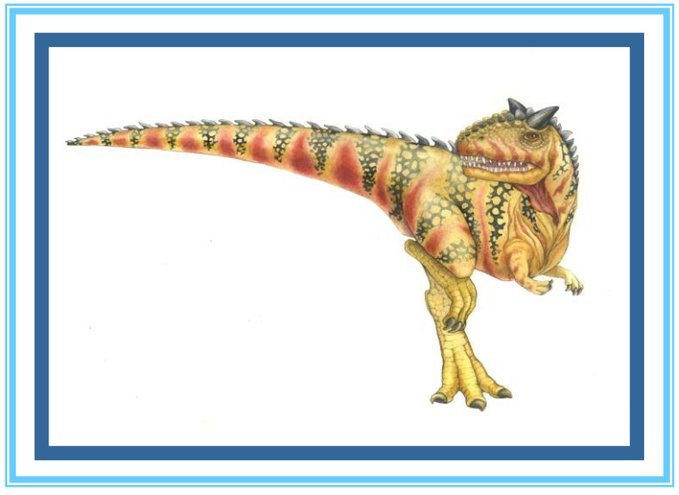
Bölüm 6: Proses Senkronizasyonu



Modül 6: Proses Senkronizasyonu

- Arka plan
- Kritik Bölge Problemi
- Peterson’un Çözümü
- Donanımsal Senkronizasyon
- MUTEX (Mutual Exclusion-Karşılıklı dışlama) kiliti
- Semaforlar
- Klasik Senkronizasyon Problemleri
- Monitörler
- Senkronizasyon Örnekleri
- Atomik İşlemler (Alternatif Yaklaşımlar)

Hedefler

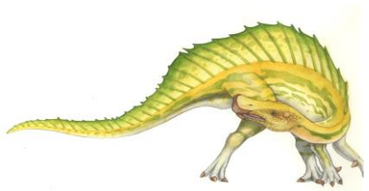
- Proses Senkronizasyonu kavramını sunmak
- Paylaşılan verinin tutarlılığını sağlamak için kullanılabilecek Kritik Bölge Problemi çözümlerini tanıtmak
- Kritik Bölge Problemlerine ilişkin yazılımsal ve donanımsal çözümleri sunmak
- Birkaç klasik proses senkronizasyonu problemlerini incelemek
- Atomik işlem kavramını tanıtmak ve atomiklik sağlama mekanizmasını açıklamak

Arka plan

- İşbirliği içindeki prosesler birbirlerini etkileyebilmektedir. Bu prosesler bir mantıksal bellek adresini paylaşabilir (Kod veya veri olabilir). İş parçacıklarının kullanımında karşımıza çıkabilecek bir durum.

- Paylaşılan verilere eşzamanlı erişim veya paralel olarak erişim, veri tutarsızlıklarına neden olabilir.

- Veri tutarlılığını korumak için işbirliği içindeki proseslerin düzenli yürütülmesini sağlayan bir mekanizma gerekir.



Arka plan

- Varsayalım ki, bir tampon kullanan üretici-tüketici problemine bir çözüm sağlamak istiyoruz.
- Bunun için tampon boyutunu tamsayı olarak bir sayaç değişkeninde tutabiliriz.
- Başlangıçta sayaca 0 değeri verilir.
- Üretici tarafından yeni bir ürün oluşturulduktan sonra sayaç değeri bir arttırılır ve tüketici bir ürünü kullandığında sayaç tüketici tarafından bir azaltılır.

Üretici

```
while (true) {

        /*  bir ürün üret ve birSonrakiÜrün değerine ata */
         while (sayaç== TAMPON_BOYUTU)
                ; // bekle
            tampon [in] = birSonrakiÜrün ;
            in = (in + 1) % TAMPON_BOYUTU;
            sayaç++;
    }
```

Tüketici

```
while (true)  {
        while (sayaç== 0)
            ; // bekle
            birSonrakiÜrün=  tampon [out];
             out = (out + 1) % TAMPON_BOYUTU;
              sayaç--;

             /*  birSonrakiÜrün tüketilir */
}
```

Yarış durumu

- sayaç++ şu şekilde uygulanabilir

  register1 = sayaç
  register1 = register1 + 1
  sayaç= register1

- sayaç-- şu şekilde uygulanabilir

  register2 = sayaç
  register2 = register2 - 1
  sayaç= register2

- Başlangıçta “sayaç = 5” iken aşağıdaki çalışma sırasını ele alalım:

  S0: üretici register1 = sayaç satırını çalıştırır {register1 = 5}
  S1: üretici register1 = register1 + 1 satırını çalıştırır {register1 = 6}
  S2: tüketici register2 = sayaç satırını çalıştırır {register2 = 5}
  S3: tüketici register2 = register2 - 1 satırını çalıştırır {register2 = 4}
  S4: üretici counter = register1 satırını çalıştırır {sayaç = 6 }
  S5: tüketici counter = register2 satırını çalıştırır {sayaç = 4}

Yarış durumu

- Bu hatalı sonuç, Counter (Sayaç) değişkenine eşzamanlı erişime izin verdiğimiz için oluşmuştur.

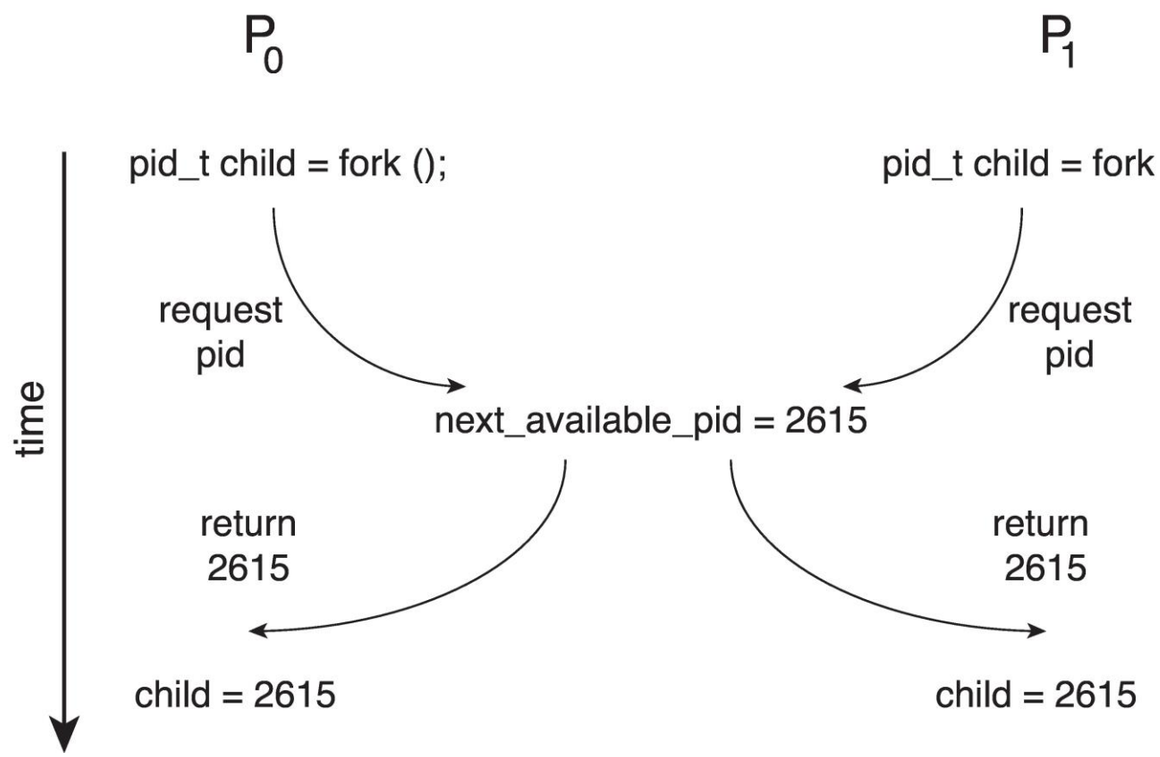
- Eğer birkaç proses aynı veriye eş zamanlı olarak erişebildiği ve değiştirebildiği durumlarda yürütmenin sonucu bu prosesler arasındaki sıraya göre gerçekleşir ve bu duruma **yarış durumu** denir.

- Buna çözüm bulmak için aynı anda sadece bir prosesin değişkene erişimi sağlanmalı.

- Özellikle de günümüzde çok çekirdekli sistemlerin geliştirilmesi ile paralel programcılıkta önemli bir konu haline gelmiştir.

Race Condition

- Proses P_0 ve P_1 `fork()` sistem çağrısı ile çocuk prosesler üretiyorlar
- `next_available_pid` değişkeni ile sıradaki uygun olan proses kimliği (pid) ile çekirdekteki yarış durumu ifade ediliyor



- Karşılıklı dışlama olmadığı sürece, aynı pid iki farklı prosese atanabilirdi!

Kritik Bölge Problemi

- n adet prosesi $\{p_0, p_1, \ldots p_{n-1}\}$ ele alalım.
- Her process içindeki kod **kritik bölge** segmentine sahiptir.
  - Proses, ortak değişkenlerine ulaşıyor, tablo güncelliyor, dosyaya yazıyor v.b. olabilir.
  - Bir proses kritik bölgede olduğunda başka bir proses o kritik bölgeye giremez.
- Kritik bölge problemini çözmek için bir protokol gerekmektedir.
- Her process kritik bölgeye girmek için izin istemelidir- **giriş bölgesi**
- Kritik bölgeden çıkana kadar kalır - **çıkış bölgesi**
- Daha sonra geri kalan işlemlerini sürdürebilir - **kalan bölge**
- Özellikle kesintili işlemlerde uygulanır

Kritik Bölge

- p_i prosesinin genel yapısı :

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

P_i prosesi için Algoritma

```
do {

        while (turn == j);

            critical section

        turn = j;

            remainder section

} while (true);
```

Kritik Bölge Probleminin Çözümü

1. **Karşılıklı Dışlama (Mutual Exclusion)** – Eğer P_i prosesi kendi kritik bölgesinde çalışıyorsa,başka hiçbir proses onun kritik bölgesine giremez.

2. **İlerleme (Progress)** - Eğer kritik bölgede hiçbir proses çalışmıyorsa ve kritik bölgeye girmek isteyen proses var ise mutlaka birisi seçilip ilerletilmelidir.

3. **Sınırlandırılmış Bekleme (Bounded Waiting)** – Kritik bölgeye erişmek için bir proses beklerken diğer proseslerin kritik bölgeye girme sayısının bir sınırı vardır.
   - Her bir proses sıfırdan farklı bir hızla çalışsın.
   - n adet prosesin **göreceli hızına** ilişkin bir varsayım yoktur.

Kritik Bölge Probleminin Çözümü

- Çekirdek modunda kodların uygulanmasında olası bir sürü yarış durumu oluşacaktır.
- Bu durumda Çekirdek modundaki proseslerin çalışmasını düşünelim:
- Örnek olarak, bir çekirdek veri yapısının (kod parçası) sistemdeki açık dosyaların listesini tuttuğunu varsayalım. Herhangi bir dosya açıldığı veya kapatıldığı zaman bu listenin modifiye edilmesi gerekir.
- İki proses eş-zamanlı olarak bir dosya açmak isterse burada bir yarış durumu söz konusu olacaktır.
- Buradaki yarış durumlarının çözümü çekirdek geliştiricilerine göre farklı yaklaşımlar vardır;
  - Kesintili çekirdek
  - Kesintisiz çekirdek

Kritik Bölge Probleminin Çözümü

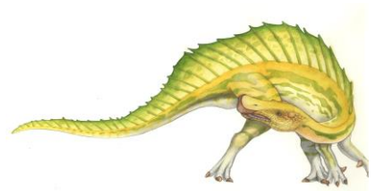
- Kesintisiz çekirdekte prosesler CPU'da işlemlerini gerçekleştirirken herhangi bir kesintiye uğramadan sonuna kadar devam ettirirler. Bir proses işini bitirdikten sonra CPU'dan ayrılır. Buda aslında bu durumda yarış durumunun engellenmesi demektir.

- Kesintili çekirdekte prosesler CPU'da çalışırken herhangi bir kesme durumunda, daha öncelikli bir iş geldiğinde veya daha kısa bir iş geldiğinde kesintiye uğrayabilmektedir. Bunun sonucunda yarış durumları oluşabilmektedir.

- Buna rağmen günümüzdeki gerçek-zamanlı programlama için kesintili çekirdek yaklaşımı daha az yanıt süresine sahip olduğu için tercih edilmektedir. Fakat yarış durumuna çözüm olabilecek yöntemleri içermesi de gerekmektedir.

Peterson Çözümü

- İki proses çözümü
- LOAD ve STORE makine-dili komutlarının atomik (kesilemez) olduğunu varsayalım.
- İki proses iki değişken paylaşsın:
  - int **turn**;
  - boolean **flag[2]**
- **turn** değişkeni kritik bölgeye girme sırasının kimde olduğunu gösterir.
- **flag** dizisi, bir prosesin kritik bölgeye girmek için hazır olup olmadığını belirtmek için kullanılır.
- **flag [i]** = true ise P_i prosesi hazır demektir



P_i Prosesi Algoritması

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[ j ] && turn == j);
            kritik bölge
    flag[i] = FALSE;
            kalan bölge
} while (TRUE);
```

- Aşağıdaki şartlar sağlanmıştır:
  1. Karşılıklı dışlama korunur (başka proses kritik bölgeye giremez). Sadece P_i girebilir aksi durumda `flag[j] = false` veya `turn = i` olabilirdi.
  2. İlerleme gereksinimi sağlanır. (kritik bölgeye girmek isteyen prosesler işletim sist. tarafından seçilip ilerlemelidir. Turn değişkeni ile)
  3. Sınırlı bekleme gereksinimi karşılanabilmektedir

     (`flag[i] = false ile`)

Peterson’s Solution

- Çözüm, algoritmanın gösterilmesi için yararlı olsa da, Peterson’ın Çözümünün modern mimariler üzerinde çalışma garantisi yoktur.
- Neden işe yaramayacağını anlamak, yarış durumlarını daha iyi anlamak için de faydalıdır.
- Performansı artırmak için, işlemciler ve / veya derleyicileri bağımlılığı olmayan işlemleri yeniden sıralayabilir.
- Tek iş parçacıklı için, sonuç her zaman aynı olacak gibi.
- Çok iş parçacıklıda, yeniden sıralama tutarsız veya beklenmedik sonuçlar doğurabilir!

Peterson’s Solution

- İki iş parçacığı veriyi paylaşmaktadır:

```
boolean flag = false;
int x = 0;
```

- Thread 1

```
while (!flag)
   ;
print x
```

- Thread 2

```
x = 100;
flag = true
```

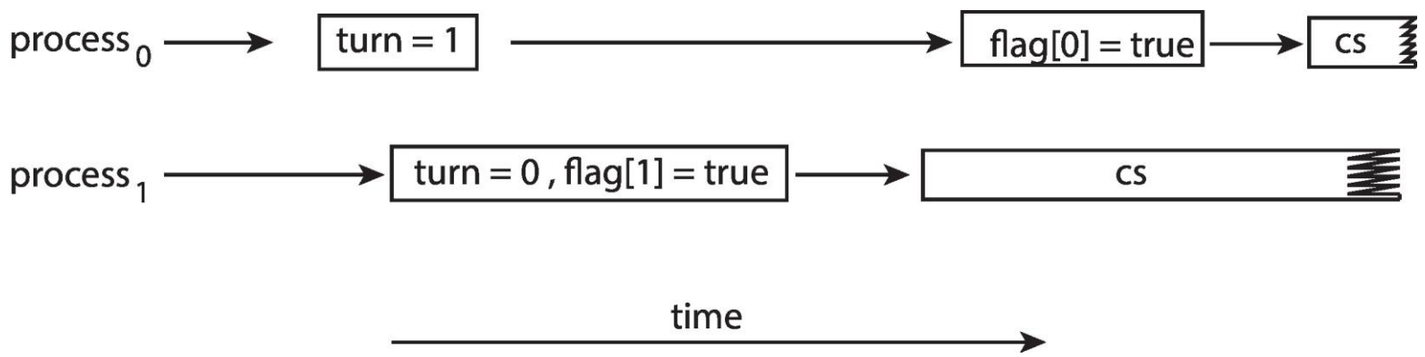
- Beklenen çıktı ne olur ?

Peterson’s Solution

- 100 beklenen çıktı
- Fakat, Thread 2 için gerekli işlemler yeniden sıralanabilir:

```
flag = true;
x = 100;
```

- Eğer bu gerçekleşirse çıktı 0 olabilir!
- Peterson'ın Çözümünde komutların yeniden düzenlenmesinin etkileri



- Bu, her iki prosesinde kritik bölümde aynı anda olmasını sağlar!

Donanım Senkronizasyonu

- Bir çok sistem kritik bölge kodu için donanımsal destek sağlar.

- Tek işlemcili sistemler – kesmeler devre dışı bırakılabilir.
  - Yürütülmekte olan kodu kesinti olmadan çalıştırır.
  - Çok işlemcili sistemlerde genelde çok verimsizdir.
    - Bunu kullanan işletim sistemleri ölçeklenebilir değildir.

- Üç çeşit donanım desteğine bakacağız:
  - 1. Bellek bariyerleri
  - 2. Donanım komutları
  - 3. Atomik değişkenler

Bellek Bariyeri

- Bellek modeli (**Memory model)** bilgisayar mimarisinin uygulama programlarına yaptığı bir garanti niteliğindedir.
- Bellek modelleri aşağıdakilerden biri olabilir:

  - **(Güçlü sıralı) Strongly ordered** – bir işlemcinin bellek modifikasyonu tüm diğer işlemciler için hemen görünür.
  - **(Zayıf sıralı) Weakly ordered** – bir işlemcinin bellek modifikasyonu tüm diğer işlemciler için hemen görünmeyebilir.

- Bir bellek bariyeri (**memory barrier**), bellekteki herhangi bir değişikliği diğer tüm işlemcilere yayılacak (görünür hale getirecek) zorlayan bir komuttur.

Bellek Bariyeri

- Thread 1 çıktısının 100 olması için aşağıdaki komutlara bir bellek bariyeri ekleyebiliriz:
- Thread 1 şimdi çalışıyor

```
while (!flag)
   memory_barrier();
print x
```

- Thread 2 şimdi çalışıyor

```
x = 100;
memory_barrier();
flag = true
```

Hardware Instructions

- Bir verinin içeriğini test etmemize ve değiştirmemize (*test-and-modify*) ya da iki verinin içeriğini atomik olarak değiştirmemize olanak veren özel donanım komutları (kesintisiz olarak.)
- **Test-and-Set** komutları
- **Compare-and-Swap** komutları

TestAndSet Komutu

- Tanım:

```
boolean TestAndSet (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv:
 }
```

1. Atomik olarak yürütülür
2. Aktarılan parametrenin orijinal değerini döndürür (Returns the original value of passed parameter)
3. Aktarılan parametrenin yeni değeri "TRUE" olarak ayarlanır.

TestAndSet Kullanılarak Çözüm

- Eğer TestandSet komutu eş-zamanlı olarak iki proses (farklı CPU'lardan) tarafından çalıştırılırsa sırayla çalıştırılırlar. Birisi çalıştırırken kilitler. İşlem bittikten sonra kilit açılır.

- Paylaşılan boolean değişken kilit(lock), FALSE olarak başlatılır.
- Çözüm:

```
do {
        while ( TestAndSet (&kilit ))     ;   // bekle
                //    kritik bölge
        kilit = FALSE;
                //      geri kalan kısım
} while (TRUE);
```

Takas(Swap) Komutu

- Tanım:

```
int compare_and_swap (int *value, int expected, int new value) {
    int temp = *value;
        if (*value == expected)
        *value = new value;
    return temp;
}
```

1. Atomik olarak yürütülür
2. Aktarılan parametrenin“value” orijinal değerini döndürür
3. Eğer “value” ==“expected” koşulu sağlanırsa “value” değişkenine aktarılacak yeni değer atanır . Yani takas yalnızca bu koşullar altında gerçekleşir.

Takas Kullanılarak Çözüm

- Paylaşılan Boolean değişken «lock» 0 ile başlatılır;

- Çözüm:

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */
    /* critical section */
    lock = 0;
    /* remainder section */
} while (true);
```

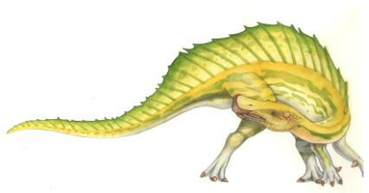
Bounded-waiting Mutual Exclusion with compare-and-swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Bounded-waiting Mutual Exclusion with test_and_set

```
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);
```

- Bu veri yapıları false olarak başlatılır.
- Pi prosesi sadece waiting[i] == false veya key == false olunca kritik bölgeye erişebilir.
- Key değeri test and set() fonk. Yürütüldüğünde false olur. Diğerleri bekler
- Sınırlı bekleyen şartının yerine getirildiğini ispatlamak için, bir işlem kritik bölümden çıktığında, döngüsel olarak sıralamada bekleyen diziyi tarar ($i + 1, i + 2, \ldots, n - 1, 0, \ldots, i - 1$).



Atomic Variables

- Tipik olarak, compare-and-swap gibi komutlar diğer senkronizasyon araçları için yapı taşları olarak kullanılır.
- Bir araç olan **atomic variable (atomik değişkenler)**, tam sayı ve boolean gibi temel veri türlerinde atomik (kesintisiz) güncellemeler sağlar.
- Örneğin, atomik değişken olan **`sequence`** üzerindeki **`increment()`** işlemi, **`sequence'in`** kesintisiz olarak arttırılmasını sağlar:

```
increment(&sequence);
```

Atomic Variables

- `increment()` fonksiyonu aşağıdaki gibi uygulanabilir;

```
void increment(atomic_int *v)
{
   int temp;

   do {
         temp = *v;
   }
   while (temp != (compare_and_swap(v,temp,temp+1));
}
```

MUTEX kilidi

- Donanımsal çözümler karmaşıktır ve genellikle uygulama programcıları tarafından erişilememektedir.
- İşletim Sistemi tasarımcıları, kritik bölge problemini çözmek için yazılım araçları oluşturmuşlardır
- En basiti mutex kiliti
- Kritik bir bölümü ilk olarak bir kiliti acquire(),elde edin ve daha sonra release(), serbest bırakın.
  - Kilidin mevcut olup olmadığını gösteren Boolean değişken **`available`**
- **`acquire()`** ve **`release()`** atomik olmalı
- Genellikle donanımsal atomik komutlarla uygulanır (compare-and-swap gibi)
- Ancak bu çözüm meşgul bekletmeyi (**busy waiting**) gerektirir.
  - Bu kilite **spinlock (dönen-kilit)** denir. Çünkü;
  - Bir proses kritik bölgedeyken, kritik bölgeye girmeye çalışan başka herhangi bir proses acquire () çağrısında sürekli döngüde kalır.(Bekletilir)

acquire() ve release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
 }
release() {
    available = true;
 }
do {
 acquire lock
    critical section
 release lock
   remainder section
} while (true);
```

- Bu iki işlev atomik olarak uygulanmalıdır.
- bu fonksiyonları uygulamak için her iki test-and-set ve compare-and-swap da kullanılabilir.

Semafor

- Mutex kilidine benzer fakat daha sağlam yapıya sahip ve biraz daha karmaşık senkronizasyon aracıdır.
- *S* Semaforu – bir tamsayı değişkenidir
- S üzerinde iki atomik fonksiyon değişiklik yapabilir : wait() ve signal()
  - P()- ***proberen, test için*** ve V()- ***verhogen artım için***, biçiminde adlandırılırlar.
- Sadece iki bölünmez (atomik) işlem üzerinden erişilebilir.
  - wait (S) {
    while S <= 0
    ; // bekleme
    S--;
    }
  - signal (S) {
    S++;
    }

Genel Senkronizasyon Aracı Olarak Semafor

- **Sayma (Counting)** semaforu – tamsayı değeri sınırsız bir etki alanı içinde değişebilir.
- **İkili (Binary)** semafor – Tamsayı değeri yalnızca 0 ve 1 olarak değişebilir; uygulaması daha basit olabilir.
  - **mutex kilidi** ile benzer.
- Birçok senkronizasyon problemini çözebiliriz;
- Karşılıklı dışlama sağlamak için ikili semaforlar kullanılabilir.
- Sayma semaforları, sonlu sayıda örnekten oluşan belirli bir kaynağa erişimi kontrol etmek için kullanılabilir.
- Semafor mevcut kaynakların sayısına göre başlatılır.
- Kaynak kullanmak isteyen her proses, semafor üzerinde bir wait () işlemi gerçekleştirir.
- Bir proses bir kaynağı bıraktığında, bir signal () işlemi gerçekleştirir
- Semafor için sayım 0'a gittiğinde, tüm kaynaklar kullanılmıştır. Bundan sonra, bir kaynak kullanmak isteyen prosesler sayım 0'dan büyük oluncaya kadar engellenir.

Genel Senkronizasyon Aracı Olarak Semafor

- Karşılıklı dışlama şartını sağlar.

```
Semaphore mutex;    //  1 e kurulur
do {
      wait (mutex);
         // Krtik Bölge
      signal (mutex);
               // geri kalan bölge
} while (TRUE);
```

- Birçok probleme de çözüm olabilir. P1 prosesin S1 işlemi bittikten sonra P2 prosesinin S2 işlemini yapmasını istiyoruz. Çözüm;(synch paylaşılmış semafor ve 0 dan başlatılır)

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Semafor Uygulanması

- Herhangi iki prosesin aynı anda, aynı semafor üzerinde wait() ve signal() fonksiyonlarını çalıştıramayacağı garanti altına alınmalıdır.
- Yoksa, uygulama wait ve signal kodunun kritik bölüme yerleştirildiği kritik bölüm problemi haline gelir
  - Şimdi kritik bölge uygulamasında **busy waiting (meşgul bekleme)** olabilir.
    - Fakat uygulama kodu kısadır.
    - Kritik bölge nadiren meşgul olursa, kısa süreli meşgul beklemeler olur.
- Uygulamalar kritik bölgelerde çok fazla zaman harcayabilir bu nedenle bunun iyi bir çözüm olmadığını unutmayın.

Semaphore Implementation with no Busy waiting

- Her semafor ile ilgili bekleme sırası vardır
- Bekleme kuyruğundaki her girdinin iki veri öğesi vardır:
  - value (tamsayı türünde)
  - pointer (listedeki bir sonraki kaydın işaretçisi)
- İki işlem:
  - **block** –kuyrukta işlemi bekleyen uygun prosesi yerleştir
  - **wakeup** – bekleme kuyruğundaki proseslerden birini uyandırın ve hazır kuyruğa yerleştirin
- ```
  typedef struct{
  int value;
  struct process *list;
  } semaphore;
  ```


  ```

Meşgul Beklemesiz Semafor Uygulanması

- Bekleme uygulaması :

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

- Sinyal uygulaması :

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

Monitörler (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.

Semaforların Sorunları

- Semafor işlemlerinin yanlış kullanımı:
  - signal (mutex) …. Kritik bölge…. wait (mutex)
  - Örnekte **birden fazla process kritik bölüme aynı anda girebilir.**
  - wait (mutex) … wait (mutex)
  - Bu durumda da **deadlock oluşur.**
  - wait (mutex) ya da signal (mutex) (ya da her ikisi de) unutulursa karşılıklı dışlama yapılamaz veya deadlock oluşur.
  - Bu tür hataların tespit edilmesi tespit edilmesi kolay olmayabilmektedir (Her zaman gerçekleşmeyebilir)
- Deadlock (Kilitlenme) ve starvation(açlık) olabiliyor

Monitörler

- Programcıdan kaynaklanabilecek bu hataların giderilmesi için **(monitör)** kullanılır.
- Senkronizasyonda doğru programların yazılmasını kolaylaştırmak adına Brinch Hansen (1973) ve Hoare (1974) yüksek seviyeli senkronizasyon yapıları (soyutlama) geliştirmişlerdir.
- **Monitör içinde tanımlanan bir fonksiyon**, sadece **monitor içinde tanımlanan değişkenlere** ve **kendi parametrelerine erişebilir.**
- Bir monitörde tanımlanan bir fonksiyon, yalnızca monitörde yerel olarak bildirilen değişkenlere (private) sadece public (fonksiyonlar) içindeki kod tarafından erişilebilirler.
- Monitör yapısı, aynı anda yalnızca bir işlemin monitör içinde etkin olmasını sağlar. Sonuç olarak, programcının bu senkronizasyon kısıtını açıkça kodlaması gerekmez. (Mutex kendiliğinden oluşur)

Monitörler

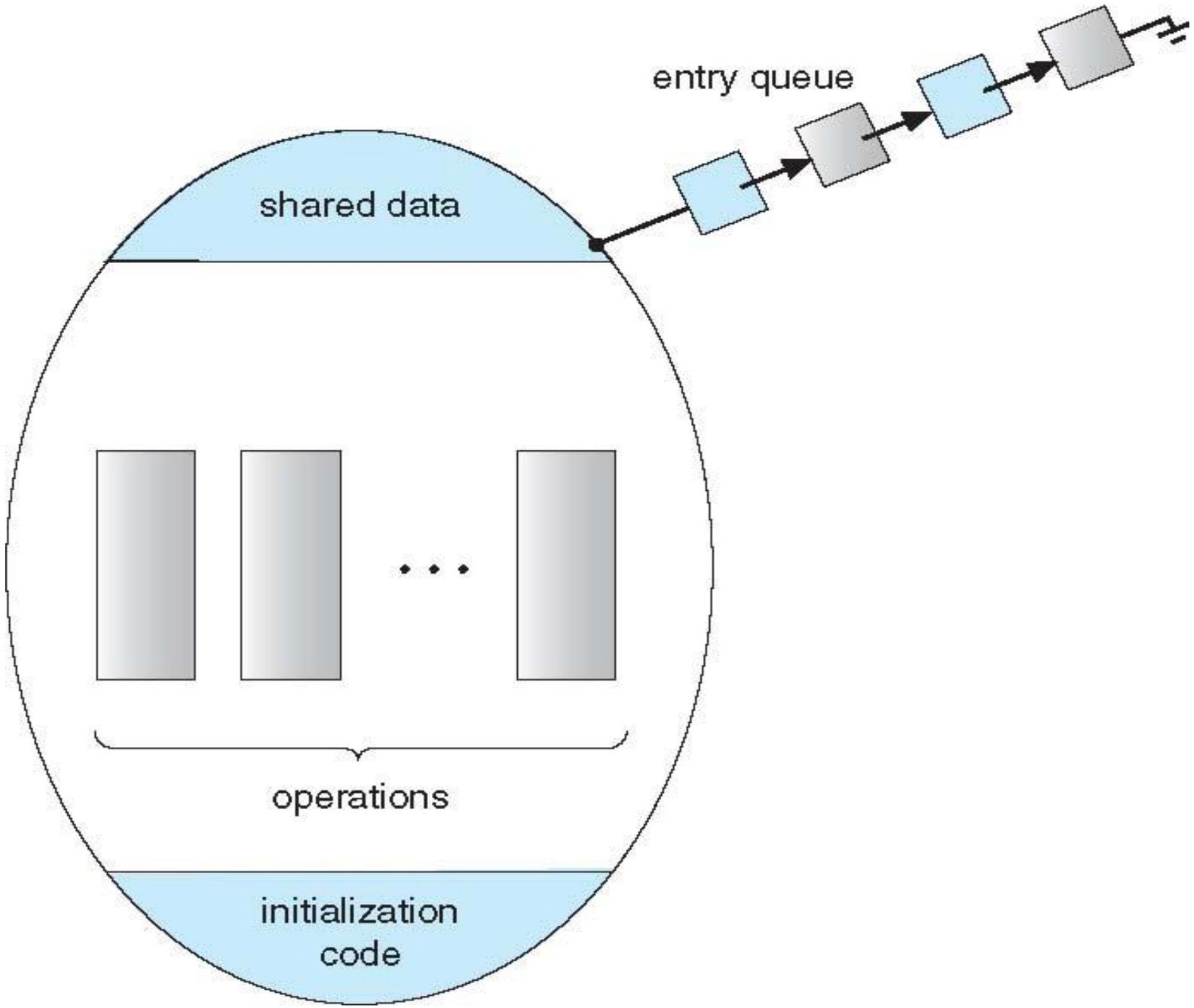
- Monitör programları paylaşılan nesneye ulaşmada meydana gelebilecek problemleri ortadan kaldırmaya yönelik geliştirilmiştir.

  - Paylaşılan nesneyi oluşturan veri,
  - Bu nesneye ulaşmak için kullanılan bir grup procedure (fonksiyonlar),
  - Nesneyi başlangıç konumuna getiren bir program parçasından

  oluşmaktadır.

```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }
  function P2 (…) { …. }
  function Pn (…) {……}
   initialization code (…) { … }
}
```

Monitor'ün Şematik Görünümü



Durum Değişkenleri

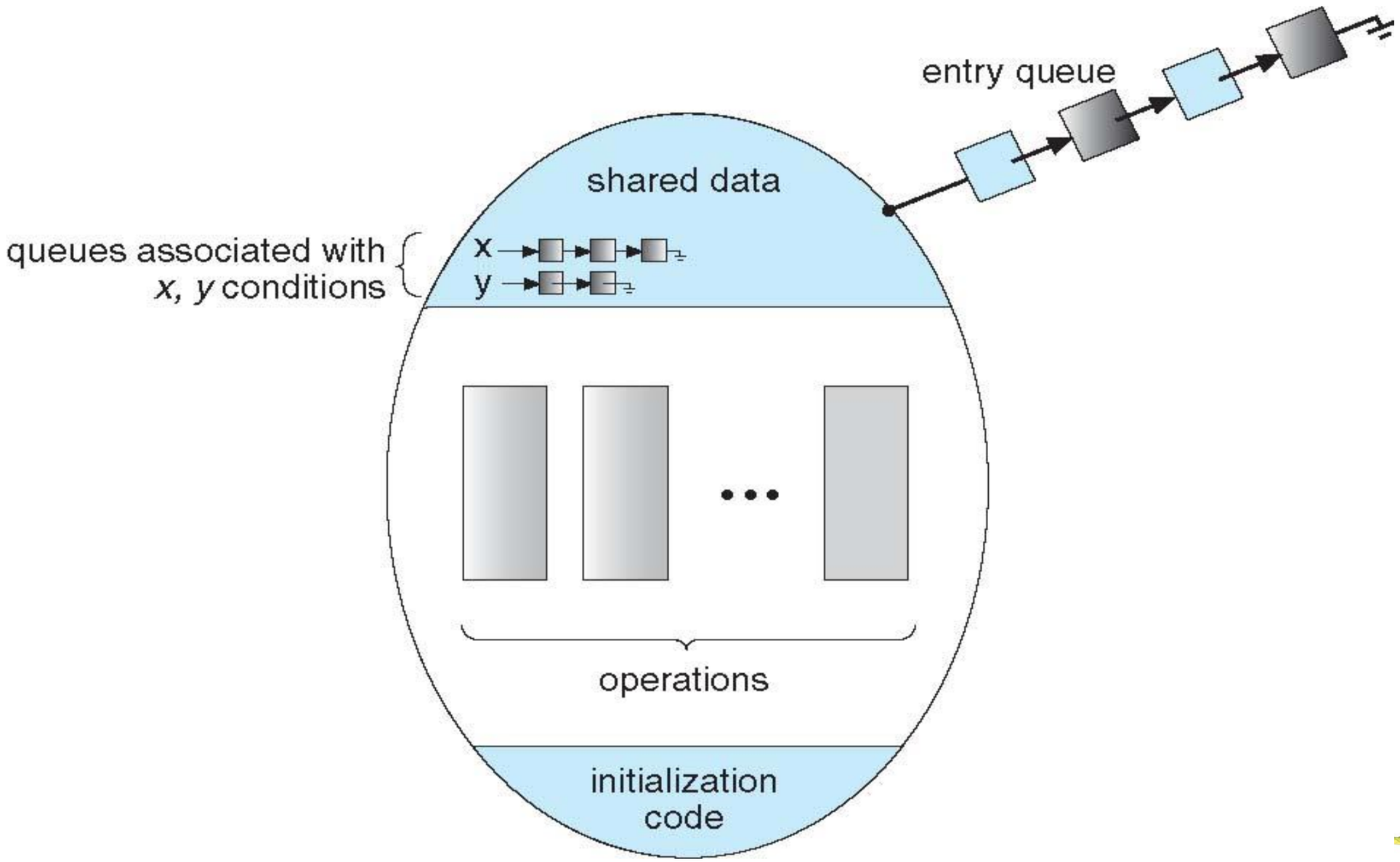
- Fakat bazı senkronizasyon şemalarını modellemek için yeterince güçlü değildir.
- Bu amaçla, ek senkronizasyon mekanizmaları tanımlamamız gerekir. Bu mekanizmalar durum (condition) yapısı tarafından sağlanır.
- Programcı işe veya **değişkene özel senkronizasyon** oluşturmak için durum değişkenleri oluşturabilir.

- `condition x, y;//x ve y durumları`

- Durum değişkenleri üzerinde iki işlem yapılabilir:
  - x.wait() –işlemi çağıran bir proses x.signal () işlevine kadar askıya alınır.
  - x.signal() – (varsa) x.wait () ile beklemeye alınmış proses lerden biri devam ettirilir.
    - x.wait () ile beklemeye alınmış değişken yoksa, değişken üzerinde hiçbir etkisi yoktur.

Durum Değişkenleri İle Monitör

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



Durum Değişkenleri Seçimleri

- **bir P process'i x.signal()** işlemini **çağırmış olsun**. Aynı anda, **x** durumuna bağlı **x.wait()** ile **beklemekte olan bir Q process'i de olsun.** , daha sonra ne olmalı?
- Monitör içindeki P ve Q aynı anda aktif olamaz.
- **Q process'i** çalışmaya **başlarsa**, **P process'i beklemek zorundadır.**

- Seçenekler şunlardır :
- **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.**
- **İkisininde artıları eksileri vardır.**
- Birçok programlama dili, Java ve C # de dahil olmak üzere bu bölümde anlatıldığı şekilde monitör fikrinde birleşmişlerdir.

Semafor Kullanarak Monitör Uygulama

- Değişkenler

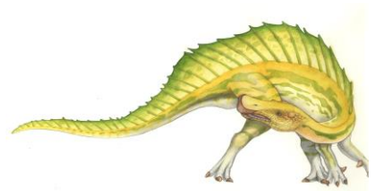
```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Her prosedür ***F*** bu şekilde değiştirilecektir.

```
wait(mutex);
    …
        body of F;

    …
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```

- Monitörün içinde mutual exclusion (karşılıklı dışlama) sağlanır.



Monitor Uygulama – Durum Değişkenleri

- Her koşul değişkeni ***x*** için şunlara sahibiz :

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- x.wait() işlemi şu şekilde uygulanabilir :

```
x_count++;
if (next_count > 0)
      signal(next);
else
      signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Uygulama (Devam)

- x.signal() işlemi şu şekilde uygulanabilir :

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Resuming Processes within a Monitor

- Birden fazla proses x koşul değişkeninde kuyruğa alınmışsa ve **`x.signal()`** çalıştırılırsa, hangi süreç devam ettirilmelidir?
- FCFS çoğu zaman yeterli değil
- **conditional-wait (koşullu bekleme),** **`x.wait(c)`** formu yapısı
  - **`c`** öncelik sayıdır (**priority number**)
  - En düşük sayı (en yüksek öncelikli) proses sıradaki olur

Single Resource allocation

- Bir işlemin kaynağı kullanmayı planladığı maksimum adeti/süreyi belirten öncelik numaraları kullanarak rakip işlemler arasında tek bir kaynak tahsis edilir.

```
R.acquire(t);
    ...
  access the resource;
    ...
R.release;
```

- R, **`ResourceAllocator`** türünde bir örnektir.

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
           if (busy)
              x.wait(time);
           busy = true;
   }
   void release() {
           busy = FALSE;
           x.signal();
   }
    initialization code() {
     busy = false;
   }
}
```

Liveness/Canlılık

- Muteks kiliti veya semafor gibi bir senkronizasyon aracı elde etmeye çalışırken prosesler süresiz beklemek zorunda kalabilir.
- Bekleme süresi, bu bölümün başında tartışılan ilerlemeyi ve sınırlı bekleme kriterlerini ihlal eder.
- **Liveness / Canlılık**, bir prosesin ilerlemesini sağlamak için bir sistemin yerine getirmesi gereken bir dizi özellik anlamına gelir.
- Belirsiz bekleme, bir canlılığın yitirilmesi de denebilir.

Kilitlenme ve Açlık

- **Deadlock (Kilitlenme)**–iki yada daha çok prosesin belirsiz bir süre beklemesidir,
- S ve Q iki semafor ve 1 ‘ den başlatılmış

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- P0'un wait (S) ve ardından P1 wait (Q) işlemini gerçekleştirdiğini varsayalım. P0 wait (Q) uyguladığında, P1'in signal (Q) çalıştırana kadar beklemesi gerekir. Benzer şekilde, P1 wait (S) uyguladığında, P0'un signal (S) çalıştırana kadar beklemesi gerekir. Bu signal () işlemleri gerçekleştirilemediğinden, P0 ve P1 kilitlendi.

- **Starvation (Açlık)** – Belirsiz engelleme
  - Askıya alınmış bir process asla semafor kuyruğundan silinmez.
- **Priority Inversion (Öncelik Değişimi)** –Yüksek öncelikli bir prosesin ihtiyaç duyduğu bir kaynağı daha düşük öncelikli bir process kilitli tutuyorsa planlama problemi meydana gelir.
  - **priority-inheritance protocol (Önceliğin Kalıtımı Protokolü)** aracılığıyla çözülmüştür.

Priority Inheritance Protocol

- Üç proses P1, P2 ve P3 ile bir senaryo düşünün. P1 en yüksek önceliğe, bir sonraki en yüksek P2'ye ve en düşük P3 değerine sahiptir. P1'in istediği bir R kaynağının P3'e atandığını düşünelim
- Böylece P1, P3'ün kaynaktaki işlemini bitirmesini beklemek zorundadır. Bununla birlikte, P2 çalıştırılabilir hale gelirse ve P3'ü önleyebilir. Olan şey P1'den daha düşük önceliğe sahip bir işlem olan P2'nin dolaylı olarak P3'ün kaynağa erişmesini engellemesidir.
- Bunun gerçekleşmesini önlemek için öncelikli bir kalıtım protokolü (**priority inheritance protocol**)kullanılır. Bu, şu anda kaynağı kullanan iş parçacığına atanacak bir paylaşılan kaynağa erişmeyi bekleyen en yüksek iş parçacığının önceliğine izin verir. Böylece, kaynağın mevcut sahibi, kaynağı elde etmek isteyen en yüksek öncelikli iş parçacığının önceliğine atanır.

Senkronizasyonun Klasik Problemleri

- Yeni önerilen senkronizasyon planlarını sınamak için kullanılan klasik problemler.
  - Bounded-Buffer Problem (Sınırlı Tampon Problemi)
  - Readers and Writers Problem (Okuyucu ve Yazıcı Problemi)
  - Dining-Philosophers Problem (Yemek Yiyen Filozoflar Problemi)

Bounded-Buffer Problem

- ***n*** buffer, herbiri bir item tutabilir
- Semaphore **`mutex`** 1 değerinden başlatılır
- Semaphore **`full`** 0 değerinden başlatılır
- Semaphore **`empty`** n değerinden başlatılır

Sınırlı Buffer (Tampon) Problemi

```
public class BoundedBuffer<E> implements Buffer<E>
{
private static final int BUFFER SIZE = 5;
private E[] buffer;
private int in, out;
private Semaphore mutex;
private Semaphore empty;  // boş bufferların sayısı
private Semaphore full;  //dolu bufferların sayısı
public BoundedBuffer() {
// buffer is initially empty
in = 0; out = 0;
mutex = new Semaphore(1);
empty = new Semaphore(BUFFER SIZE);
full = new Semaphore(0);
buffer = (E[]) new Object[BUFFER SIZE];
}
public void insert(E item) {
// Figure X }
public E remove() {
// Figure Y }
}
```

Sınırlı Buffer Problemi (Devam)

- Üretici process'in yapısı (X The insert() method)

```
public void insert(E item) {
wait (empty);
wait (mutex);
// add an item to the buffer
buffer[in] = item;
in = (in + 1) % BUFFER SIZE;
signal (mutex);
signal (full);
}
```

Sınırlı Buffer Problemi (Devam)

- Tüketici process'in yapısı (Figure Y The remove() method)

```
public E remove() {
E item;
wait (full);
wait (mutex);
// remove an item from the buffer
item = buffer[out];
out = (out + 1) % BUFFER SIZE;
 signal (mutex);
signal (empty);
return item;
}
```

Bounded-buffer producer/Ek

```
import java.util.Date;

public class Producer implements Runnable
{
   private Buffer<Date> buffer;

   public Producer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // produce an item & enter it into the buffer
         message = new Date();
         buffer.insert(message);
      }
   }
}
```

Bounded-buffer consumer / Ek

```
import java.util.Date;

public class Consumer implements Runnable
{
   private Buffer<Date> buffer;

   public Consumer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // consume an item from the buffer
         message = (Date)buffer.remove();
      }
   }
}
```

Bounded-buffer factory /Ek

```
import java.util.Date;

public class Factory
{
    public static void main(String args[]) {
       Buffer<Date> buffer = new BoundedBuffer<Date>();

       // Create the producer and consumer threads
       Thread producer = new Thread(new Producer(buffer));
       Thread consumer = new Thread(new Consumer(buffer));

       producer.start();
       consumer.start();
    }
}
```

Okuyucular-Yazıcılar Problemi

- Bir veri tabanı eşzamanlı processler dizisi arasında paylaşılmıştır.
  - Okuyucular – veri tabanı yalnızca okuyabilirler; herhangi bir güncelleme yapmazlar.
  - Yazıcılar – hem okuyabilir hem yazabilir.
- Problem –
  - Aynı anda birden fazla okuyucuya izin verilir.
  - Tek bir anda yalnızca tek bir yazıcı paylaşılmış veriye ulaşabilir. Başka yazıcı veya okuyucu erişemez
- Birkaç varyasyon ile okuyucular ve yazıcılar işlem görür
- Paylaşılmış veri
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore rw_mutex initialized to 1
  - Integer readcount initialized to 0

Okuyucular-Yazıcılar Probleminin Çeşitleri

- *Birinci* varyasyon – bir yazar zaten paylaşılan nesneyi kullanma iznini almadıkça hiçbir okuyucunun bekletilmemesini gerektirir.

- *İkinci* varyasyon – İkinci okuyucu-yazar sorunu, bir kez bir yazar hazır olduğunda, bu yazar en kısa sürede yazmasını gerektirir. Başka bir deyişle, bir yazar nesneye erişmeyi bekliyorsa, yeni okuyucu okumaya başlayamaz.

- İlk durumda, yazarlar açlıktan ölebilir; İkinci durumda okurlar açlıktan ölebilir.

Okuyucular-Yazıcılar Problemi (Devam)

- Ayrıca, bir yazar signal(rw_mutex) çalıştırdığında, bekleyen okuyucuların veya bekleyen tek bir yazıcının yürütülmesine yeniden başlayabiliriz.
- Seçim çizelgeleyici tarafından yapılır.
- Yazıcı process yapısı

```
do {
        wait (rw_mutex) ;
                //    writing is performed
        signal (rw_mutex) ;
} while (TRUE);
```

Okuyucular-Yazıcılar Problemi (Devam)

- Okuyucu process'in yapısı

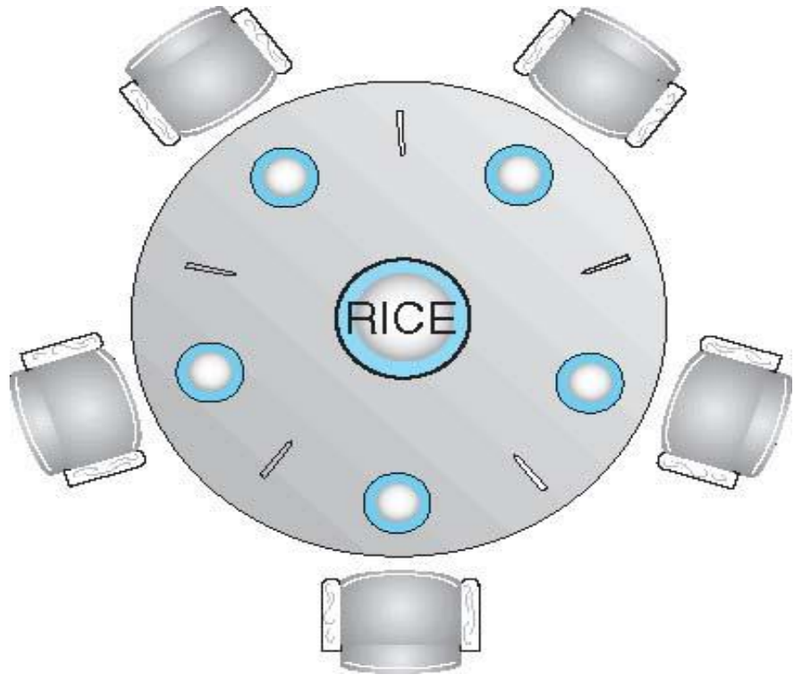
```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  //ilk okuyucu için (yazıcı yoksa giriş yap)
            wait (rw_mutex) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
            signal (rw_mutex) ;// son okuyucu için (yazıcılar giriş yapabilir)
        signal (mutex) ;
} while (TRUE);
```

Yemek Yiyen Filozoflar Problemi

- Filozoflar yaşamlarını yemek yiyerek ve düşünerek geçirirler.
- Komşularıyla etkileşime geçmeden, arasıra kasesindeki yemeği yemek için 2 yemek çubuğunu almaya çalışıyor (her seferinde bir tane)
  - Yemek için ikisine de ihtiyaç duyar, işini bitirdiğinde ikisini de serbest bırakır.
- 5 filozof durumunda
  - Paylaşılmış veri
    - Bir kase pirinç (veri seti)
    - Basit bir çözüm, her bir çubuğu semafor ile temsil etmektir. Semafor chopstick [5] 1 ile başlatılır.
  - Wait() ile çubuk alınır signal() ile bırakılır



Yemek Yiyen Filozoflar Problem Alg.

- Filozof *i* nin yapısı:

```
Semaphore chopStick[] = new Semaphore[5];
for(int i = 0; i < 5; i++)
chopStick[i] = new Semaphore(1);
do  {
        wait ( chopstick[i] );
          wait ( chopStick[ (i + 1) % 5] );
                    //  eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
                  //  think
} while (TRUE);
```

- Bu algoritmadaki problem nedir?
- Bu çözüm, hiçbir komşunun aynı anda yemediğini garanti etmesine rağmen, yine de bir ölümcül kilitlenme oluşturabileceği için reddedilmelidir.
- Beş filozofun hepsinin aynı anda acıktığını ve her birinin sol çubuğunu kaptığını varsayalım. Çubuk elemanları artık 0'a eşit olacak. Her filozof sağa doğru çubuğu tutmaya çalışınca, sonsuza dek bekleyecekler.

Yemek Yiyen Filozoflar Problem Algoritması

- Kilitlenme sorununun çeşitli olası çözümleri aşağıda dır:
- En fazla 4 filozofun masada aynı anda oturmasına izin verin.
- Bir filozofun çatalları sadece ikisi de mevcutsa almasına izin verin (toplama, kritik bölgede yapılmalıdır)
- Asimetrik bir çözüm kullanın - tek sayıdaki bir filozof önce sol çubuğu ve ardından sağdaki çubuğu alır. Çift numaralı filozof önce sağ çubuğu ve sonra da sol çubuğu alır.

Yemek Yiyen Filozofların Çözümü (Monitor)

```
 monitor DiningPhilosophers
{
enum State {THINKING, HUNGRY, EATING};
State[] states = new State[5]; Condition[] self = new Condition[5];

     public void pickup (int i) {
state[i] = State.HUNGRY;
test(i);
if (state[i] != State.EATING)
self[i].wait;
}
     public void putdown(int i) {
state[i] = State.THINKING;
// test left and right neighbors
test((i + 4) % 5);
test((i + 1) % 5);
}
```

Yemek Yiyen Filozofların Çözümü (Devam)

```
private void test (int i) {
if ( (state[(i + 4) % 5] != State.EATING) && (state[i] == State.HUNGRY) &&
(state[(i + 1) % 5] != State.EATING) ) {
state[i] = State.EATING;
self[i].signal;
}
}

public DiningPhilosophers {
for (int i = 0; i < 5; i++)
state[i] = State.THINKING;
}

}
```

Yemek Yiyen Filozofların Çözümü (Devam)

- Her filozof aşağıdaki sırayla pickup() ve putdown() fonksiyonlarını çağırır :

```
DiningPhilosophers.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

- Deadlock (kilitlenme) olmaz fakat starvation (açlık) mümkün.

Senkronizasyon Örnekleri

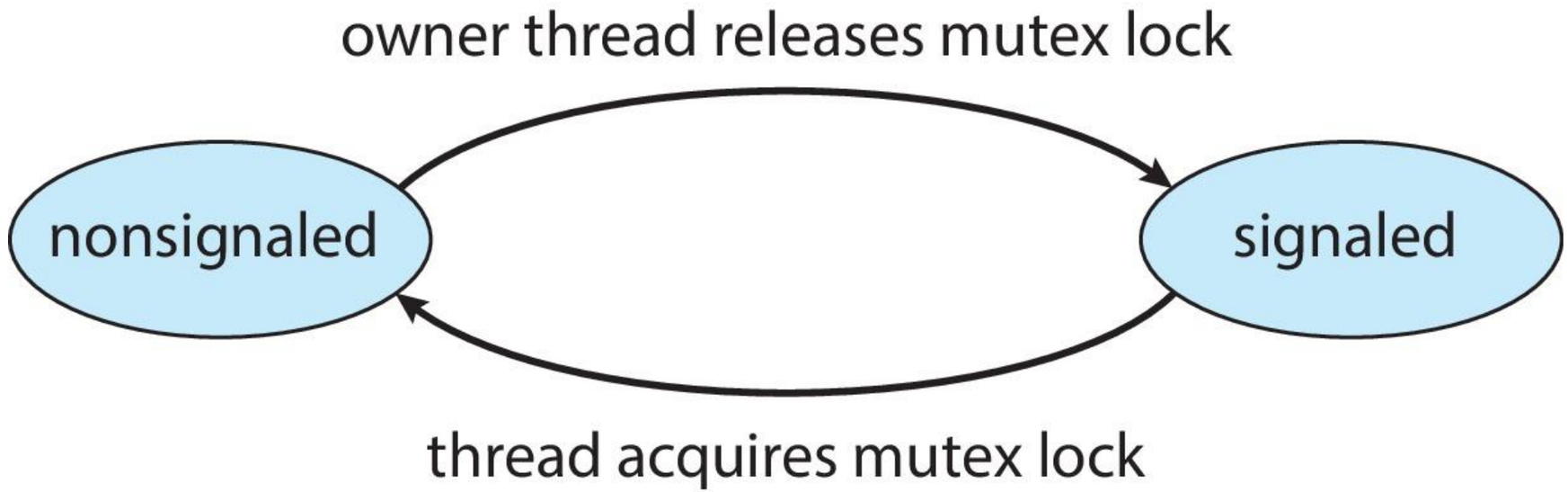
- Solaris
- Windows
- Linux
- Pthreads

Windows Senkronizasyon

- Tek işlemcili sistemlerde global kaynaklara güvenli erişimi korumak için interrupt masks (kesme maskeleri) kullanır.
- Birden fazla işlemcili sistemlerde **spinlocks** kullanır.
  - Spinlocking-thread asla preempted (kesintili) olmamalıdır.
- Ayrıca kullanıcı tarafında muteksler, semaforlar, olaylar ve zamanlayıcılar gibi davranan **dispatcher objects (dağıtıcı nesneler)** sağlar.
  - **Events (olaylar)**
    - Bir olay daha çok bir durum değişkeni gibi davranır.
  - Timers (zamanlayıcılar) süre aşıldığında (time expired) bir ya da daha fazla thread a bildirir.
  - Dispatcher nesneleri ya sinyal-edilmiş durumunda (nesne mevcut) veya sinyal-edilmemiş durumda (thread bloke edecek)

Kernel Synchronization - Windows

- Mutex dispatcher object



Solaris Senkronizasyon

- Multitasking, multithreading (gerçek zamanlı threadler de dahil olmak üzere) ve multiprocessing desteklemek için çeşitli kilitler uygular.(uyarlanabilir muteksler (adaptive mutex locks), durum değişkenleri, semaforlar, okuyucu-yazıcı kilitleri , ve turnstiles)

- Kısa kod bölümlerinde verimlilik sağlamak için **adaptive mutexes (uyarlanabilir muteksler)** kullanır.
  - standart semafor spin-lock gibi başlar.
  - Eğer kilit tutulursa, diğer bir CPU'da çalışan thread spin atar.
  - Eğer kilit çalışma durumunda olmayan bir thread tarafından tutulursa , bloke etmiş olur ve kilidin serbest bırakılma sinyali beklenirken diğer thread ler uyur. Bu şekilde boşa spin atmazlar

- **condition variables (durum değişkenleri)** kullanılır

Solaris Senkronizasyon

- Uzun bölümlerde kod veriye erişim ihtiyacı duyduğunda **okuyucu-yazıcı kilitleri** kullanır. Okuyucu-yazar kilitleri, sık erişilen verilerin korunması için kullanılır, ancak genellikle salt okunur (read-only) bir şekilde erişilir.

- Uyumlu muteks veya okuyucu-yazar kilidi elde etmeyi bekleyen iş parçacıklarının listesini düzenlemek için **turnikeler** (**turnstiles**) kullanır.
  - Bir turnike, bir kilit üzerinde bloke edilen iş parçacıklarını içeren bir sıra yapısındadır.
- Turnikelerde, daha düşük öncelikli bir iş parçacığı şu anda daha yüksek öncelikli bir iş parçacığının engellediği bir kilidi bulundurursa, daha düşük öncelikli iş parçacığı, daha yüksek öncelikli iş parçacığının önceliğini geçici olarak kalıtım alır. Kilit serbest bırakıldıktan sonra iş parç. orijinal önceliğine geri dönecektir.

Linux Senkronizasyon

- Linux:
  - Kernel 2.6 versiyonundan önceki versiyonlarda, kritik bölgeleri tamamlamak için kesmeler devre-dışı bırakılır.
  - Versiyon 2.6 ve sonrası, tamamen kesintili
- Linux şunları sağlar :
  - semaforlar
  - spinlocks
  - Hem okuyucu-yazıcı sürümleri
  - Atomik tam sayılar
- Tek CPU'lu sistemlerde, spinlock'lar, kesintili-çekirdek özelliğini etkinleştirme ve devre dışı bırakma ile değiştirildi

Linux Synchronization

- Atomik değişkenler

  `atomic_t` bir Atomik değişken tipi

- Değişkenleri ele alalım;

```
atomic_t counter;
int value;
```

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Pthreads Senkronizasyon

- Pthreads API, OS-independent (işletim sisteminden bağımsız)'dır.
- Şunları destekler:
  - muteks kilitleri
  - Koşul/durum değişkenleri
- Taşınabilir olmayan uzantılar şunları içerir:
  - okuma-yazma kilitler
  - spinlocks

POSIX Mutex Locks

- Kilit oluşturma ve başlatma

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Kiliti elde etme ve serbest bırakma

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX, adlandırılmış ve adlandırılmamış iki sürüm sağlar.(**named** and **unnamed**)
- Adlandırılmış semaforlar, bağımsız prosesler tarafından kullanılabilir, adsız semaforlar kullanılamaz.

```
sem_t *sem_open(const char *name, int oflag, mode_t mode,
unsigned int value);
```

POSIX Named Semaphores

- Semafor oluşturma ve başlatma:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Başka bir proses **`SEM`** semaforuna **`referans vererek`** erişebilir.
- Semaforun alınması ve serbest bırakılması::

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

- Semafor oluşturma ve başlatma :

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Semaforun alınması ve serbest bırakılması:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

- POSIX tipik olarak C / C ++ 'da kullanılır ve bu dillerin bir monitör desteği sağlamadığından, POSIX koşul değişkenleri, karşılıklı dışlama sağlamak için bir POSIX mutex kilidi ile ilişkilendirilir:
- Koşul değişkeninin oluşturulması ve başlatılması:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX Condition Variables

- Thread **`a == b`** koşulunun true olmasını bekliyor:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Durum değişkeni üzerinde bekleyen başka bir iş parçacığı bildiren iş parçacığı :

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Java Synchronization

- Java, dil seviyesinde senkronizasyon sağlar.
- Her bir Java nesnesinin ilişkili bir kilidi vardır.
- Bu kilit, senkronize edilmiş bir metot (synchronized method) çağırarak elde edilir.
- Senkronize edilmiş metotdan (synchronized method.) çıkılırken bu kilit serbest bırakılır.
- Nesne kilidini elde etmek için bekleyen iş parçacıkları, nesne kilidinin giriş kümesine yerleştirilir.

Bounded Buffer – Java Synchronization

```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

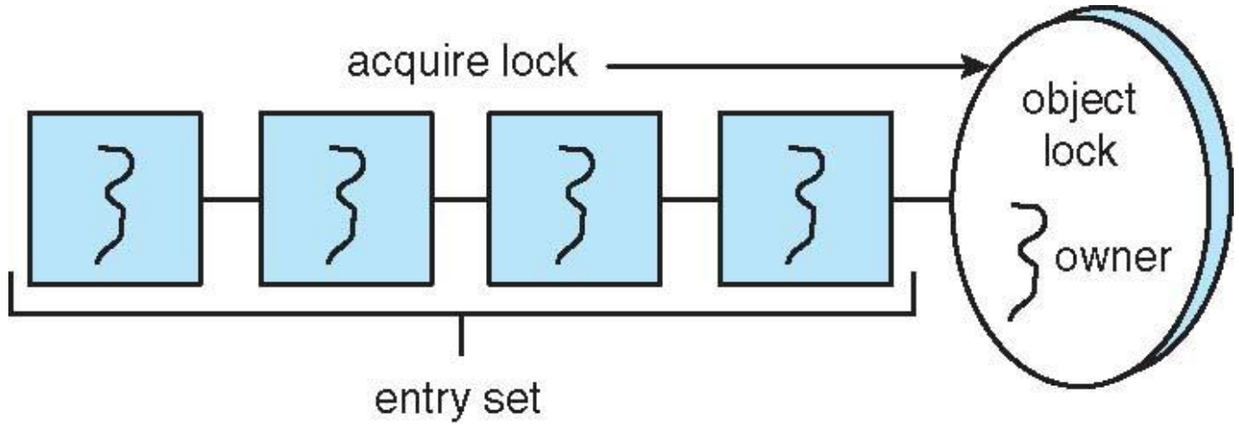
   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

Java Synchronization

- Her nesnenin ilişkili giriş kümesi (**entry set**) vardır.



Java Synchronization wait/notify()

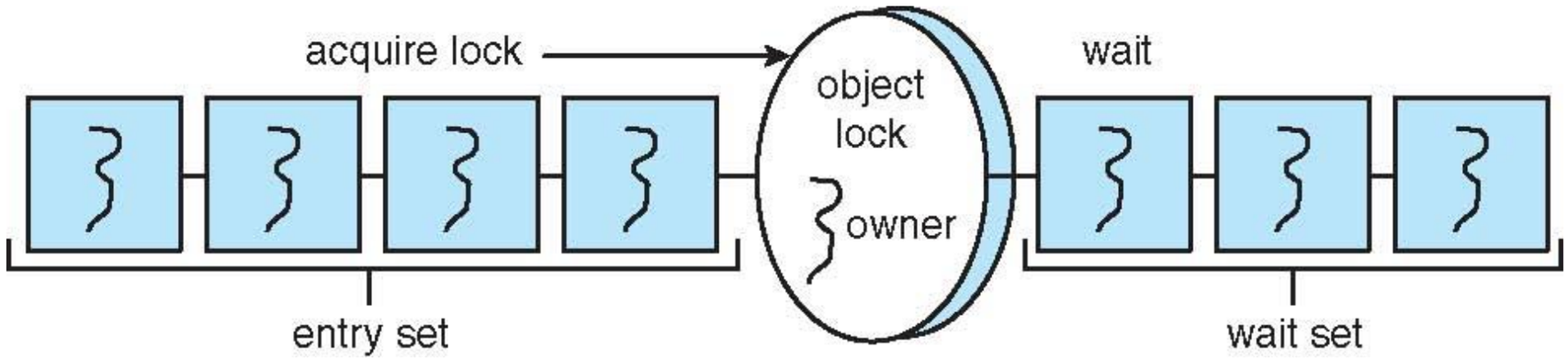
- herbir nesne bir **wait set** e sahiptir
- Bir iş parçacığı **wait ()** metodunu çağırdığında:
  1. iş parçacığı, nesne kilidini serbest bırakır;
  2. İş parçacığının durumu bloke edildi (blocked) olarak ayarlanır;
  3. İş parçacığı, nesne için bekleme kümesine yerleştirilir.

- Bir iş parçacığı, bir koşulun gerçekleşmesini beklerken genellikle wait() metodunu çağırır.
- Bir iş parçacığı nasıl bilgilendirilir?

- Bir iş parçacığı, **notify()** metodunu çağırdığında
  1. Bekleme kümesindeki keyfi bir İş parçacığı T seçilir;
  2. T bekleme durumundan giriş kümesine geçer;
  3. T durumu Runnable olarak ayarlanır.

Java Synchronization

- Entry and wait sets



Java Synchronization – wait/notify

- Synchronized insert() method – Correct!

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;

    notify();
}
```

Java Synchronization – wait/notify

- Synchronized remove() method – Correct!

```
// Consumers call this method
public synchronized E remove() {
    E item;

    while (count == 0) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    notify();

    return item;
}
```

Java Reentrant Locks/Yeniden girişli

- mutex kilidine benzer
- **`finally`** kısmında, try bloğunda bir istisna meydana geldiğinde kilitin serbest bırakılmasını sağlar.

```
Lock key = new ReentrantLock();

key.lock();
try {
   /* critical section */
}
finally {
   key.unlock();
}
```

Java Semaphores

- Constructor:

```
Semaphore(int value);
```

- Kullanımı:

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Java Condition Variables

- Durum değişkenleri **`ReentrantLock`** ile ilişkilidir
- **`ReentrantLock`**: metodundan **`newCondition()`** kullanarak durum değişkeninin oluşturulması

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- Bir iş parçacığı, **`await()`** metodunu çağırarak bekler ve **`signal()`** metodunu çağırarak sinyal verir.

Java Condition Variables

- Örnek:
- Beş Thread 0 – 4 arası numaralanmış
- Hangi Thread in sırada olduğunu belirten turn değişkeni (Paylaşılan veri)
- İş parçacığı, bir iş yapmak istediğinde doWork () çağırır. (Ama sıra geldiğinde iş yapabilir
- sıran değilse, bekleyin
- Sıran geldiğinde, bir süre için biraz çalış ...
- Tamamlandığında, sıradaki iş parçacığını bilgilendirin.
- Gerekli veri yapıları:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
   condVars[i] = lock.newCondition();
```

Java Condition Variables

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
   lock.lock();

   try {
      /**
       * If it's not my turn, then wait
       * until I'm signaled.
       */
      if (threadNumber != turn)
         condVars[threadNumber].await();

      /**
       * Do some work for awhile ...
       */

      /**
       * Now signal to the next thread.
       */
      turn = (turn + 1) % 5;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```

Alternatif Yaklaşımlar

- ***İşlemsel bellek (Transactional memory)***
- ***OpenMP (Open Multi-Processing)***
- **Fonksiyonel Programlama Dilleri (Örn: Java)**

İşlemsel bellek

- **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 di!

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

   /* modify shared data */

   release();
}
```

İşlemsel bellek

- Klasik kilitleme yöntemlerine alternatif olarak **programlama dillerine yeni özellikler eklenmiştir.**
- Bellek işlemi (Memory trasnsaction), atomik olan okuma-yazma işlemleri dizisidir.
- Örneğin, **atomic(S)** kullanılarak **S işlemlerinin tümünün işlemsel (transactional) olarak gerçekleştirilmesi sağlanır.(yukarı grafikteki metot yerine aşağıdaki kod yazılır)**

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

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

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;
   }
}
```

Fonksiyonel Programlama Dilleri

- Veri yarışlarını ele alma yaklaşımlarından dolayı Erlang ve Scala veya Java gibi Fonksiyonel dillere olan ilginin artması dikkat çekmektedir
- Java, dil seviyesinde senkronizasyon sağlar.
- Her bir Java nesnesinin ilişkili bir kilidi vardır.
- Fonksiyonel programlama dilleri, durumları korumadıklarından prosedürel dillerden farklı bir paradigma sunmaktadır.
- Değişkenler değişmez olarak değerlendirilir ve bir değer atandıktan sonra durumu değişmez kabul edilir.

End of Chapter 6