YZM 3102
İşletim Sistemleri

Yrd. Doç. Dr. Deniz KILINÇ
Celal Bayar Üniversitesi
Hasan Ferdi Turgutlu Teknoloji Fakültesi
Yazılım Mühendisliği

BÖLÜM – 6.1

Bu bölümde,

- Proses Senkronizasyonu
- Kritik Bölge Problemi
- Yarış Durumu
- Peterson Çözümü
- Senkronizasyon Donanımı
- Test and Set Lock
- Mutex
- Semafor

konularına değinilecektir.

Problem Tanımı

- Paylaşımlı veriye (bellek alanı, dosya) veya paylaşımlı bir kaynağa concurrent (aynı anda) ulaşım her zaman **probleme açıktır** ve **tutarsızlığa** *neden olabilir*.
- Tutarlılığı sağlamanın yolu,

Birlikte çalışan proseslerin veya threadlerin (iş parçacıklarının) kesin bir sırayla işlerini yapabilecekleri bir **mekanizma** oluşturmaktır.

Prod./Cons. Problemi Üzerine

- Daha önce değinilen **sınırlı buffer** üzerinden işlem yapılan **Producer-Consumer** problemini ele alalım.
- Bu metafordaki önemli sıkıntılardan bir tanesi, maksimum **BUFFER_SIZE – 1** kadar elemanın kullanılabiliyor olmasıydı.
- Şimdi tüm buffer'ı tamamen kullanacak bir çözüm düşünelim:
  - Buffer'daki **eleman sayısını tutacak** integer bir **counter** olduğunu düşünelim.
  - **İlk değeri** **0** olsun.
  - *Producer* yeni bir **eleman ürettiğinde** **counter bir artsın**.
  - Consumer bufferdan bir eleman tükettiğinde **counter bir azalsın.**

Prod./Cons. Problemi Üzerine (devam…)

- Producer Prosesi

Eski Hali	Yeni Hali

item next_produced;
while (true)
{
/* produce an item in next_produced */
 	
while (((in + 1) % BUFFER_SIZE) == out)
; /* do nothing, buffer is full*/
 	
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
}
	item next_produced;
	while (true) {
	/* produce an item and */
	/* put in next_produced*/
	
	while (counter == BUFFER_SIZE)
	; // do nothing buffer is full
	
	buffer[in] = next_produced;
	in = (in + 1) % BUFFER_SIZE;
	counter++;
	}

Prod./Cons. Problemi Üzerine (devam…)

- Consumer Prosesi

Eski Hali	Yeni Hali

item next_consumed;
while (true) {
 	
while (in == out)
; /* do nothing, buffer is empty*/
 	
next-consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
/* consume the item in next cons. */
}
	item next_consumed;
	while (true) {
	while (counter == 0)
	; // do nothing buffer empty
	
	next_consumed = buffer[out];
	out = (out + 1) % BUFFER_SIZE;
	counter--;
	/* consume the item in next cons. */
	}

Prod./Cons. Problemi Üzerine (devam…)

- Her iki prosesi de **tek tek incelediğimizde** doğru tasarlanmış gibi görünebilirler ancak concurrent çalışma *söz konusu olduğunda* tutarlı sonuçlar üretmeyebilirler.
- Örneğin: **counter=5** iken **counter++** ve **counter--** deyimlerinin **aynı anda çalıştığını** düşünelim.
- Producer ve consumer concurrent çalıştıktan sonra
  - Değişkenin değeri **4, 5 veya 6 olabilir**.
- Aslında tek bir sonuç vardır o da 5'tir.
- Bahsettiğimiz bu problemi farklı bir açıdan bakarak oluşturmaya çalışalım.

Prod./Cons. Problemi Üzerine (devam…)

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

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

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

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

- $register_1$, $register_2$ yerel CPU register'larıdır.

Prod./Cons. Problemi Üzerine (devam…)

- counter++ ve counter-- hesaplamalarının **concurrent** olduğunu düşünelim.
- Yani diğer bir deyişle **üretici** ve **tüketici** **belleği** aynı anda ***güncellemeye çalışsın***.
- Bu durumda, **makine dilinde** durumlar **iç içe geçebilir**.

S0: producer execute register$_1$ = counter **{register$_1$ = 5}**
S1: producer execute register$_1$ = register1 + 1 **{register$_1$ = 6}**
S2: consumer execute register$_2$ = counter **{register$_2$ = 5}**
S3: consumer execute register$_2$ = register$_2$ – 1 **{register$_2$ = 4}**
S4: producer execute counter = register$_1$ **{counter = 6 }**
S5: consumer execute counter = register$_2$ **{counter = 4}**

Prod./Cons. Problemi Üzerine (devam…)

- **counter** değişkeninin ilk değerinin **5** olduğunu düşünürsek önceki örnekteki **son durumda**
  - Değeri yanlış bir şekilde **4 olur**.
- S4 ve S5'in yerleri değiştiğinde
  - Değer yine yanlış bir şekilde **6 olur**.
- Bu yanlış duruma gelinmesinin nedeni:
  - Producer ve consumer proseslerinin counter değişkenini **eş zamanlı** olarak güncellemelerine **izin verilmesidir**.

Race Condition (Yarış Durumu)

- İki veya daha fazla prosesin **paylaşımlı kullandıkları** bir veri üzerinde yapmış oldukları okuma ve yazma işlemleri, hangi işlemin ne zaman çalıştığına bağlı olarak bir **son değere sahip oluyorsa** bu duruma **yarış durumu** denilmektedir.
- Örneğin, producer-consumer problemindeki yarış durumundan kurtulmak için **bir anda** *sadece bir prosesin* **counter değişkenini değiştireceği garanti edilmelidir**. Prosesler bir şekilde senkronize olmalıdır.

Örnek: Yazıcı İşlem Kuyruklama

Bilgi:

- Proses bir dosya yazdırmak istediğinde, dosyanın adını özel bir spooler (kuyruklama) *dizine yazar*.
- Diğer proses (printer daemon) **yazdırılacak dosya var mı** diye sürekli kontrol eder.
- Dosya **varsa** *print eder* ve *isimlerini* ***dizinden çıkarır***.
- Spooler dizininin çok sayıda giriş içerdiğini düşünelim.
- **out:** print edilecek bir sonraki dosyayı işaret eder.
- **in:** dizindeki bir sonraki boş slotu işaret eder.

Örnek: Yazıcı İşlem Kuyruklama (devam…)

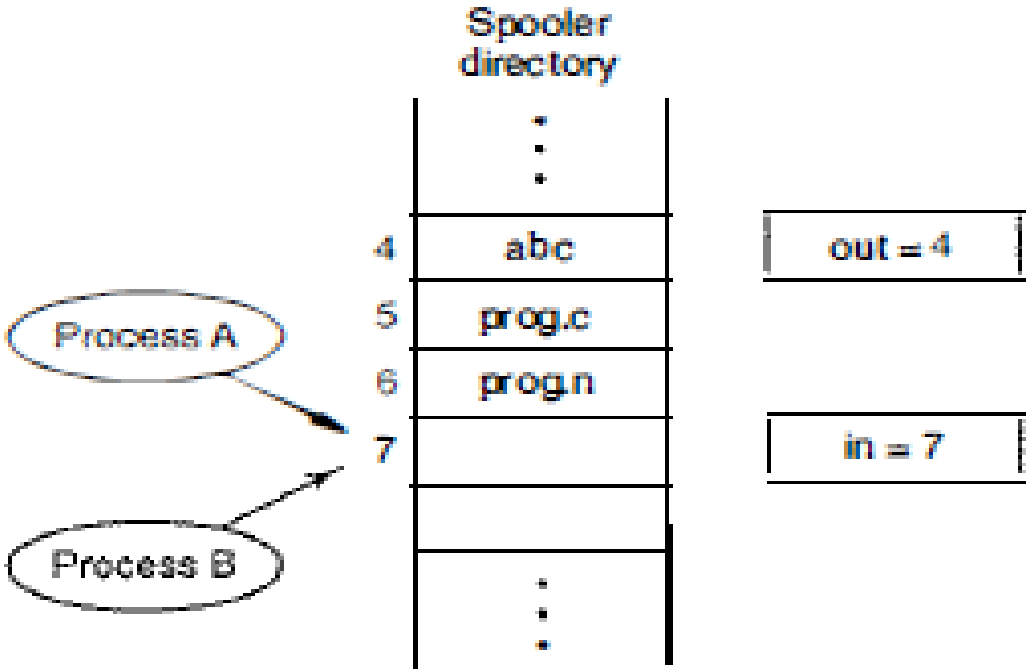


Figure 2-21. Two processes want to access shared memory at the same time.

Örnek: Yazıcı İşlem Kuyruklama (devam…)

Senaryo:

- Proses A ve B dosya yazdırmak istiyorlar. A in'i okur ve 7'yi local değişkeninde saklar.
- **Clock interrupt oluşur** ve CPU A prosesini yeterince uzun işlettiğini düşünür ve B prosesine atlar.
- B in değişkenini okur ve **yine** 7'yi alır.
- *Bu durumda ikisi de bir sonraki boş slotun 7 olduğunu düşünür*.
- B çalışmaya devam eder ve yazdıracağı dosya adını 7 ye yazar. in değişkenini 8 yapar.
- A kaldığı yerden işletileceği zaman dosya adını 7 ye yazmak ister. B nin eklediği dosya adını silerek kendi dosya adını yazar.
- **Printer daemon** yanlış hiçbir şey fark etmez, çalışmaya devam eder.
- **B hiçbir zaman cevap alamaz**.

Critical Section Problemi

- Bir sistemde *n* tane proses olduğunu varsayalım: {P_0, P_1, …, P_{n-1}}.
  - Her proses **kritik bölge/kısım** olarak adlandırılan bir kod bloğuna sahiptir.
  - Bu bölgede **paylaşımlı değişkenler**, bir **tablo** veya bir **dosya** değiştiriliyor olabilir.
- Proseslerin ortaklaşa kullandıkları veri alanlarına *erişip*, **veri okuma** ve **yazma** işlemlerini yaptıkları program parçalarına **kritik bölge/kısım** denilmektedir.
- Her proses kritik kısmına girmeden önce *izin istemelidir* (**entry section**). Critical section tamamlandıktan sonra **exit section** bunu izler. Geri kalan kod ise **remainder (kalan) section**'da yer alır.

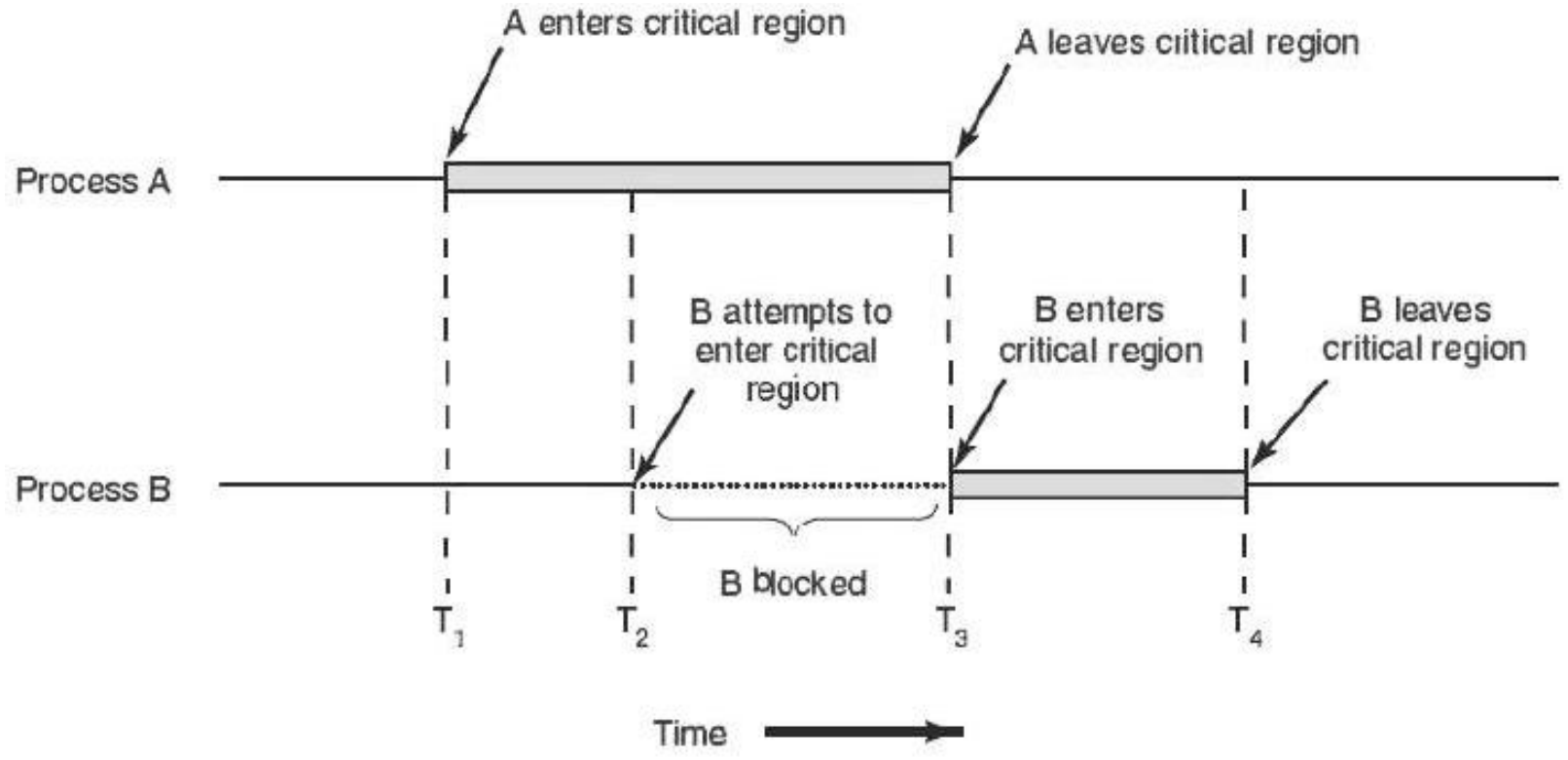
Critical Section Problemi (devam…)

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

Critical Section Problemi (devam…)

Critical section probleminin çözümü **3 durumu** karşılamalıdır.

1. **Mutual Exclusion (Karşılıklı Dışlama):** Eğer bir proses bir crtical section'da çalışıyorsa başka hiçbir proses critical sectionda çalışamaz.



Critical Section Problemi (devam…)

2. **Progress (İlerleme):** Hiçbir proses critical section'da çalışmıyorsa ve critical section'a girmek isteyen birtakım prosesler varsa bu prosesler sıra ile critical section'a girer. Critical section'a girme isteği uzun bir süre ertelenemez. Yani prosesler birbirilerinin *sürekli iş yapmalarını* **engellememelidir**. *Karşılıklı tıkanmaya* neden olunmaması gereklidir.
3. **Bounded Waiting (Sınırlı Bekleme):** Bir proses beklerken diğer proseslerin critical section'a girme sayısının bir sınırı vardır.

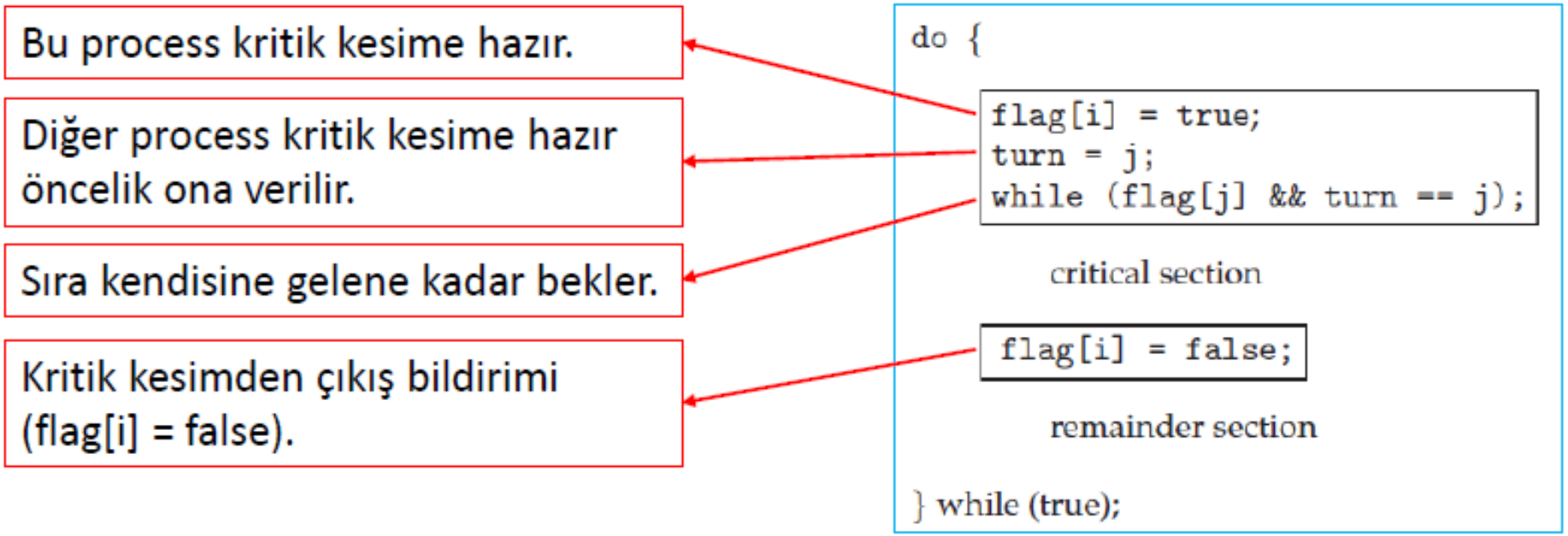
Critical Section Problemi (devam…)

- **Çözümler**
  - Peterson Çözümü
  - Senkronizasyon Donanımı
  - Mutex Kilitleri
  - Semaforlar
  - Monitörler

Peterson Çözümü

- Critical section problemi için geliştirilmiş klasik **yazılım tabanlı** bir çözümdür.
- Modern bilgisayarlar load() ve store() gibi basit makine dili komutları ile çalıştıkları için bu mimarilerde **%100 çalışma** garantisi olmayan bir çözümdür.
- Peterson algoritması, 2 proses için tasarlanmıştır. Bu prosesler 2 veri nesnesini paylaşır ve bu şekilde senkronize olurlar.
  - `int turn:` Kritik kısma girme sırasını belirler. **if turn==i then** Pi kritik kısmına girebilir.
  - `boolean flag[2]:` Bir prosesin kritik kısma girmeye hazır olduğunu belirtir. **if flag[i] is true** then Pi kritik kısmına girmeye hazır.

Peterson Çözümü (devam…)



Peterson Çözümü (devam…)

P_0 için kod bloğu	P_1 için kod bloğu

`flag[0] = true;`
`turn = 1;`
`while (flag[1] && turn == 1);`
`--------------`
`Critical_section_işini_yap();`
`--------------`
`flag[0] = false;`
`--------------`
`Remainder_section_devam_et();`
	`flag[1] = true;`
	`turn = 0;`
	`while (flag[0] && turn == 0);`
	`--------------`
	`Critical_section_işini_yap();`
	`--------------`
	`flag[1] = false;`
	`--------------`
	`Remainder_section_devam_et();`

- ***turn*** değişkenini iki proses de aynı anda değiştirse bile, son değer alınır ve o process kritik bölüme girer **(mutual exclusion).**
- Kritik bölümü tamamlayan proses, kritik bölüme giriş gereksinimini **iptal eder** ve diğer process **kritik böl. girer (progress).**
- Bir proses **kritik bölüme** bir kez girdikten sonra *sırayı diğerine aktarır* **(bounded waiting).**

Senkronizasyon Donanımı

- Yazılım çözümlerinin dışında donanım tabanlı çözümler de bulunmaktadır.
- Çözümlerde **temel mantık** **kritik bölgeyi kilitlemekten** (**lock**) geçer.

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

Kesmeleri Devre Dışı Bırakmak

- **Kilitleme** mekanizması **olmayan** senkronizasyon donanımı başlığı altında sunulan bir çözümdür.
- Prosesler kritik bölgeye girdiklerinde tüm kesmeleri devre dışı bırakabilirler.
- Kritik bölge dışına çıktıklarında ise kesmeler **tekrar** devreye girer. Bu durumda **eğer** bir proses bu işlemi uygulasa ve çıktığında eski durumuna getirmez ise ***sistem çalışmaz duruma gelir***.
- Sistemde birden fazla işlemci varsa bu işlem **sadece tek bir işlemciyi etkiler**. Kullanıcı proseslerine bu hakkı vermek oldukça tehlikelidir, sistemi tamamen çalışmaz duruma getirebilir.

Kesmeleri Devre Dışı Bırakmak (devam…)

- **Birden fazla işlemcinin** olduğu ortamda bu yöntemi desteklemenin maliyeti ise her işlemciye kesmeleri devre dışı bırak mesajının sürekli gönderilmesi ve kesmelerin daha sonra tekrar aktif edilmesidir. Bu da zaman alan (**time consuming**) bir mekanizmadır.

Test and Set (TSL) Lock

- Modern bilgisayar sistemlerinin çoğu, bir word'ü test etmemizi/güncellememizi veya iki word'ü yer değiştirmemizi sağlayan özel atomik donanım komutları sağlar ve bu işlemi tek bir **uninterruptible unit** olarak gerçekleştirirler. Bu komutlardan en bilineni **test_and_set()** komutudur.

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

```
do {
      while ( TestAndSet (&lock ))
      ;    // do nothing…

      // critical section…

     lock = FALSE;

     // remainder section…
} while (TRUE);
```

Test and Set (TSL) Lock (devam…)

- KritikKısmaGir

```
TSL register, flag ; //flag reg saklayıcıya kopyalanır ve sonra otomatik olarak 1 yapılır.
CMP register, #0 ; //flag=0 mı?
JNZ KritikKısmaGir; //reg 0 değilse kritik kısma gir
RET                ; //geri dön
```

- KritikKısımdanAyrıl

```
MOV flag, #0       ; //flag = 0
RET                ; //geri dön
```

Mutex Kilitleri

- **Donanım tabanlı** kritik bölüm çözümleri **karmaşıktır** ve yazılımcılar tarafından erişilemez.
- İşletim sistemi tasarımcıları, *kritik bölüm problemi* için çeşitli **yazılım araçları** (kütüphaneler veya API'ler) geliştirmişlerdir.
- **En basit** yazılım aracı **mutex lock aracıdır**.
- Mutex lock kritik kısımları korumak ve yarış koşullarını **engellemek** üzere kullanılır.
- Bir proses, **kritik kısmına girmeden önce** bir **lock talep** etmelidir.

Mutex Kilitleri (devam…)

- Kritik kısımdan ayrılınca lock değişkenini serbest bırakmalıdır:
  - Her proses kritik bölüme girmek ve kilitlemek için **izin** ister (**acquire()**).
  - Kritik bölümünden çıktıktan sonra da lock durumu **sonlandırılır** (**release**()).
  - Lock durumunun uygun olup olmadığına karar vermek için ***boolean*** değişken kullanılır (**available**).
- acquire ve release fonksiyon çağırımlarının **atomik gerçekleşmesi gerektiği** için bu fonksiyonun *arka planında* donanım tabanlı çözümler kullanılır.

Mutex Kilitleri (devam…)

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

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

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

- İşletim sistemlerinde bu tip bir mutex lock mekanizması **spinlock** olarak da adlandırılır

Mutex Kilitleri (devam…)

- Mutex kilitlerindeki temel sorun: **busy waiting** (meşgul bekleme) döngüsüdür.
- Bir proses **kritik kısmında iken** kritik kısımlarına girmek isteyen **diğer prosesler acquire()** fonksiyonu içinde **döngüde** sürekli işlem görürler. Başka bir iş yapamadan beklerler.
- Ayrıca:
  - Meşgul bekleme olduğundan bekleyen proses de **işlemci zamanı harcar**.
  - Bir proses kritik bölgesinden çıktığında bekleyen birden fazla proses varsa **açlık durumu oluşabilir**.

pthread Mutex API Fonksiyonları

- **pthread_mutex_t lock**
  - Mutex nesnesini tanımlar.
- **pthread_mutex_init**(&lock, NULL)
  - Mutex nesnesini oluşturur.
- **pthread_mutex_lock**(&lock)
  - Kritik kod bloğunu kilitler.
- **pthread_mutex_unlock**(&lock)
  - Kritik kod bloğundaki kilidi açar.
- **pthread_mutex_destroy**(&lock)
  - Mutex nesnesini yok eder.

.NET Mutex Sınıfı

- **Mutex m = new Mutex();**
  - Mutex m nesnesini oluşturur.
- **m.WaitOne();**
  - Kritik kod bloğunu kilitler.
- **m.ReleaseMutex();**
  - Kritik kod bloğundaki kilidi açar.

Semaforlar

- **Dijkastra** tarafından 1965 yılında tanımlanmıştır.
- Genelleştirilmiş kilit olarak adlandırılır.
- UNIX işletim sisteminin temel senkronizasyon alt yapısını oluştururlar.
- Bir prosesin **kaynağı kullanıp kullanmadığını** *sürekli olarak kontrol etmek* performans düşüşüne neden olur. (**busy waiting**)
- Gerçek işletimlerde, bu test işlemleri **CPU'yu yorar**.
- Bunu önlemek için Semafor adı verilen **değişken** tanımlanır.

Semaforlar (devam…)

- Semafor *S* – tamsayı bir değişkendir
- İki tane standart operasyona sahiptir
  - **S: | acquire() release() |**
- Orijinal gösterimleri **P()** ve **V()**
  - Almancadaki *Proberen* (test) ve Verhogen (incremet) (Dijkstra)
- Ayrıca **down()** ve **up()**
- Ayrıca **wait()** and **signal()**
- Üst seviye soyutlama ile karmaşıklık azaltılabilir.

Semaforlar (devam…)

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

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

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

Semaforlar (devam…)

- İşletim sistemleri,
  1. **Sayan semafor (counting semaphore)** ve
  2. **İkilik semafor (binary semaphore)** kullanırlar.
- *Sayan semaforların* değeri **kısıtlı değildir**.
- **İkilik semaforların** değeri **0 veya 1** olabilir.
- İkilik semafor **mutex kilitleri** gibi davranır.

```
Semaphore S = new Semaphore();

S.acquire();

   // critical section

S.release();

   // remainder section
```

Semaforlar (devam…)

- **Sayan semaforlar**, belirli sayıdaki kaynağa erişimi denetlemek için kullanılır.
- Sayan semafor kaynak sayısı ile başlatılır.
- Kaynağı kullanmak isteyen her proses, **semafor üzerinde wait() işlemi** gerçekleştirir (**sayaç azaltılır**).
- Bir process **kaynağı serbest bıraktığında** ise **signal()** işlemi gerçekleştirir (**sayaç artırılır**).
- **Semafor değeri = 0**, olduğunda tüm kaynaklar kullanılabilir durumdadır.

Semaforlar (devam…)

- Otel örneği
  - Otelin resepsiyoncusu semafor değişkenini odaları (kaynakları) müşterilere (proses/thread) vermek için kullanır.
  - Otel tamamen boşken otelde kaç odanın müsait olduğu tutar (S = otelin oda sayısı). Müşteri geldikçe oda sayısı azaltılır.
  - Odalardan çıkış yapıldığında bu sayı arttırılır. Sayı sıfıra ulaştığında boşta oda kalmaz ve gelenler isterlerse lobide bekler (kuyruk). Bekleyenler sırayla alınırlar (öncelik olabilir).
  - İkili semaforda oda 1 tanedir.

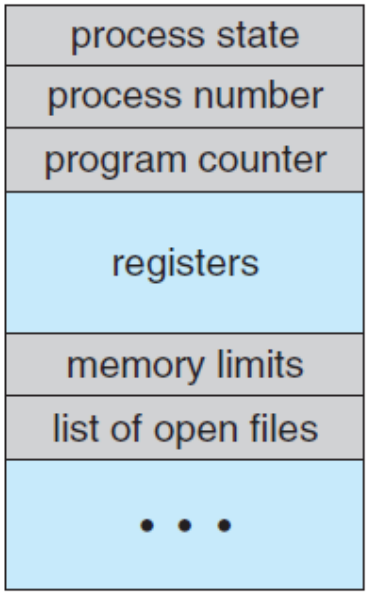
Semafor Gerçekleştirimi

- **Mutex kilit** gibi **wait()** ve **signal()** fonksiyonlarının kullanımı, kritik kısım probleminin çözümündeki daha önceki yaklaşımlarla benzerdir. ***Örneğin:*** Bir proses, wait çalıştırır ve semaforun değerinin pozitif olmadığını bulursa bekler.
- Yani bu haliyle **semafor kullanımı** da **busy waiting** (meşgul bekleme) durumu oluşmasına ve **CPU'nun boşa kullanılmasına** neden olacaktır.
- Meşgul bekleme **problemini çözmek** için
  - Prosesi CPU üzerinde ***meşgul bekletmek yerine***,
  - Prosesi **bloke edip**, **bekleme kuyruğuna alınmalıdır**.

Semafor Gerçekleştirimi (devam…)

- Bu gerçekleştirimde, proses, meşgul bekleme yerine kendini bloke eder. *Bloke edilen proses*, semaforla ilişkilendirilir, durumu "**bekleme**" konumuna getirilir ve semafor ile ilişkili bir **bekleme kuyruğuna yazılır**.
- Semafor S in *üzerinde bekleyen*, **bloke edilmiş olan proses**, *diğer proseslerden bazıları* **signal operasyonu** işlettiklerinde **yeniden başlatılabilir**.
- Proses bir **wakeup()** operasyonu ile restart edilir. Bu prosesin durumunu **bekelemeden → hazıra** geçirir.
- Kernel tarafından kullanılan *2 operasyon* vardır:
- block()
  - Prosesin durumunu **running den waiting'e** çevirir.
  - Prosesi bekleme (**waiting**) kuyruğuna yerleştirir.
- wakeup()
  - Bekleme kuyruğundan semaforla ilişkili olan **1 prosesi çıkarır** ve hazır kuyruğuna koyar.

Semafor Gerçekleştirimi (devam…)




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

- Her semafor **tamsayı bir değere** ve bu semafor üzerinde bekleyen **proses listesine** sahiptir (PCB listesine bir pointer kaydı).
- Bir proses, bir semafor üzerinde beklemeli ise semaforun proses listesine eklenir.
- Signal operasyonu prosesi semafor listesinden çıkarır.

Semafor Gerçekleştirimi (devam…)

wait()

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

signal()

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

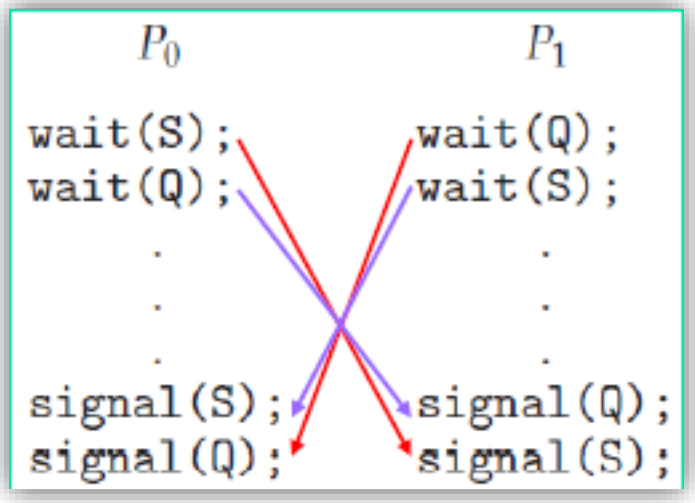
Semafor Gerçekleştirimi (devam…)

Soru: Semaforun bu şekilde kullanımı busy waiting problemini kökten çözdü mü?

Ölü-Kilit Problemi (Deadlock)

- Bir prosesin beklemesinden dolayı, iki veya daha prosesin sonsuza kadar beklemesine **ölü-kilit (deadlock)** denir.
- Aşağıdaki ***P0*** ve ***P1***process'leri, **S** ve **Q** semaforlarına erişmektedir.

  - ***P0***, wait(S) ve ***P1***, wait(Q) işlemlerini çalıştırsın.
  - ***P0*** wait(Q)'yu çalıştırırken, ***P1***signal(Q)'yu çalıştırana kadar bekler.
  - ***P1*** wait(S)'yi çalıştırırken, ***P0*** signal(S)'yi çalıştırana kadar bekler.



Klasik Senkronizasyon Problemleri

- Klasikleşmiş üç tane senkronizasyon problemi üzerinde çalışılacaktır.
  1. The Bounded-Buffer Problem
  2. The Readers–Writers Problem
  3. The Dining-Philosophers Problem

İYİ ÇALIŞMALAR…

Yararlanılan Kaynaklar

- **Ders Kitabı:**
  - Operating System Concepts, Ninth Edition, Abraham Silberschatz, Peter Bear Galvin, Greg Gagne
- **Yardımcı Okumalar:**
  - İşletim Sistemleri, Ali Saatçi
  - Şirin Karadeniz, Ders Notları
  - İbrahim Türkoğlu, Ders Notları
- **M. Ali Akcayol, Gazi Üniversitesi Bilgisayar Mühendisliği Bölümü**
- **http://www.albahari.com/threading/**