Process senkronizasyonu

- **Cooperating process'ler diğer process'leri etkilerler veya diğer process'lerden etkilenirler.**
- Cooperating process'ler **paylaşılmış hafıza alanıyla** veya **dosya sistemleri** ile veri paylaşımı yaparlar.
- **Paylaşılmış veriye eşzamanlı erişim tutarsızlık problemlerine yol açabilir.**
- **Paylaşılmış veri üzerinde işlem yapan process'ler arasında veriye erişimin yönetilmesi gereklidir.**
- Paylaşılan veriye erişim **üretici-tüketici (producer-consumer) problemi** olarak modellenebilir.

Process senkronizasyonu

- Üretici ve tüketici processler için örnek kod aşağıda verilmiştir.

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

Yeni eleman eklendi.

Bir eleman alındı.

- **`counter`** değişkeninin değeri buffer'a yeni eleman eklendiğinde artmakta, eleman alındığında azalmaktadır.

Process senkronizasyonu

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

- **İki örnek ayrı ayrı doğru olsa da eşzamanlı doğru çalışamayabilirler.**
- `counter=5` iken `counter++` ve `counter--` deyimlerinin aynı anda çalıştığını düşünelim.
- **Farklı zaman aralıklarında çalışmış olsalardı `counter=5` olacaktı.**

Process senkronizasyonu

- `counter++` için makine komutları aşağıdaki gibi olabilir.

$$\begin{aligned} register_1 &= \texttt{counter} \\ register_1 &= register_1 + 1 \\ \texttt{counter} &= register_1 \end{aligned}$$

- `counter--` için makine komutları aşağıdaki gibi olabilir.

$$\begin{aligned} register_2 &= \texttt{counter} \\ register_2 &= register_2 - 1 \\ \texttt{counter} &= register_2 \end{aligned}$$

- $register_1$ ve $register_2$ aynı (AC) veya farklı register olabilir.

Process senkronizasyonu

- `counter++` ve `counter--` için sıralı zaman aralıklarında yapılan mikroişlemler aşağıdaki gibi olabilir.

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

- Yukarıdaki sırada buffer'daki eleman sayısı **4 olarak görülür**, ancak **gerçekte buffer'daki eleman 5 tanedir.**
- T_4 ile T_5 yer değiştirirse buffer'daki **eleman sayısı 6 olarak görülecektir.**
- **İki process aynı anda `counter` değişkeni üzerinde işlem yaptığından sonuç yanlış olmaktadır.**

Process senkronizasyonu

- Aynı değişkene **çok sayıda process'in erişmesi** durumunda sonuç değer erişim sırasına bağlı olarak değişecektir (race condition).
- Paylaşılan bir değişkene **aynı anda sadece bir process'in erişimi** sağlanmak zorundadır (process synchronization).
- Günümüzde işletim sistemlerinin farklı kısımlarındaki **process'lerin aynı veriye erişiminde senkronizasyon yapılmak zorundadır.**
- **Multicore işlemcilerde çalışan multithread uygulamalarda da process senkronizasyonu yapılmak zorundadır.**

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

Kritik bölüm problemi

- Bir sistem, $\{P_0, P_1, \ldots, P_{n-1}\}$ şeklinde n tane process'e sahip olsun.
- **Her process,** ortak değişkenler, tablolar veya dosyalar üzerinde işlem yapan **kritik bölüme sahip olabilir.**
- Bir process kendi kritik bölümünü çalıştırırken diğer process'lerin kendi kritik bölümlerini çalıştırmamaları zorunludur.
- **Aynı anda iki process kritik bölümünü çalıştırmamalıdır.**
- Kullanılan protokoller ile **her process kritik bölümüne girmek için izin istemektedir.**

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

- Kritik bölümden sonra çıkış bölümü de yer alabilir.
- Örnekte, entry section giriş izni için kullanılır.

Kritik bölüm problemi

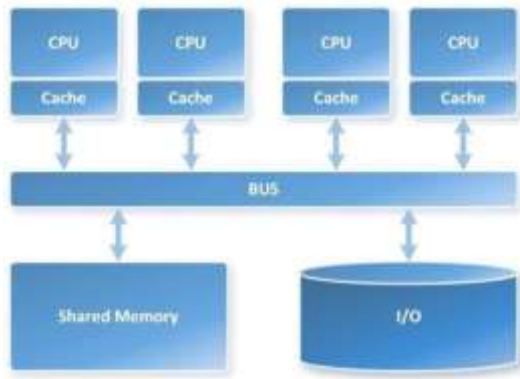
- Kritik bölüm probleminin çözümü aşağıdaki üç gereksinimi sağlamak zorundadır:
  - Mutual exclusion (karşılıklı dışlama): **Bir P_i process'i kritik bölümünü çalıştırıyorsa** diğer process'lerin hiç birisi kritik bölümlerini çalıştıramazlar.
  - Progress: **Hiçbir process kritik bölümünü çalıştırmıyorsa**, kritik bölüme girmek isteyenlerden (remainder section çalıştırmayanlar arasından) bir tanesinin kritik bölüme girmesine izin verilir.
  - Bounded waiting (sınırlı bekleme): **Bir process kritik bölüme giriş izni istedikten sonra ve izin verildikten önceki aralıkta,** kritik bölüme giriş izni verilen process sayısının sınır değeri vardır.
- İşletim sisteminde **açık durumdaki tüm dosya listesi için kernel veri yapısı oluşturulur.**
- İki process aynı anda dosya açma veya kapatma işlemi yaptığında bu listeye erişmeleri gerekir (race condition).

Kritik bölüm problemi

- **Hafıza tahsis edilmesi** ve **interrupt işlemleri** gibi işlemler race condition içeren örneklerdir.
- **Kritik bölüm yönetimi için** iki yaklaşım vardır:
  - Preemptive kernel: Bir process **kernel mode'da çalışırken sonsuz öncelikli** (preemptive) olabilir.
  - Nonpreemptive kernel: Bir process **kernel mode'da çalışırken sonsuz öncelikli olamaz**, onun **yerine bir kernel-function çalışır.**
- Nonpreemptive kernel, **kernel veri yapıları üzerinde race condition oluşturmaz**, aynı anda bir process kernel içinde aktif durumdadır.

Kritik bölüm problemi

- **SMP (Symmetric Multiprocessing)** mimarisinde (her işlemci eş düzey işlem kapasitesine sahiptir.) **preemptive kernel tasarımı daha zordur.**
- Birden fazla kernel mode process, aynı anda farklı işlemcilerde çalışabilir.



- Preemptive kernel'ın cevap verebilirliği (responsiveness) daha iyidir ve gerçek zamanlı uygulamalar için daha uygundur.

13

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

14

Peterson çözümü

- Peterson çözümü, **yazılım tabanlı** kritik bölüm çözümüdür.
- P_i ve P_j process'leri için kritik bölüm çözümü aşağıdaki gibidir.

Bu process kritik kesime hazır.

Diğer process kritik kesime hazırsa öncelik ona verilir.

Sıra kendisine gelene kadar bekler.

Kritik kesimden çıkış bildirimi (flag[i] = false).

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

- Peterson çözümü, P_i ve P_j process'leri için **iki veriyi paylaşarak kullanır.**

```
int turn;
boolean flag[2];
```

Peterson çözümü

- Eğer ***turn = i*** ise, kritik bölüme ***i*** process'i girecektir.
- ***flag[ ]*** bitleri ise process'lerin kritik bölüme girmeye hazır durumunu gösterir.
- **Eğer *flag[i] = true*** ise, **i.process kritik bölümüne girmeye hazırdır.**
- i.process ***flag[i] = true*** **VE** ***turn = i*** olunca kritik bölümüne girer.
- ***turn*** değişkenini iki process'te aynı anda değiştirse bile, son değer alınır ve o process kritik bölüme girer (mutual exclusion).
- **Kritik bölümü tamamlayan** process **kritik bölüme giriş isteğini iptal eder ve diğer process kritik bölüme girer** (progress).
- **Bir process kritik bölüme bir kez girdikten sonra sırayı diğerine aktarır** (bounded waiting).


```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

Senkronizasyon donanımı

- Kritik bölüm problemi için çok sayıda donanım ve yazılım tabanlı çözüm vardır. **Bunlar, temel olarak kilitleme (locking) tabanlı yaklaşımlardır.**
- **Tek işlemcili sistemlerde**, **interrupt'ların paylaşılmış veriye erişimi engellenirse**, kritik bölüm problemi basit bir şekilde çözülebilir.
- Bu sistemlerde, **komut sırası değiştirilmeden ve önceliklendirme yapmadan çalışma sağlanırsa kritik bölüm problemi ortaya çıkmaz.**
- Nonpreemptive kernel'ların sıklıkla kullandığı yaklaşımdır.
- **Bu yöntem çok işlemcili sistemlerde uygun çözüm değildir.**
- Çok işlemcili sistemlerde, **interrupt'ların disable/enable yapılması için tüm işlemcilere mesaj göndermek için zaman gereklidir** ve sistem performansı düşer.

Senkronizasyon donanımı

- Modern bilgisayar sistemlerinde, bir word içeriğini **test edip değiştirme** veya iki word içeriğini **yer değiştirme (swap)** işlemlerini **atomik** olarak yapan özel donanım komutları vardır.
- **Test et ve değiştir komutu** (atomik çalışır) ile **karşılıklı dışlama** (mutual exclusion) yapan P_i process'i aşağıda verilmiştir.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

pass by reference

Mevcut değeri alındı ve geri döndürüldü.

Değeri true yapıldı.

lock = true

Değeri başlangıçta false.

Senkronizasyon donanımı

- **Karşılaştır ve yer değiştir komutu** (compare_and_swap) ile **karşılıklı dışlama** (mutual exclusion) yapan P_i process'i aşağıda verilmiştir.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Değeri başlangıçta 0.

pass by reference

Mevcut değeri alındı ve geri döndürüldü.

Değeri değiştirildi.

lock = 1

Senkronizasyon donanımı

- **Önceki algoritmalar** karşılıklı dışlamayı sağlar, ancak **bounded-waiting gereksinimini sağlamaz**.
- **Alttaki algoritma (P_i için çalışır) ile bounded-waiting karşılanır.**
- **Her process en fazla (n-1) çalışma sonrasında sırayı alır.**

P_0

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

P_1

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

. . .

P_n

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

Senkronizasyon donanımı

```
boolean waiting[n];
boolean lock;
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

Döngüden çıkmasını sağlar.

Kendisinden sonraki bekleyen process dairesel dizide aranıyor.

Bekleyen yok

Bekleyen var

Pj kritik bölüme girer.

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

Mutex kilitlenmeleri

- **Donanım tabanlı** kritik bölüm çözümleri **karmaşıktır** ve **uygulama geliştirici tarafından erişilemez.**
- İşletim sistemi tasarımcıları, kritik bölüm problemi için yazılım araçları geliştirmişlerdir.
- En basit yazılım aracı mutex (**mut***ual* **ex***clusion*) lock aracıdır.
- Her process kritik bölüme girmek ve lock yapmak için izin ister **(acquire() function)**.
- Kritik bölümünden çıktıktan sonra da lock durumu sonlandırılır **(release() function)**.
- **Lock durumunun uygun olup olmadığına karar vermek için** bir **boolean değişken kullanılır (available)**.

Mutex kilitlenmeleri

- Aşağıda, acquire(), release() fonksiyonları ile mutex lock kullanılan **kritik bölüm çözümü verilmiştir.**

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

```
release() {
   available = true;
}
```

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

- Bir process, kritik bölümünde iken diğer tüm process'ler acquire() fonksiyonunda sürekli döngüdedirler (spinlock).

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

Semaforlar

- S semaforu bir tamsayıdır ve sadece wait() ve signal() **atomik** işlemleri tarafından erişilebilir.
- Literatürde **wait()** işlemi **P** ile, **signal()** işlemi ise **V** ile gösterilir.
- **wait()** ve **signal()** işlemleri aşağıda verilmiştir.

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

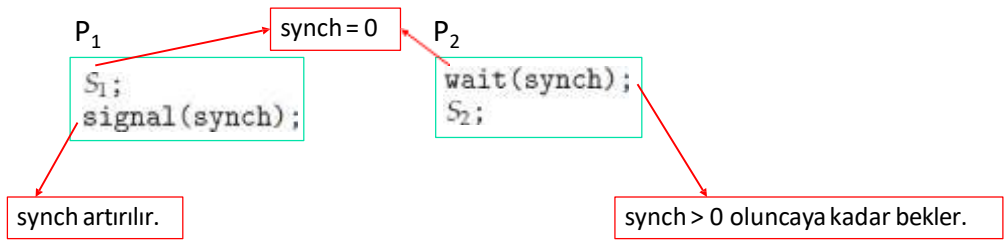
- **wait()** ile S'nin değeri azaltılır, **signal()** ile S'nin değeri artırılır.
- S üzerindeki wait() ve signal() işlemleri **kesintisiz (atomik) bir şekilde gerçekleştirilir.**

Semaforlar

- İşletim sistemleri, **sayan semafor (counting semaphore)** ve **ikilik semafor (binary semaphore)** kullanırlar.
- **Sayan semaforların değeri kısıtlı değildir.**
- **İkilik semaforların değeri 0 veya 1 olabilir.**
- İkilik semafor mutex lock gibi davranır.
- **Sayan semaforlar, belirli sayıdaki kaynağa erişimi denetlemek için kullanılır.** Sayan semafor kaynak sayısı ile başlatılır.
- Kaynağı kullanmak isteyen her **process semafor üzerinde wait() işlemi gerçekleştirir (sayaç azaltılır).**
- Bir **process kaynağı serbest bıraktığında ise signal() işlemi gerçekleştirir (sayaç artırılır).**
- **Semafor değeri = 0** olduğunda tüm kaynaklar kullanılır durumdadır.

Semaforlar

- Semaforlar, **işlem bağımlılığı gibi farklı senkronizasyon problemlerinde de kullanılabilir.**
- Örneğin, P_2 process'indeki S_2 deyimi P_1 process'indeki S_1 deyiminden sonra çalışmak zorunda olsun.
- Örnekte P_1 ve P_2 için paylaşılan synch semaforu tanımlanmıştır.
- **synch** semaforu **başlangıçta 0** değerine sahiptir.
- **P_1 ve P_2 içerisine eklenen deyimler aşağıda verilmiştir.**



29

Semaforlar

Semafor oluşturulması

- **Mutex lock** gibi **semafordaki wait() ve signal()** tanımları da **süresiz beklemeye neden olabilir.**
- Aşağıda örnek semafor tanımı verilmiştir:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- Her semafor bir tamsayı değeri ve process listesine sahiptir.
- **Bir process semaforu bekliyorsa process listesine eklenir.**
- Listeden bir process signal() ile alınır ve çalıştırılır.

30

Semaforlar

Semafor oluşturulması

- Semafor için wait() ve signal() işlemleri aşağıdaki gibi tanımlanabilir:
- **block()** process'i **beklemeye alır**, **wakeup()** ise çalışmaya **devam ettirir.**

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

Negatif değer büyüklüğü bekleyen process sayısını gösterir.

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Semaforlar

Semafor oluşturulması

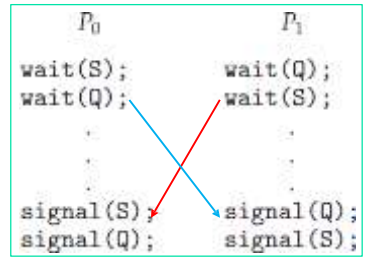
- Bekleyen process listesi, her process'in PCB (Process Control Block) linkiyle oluşturulabilir.

- **Her semafor bir tamsayı ile PCB pointer'ına sahiptir.**
- **Bounded waiting için FIFO kuyruk oluşturulur.**
- FIFO kuyruk yapısının dışında **priority queue** yapısı da oluşturulabilir.
- Semafor işlemlerinin atomik olarak çalıştırılması gereklidir.

Semaforlar

Kilitlenme (Deadlock)

- Bir process'in beklemesine bağlı olarak, iki veya daha çok process'in sonsuza kadar beklemesine **kilitlenme** (deadlock) denir.
- Aşağıdaki P_0 ve P_1 process'leri, **S** ve **Q** semaforlarına erişmektedir.



- P_0 wait(S) ve P_1 wait(Q) işlemlerini aynı anda çalıştırsın.
- P_0 wait(Q)'yu çalıştırırken, P_1 wait(S)'yi çalıştırır.
- P_0 wait(Q)'da, P_1 wait(S)'de kilitlenir.

33

Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

34

İzleyiciler

- Semaforlar kullanıldığında da senkronizasyon hataları olabilmektedir.
- Tüm process'lerin **kritik bölüme girmeden önce wait()**, **girdikten sonra ise signal() işlemlerini** yapmaları gereklidir.
- **Program geliştirici bu sıraya dikkat etmezse, iki veya daha fazla process aynı anda kritik bölüme girebilir.**
- Bu durumlar programcılar arasında yeterli işbirliği olmadığı durumlarda da olabilmektedir.
- Kritik bölüm problemine ilişkin tasarımda oluşan sorunlardan dolayı **kilitlenmeler** veya **eşzamanlı erişimden dolayı yanlış sonuçlar** ortaya çıkabilmektedir.

İzleyiciler

- **wait() ile signal() yer değişirse,** aşağıdaki çalışma ortaya çıkar.

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

- Örnekte **birden fazla process kritik bölüme aynı anda girebilir.**
- signal() yerine wait() yazılırsa aşağıdaki çalışma ortaya çıkar.

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

- Bu durumda da **deadlock oluşur.**
- wait() veya signal() unutulursa, karşılıklı dışlama yapılamaz veya deadloack oluşur.

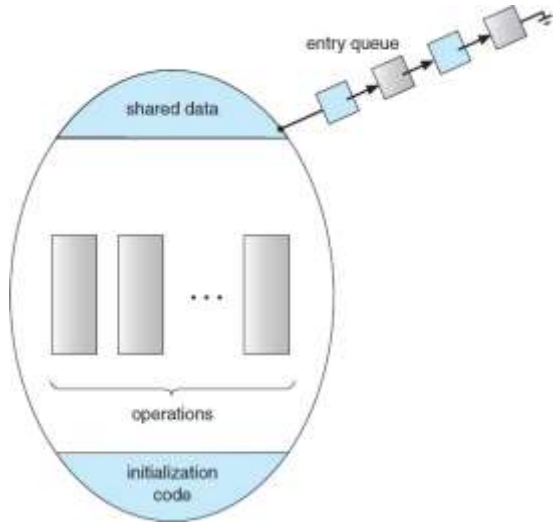
İzleyiciler

- Programcıdan kaynaklanabilecek bu hataların giderilmesi için **izleyici (monitor)** kullanılır.
- **Monitor içinde tanımlanan bir fonksiyon**, sadece **monitor içinde tanımlanan değişkenlere** ve **kendi parametrelerine erişebilir.**
- Monitor içindeki fonksiyonlardan sadece bir tanesi aynı anda aktif olabilir.

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

İzleyiciler

- Programcının senkronizasyon kısıtlarını yazması gerekli değildir.



İzleyiciler

- Programcı işe veya **değişkene özel senkronizasyon** oluşturmak için durum değişkenleri oluşturabilir.

```
condition x, y
```

- Sadece **bir durum değişkenine bağlı çalışan** wait() ve signal() işlemleri tanımlanabilir.

```
x.wait();
```

ile x durum değişkenine bağlı bir process beklemeye alınır.

```
x.signal();
```

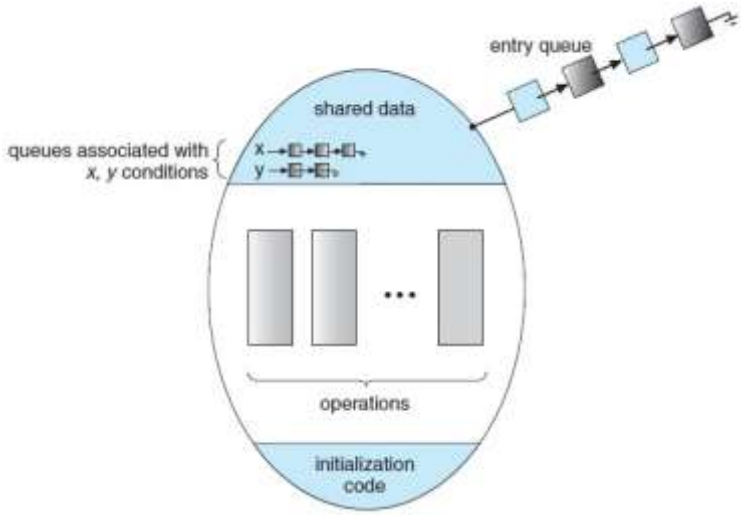
ile x durum değişkenine bağlı beklemekte olan bir process çalışmaya devam eder.

İzleyiciler

- **`x.signal()`** işlemi **bir `P` process'i başlatmış olsun**. Aynı anda, **`x`** durumuna bağlı **beklemekte olan bir `Q` process'i olsun.**
- **Q process'i** çalışmaya **başladığında**, **P process'i** tekrar işlem yapmak isterse **beklemek zorundadır.**
- Aksi durumda, monitör içindeki P ve Q aynı anda aktif olur.
- Bu durumda iki olasılık vardır:
  - Signal and wait: **P process'i**, **Q process'inin monitör'den ayrılmasını** veya başka bir duruma geçmesini **bekler.**
  - Signal and continue: Q process'i, **P process'inin monitör'den ayrılmasını** veya başka bir duruma geçmesini **bekler.**

İzleyiciler

- **x** ve **y** durum değişkenlerine bağlı process'lerin monitör içinde çalışması.



Konular

- Process senkronizasyonu
- Kritik bölüm problemi
- Peterson çözümü
- Senkronizasyon donanımı
- Mutex kilitlenmeleri
- Semaforlar
- İzleyiciler
- Alternatif yaklaşımlar

Alternatif yaklaşımlar

Transactional memory

- **Multicore sistemlerde**, **mutex lock**, **semafor** gibi mekanizmalarda **deadlock** gibi problemlerin **oluşma riski bulunmaktadır.**
- Bunun yanı sıra, **thread sayısı arttıkça deadlock problemlerinin ortaya çıkma olasılığı artmaktadır.**
- **Klasik mutex lock (veya semafor)** kullanılarak paylaşılmış veride güncelleme yapan **update()** fonksiyonu aşağıdaki gibi yazılabilir.

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```


Alternatif yaklaşımlar

Transactional memory

- Klasik kilitleme yöntemlerine alternatif olarak **programlama dillerine yeni özellikler eklenmiştir.**
- Örneğin, **atomic(S)** kullanılarak **S işlemlerinin tümünün transaction olarak gerçekleştirilmesi sağlanır.**

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

- **Lock işlemine gerek kalmadan** ve **kilitlenme olmadan işlem tamamlanır.**

Alternatif yaklaşımlar

OpenMP (Open Multi-Processing)

- OpenMP, C, C++ ve Fortran için compiler direktiflerinden oluşan API'dir.
- OpenMP, paylaşılmış hafızada eşzamanlı çalışmayı destekler.
- OpenMP, **`#pragma omp critical`** komutu ile kritik bölümü belirler ve aynı anda sadece bir thread çalışmasına izin verir.

```
void update(int value)
{
   #pragma omp critical
   {
      counter += value;
   }
}
```