Bölüm 6: Senkronizasyon

İşletim Sistemleri

Süreç

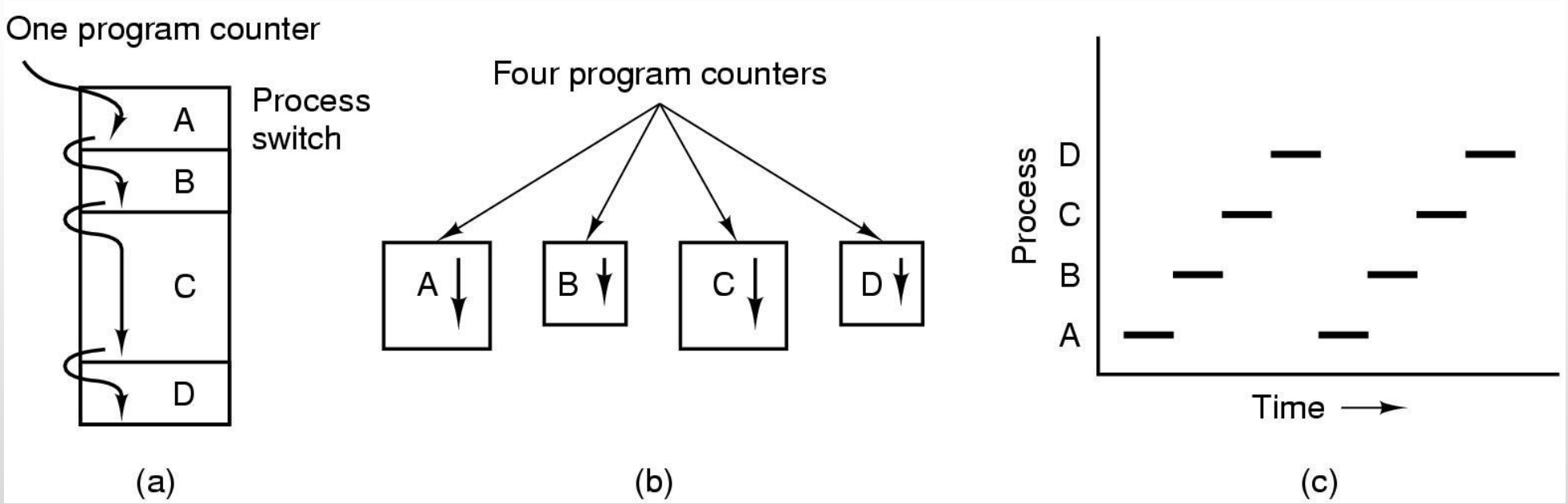
- Yazmaçlar, değişkenler ve bir program sayacına sahip bir program örneği
- Program, girdi, çıktı ve durumu vardır.
- Bu fikir neden gerekli?
  - Bir bilgisayar aynı anda birçok hesaplamayı yönetir - bunu nasıl yaptığını açıklamak için bir soyutlamaya ihtiyaç duyar

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)

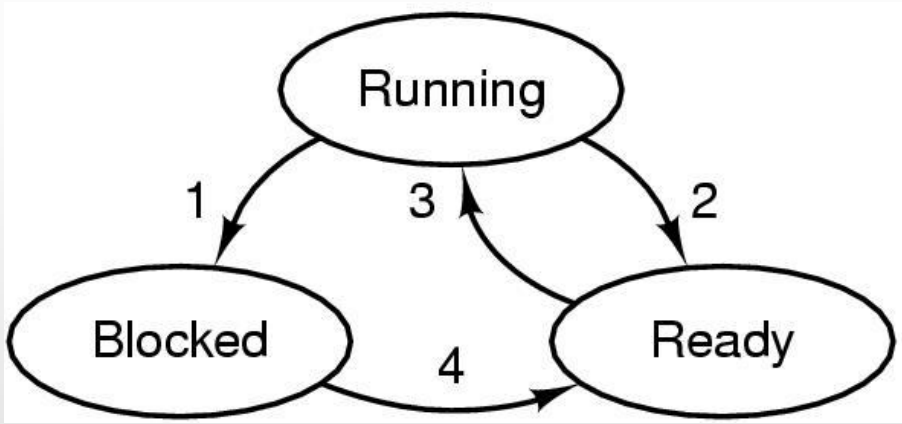
Süreç Sonlandırma

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

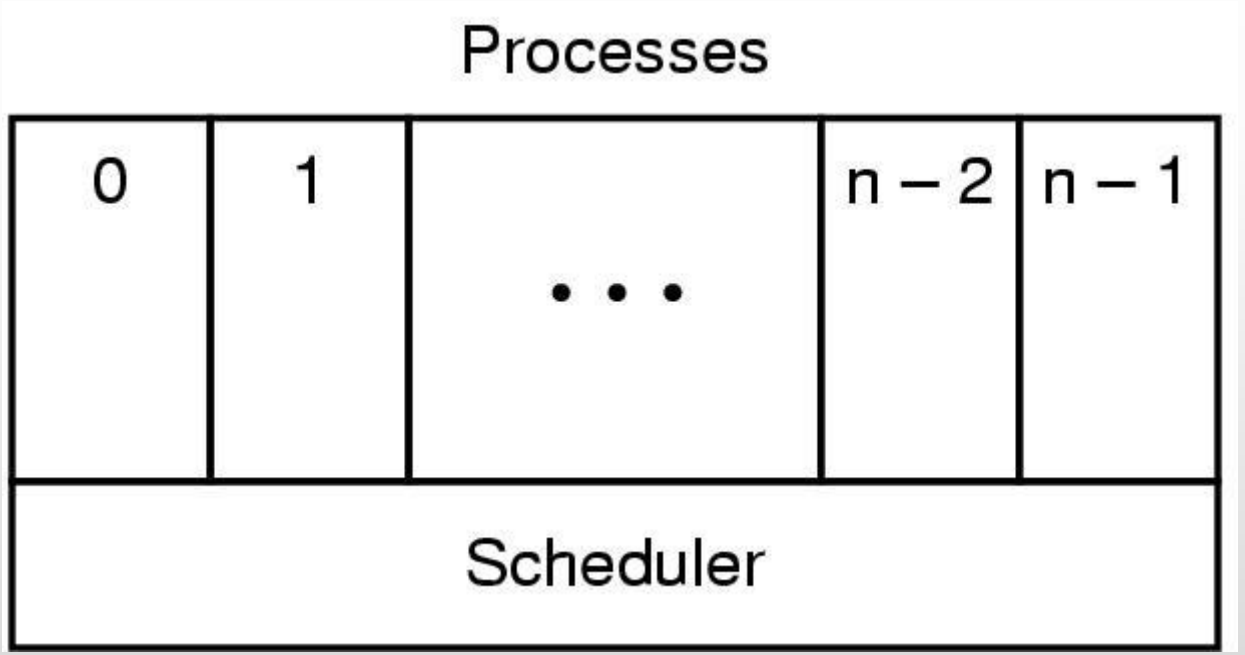
Bir süreç çalışıyor, engellenmiş veya hazır durumda olabilir.



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		

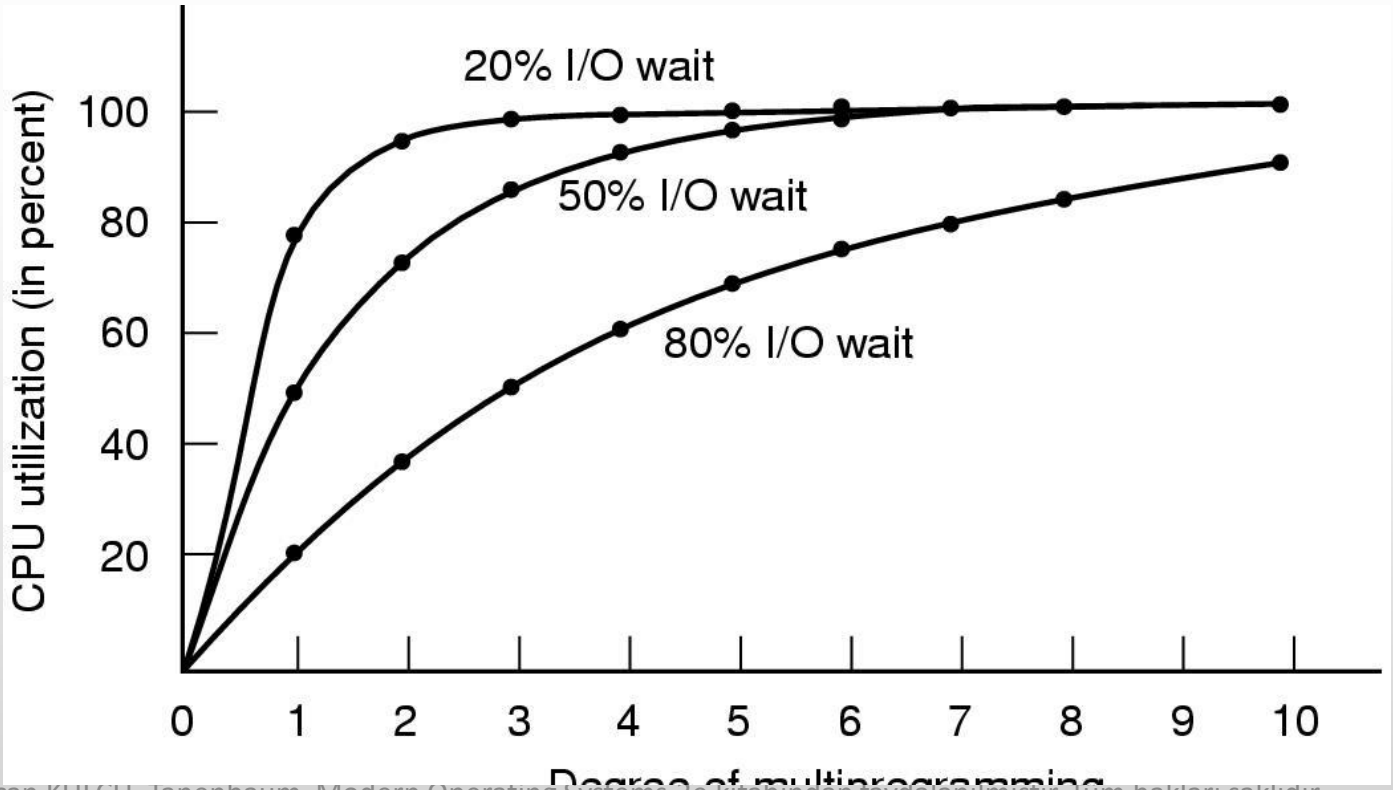
Süreçleri Gerçekleştirme

Bir kesme oluştuğunda işletim sisteminin en alt seviyesi ne yapar.

1. Donanım program sayacı vb. verileri kaydeder
2. Donanım, kesme vektöründen yeni program sayacı yükler.
3. Assembly dili prosedürü yazmaçları kaydeder.
4. Assembly dili prosedürü yeni yığın hazırlar.
5. C kesme hizmeti çalışır (genellikle girişi okur ve ara belleğe alır).
6. Çizelgeleyici hangi sürecin çalıştırılacağına karar verir.
7. C kesme hizmeti Assembly dili prosedürüne geri döner.
8. Assembly dili prosedürü seçilen süreci başlatır.

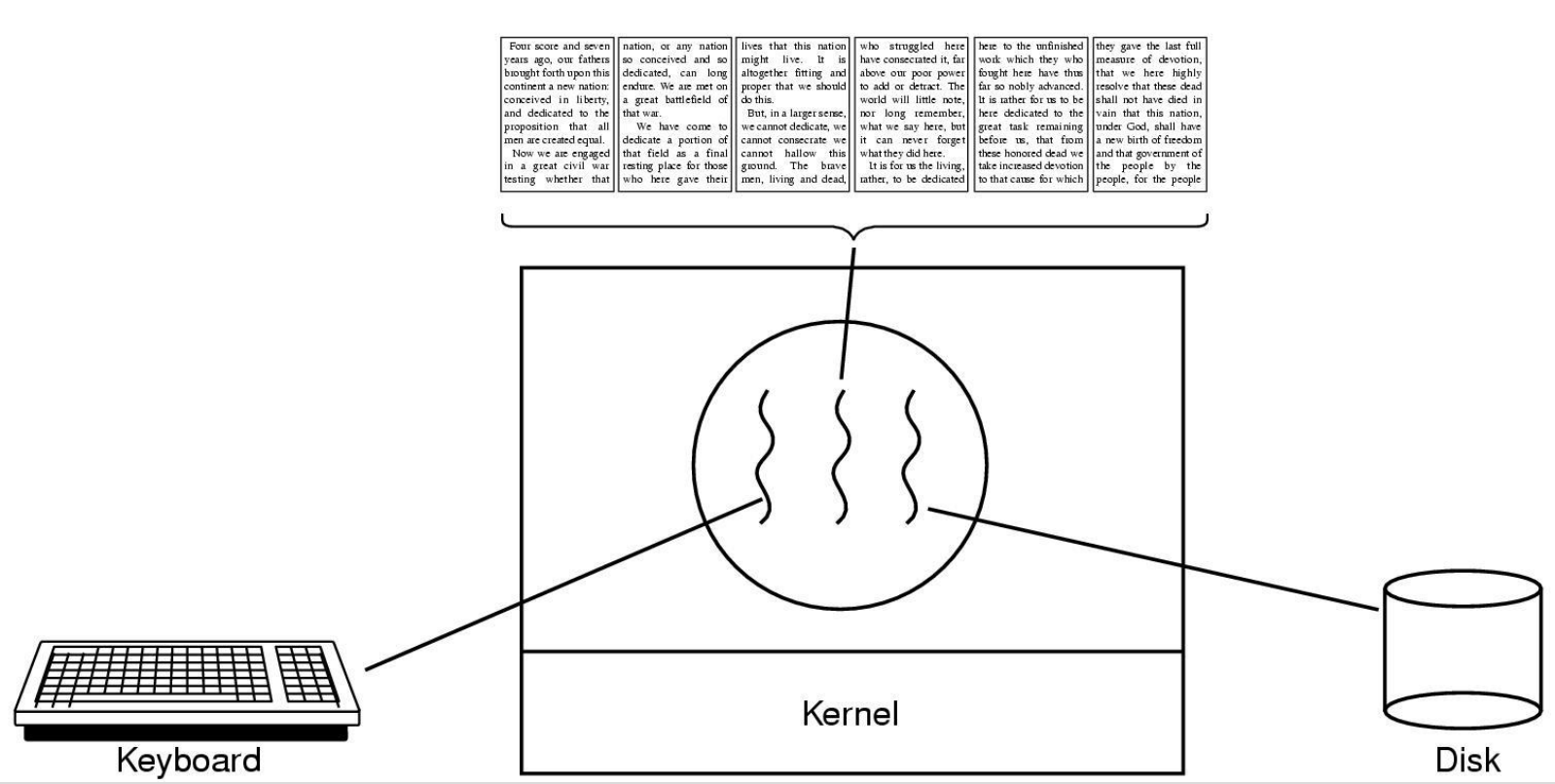
Çoklu Programlama Modellemesi

Bellekte bulunan süreç sayısının bir fonksiyonu olarak CPU kullanımı grafiği.



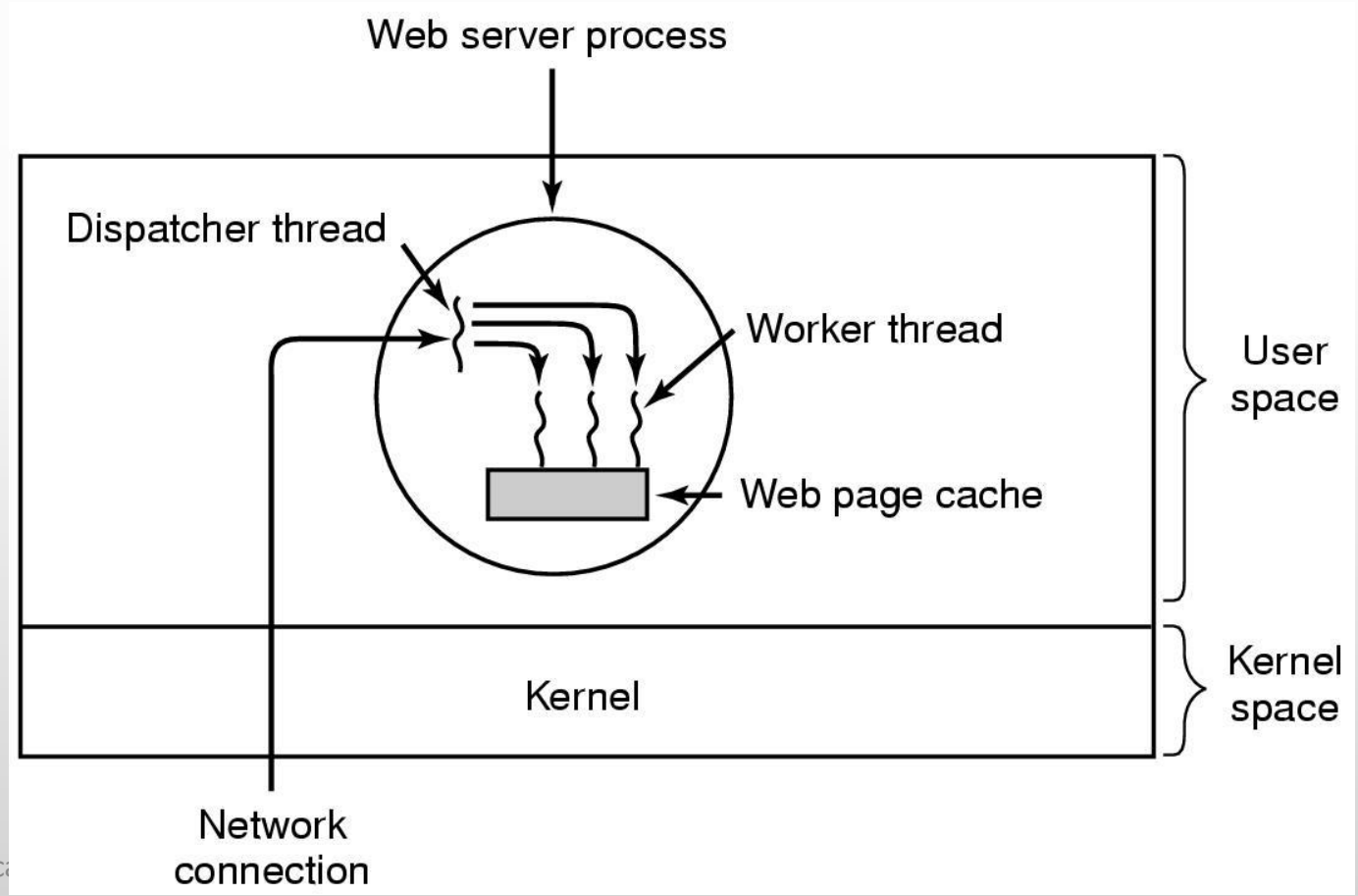
İş Parçacığı

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 (context switch)(pahalı)

Web Sunucusu Geliştirmek için Üç Farklı 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 paylaşılan veriler
  - Adres alanı (address space), Global değişkenler (variables), Açık dosyalar (open files), Çocuk süreçler (child processes), Bekleyen alarmlar (waiting alarms), Sinyal ve Sinyali ele alacak süreçler (signal handlers)

- Her bir iş parçacığına özel veriler
  - Program sayacı (counter), Yazmaçlar (register), Yığın (stack), Durum (state)

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

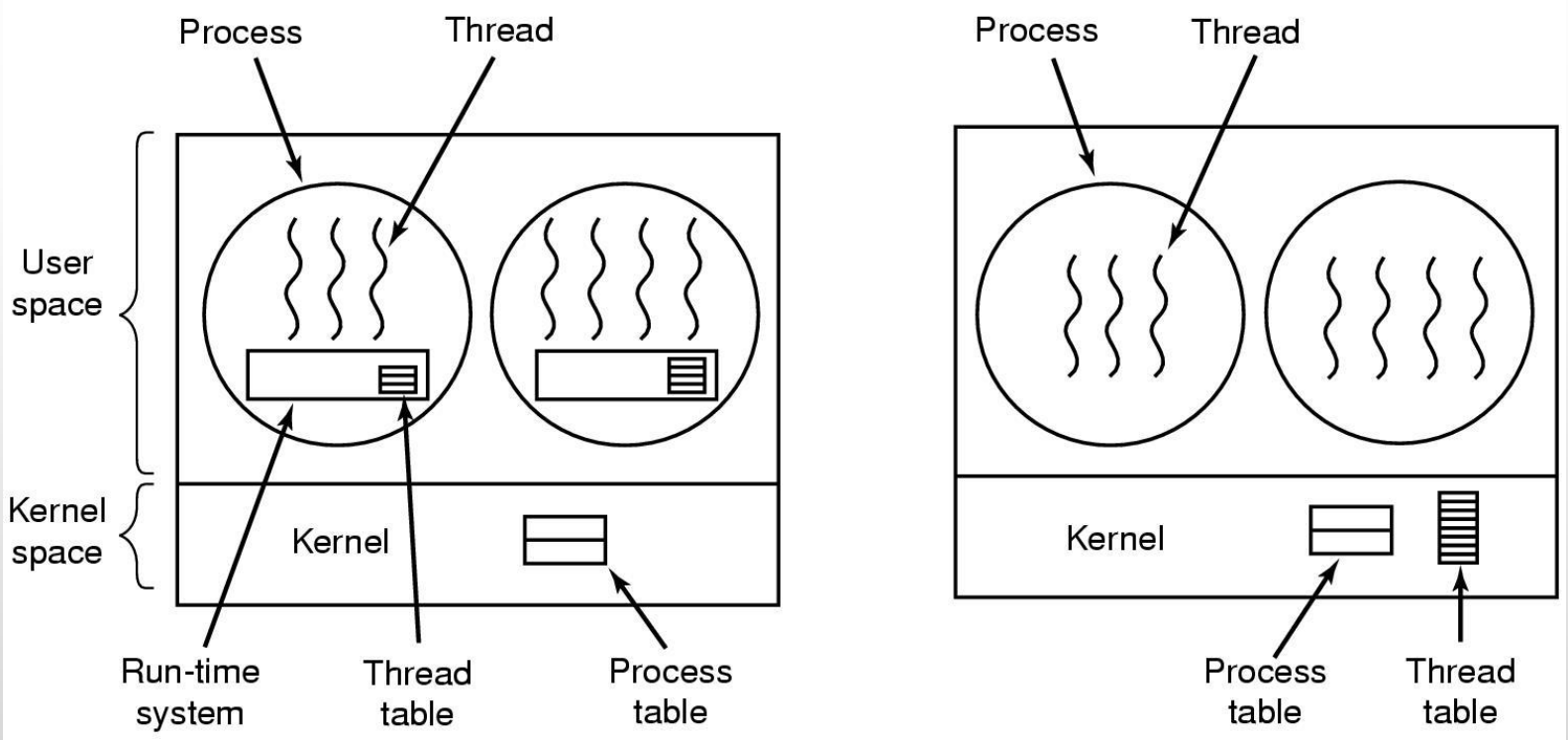
POSIX kütüphanesi (pthreads)

- IEEE Unix standart kütüphane çağrıları

İşlev çağrısı	Açıklama
Pthread_create	Yeni bir iş parçacığı oluştur
Pthread_exit	Çağıran iş parçacığını sonlandır
Pthread_join	Belirli bir iş parçacığının sonlanmasını bekle
Pthread_yield	Başka bir iş parçacığının çalışması için CPU'yu serbest bırak
Pthread_attr_init	Bir iş parçacığının öznitelik yapısını oluştur ve başlat
Pthread_attr_destroy	Bir iş parçacığının öznitelik yapısını kaldır

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 (-)

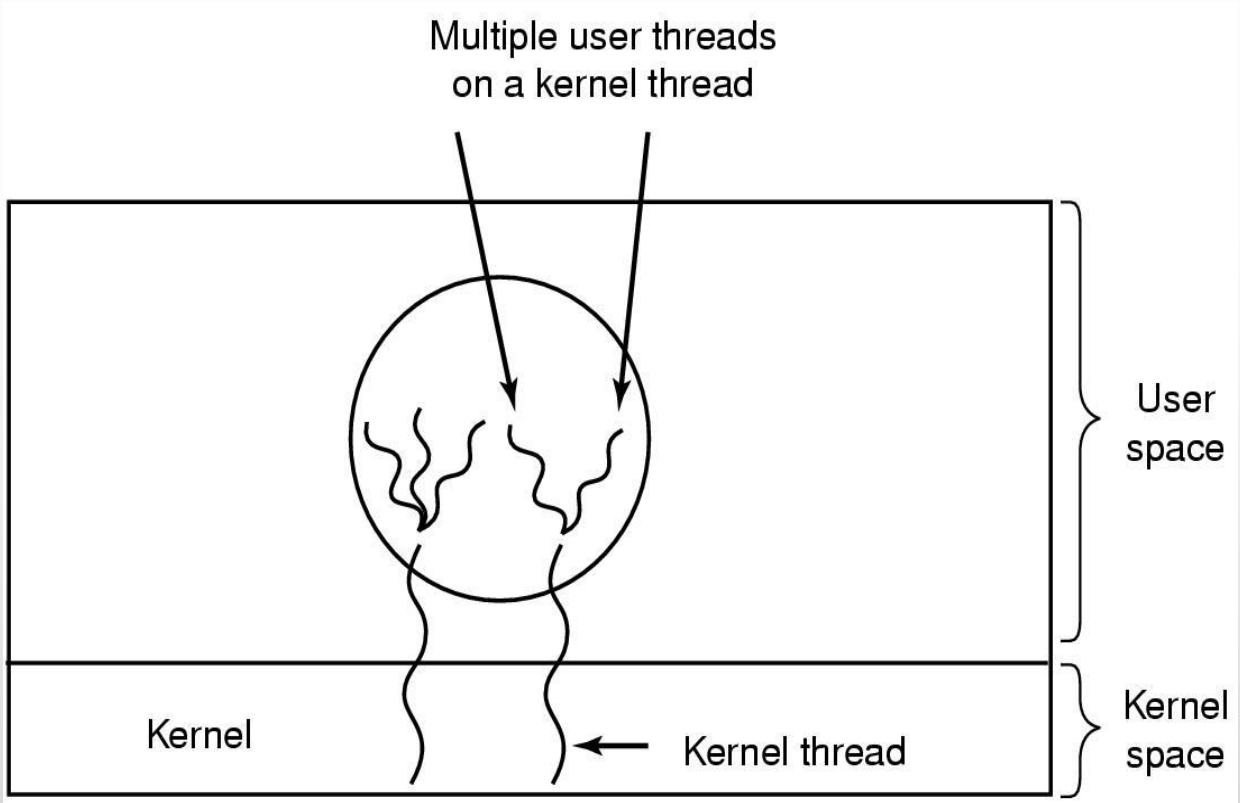
- İş parçacığının sistem çağrısını yürütmesine izin verilemez, çünkü diğer tüm iş parçacıklarını engelleyecektir.
- Zarif bir çözüm yok
  - Çağrıları engellememek için sistem kütüphanesi kırılabilir (hack)
  - Unix'in bazı sürümlerinde aynı işi yapan benzer sistem çağrıları kullanılabilir
- İş parçacıkları 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

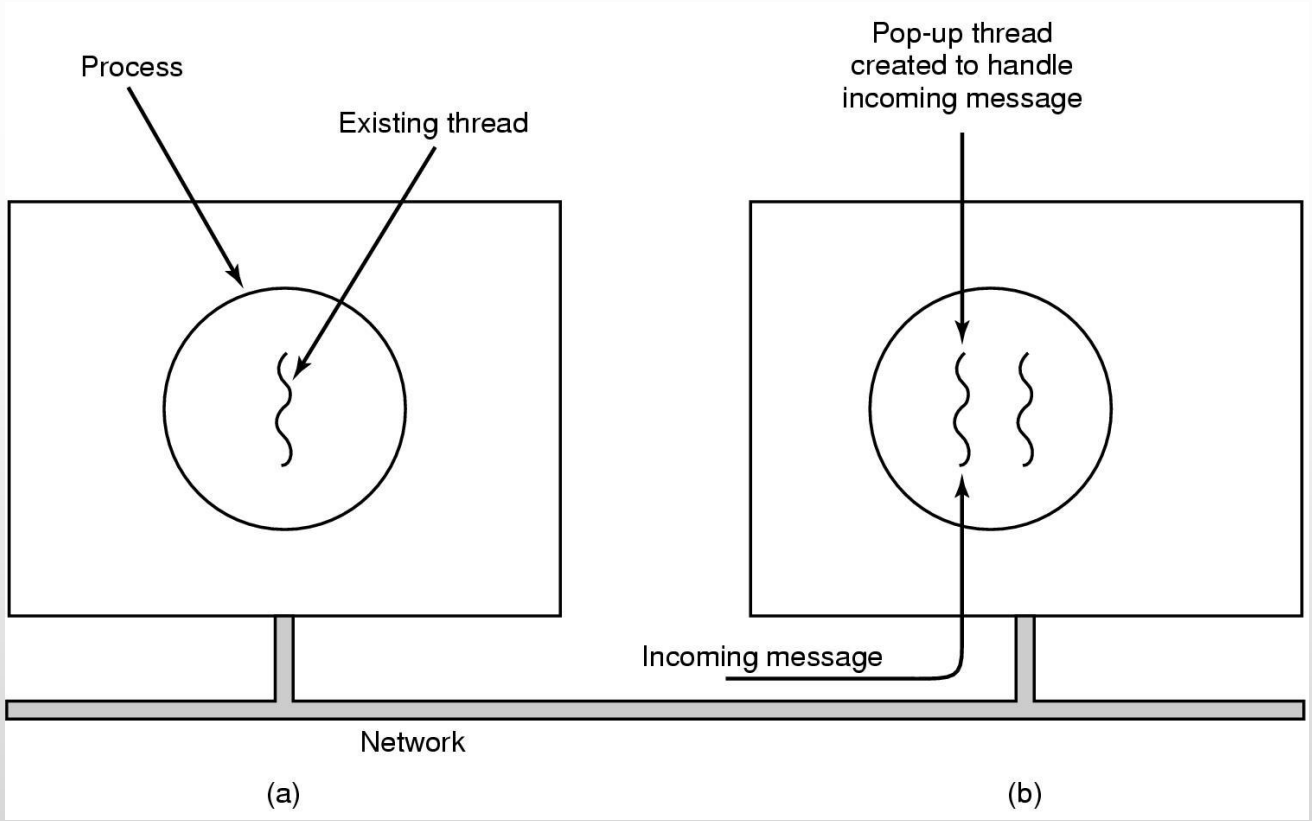
- Çekirdek, sadece çekirdek iş parçacıklarından haberdardır
- Kullanıcı düzeyinde iş parçacıkları, çekirdekten bağımsız oluşturulur, çizelgelenir, sonlandırılır
- Programcı, kaç tane kullanıcı seviyesi ve kaç tane çekirdek seviyesi iş parçacığı kullanacağını belirler

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

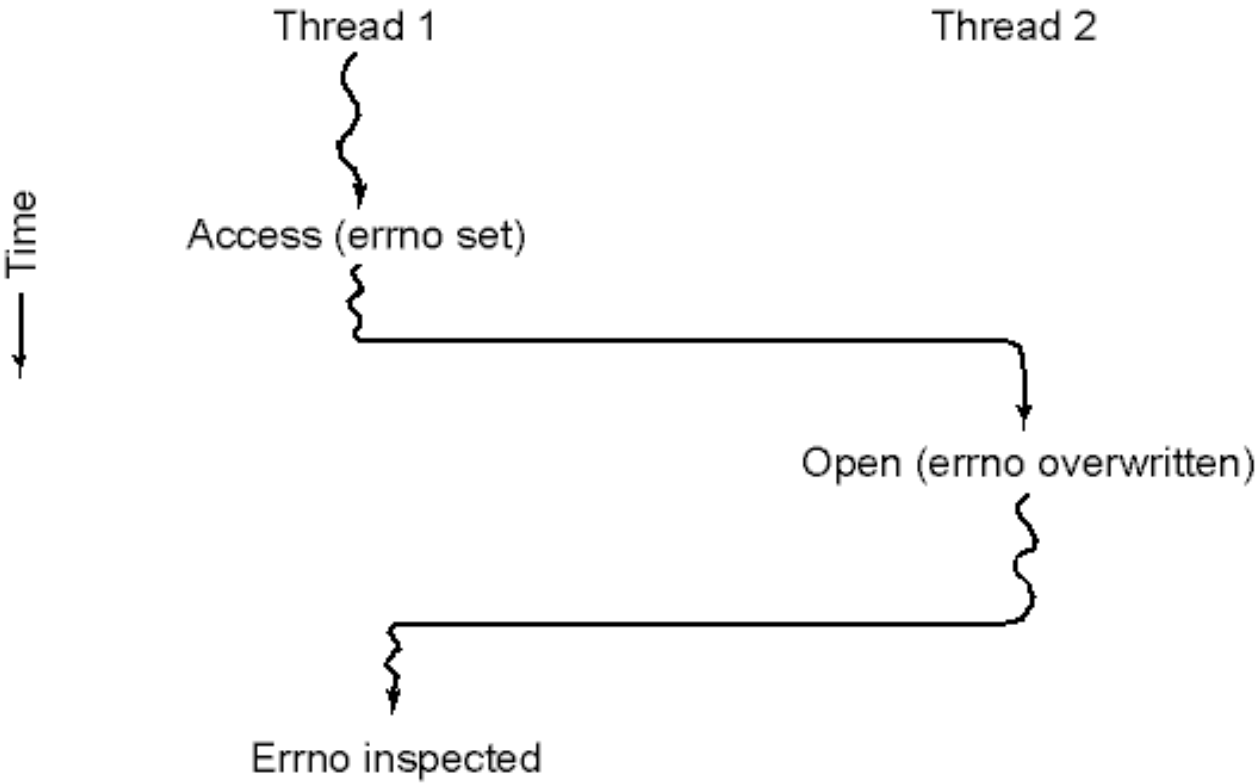


Açılır İş Parçacıkları

- 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

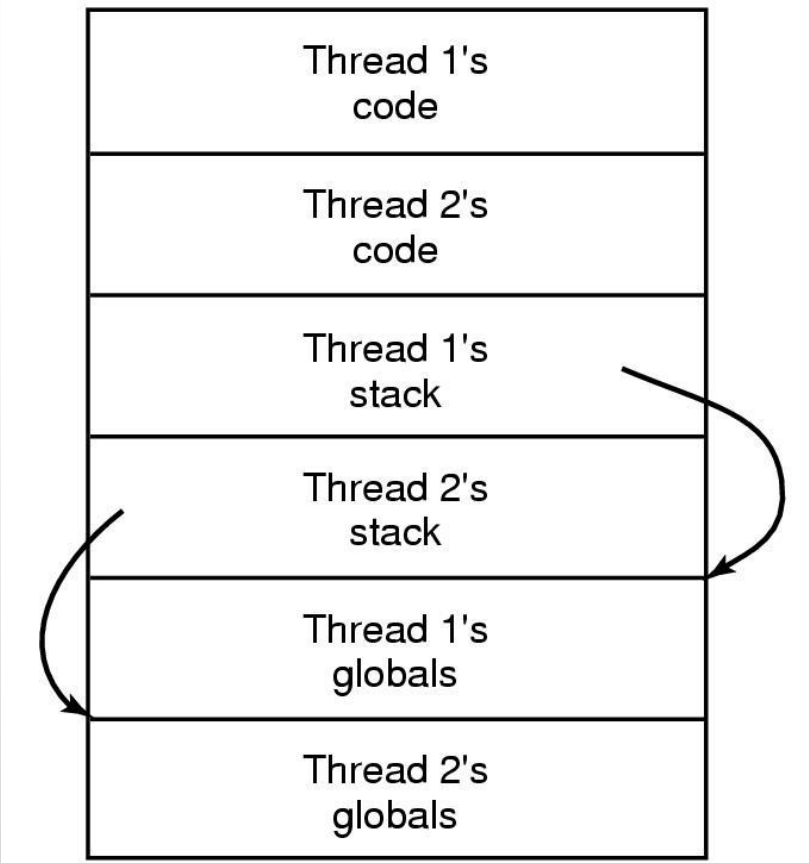
İş Parçacıkları Arasında Çakışma

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 etkinleştirir (web sunucusu)
- 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

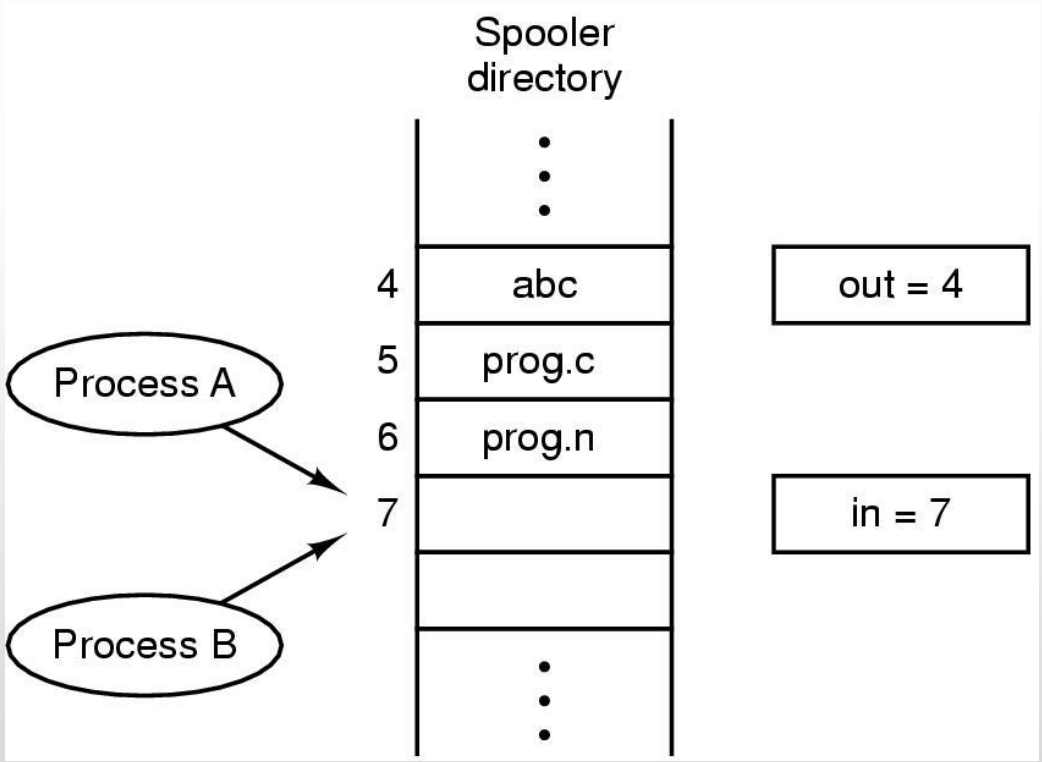
- Senkronizasyon
  - Paylaşımlı bellek ve değişkenlere erişim kontrol edilmelidir.
  - Program koduna karmaşıklık, hatalar ekleyebilir. Yarış koşullarından, kilitlenmelerden ve diğer sorunlardan kaçınmak gerekir
- Bağımsızlık eksikliği
  - Ağır Ağırlık İşlemde (HWP) iş parçacıkları bağımsız değildir
  - 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 Problemler

- 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

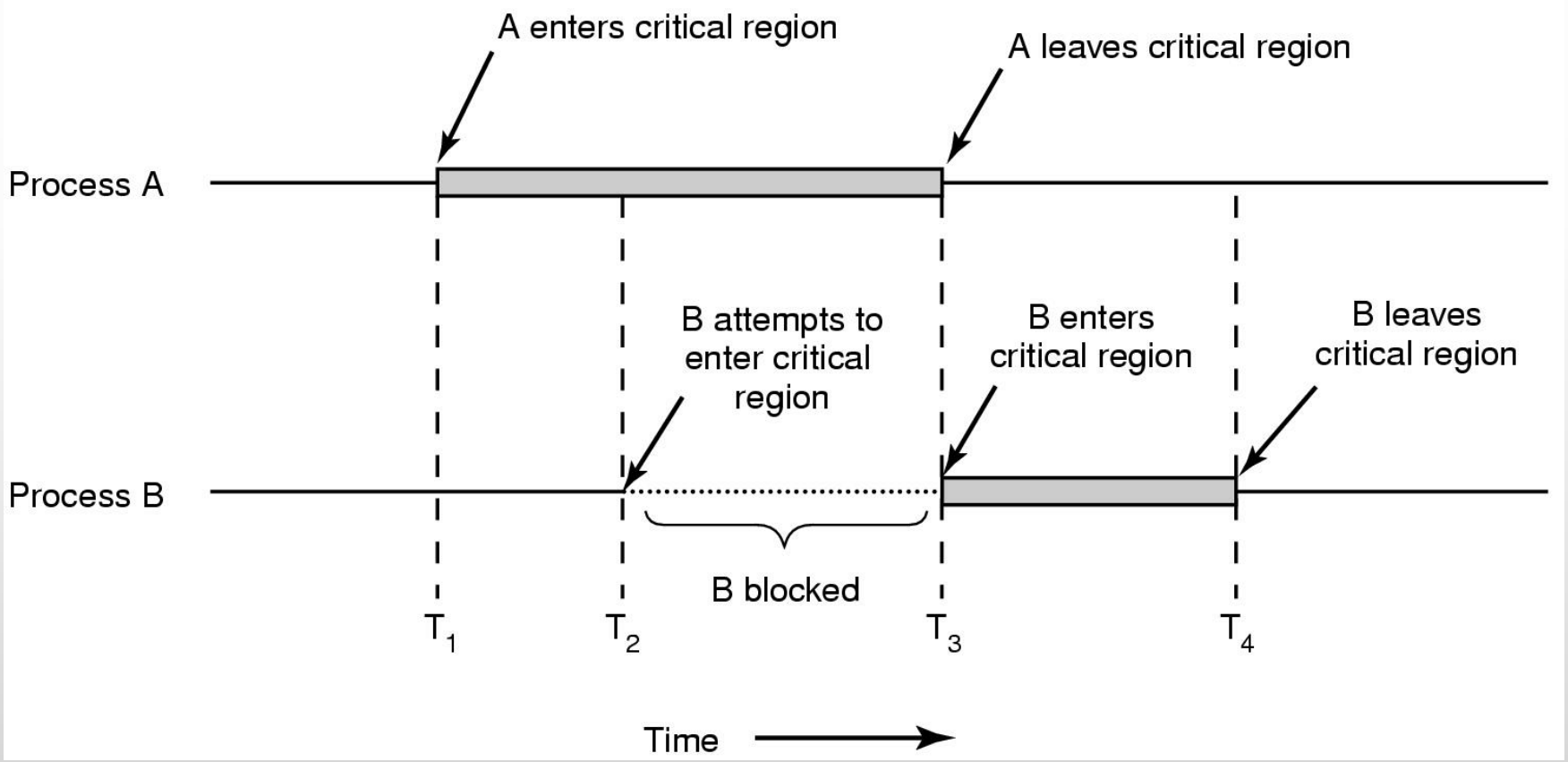
Karşılıklı dışlama sağlamak için dört koşul

- 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!, her işlem CPU'yu kullanma sırası aldığından adaleti sağlar.

```
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

- Algoritma, hangi işlemin kritik bölüme girmesi gerektiğini belirtmek için "turn" ve "flag[2]" olmak üzere iki değişken kullanır.
- flag, süreç tarafından kritik bölüme girme niyetini belirtir.
- turn, daha sonra hangi sürecin gireceğini belirtir.
- Algoritma, her iki işlemin de kritik bölüme aynı anda girmesini önlemek için bir meşgul bekleme döngüsü ve bir dizi koşul kullanır.
- Algoritma, karşılıklı dışlamayı sağlar ve süreçlerin sonsuz bir bekleme döngüsüne girmesini engeller.
- Meşgul bekleme döngüsü önemli miktarda CPU zamanı tüketebilir!

Peterson'un Çözümü

```
private static final int N = 2; // Number of threads
private static volatile boolean[] flag = new boolean[N];
private static volatile int turn = 0;
private static int counter;

private static void incrementCounter() {
     int i = (int) (Thread.currentThread().getId() % N);
     int j = (i + 1) % N;
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j) {} // Spin loop
     // Critical region
     counter++;
     System.out.println("Counter: " + counter + " i: " + i + " j: " + j);
     flag[i] = false;
}
```

TSL Komutu

- TSL (Test and Set Lock), paylaşılan kaynaklara erişimi senkronize etmek için basit ve verimli bir mekanizma sağlayarak veri bozulması ve yarış koşulları riskini azaltır.
- TSL komutu, donanım düzeyinde atomik işlemler kullanılarak hız ve verim sağlar.
- İşletim sistemi tarafından paylaşılan veri yapıları ve aygıt sürücüleri gibi kritik bölümlere erişimi senkronize etmek için kullanılır.
- Çoğu modern CPU mimarisi ve işletim sistemiyle uyumludur.

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 (Exchange), iki işlenenin içeriğini atomik olarak değiştirerek, değişimin tek bir adımda tamamlanmasını sağlar.
- XCHG komutu, işletim sistemi tarafından kilitleri, semaforları ve diğer senkronizasyon mekanizmalarını uygulamak için kullanılır.
- XCHG komutu, çok iş parçacıklı bir ortamda süreçler arası iletişim ve senkronizasyon için kullanılır.

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 contents of the register and lock
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 - Producer

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

void producer()
{
  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? */
  }
}
```

Ölümcül Yarış Durumu - Consumer

```
void consumer()
{
  while (true) { /* repeat forever */
    if (count == 0) sleep(); /* if buffer empty, 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 - Producer

```
int N = 100; /* number of slots in the buffer */
Semaphore full = new Semaphore(0);  /* controls access to critical region */
Semaphore empty = new Semaphore(QUEUE_SIZE); /* counts empty buffer slots */
Semaphore mutex = new Semaphore(1); /* counts full buffer slots */

void producer()
{
  while (true) { /* repeat forever */
    item = produce_item(); /* generate something to put in buffer */
    down(empty); /* decrement empty count */
    down(mutex); /* enter critical region */
    insert_item(item); /* put item in buffer */
    up(mutex); /* leave critical region */
    up(full); /* increment count of full slots */
  }
}
```

Semafor Kullanımı - Consumer

```
void consumer()
{
  while (true) { /* repeat forever */
    down(full); /* decrement full count */
    down(mutex); /* enter critical region */
    item = remove_item(); /* take item out of buffer */
    up(mutex); /* leave critical region */
    up(empty); /* increment count of empty slots */
    consume_item(item); /* print 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 - Producer

```
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 */
  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);
}
```

Pthreads Mutex - Consumer

```
void *consumer(void *ptr) /* consume data */
{
  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);
}
```

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

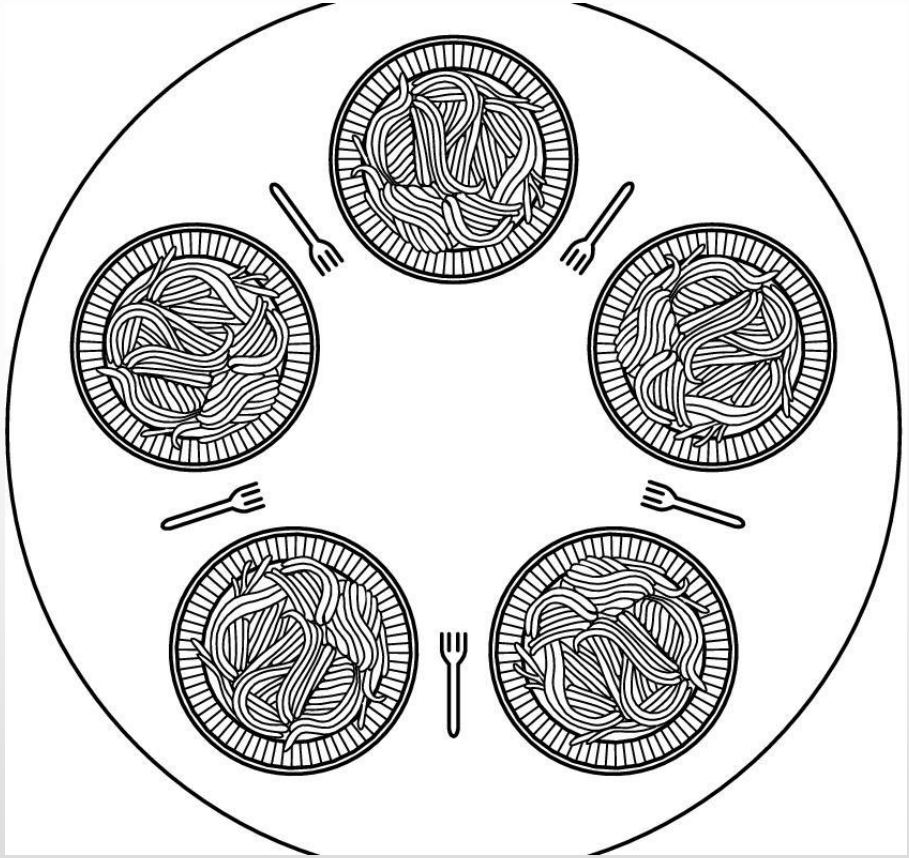
- Gözleyici devam edemeyeceğini anladığında (arabellek dolu), bir koşul değişkeninde (dolu) bir sinyal yayınlayarak sürecin (üretici) bloke olmasına neden olur
- Başka bir sürecin (tüketici) gözleyiciye girmesine izin verilir.
- Bu işlem, bloke olan sürecin (üretici) uyanmasına neden olacak bir sinyal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



Dining Philosophers

```
while(true) {
// Initially, thinking
  think();
// Take a break from thinking, hungry now
  pick_up_left_fork();
  pick_up_right_fork();
  eat();
  put_down_right_fork();
  put_down_left_fork();
// Not hungry anymore. Back to thinking!
}
```

Dining Philosophers - loop

```
#define LEFT (i+N-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */

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(); /* philosopher is eating */
        put_forks(i); /* put both forks back on table */
    }
}
```

Dining Philosophers – take forks

```
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 */
}
```

Dining Philosophers – put forks

```
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 */
}
```

Dining Philosophers – test state

```
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 - writer

```
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 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 - reader

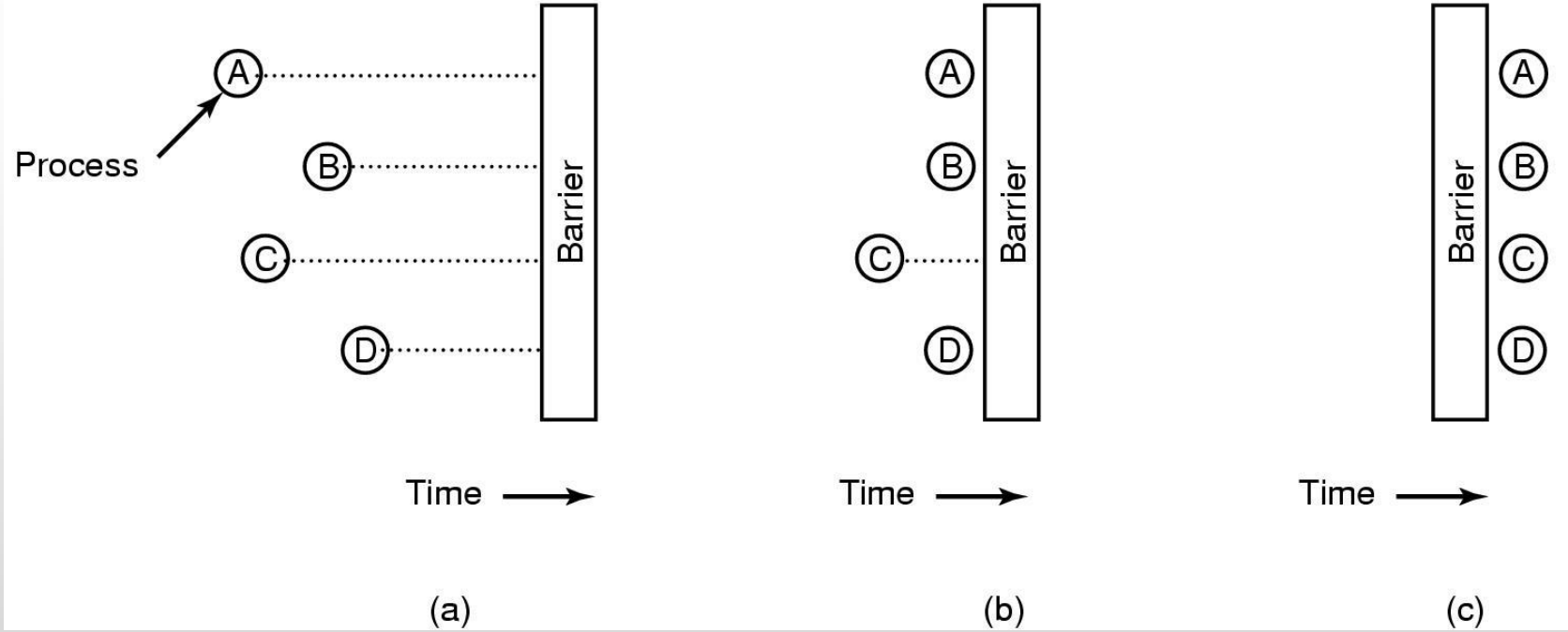
```
void reader(void)
{
  while (TRUE) { /* repeat forever */
    down(&mutex); /* get exclusive access to 'rc' */
    rc = rc + 1; /* one reader more now */
    if (rc == 1) down(&db); /* if this is the first reader ... */
    up(&mutex); /* release exclusive access to 'rc' */
    read_data_base();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

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

Bariyer (Barriers)

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



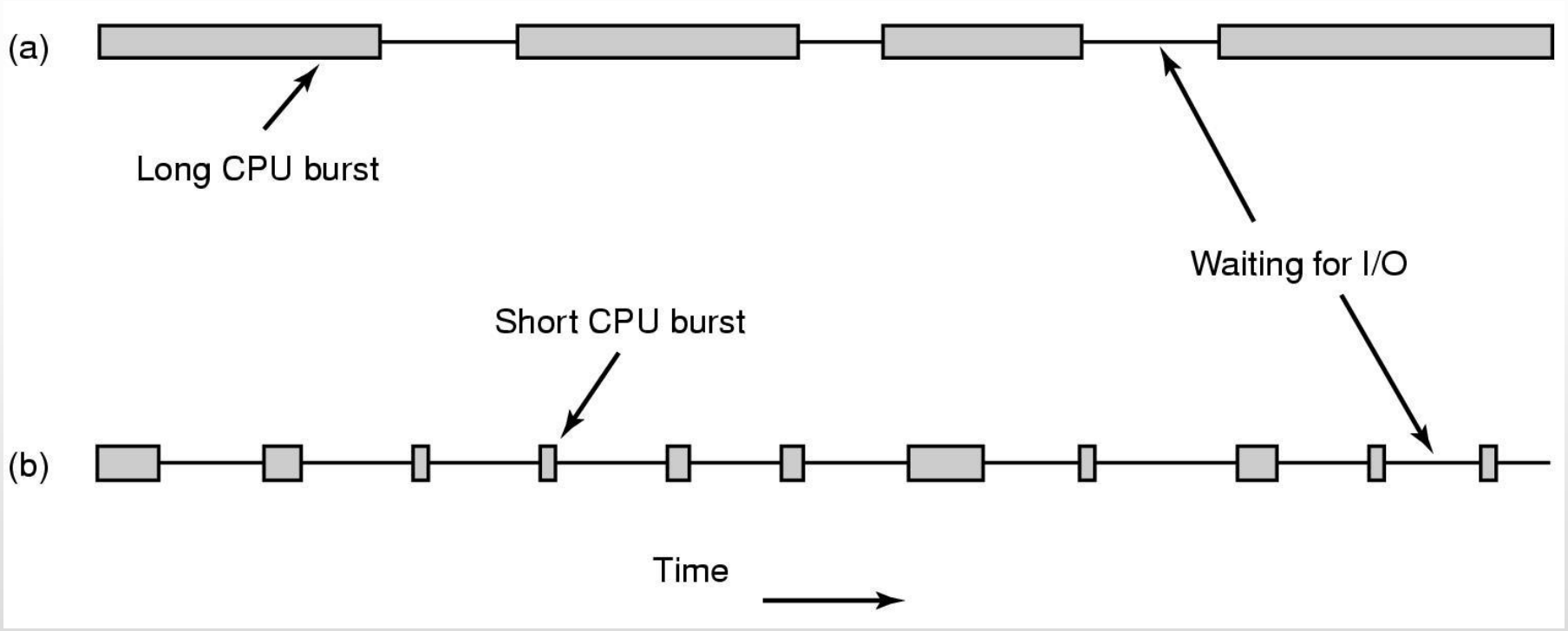
Çizelgeleme

- 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)
  - Süreçler, kuyruğa geldikleri sırayla yürütülür. Basit ve uygulaması kolay. Uzun işlerde düşük performans, daha kısa işler için bekleme süresine neden olur.
- Önce en kısa iş (shortest job first)
  - Kısa yürütme süresine sahip süreçlere, uzun olanlara göre öncelik verilir. CPU kullanımını üst düzeye çıkarır. Süreçlerin yürütme süresini tahmin etmek zor.
- Sonraki en kısa kalan süre (shortest remaining time next)
  - En kısa kalan süreye sahip işler önce yürütülür. Dinamik yaklaşım, yürütme süresi değiştikçe kendini ayarlar. Her iş için kalan süreyi takip etme ve kuyrukta sık sık değişiklik yapma yükü var.

FCFS Hangi Tür Süreçlerde Avantajlı

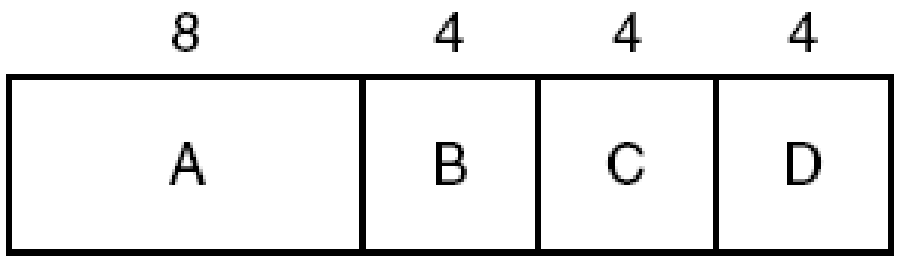
Process	Arrive time	Service time	Start time	Finish time	Turnaround	Weighted turnaround
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	1.99

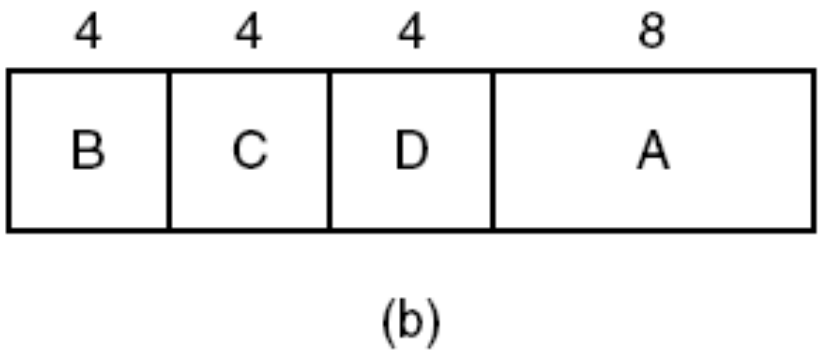
FCFS Dezavantajları

- CPU'ya bağlı bir süreç 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





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

Sonuç

- Her algoritmada, CPU tahsisi farklı kriterlere göre belirlenir.
- FCFS ve SJF önleyici değildir, yani bir iş başlatıldıktan sonra tamamlanıncaya kadar devam eder.
- SRTN, kalan yürütme süreleri değiştikçe işler arasında geçiş yapılmasına izin veren önleyici bir algoritmadır.
- Algoritma seçimi, işletim sisteminin ve üzerinde çalıştığı sistemin özel gereksinimlerine bağlıdır.

İnteraktif Sistemlerde Çizelgeleme

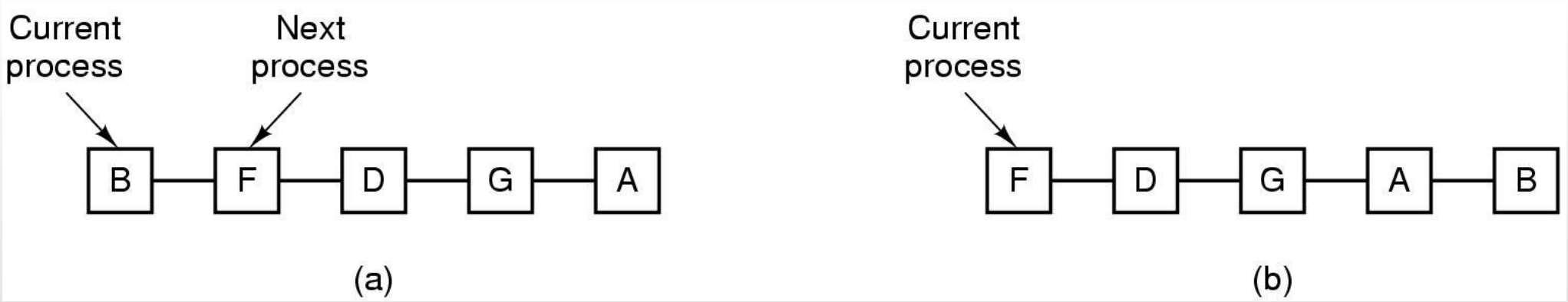
- Sıralı planlama (Round-robin scheduling)
- Öncelik zamanlaması (Priority scheduling)
- Çoklu kuyruk (Multiple queues)
- Sonraki en kısa süreç (Shortest process next)
- Garantili planlama (Guaranteed scheduling)
- Piyango planlaması (Lottery scheduling)
- Adil paylaşım planlaması (Fair-share scheduling)

Sıralı Planlama

- Süreçler döngüsel bir sırada yürütülür ve her iş sabit bir zaman dilimi alır.
- Basit, hem G/Ç'ye bağlı hem de CPU'ya bağlı süreçleri etkili bir şekilde yönetir.
- Küçük zaman dilimi durumunda yüksek ek yüke ve daha düşük yanıt süresine neden olabilir.

Sıralı Planlama

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



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 (context switch)

- 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 maliyet, düşük CPU kullanımı
  - Bağlam anahtarlama masrafına göre büyük olmalıdır

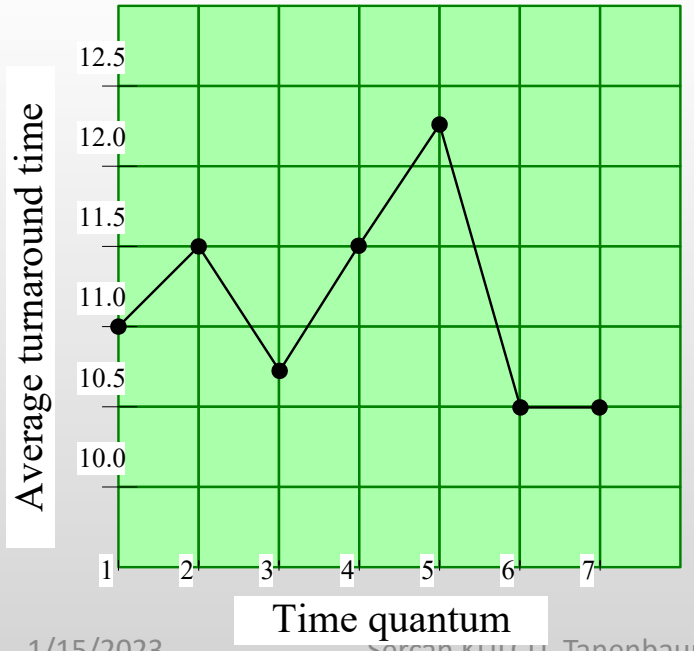
Bağlam Anahtarlama

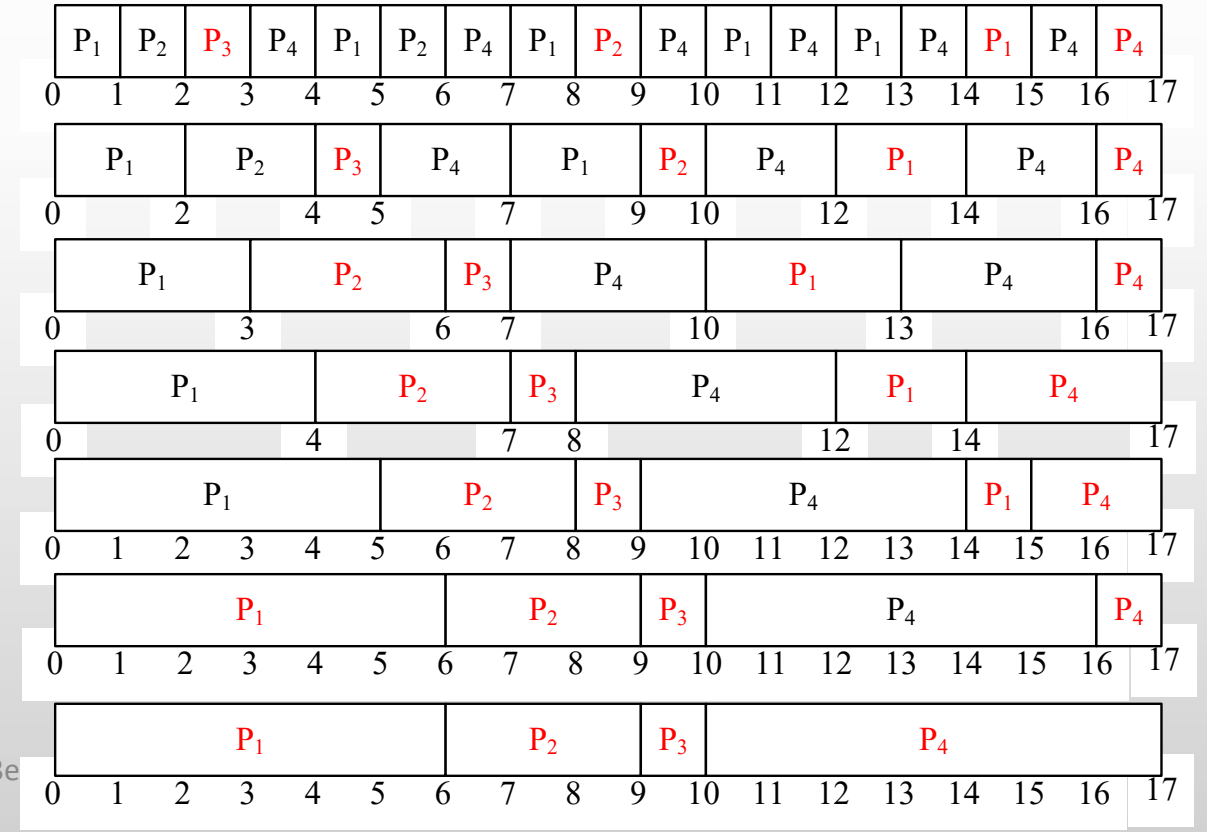
- Kuantum 4 milisaniye ve bağlam anahtarlama 1 milisaniye ise CPU zamanının %20'si boşa gider
- Kuantum 100 milisaniye ise, boşa harcanan zaman %1'dir, ancak daha az duyarlıdır
- 20-50 milisaniye civarında bir kuantum makul

Dönüş Süreleri

Process	Time
P_1	6
P_2	3
P_3	1
P_4	7

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



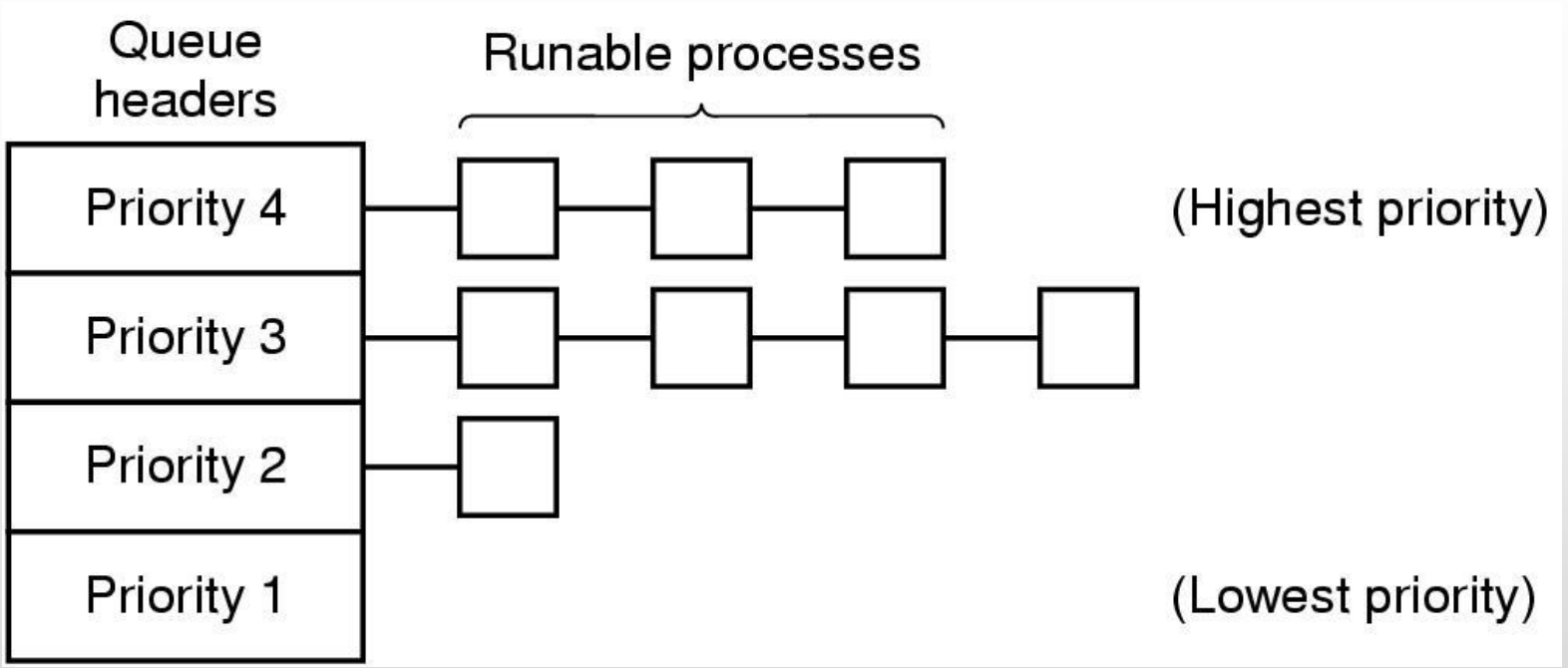


Öncelik Zamanlaması

- Süreçler önceliğe göre yürütülür, önce daha yüksek öncelikli işler yürütülür.
- Önemli süreçlere öncelik sağlar, genel sistem yanıt verebilirliğini artırır.
- Düşük öncelikli işler için açlığa (starvation) neden olabilir, bunu önlemek için yaşlanma (aging) mekanizmasına ihtiyaç vardır.

Öncelik Zamanlaması

- Her önceliğin bir öncelik numarası vardır
- 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)

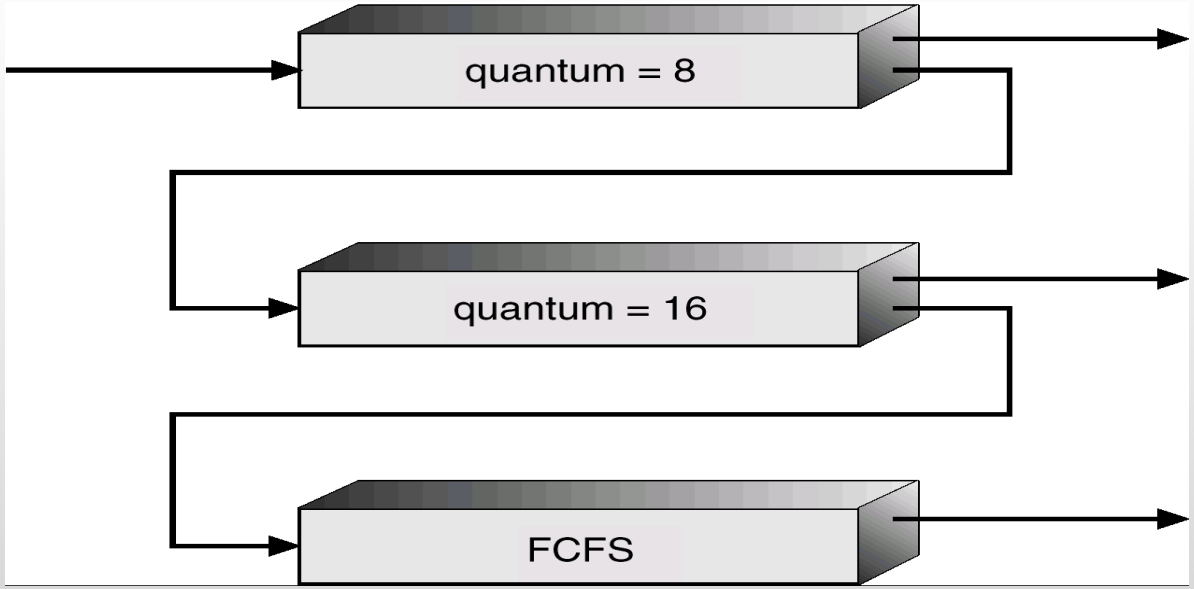
- Ortalama bekleme süresi = (6 + 0 + 16 + 18 + 1) /5 = 8,2

Çoklu Kuyruk

- Süreçler farklı öncelik sınıflarına ayrılır ve ayrı kuyruklara yerleştirilir.
- Önceliği ve eskimeyi yönetir, ek yükü azaltarak sistem performansını artırır.
- Uygulaması karmaşıktır, sık sıra değişiklikleri nedeniyle daha yüksek yanıt süresine neden olabilir.
- CPU'ya bağlı sürece büyük kuantum ve etkileşimli sürece küçük kuantum verilir.
- 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

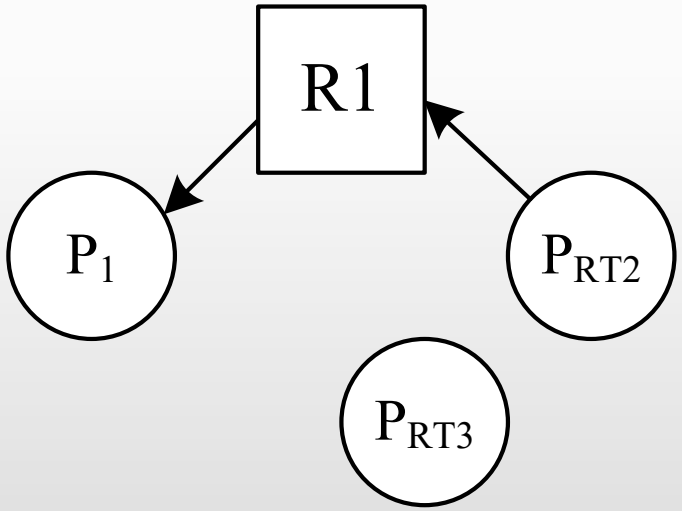
- Süreçler, bir sonraki sürecin ihtiyacı olan sürenin uzunluğuna göre yürütülür.
- CPU kullanımını en üst düzeye çıkarır ve ortalama bekleme süresini azaltır.
- İnteraktif sistemlerde bir sürecin kalan süresini tahmin etmek zordur.
- Geçmiş davranışa dayalı olarak tahminde bulunulabilir
- T0, T0 / 2 + T1 / 2, T0 / 4 + T1 / 4 + T2 / 2..
- Yaşlanma (aging):
- Geçerli ve önceki tahminin ağırlıklı ortalaması alınarak sonraki değer tahmin edilebilir

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



Garantili Çizelgeleme

- Önceden belirlenmiş bir programa göre her süreç için CPU zamanı tahsis eder.
- Gerçek zamanlı sistemler için uygun, öngörülebilir bir davranış sağlar.
- Sınırlı esneklik, öngörülemeyen iş yükleri durumunda düşük performansa neden olabilir.
- Katı gerçek zamanlıya karşı yumuşak gerçek zamanlı
  - Katı: Bir fabrikada robot kontrolü
  - Yumuşak: CD çalar
- Algoritmalar statik (çalışma sürelerini önceden bilinen) veya dinamik (çalışma zamanı kararları) olabilir.

Piyango Çizelgeleme

- Süreçlere bilet atanır ve yürütme için rastgele seçilir.
- İşlemci süresi için saniyede birkaç kez çekiliş yapılır
- Önceliği ve eskimeyi yönetir, ek yükü azaltır ve sistem performansını artırır.
- Uygulanması karmaşık, dengesiz bilet dağılımı durumunda öngörülemeyen davranışlara neden olabilir.
- "Daha önemli" süreçler için daha fazla çekiliş hakkı tanıyarak önceliklerin değiştirilebilmesine izin verir.

Adil Paylaşım Çizelgeleme

- Son kullanım geçmişlerine göre süreçlere CPU zamanı ayırır.
- Süreçler arasında kaynakları dengeler, genel sistem yanıt verebilirliğini artırır.
- Her sürecin son kullanım geçmişinin izlenmesini gerektirir, ek yüke neden olabilir.

Sonuç

- İşletim sisteminin özel gereksinimlerine bağlı olarak her algoritmanın kendi güçlü ve zayıf yönleri vardır.
- Round-Robin, Priority, Multiple Queues ve Fair-Share çizelgelemelerinin tümü önleyicidir ve CPU zaman paylaşımına izin verir.
- Shortest Process Next, Guaranteed, Lottery çizelgelemeleri öncelikli değildir ve CPU kullanımını en üst düzeye çıkarmayı amaçlar.
- Algoritma seçimi, işletim sisteminin yanıt verebilirlik, öngörülebilirlik, kaynak tahsisi ve adalet gibi hedeflerine bağlıdır.

SON