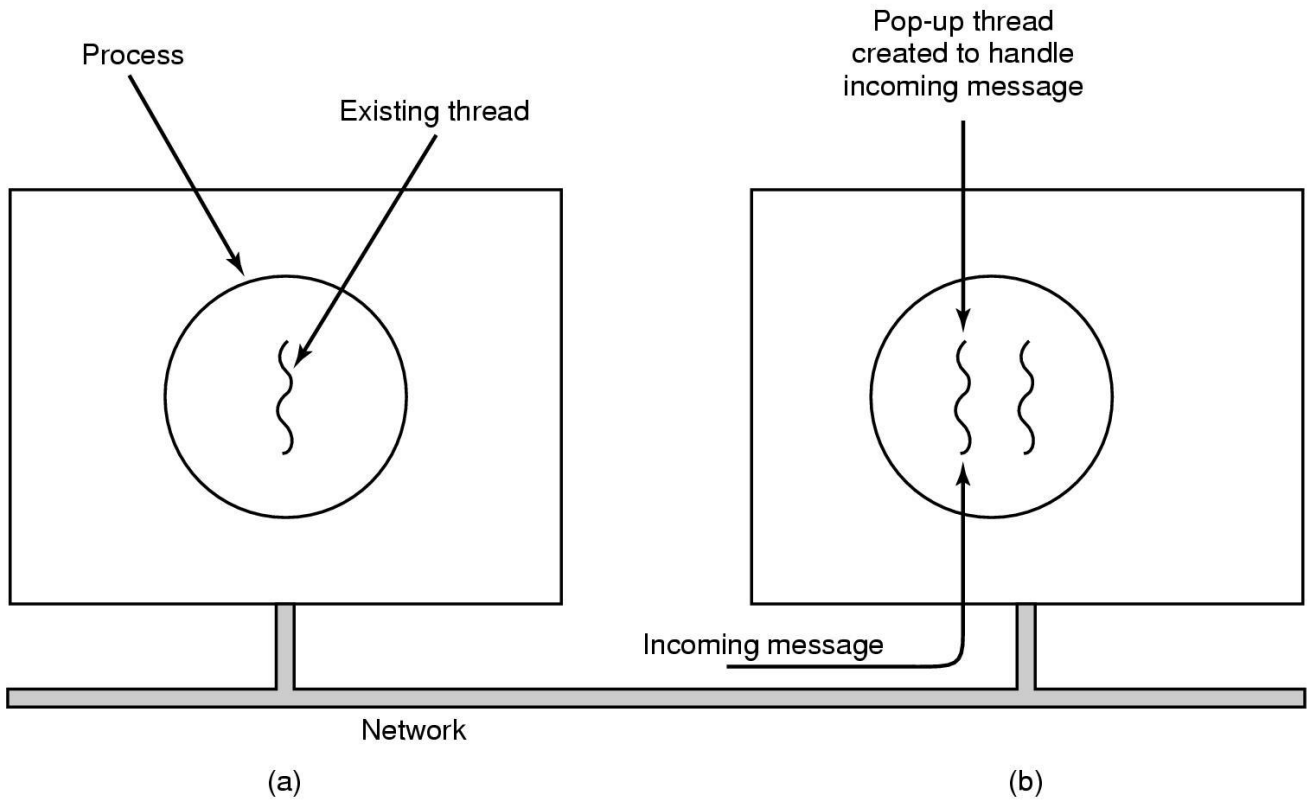
- Çekirdek, sadece çekirdek iş parçacıklarından haberdardır
- Kullanıcı düzeyinde iş parçacıkları, çekirdekten bağımsız oluşturulur, çizelgelenir, sonlandırılır
- Programcı, kaç tane kullanıcı seviyesi ve kaç tane çekirdek seviyesi iş parçacığı kullanacağını belirler

Çizelgeleyici Aktivasyonları – Yukarı Çağrılar

- Bir iş parçacığı bloke olduğunda çalışma zamanı sisteminin iş parçacıklarını değiştirmesini istediğinde
  - Her süreçle bir sanal işlemciyi ilişkilendirir
  - Çalışma zamanı sistemi, iş parçacıklarını sanal işlemcilere tahsis eder
- Çekirdek, çalışma zamanı sistemine bildirir
  - bir iş parçacığı bloke olduğunda
  - bilgileri RTS'ye iletir (iş parçacığı kimliği …)
  - RTS başka bir iş parçacığı çizelgeler

Açılır (Pop-up) İş Parçacıkları

- Bir mesaj geldiğinde yeni bir iş parçacığı yaratılır

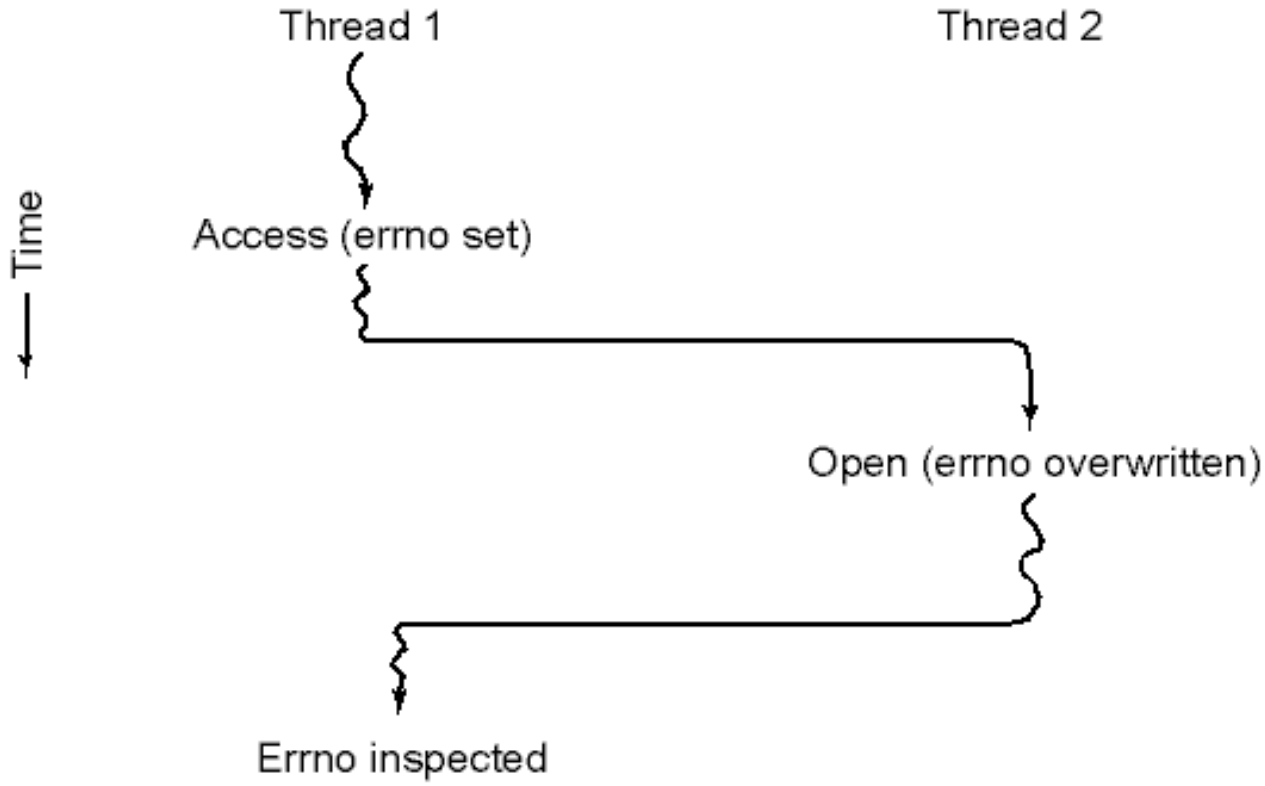


Neden Açılır İş Parçacıkları

- Sistem alma çağrısına bloke olmuş ve mesaj geldikçe işleyen bir iş parçacığı kullanılabilir
- Her mesaj geldiğinde iş parçacığının geçmişinin geri yüklenmesi gerekir
- Açılır iş parçacıkları yenidir ve geri yüklenecek bir verisi yoktur
- Bu nedenle daha hızlıdırlar

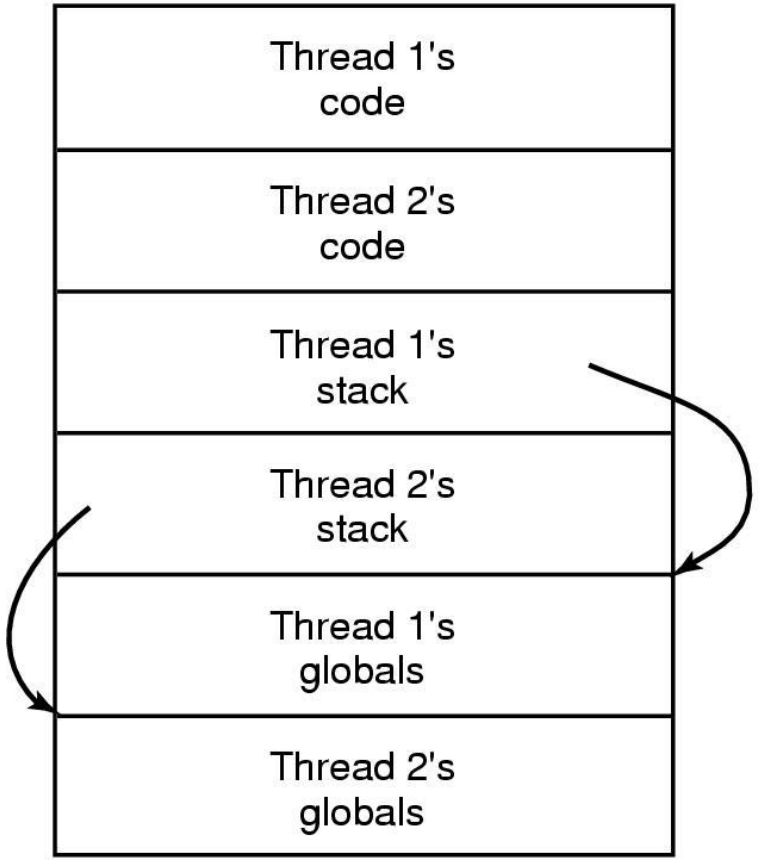
İş Parçacıkları Arasında Çakışma

Global bir değişkenin kullanımıyla ilgili iş parçacıkları arasında yaşanabilecek çakışma



Çoklu İş Parçacıklı Programlama

İş parçacıkları kendilerine ait global değişkenlere sahip olabilir



Problemler

- Yeniden girilmeyen (not re-entrant) kütüphane yordamı.
  - Bir iş parçacığı mesajı bir ara belleğe koyar, yeni bir iş parçacığı mesajın üzerine yazar
- Bellekten yer alma programları (geçici olarak) tutarsız bir durumda olabilir
  - Yeni iş parçacığı yanlış işaretçi almış olabilir
- İş parçacığına özgü sinyalleri uygulamak zor mu?
  - İş parçacıkları kullanıcı alanındaysa, çekirdek doğru iş parçacığını adresleyemez.

İş Parçacıklarının Kullanılma Nedeni

- Sistem çağrılarını bloklayarak paralelliği (web sunucusu) etkinleştirir
- İş parçacıkları oluşturmak ve yok etmek, süreçlerden daha hızlıdır
- Çoklu çekirdekli sistemler için doğal
- Kolay programlama modeli

İş Parçacıklarının Avantajları

- Kullanıcı duyarlılığı
  - Bir iş parçacığı bloke olduğunda, diğeri kullanıcı G/Ç'sini işleyebilir. Ancak: iş parçacığı uygulamasına bağlı
- Kaynak paylaşımı: ekonomi
  - Bellek paylaşılır (yani adres alanı paylaşılır), Açık dosyalar, soketler
- Hız
  - İş parçacığı oluşturma süreç oluşturmaya göre yaklaşık 30 kat daha hızlı, bağlam geçişi 5 kat daha hızlı
- Donanım paralelliğinden yararlanma
  - Ağır süreçler, çoklu işlemcili mimarilerden de faydalanabilir

İş Parçacıklarının Dezavantajları

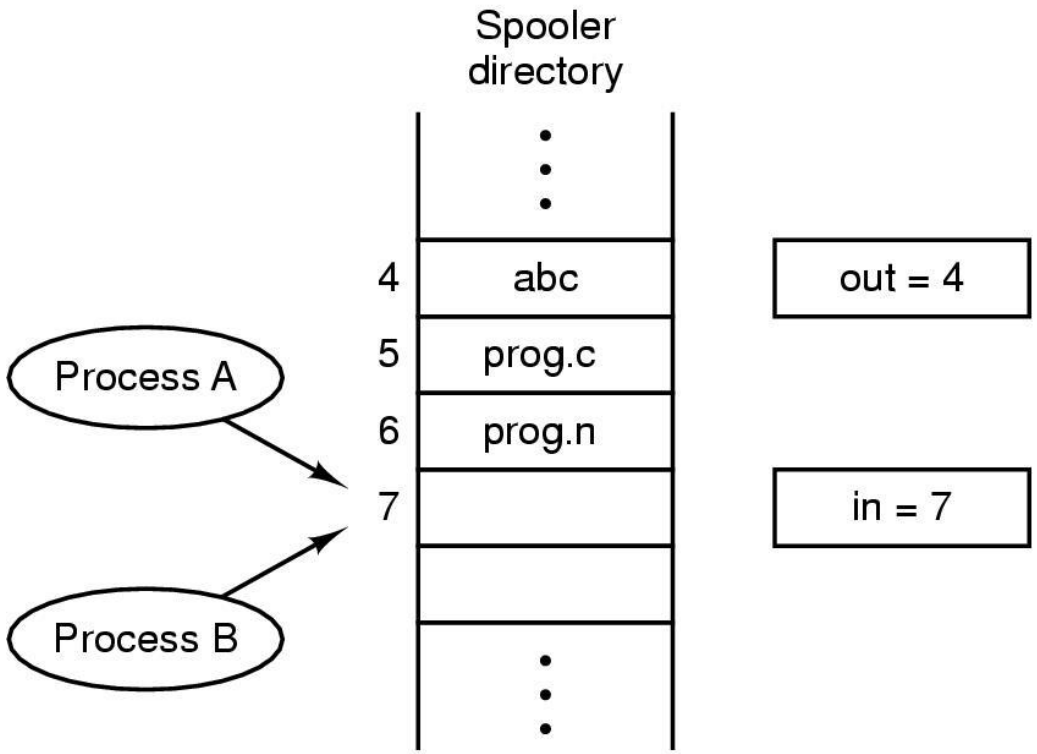
- Senkronizasyon
  - Paylaşımlı bellek ve değişkenlere erişim kontrol edilmelidir.
  - Program koduna karmaşıklık, hatalar ekleyebilir. Yarış koşullarından, kilitlenmelerden ve diğer sorunlardan kaçınmak gerekir
- Bağımsızlık eksikliği
  - Ağır Ağırlık İşlemde (HWP) iş parçacıkları bağımsız değildir
  - RAM adres uzayı paylaşıldığından bellek koruması yoktur
  - Her iş parçacığının yığınları bellekte ayrı yerde olması amaçlanır, ancak bir iş parçacığının hatası nedeniyle başka bir iş parçacığının yığınının üzerine yazma yapılabilir.

Süreçler Arası İletişim

- Üç problem
  - Gerçekte nasıl yapılacak
  - Süreç çakışmalarıyla nasıl başa çıkılır (aynı koltuk için 2 havayolu rezervasyonu)
  - Bağımlılıklar mevcutken doğru sıralama nasıl yapılır, silahı ateşlemeden önce nişan alınması

Süreçler Arası İletişim

- Yarış durumu: iki süreç aynı bellek alanına aynı anda erişmek istediğinde



Önerilen Çözümler

- Kesmeleri devre dışı bırakma (disabling interrupts)
- Kilit değişkenleri (lock variables)
- Sıkı değişim (strict alternation)
- Peterson'ın çözümü
- TSL komutu

Kesmeleri Devre Dışı Bırakma

- Fikir: süreç kesmeleri devre dışı bırakır, kritik bölgeye girer, kritik bölgeden çıktığında kesmeleri etkinleştirir
- Problemler
  - Süreç, kesmeleri devre dışı bırakamazsa sistem çöker
  - Clock yalnızca bir kesme olduğundan, hiçbir CPU önalımı (preemption) gerçekleşemez.
  - Kesmeleri devre dışı bırakmak çoklu çekirdekli sistemlerde çalışmaz
  - Kesmeleri devre dışı bırakmak, işletim sisteminin kendisi için yararlıdır, ancak kullanıcılar için değildir

Kilit Değişkeni

- Bir yazılım çözümü - Herkes bir kilidi paylaşır
  - Kilit 0 olduğunda, süreç 1'e çevirir ve kritik bölgeye girer.
  - Kritik bölgeden çıktığında, kilidi 0'a çevirir
- Problem: Yarış durumu

Yarış Durumu

- İki veya daha fazla süreç, bazı paylaşılan verileri okuyor veya yazıyor ve nihai sonuç hangisinin ne zaman çalıştığına bağlı.

- Karşılıklı dışlama
  - Birden fazla işlemin paylaşılan verileri aynı anda okumasını ve yazmasını engelleme
- Kritik bölge
  - Programın paylaşılan alana erişim yaptığı kod bölümü

Karşılıklı Dışlama

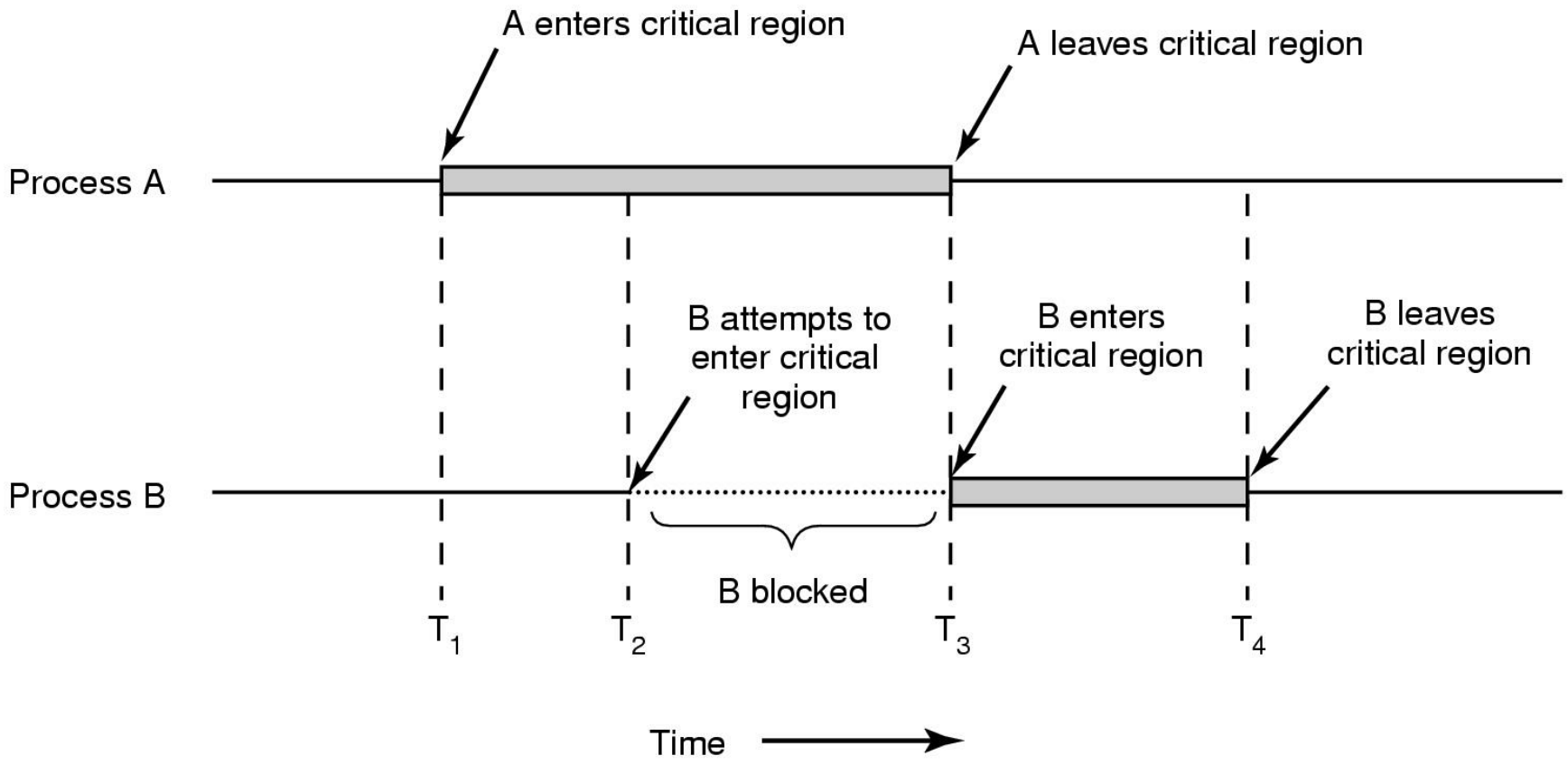
Karşılıklı dışlama sağlamak için dört koşul

- İki süreç aynı anda kritik bölgede olmamalı
- İşlemci hızı ve sayısı hakkında varsayım yapılmamalı
- Kritik bölgesinin dışında çalışan hiçbir süreç başka bir süreci engellememeli
- Hiçbir süreç kritik bölgesine girmek için sonsuza kadar beklememeli

Kritik Bölge

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Kritik Bölge Kullanarak Karşılıklı Dışlama



Sıkı Değişim – Strict Alternation

Önce ben, sonra sen!

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Kavramlar

- Busy waiting
  - Bir değere ulaşana kadar bir değişkeni sürekli olarak test etme
- Spin lock
  - Meşgul beklemeyi kullanan bir kilit, döndürme kilidi olarak adlandırılır

Peterson'un Çözümü

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

TSL Komutu

```
enter_region:
        TSL REGISTER,LOCK                | copy lock to register and set lock to 1
        CMP REGISTER,#0                  | was lock zero?
        JNE enter_region                 | if it was non zero, lock was set, so loop
        RET | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                     | store a 0 in lock
        RET | return to caller
```

XCHG Komutu

- XCHG A,B; a ve b değerlerini yer değiştir

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

Uyuma ve Uyanma

- Meşgul beklemenin dezavantajı
  - Düşük öncelikli bir süreç kritik bölgede iken,
  - Yüksek öncelikli bir süreç geldiğinde daha düşük öncelikli süreci engeller,
  - Lock’tan dolayı meşgul beklemede CPU'yu boşa harcar,
  - Daha düşük öncelikli süreç kritik bölge dışına çıkamaz
  - Öncelikleri değiştirmek/ölümcül kilitlenme

- Meşgul beklemek yerine bloke etme
  - Önce Uyandır, sonra uyut (wake up, sleep)

Üretici Tüketici Problemi

- İki işlem ortak, sabit boyutlu bir arabelleği paylaşmakta
- Üretici arabelleğe veri yazar
- Tüketici arabellekten veri okur

- Basit bir çözüm

Ölümcül Yarış Durumu

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Veri Kaybı Sorunu

- Paylaşılan değişken: sayaç
- Eşzamanlılıktan kaynaklanan sorun
- Tüketici 0 ile sayaç değişkenini okuduğunda; ancak zamanında uykuya geçmediğinde, sinyal kaybolacaktır.

Semafor

- Dijkstra tarafından önerilen yeni bir değişken türü
- Atomik Eylem, tek ve bölünmez
- Down (P)
  - semafor kontrol edilir, değeri 0 ise uyku, değilse değeri azalt ve devam et
- Up (v)
  - semafor kontrol edilir,
  - Süreçler semaforda bekliyorsa, işletim sistemi devam etmeyi seçecek ve düşüşünü tamamlayacaktır.
  - Kaynak sayısının bir işareti olarak farz edilir

Üretici-Tüketici Sorununa Çözüm

- Full: dolu yuvaların sayısı, başlangıç değeri 0
- Empty: boş yuvaların sayısı, başlangıç değeri N
- Mutex: arabelleğe (buffer) aynı anda erişimi engeller, başlangıç değeri 0 (ikili semafor)

- Senkronizasyon/karşılıklı dışlama

Semafor Kullanımı

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}
```

```
void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

mutex_lock ve mutex_unlock

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

Bazı Pthreads Çağrıları

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Bazı Pthreads Çağrıları

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Pthreads Mutex

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                       /* put item in buffer */
                pthread_cond_signal(&condc);      /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                       /* take item out of buffer */
                pthread_cond_signal(&condp);      /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Gözleyici (Monitors)

- Muteksleri ve koşul değişkenlerini kullanarak işleri karıştırmak kolaydır. Küçük hatalar felaketlere neden olur.
- Semaforlu üretici tüketici kodundaki iki down fonksiyonunun yer değiştirmesi kilitlenmeye neden olur
- Gözleyici, karşılıklı dışlama ve bloke etme mekanizmasını uygulayan bir dil yapısıdır.
- Gözleyici, bir "modül" içinde gruplandırılmış {prosedürler, veri yapıları ve değişkenlerden} oluşur
- Bir işlem, gözleyicinin içindeki prosedürleri çağırabilir, ancak içindeki öğelere doğrudan erişemez.

Gözleyici

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .       .       .
        end;
end monitor;
```

Gözleyici

- Karşılıklı dışlamayı zorlamak programcının değil, derleyicinin işidir.
- Monitörde aynı anda yalnızca bir işlem olabilir
  - Bir süreç bir gözleyiciyi çağırdığında, yapılan ilk şey gözleyicide başka bir işlemin olup olmadığını kontrol etmektir. Bu durumda, çağrı işlemi askıya alınır.
- Bloke etmeyi zorlamak gerekiyor
  - koşul değişkenlerini kullanarak
  - bekle, sinyal işlemleri kullanılarak

Gözleyici

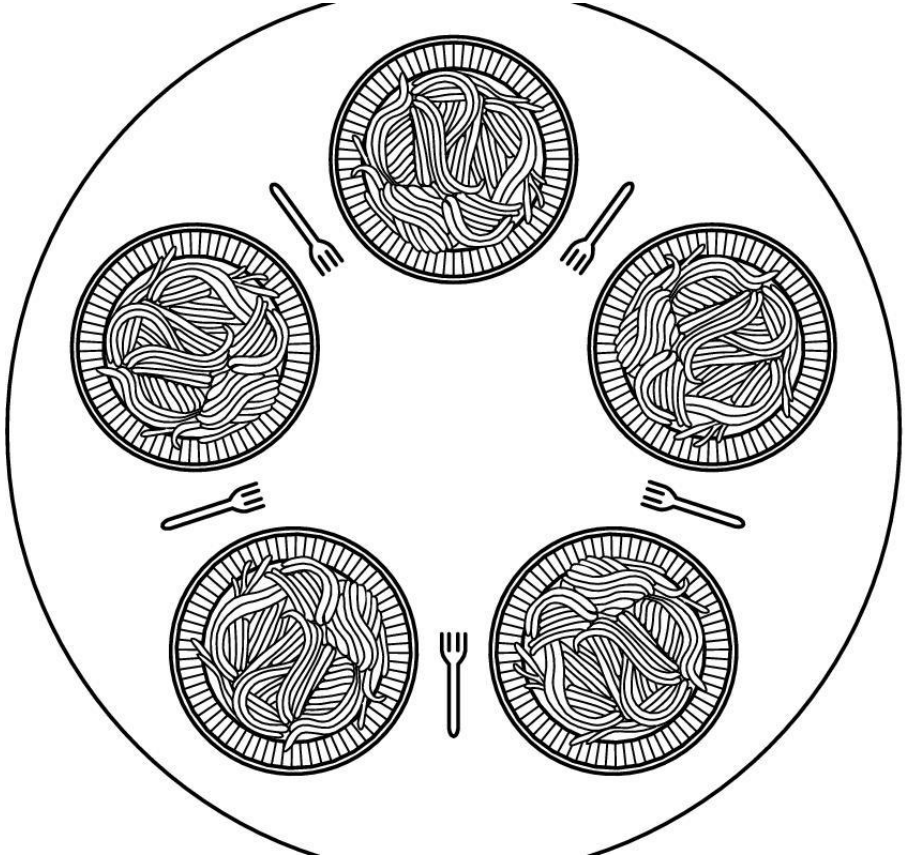
- Gözleyici devam edemeyeceğini anladığında (arabellek dolu), bir koşul değişkeninde (dolu) bir sinyal yayınlayarak sürecin (üretici) bloke olmasına neden olur
- Başka bir sürecin (tüketici) gözleyiciye girmesine izin verilir.
- Bu işlem, bloke olan sürecin (üretici) uyanmasına neden olacak bir sinyal verir.
- Sinyali işler ve gözleyiciden çıkar

Süreçler Arası İletişim Problemleri

- Dining philosopher problemi
  - Bir filozof ya yer ya da düşünür
  - Aç kalırsa, iki çatal alıp yemeye çalış

- Okur-Yazar problemi
  - Bir veritabanına erişimi modeller

Dining Philosophers Problemi



Çözüm?

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

Çözüm (1)

```
#define N                5                  /* number of philosophers */
#define LEFT             (i+N–1)%N          /* number of i's left neighbor */
#define RIGHT            (i+1)%N            /* number of i's right neighbor */
#define THINKING         0                  /* philosopher is thinking */
#define HUNGRY           1                  /* philosopher is trying to get forks */
#define EATING           2                  /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
        while (TRUE) {                      /* repeat forever */
                think( );                   /* philosopher is thinking */
                take_forks(i);              /* acquire two forks or block */
                eat( );                     /* yum-yum, spaghetti */
                put_forks(i);               /* put both forks back on table */
        }
}
```

Çözüm (2)

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}
```

Çözüm (3)

```
void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                      /* enter critical region */
        state[i] = THINKING;               /* philosopher has finished eating */
        test(LEFT);                        /* see if left neighbor can now eat */
        test(RIGHT);                       /* see if right neighbor can now eat */
        up(&mutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Okur-Yazar Problemi

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
        while (TRUE) {                  /* repeat forever */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc + 1;            /* one reader more now */
                if (rc == 1) down(&db); /* if this is the first reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                read_data_base( );      /* access the data */
                down(&mutex);           /* get exclusive access to 'rc' */
                rc = rc – 1;            /* one reader fewer now */
                if (rc == 0) up(&db);   /* if this is the last reader ... */
                up(&mutex);             /* release exclusive access to 'rc' */
                use_data_read( );       /* noncritical region */
        }
}
```

Okur-Yazar Problemi (2)

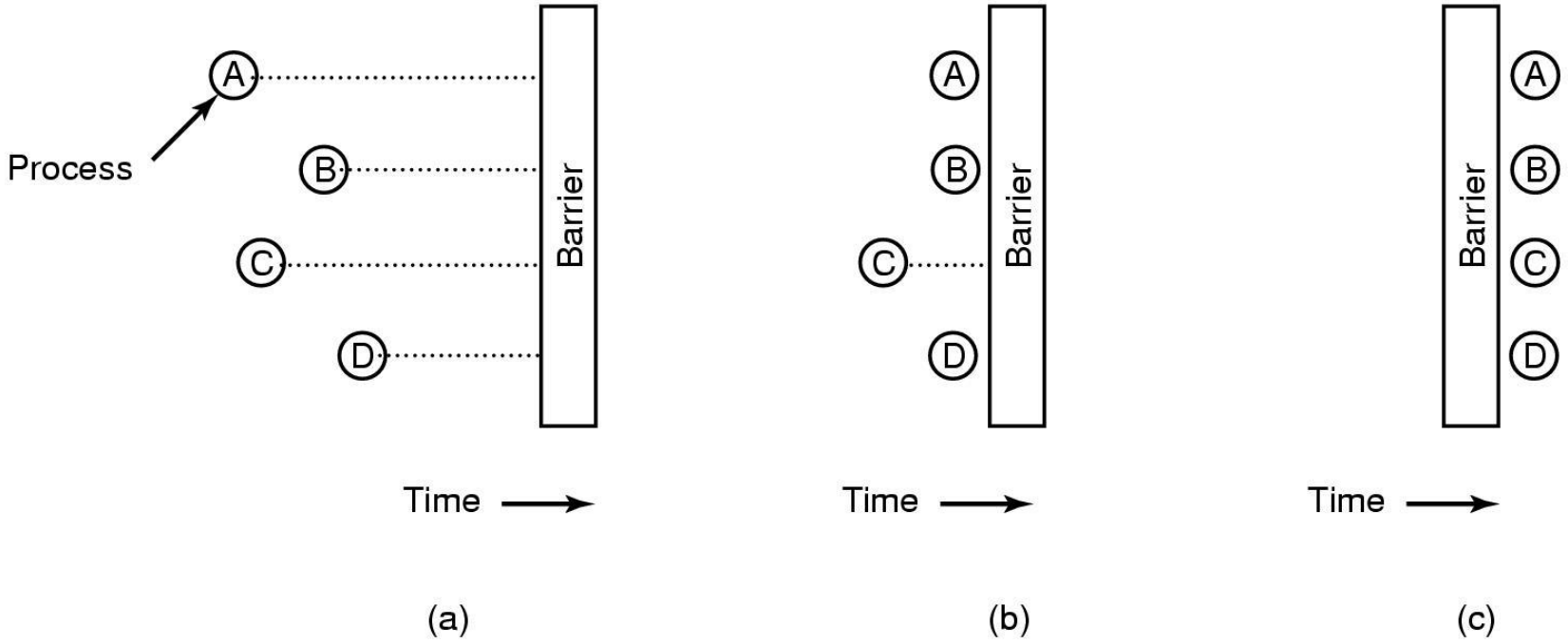
```
void writer(void)
{
        while (TRUE) {                  /* repeat forever */
                think_up_data( );       /* noncritical region */
                down(&db);              /* get exclusive access */
                write_data_base( );     /* update the data */
                up(&db);                /* release exclusive access */
        }
}
```

Okur-Yazar Problemi

- Çözümün dezavantajı nedir?
  - Yazar süreci açlık (starvation) tehlikesiyle karşı karşıya

Bariyer (Barriers)

- Bariyerler, süreç gruplarını senkronize etmek için tasarlanmıştır.
- Genellikle bilimsel hesaplamalarda kullanılır.



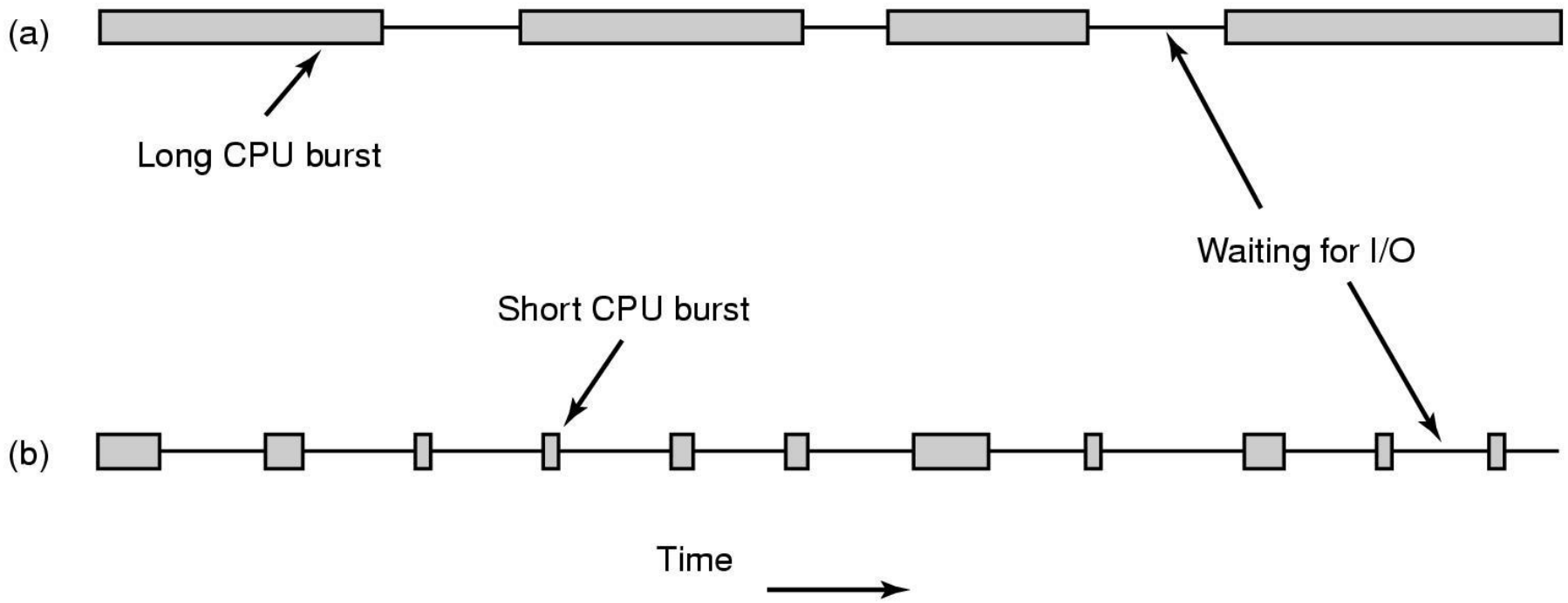
Çizelgeleme

- Bir sonraki adımda hangi süreç çalıştırılacak?
- Bir süreç çalışırken, işlemci süreç sonuna kadar çalışmalı mı yoksa farklı süreçler arasında geçiş yapmalı mı?

- Süreç değiştirme pahalı
  - Kullanıcı modu ve çekirdek modu arasında geçiş
  - Geçerli süreç kaydedilir
  - Bellek haritası (memory map) kaydedilir
  - Önbellek temizlenir ve yeniden yüklenir

İşlemci Kullanımı

(a) CPU'ya bağlı (bound) bir işlem. (b) G/Ç'ye bağlı bir işlem.



Not

- CPU hızlandığında, süreçler daha fazla G/Ç bağlı olurlar

- Bir G/Ç bağlı süreç çalışmak istediğinde, hızlı bir şekilde değişiklik alması gerekir.

Kavramlar

- Önleyici (preemptive) algoritma
  - Bir süreç, zaman aralığının sonunda hala çalışıyor durumunda ise, askıya alınır ve başka bir süreç çalıştırılır.
- Önleyici olmayan (non-preemptive) algoritma
  - Çalıştırmak için bir süreç seçilir ve bloke olana kadar veya gönüllü olarak işlemciyi serbest bırakana kadar çalışmasına izin verilir.

Çizelgeleme Kategorileri

- Farklı ortamlar farklı çizelgeleme algoritmalarına ihtiyaç duyar
- Toplu (batch)
  - Hala yaygın olarak kullanılıyor
  - Önleyici olmayan algoritmalar süreç geçişlerini azaltır
- Etkileşimli
  - Önleyici algoritma gerekli
- Gerçek zamanlı
  - Süreçler hızlı çalışır ve bloke olur

Çizelgelemenin Hedefleri

- Tüm sistemler
  - Adalet - her işleme CPU'dan adil bir pay vermek
  - Politika uygulama - belirtilen politikanın yürütüldüğünü görme
  - Denge - sistemin tüm parçalarını meşgul tutmak
- Toplu sistemler
  - Verim – birim zamanda yapılan işi maksimize etmek
  - Geri dönüş süresi – başlatma ve sonlandırma arasındaki süreyi en aza indirmek
  - CPU kullanımı - CPU'yu her zaman meşgul tutmak

Çizelgelemenin Hedefleri

- Etkileşimli sistemler
  - Yanıt süresi - isteklere hızla yanıt verilmeli
  - Orantılılık - kullanıcıların beklentilerini karşılamalı
- Gerçek zamanlı sistemler
  - Son teslim zamanı (deadline) - veri kaybı olmamalı
  - Öngörülebilirlik - multimedya sistemlerinde kalite düşüşünden kaçınmalı

Toplu Sistemlerde Çizelgeleme

- İlk gelen alır (first-come first-served)
- Önce en kısa iş (shortest job first)
- Sonraki en kısa kalan süre (shortest remaining time next)

FCFS Hangi Tür Süreçlerde Avantajlı

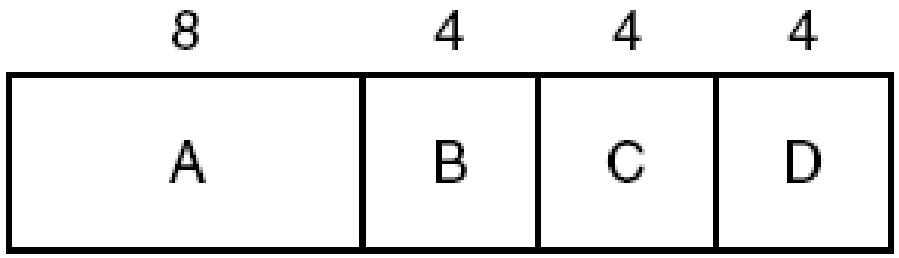
Process	Arrive time	Service time	Start time	Finish time	**Turnaround**	**Weighted turnaround**
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	1.99

FCFS Dezavantajları

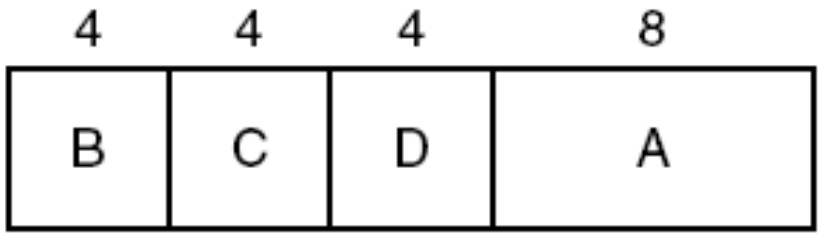
- CPU'ya bağlı bir işlem her seferinde 1 saniye çalışır
- Birçok G/Ç bağlı işlem çok az CPU zamanı kullanır ancak her birinin çok fazla disk okuması gerekir

İlk Önce En Kısa Süreç

(a) Orijinal sıra. (b) En kısa süreci birinci sırada yürütme



(a)



(b)

İlk Önce En Kısa Süreç

- Geri dönüş süreleri: İlk önce en kısa süreç kanıtlanabilir şekilde optimaldir
- Ancak yalnızca tüm süreçler aynı anda mevcut olduğunda

Karşılaştırma

	Process	A	B	C	D	E	average
	Arrive time	0	1	2	3	4	
	Service Time	4	3	5	2	4	
SJF	Finish time	4	9	18	6	13	
	turnaround	4	8	16	3	9	8
	weighted	1	2.67	3.1	1.5	2.25	2.1
FCFS	finish	4	7	12	14	18	
	turnaround	4	6	10	11	14	9
	weighted	1	2	2	5.5	3.5	2.8

Sonraki En Kısa Kalan Süre

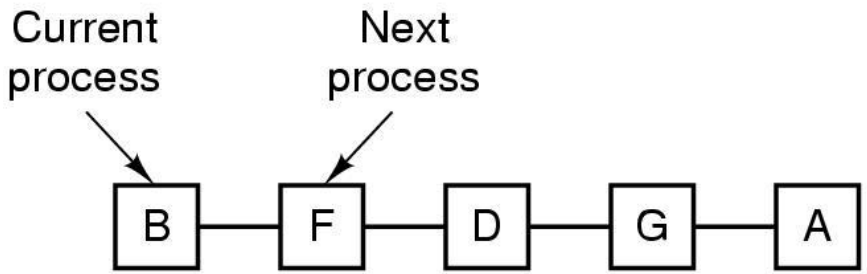
- Sıradaki yürütme için en kısa süreye sahip süreç seçilir
- Önleyici (pre-emptive): yeni sürecin çalışma süresini mevcut işin kalan süresiyle karşılaştır
  - Süreçlerin çalışma sürelerinin önceden bilinmesi gerekiyor

İnteraktif Sistemlerde Çizelgeleme

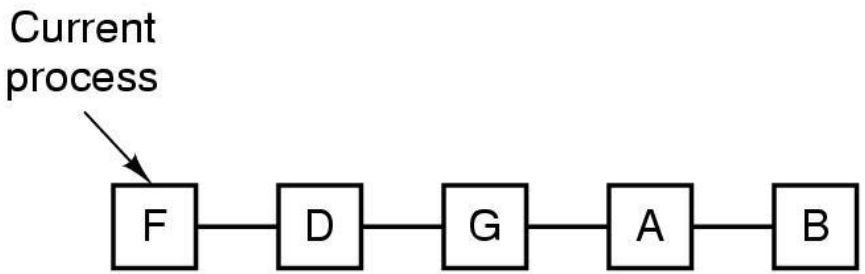
- Sıralı planlama (Round-robin scheduling)
- Öncelik zamanlaması (Priority scheduling)
- Çoklu kuyruk (Multiple queues)
- Sonraki en kısa süreç (Shortest process next)
- Garantili planlama (Guaranteed scheduling)
- Piyango planlaması (Lottery scheduling)
- Adil paylaşım planlaması (Fair-share scheduling)

Sıralı Planlama

(a) Çalıştırılabilir süreçlerin listesi. (b) B, kuantumunu kullandıktan sonra çalıştırılabilir süreçlerin listesi.



(a)



(b)

Örnek Sıralı Planlama

- Kuantum = 20

Süreç	P1	P2	P3	P4
Süre	53	17	68	24

- Gantt çizelgesi

P1	**P2**	**P3**	**P4**	**P1**	**P3**	**P4**	**P1**	**P3**	**P3**
20	37	57	77	97	117	121	134	154	162

- SJF’den yüksek geri dönüş süresi, daha iyi yanıt (response) süresi

Bağlam Anahtarlama

- Sıralı planlamanın performansı büyük ölçüde zaman kuantumunun boyutuna bağlıdır.
- Zaman kuantumu
  - Çok büyük ise FCFS ile benzer
  - Çok küçük:
    - Donanım: Süreç paylaşımı
    - Yazılım: bağlam değiştirme, yüksek ek yük maliyet, düşük CPU kullanımı
  - Bağlam değiştirme masrafına göre büyük olmalıdır

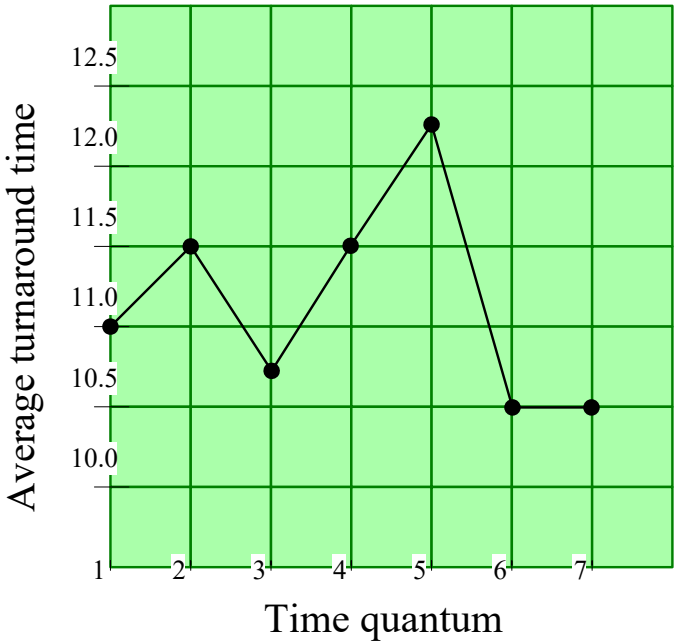
Bağlam Anahtarlama

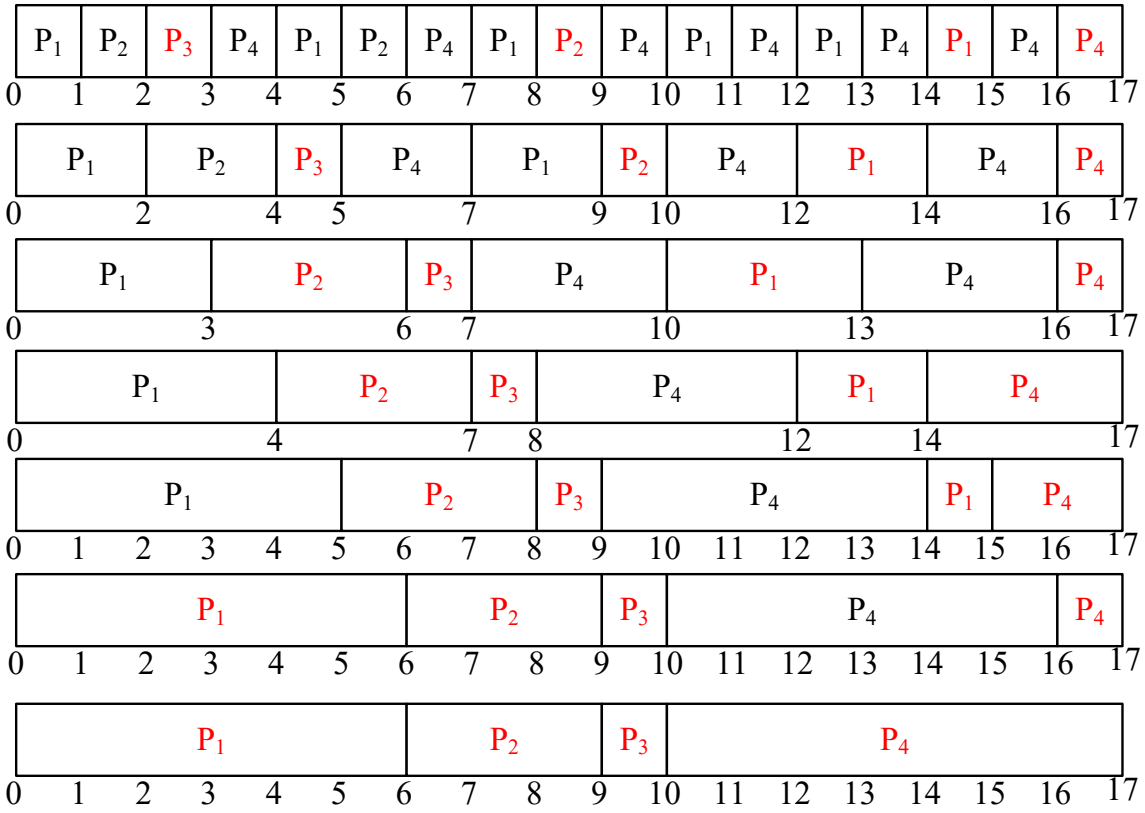
- Kuantum 4 milisaniye ve bağlam anahtarlama 1 milisaniye ise CPU zamanının %20'si boşa gider
- Kuantum 100 milisaniye ise, boşa harcanan zaman %1'dir, ancak daha az duyarlıdır
- 20-50 milisaniye civarında bir kuantum makul

Dönüş Süreleri

- q arttıkça dönüş süresi iyileşir mi?
- İşlemci kullanma zamanlarının %80'i q'dan kısa olmalıdır.

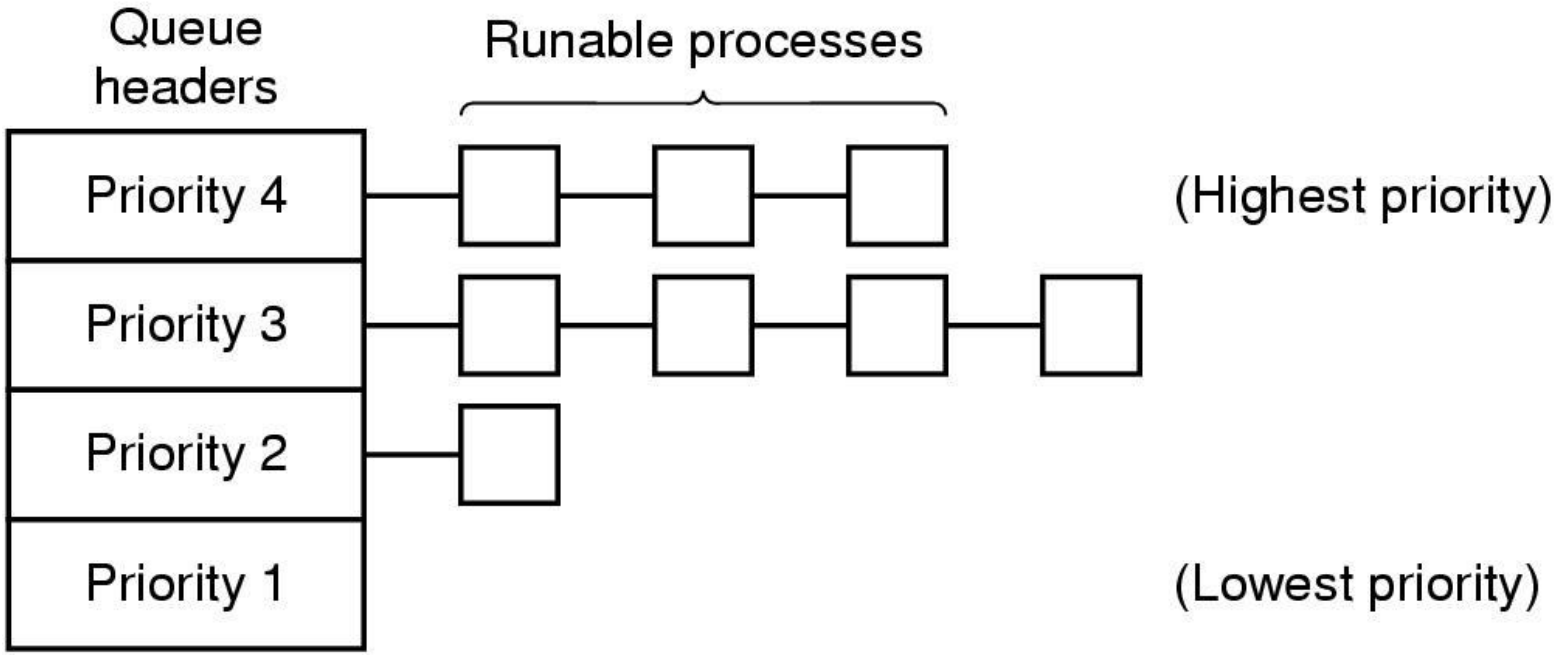
Process	Time
P_1	6
P_2	3
P_3	1
P_4	7





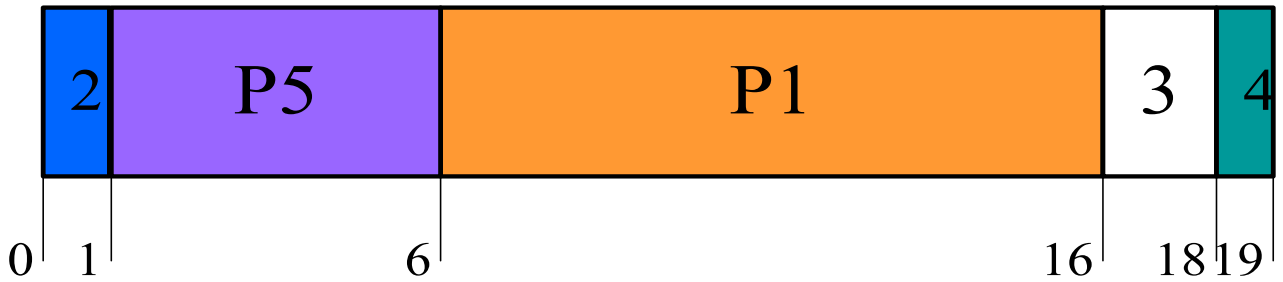
Öncelik Zamanlaması

- Her önceliğin bir öncelik numarası vardır
- En yüksek öncelik en önce çizelgelenir
- Tüm öncelikler eşitse, FCFS gibi çizelgelenir.



Örnek

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2



- Öncelik (önleyici olmayan)

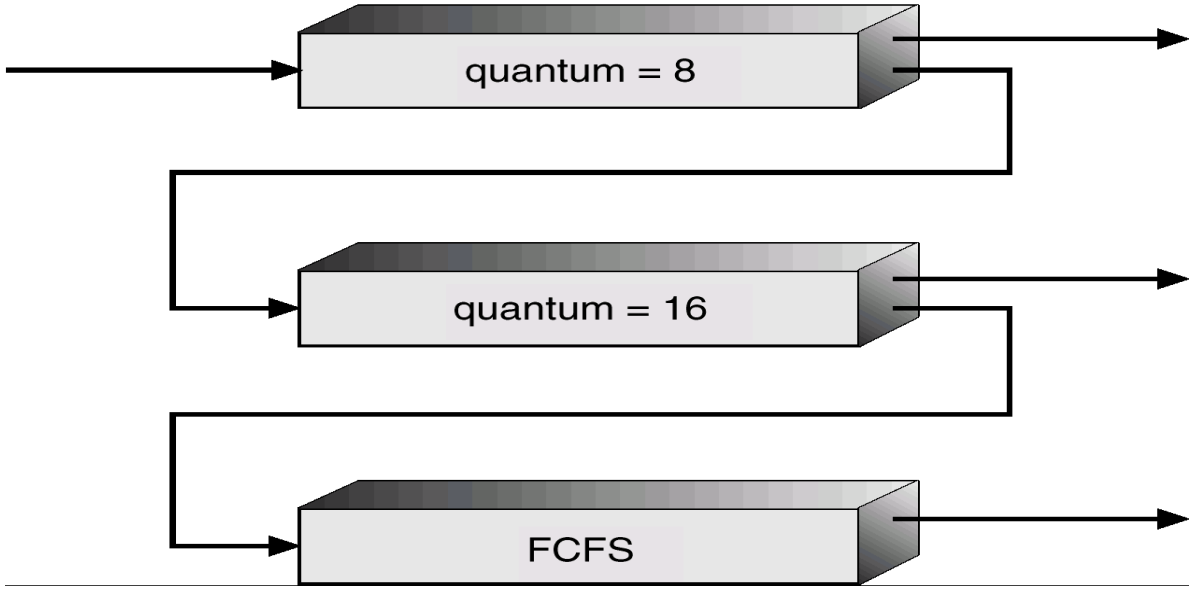
- Ortalama bekleme süresi = (6 + 0 + 16 + 18 + 1) /5 = 8,2

Çoklu Kuyruk

- Bellekte yalnızca bir süreç
- CPU'ya bağlı sürece büyük kuantum ve etkileşimli sürece küçük kuantum verilir.
- Öncelikli sınıflar
  - En yüksek sınıf bir kuantum için koşar; bir sonraki en yüksek sınıf iki kuanta için koşar, vb.
  - Bir süreç kuantumunu tükettiğinde, bir sonraki sınıfa taşınır.

Örnek

- Üç kuyruk:
- Q0 – zaman kuantumu 8 milisaniye, FCFS
- Q1 – zaman kuantumu 16 milisaniye, FCFS
- Q2 – FCFS



Çoklu Kuyruk

- İlk önce CPU'ya bağlı, daha sonra etkileşimli olan işlemler için iyi değil
- Terminalde her satırbaşı (enter tuşu) yazıldığında terminale ait işlem en yüksek öncelik sınıfına taşınıyordu.
- Ne oldu?

Sonraki En Kısa Süreç

- İnteraktif sistemlerde bir sürecin kalan süresini tahmin etmek zordur.
- Geçmiş davranışa dayalı olarak tahminde bulunun ve en kısa tahmini çalışma süresini çalıştırın

T0, T0 / 2 + T1 / 2, T0 / 4 + T1 / 4 + T2 / 2..

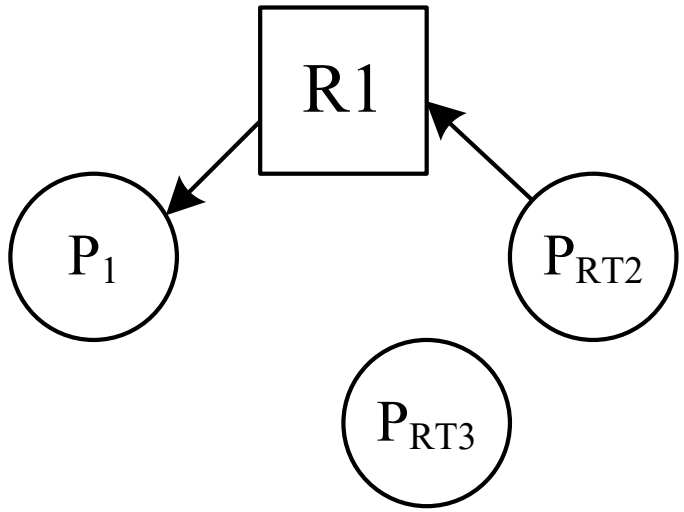
- Yaşlanma (aging):
  - Geçerli ve önceki tahminin ağırlıklı ortalamasını alarak bir serideki sonraki değeri tahmin edin

Öncelik Ters Çevirme

- Daha yüksek öncelikli işlemin, daha düşük öncelikli başka bir işlem tarafından şu anda erişilmekte olan çekirdek verilerini okuması veya değiştirmesi gerektiğinde
- Yüksek öncelikli süreç, daha düşük öncelikli bir sürecin bitmesini bekleyecektir.

Öncelik Ters Çevirme

- Öncelik: P1 < PRT3 < PRT2
- PRT3, P1’i önler; PRT2, P1'i bekler
- PRT2, PRT3'ü bekler



Piyango Çizelgeleme

- İşlemci süresi için saniyede birkaç kez çekiliş yapılır
- "Daha önemli" süreçler için daha fazla çekiliş hakkı tanıyarak önceliklerin değiştirilebilmesine izin verir.

Gerçek Zamanlı Çizelgeleme

- Katı gerçek zamanlıya karşı yumuşak gerçek zamanlı
  - Katı: Bir fabrikada robot kontrolü
  - Yumuşak: CD çalar
- Olaylar periyodik olabilir veya periyodik olmayabilir
- Algoritmalar statik (çalışma sürelerini önceden bilinen) veya dinamik (çalışma zamanı kararları) olabilir.

SON