Bölüm 6: Senkronizasyon

İşletim Sistemleri

Giriş

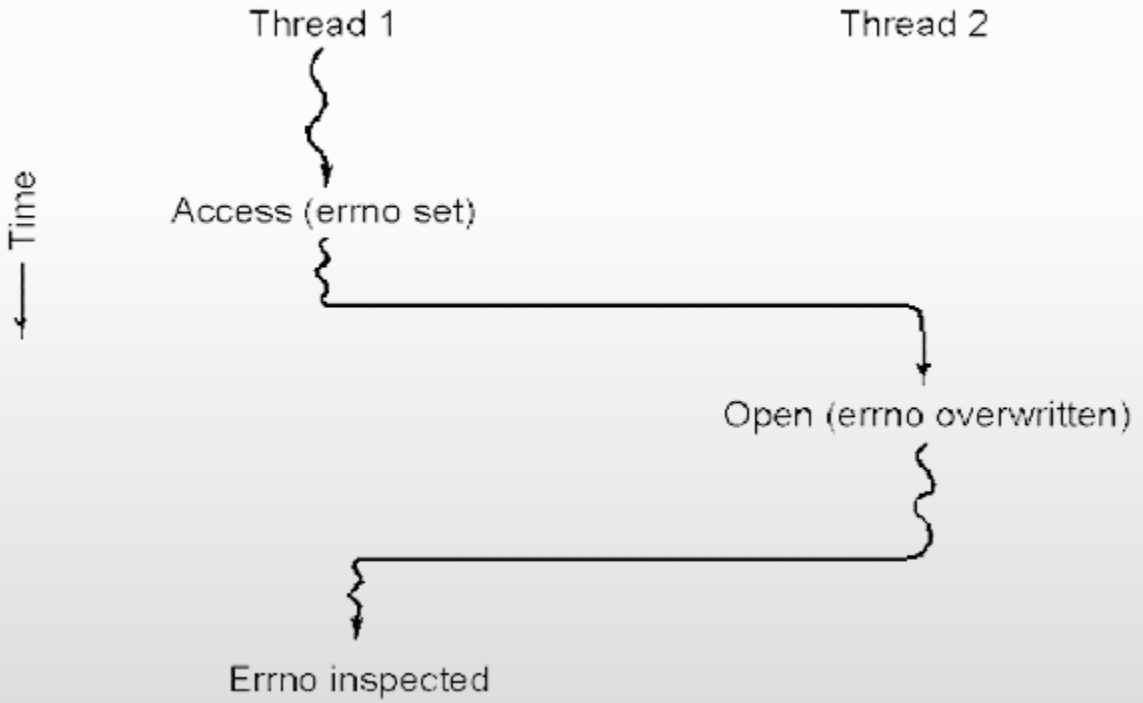
- İşlemler aynı anda yürütülebilir
  - Herhangi bir zamanda kesintiye uğrayabilir,
  - Yürütmeyi kısmen tamamlayabilir
- Paylaşılan verilere eşzamanlı erişim, veri tutarsızlığına neden olabilir
- Veri tutarlılığının sürdürülmesi, işbirliği yapan süreçlerin düzenli bir şekilde yürütülmesini sağlayacak mekanizmalar gerektirir.

Yaşanacak Problemler

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

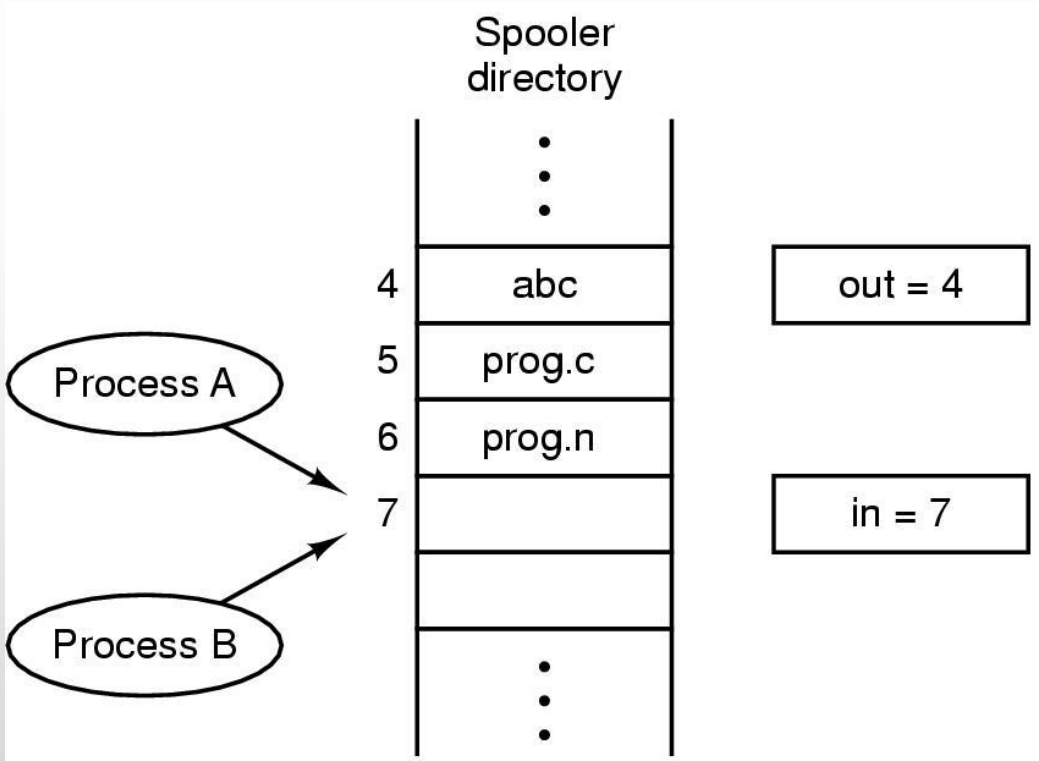
Veri Tutarsızlığı

- Global bir değişkenin kullanımıyla ilgili iş parçacıkları arasında yaşanabilecek çakışma



Süreçler Arası İletişim Problemleri

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

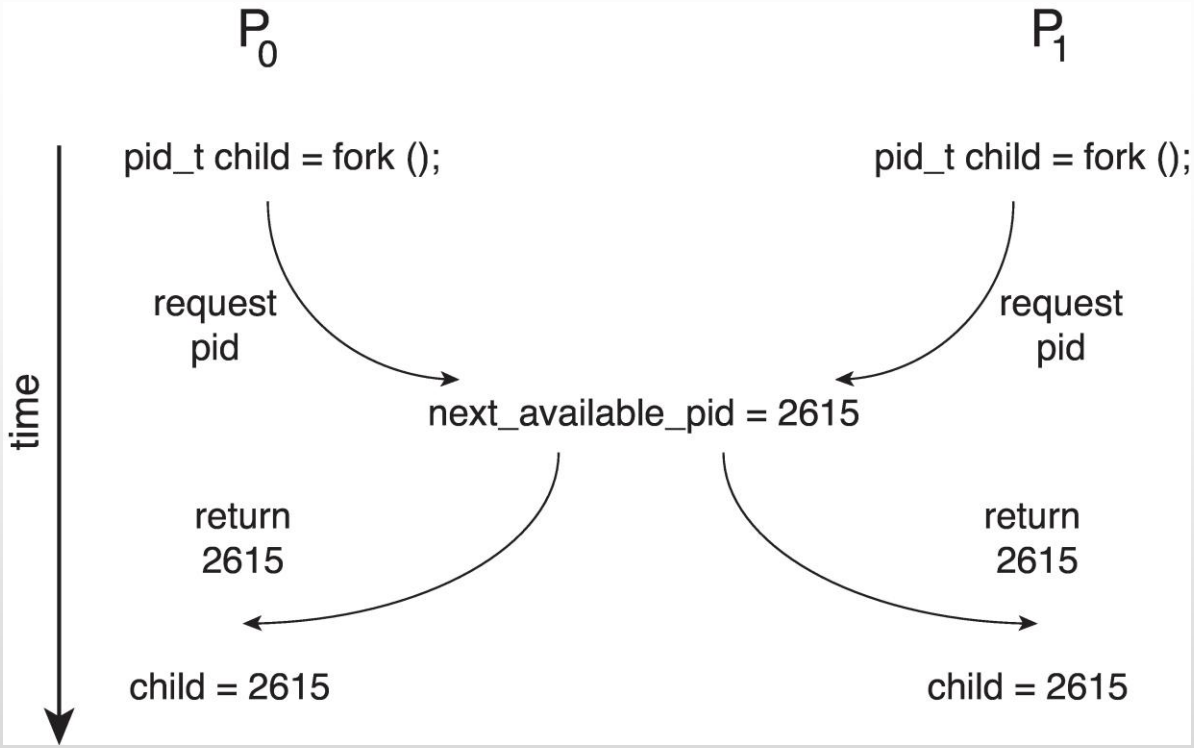


Süreçler Arası İletişim Problemleri

- Süreç çakışmalarıyla nasıl başa çıkılır,
  - Aynı koltuk için 2 havayolu rezervasyonu
- Bağımlılıklar mevcutken doğru sıralama nasıl yapılır,
  - Silahı ateşlemeden önce nişan alınması

Yarış Durumu

- P0 ve P1'in next_available_pid değişkenine aynı anda erişmesini engelleyecek bir mekanizma olmadığı sürece, aynı pid iki farklı işleme atanabilir!



Kritik Bölge Problemi

- n süreçten $\{p_0, p_1, \ldots p_{n-1}\}$ oluşan bir sistem olsun
- Her sürecin kritik kod bölümü vardır
- Süreç, bu kesimde global değişkenleri değiştiriyor, bir tabloyu güncelliyor, bir dosyaya yazıyor olabilir.
- Bir süreç kritik bölümdeyken, başka hiçbir süreç kritik bölgede olmamalı.
- Kritik bölge problemi, bu sorunu çözmek için protokol tasarlamaktır.
- Her süreç, kritik bölgeye girmek için izin istemelidir.

Kritik Bölge Problemi

- **Karşılıklı Dışlama** P süreci kritik bölgede ise, başka hiçbir süreç kritik bölgede yürütülemez.
- **İlerleme** Kritik bölümünde yürütülen süreç yoksa ve kritik bölümüne girmek isteyen bazı süreçler varsa, bir sonraki kritik bölüme girecek sürecin seçimi süresiz olarak ertelenemez.
- **Sınırlı Bekleme** Bir süreç, kritik bölümüne girmek için bir istekte bulunduktan sonra ve bu istek kabul edilmeden önce, diğer süreçlerin kritik bölümlerine girmelerine izin verilme sayısında bir sınır bulunmalıdır.

Önerilen Çözümler

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

Kesilmeleri Devre Dışı Bırakma

- Süreç kesilmeleri devre dışı bırakır, kritik bölgeye girer, kritik bölgeden çıktığında kesilmeleri etkinleştirir
- Problemler
  - Süreç, kesilmeleri devre dışı bırakamazsa sistem çöker
  - Clock bir kesilme olduğundan, diğer süreçler CPU kullanamaz.
  - Çoklu çekirdekli sistemler için çözüm değil
  - İşletim sisteminin kendisi için yararlı, ancak kullanıcılar için değil
  - Kritik bölgeye giren süreç çok uzun sürebilir
  - Bazı süreçlere sıra hiç gelmeyebilir (starvation)

Kilit Değişkeni

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

Kilit Değişkeni

```
while (true){

        while (turn = = j);

        /* critical section */

        turn = j;

        /* remainder section */

}
```

Yarış Durumu

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

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

Karşılıklı Dışlama

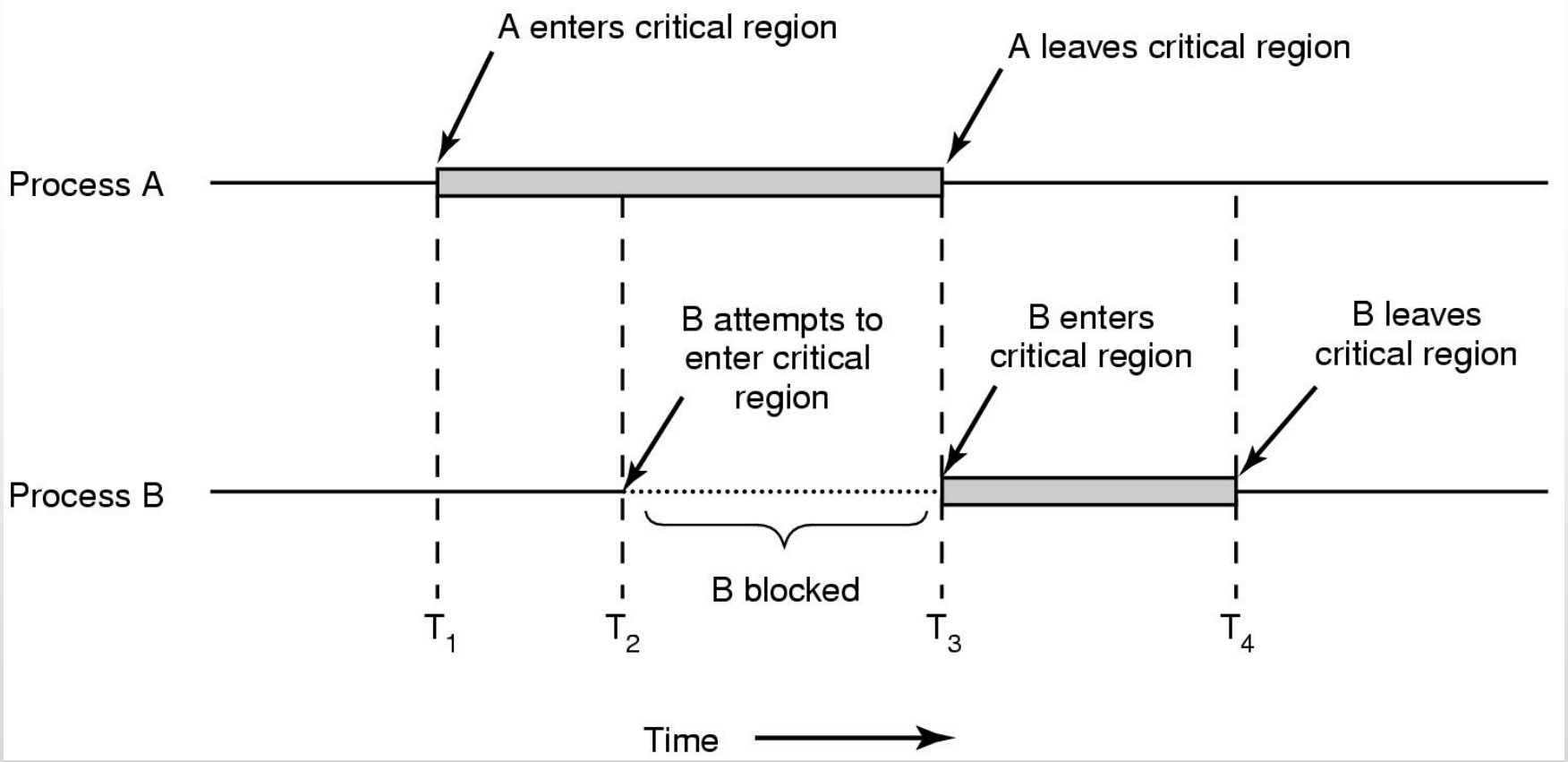
Karşılıklı dışlama sağlamak için dört koşul

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

Kritik Bölge

```
do {

    entry section

        critical section

    exit section

            remainder section

} while (TRUE);
```

Kritik Bölge Kullanarak Karşılıklı Dışlama



Sıkı Değişim – Strict Alternation

- Önce ben, sonra sen!, her işlem CPU'yu kullanma sırası aldığından adaleti sağlar.

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

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

Kavramlar

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

Peterson’un Çözümü

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

Peterson’un Çözümü

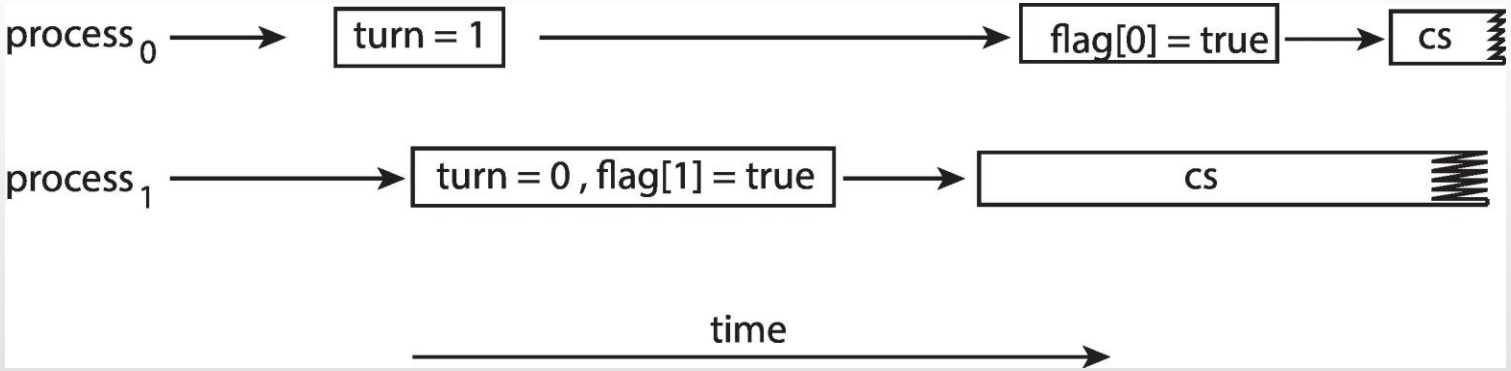
```
while (true){

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

            /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Peterson'un Çözümü

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

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

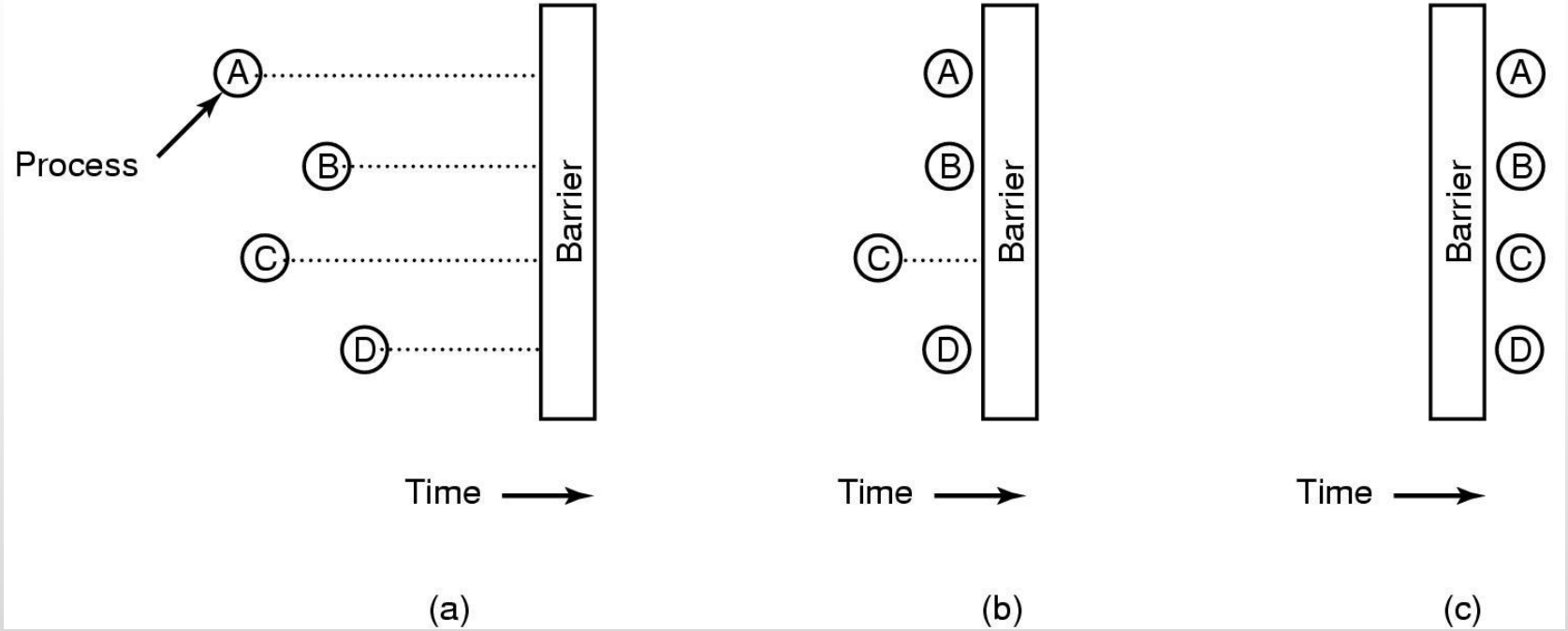
Modern Mimaride Peterson’un Çözümü

- Her iki süreç aynı anda kritik bölümlerinde olabilir!
- Peterson'ın çözümünün modern bilgisayar mimarisinde doğru şekilde çalışmasını sağlamak için **Bellek Bariyer’i** kullanmalıyız.



Bariyer (Barriers)

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



Bariyer (Barriers)

- Bellek bariyeri, bellekteki herhangi bir değişikliğin diğer tüm işlemcilere yayılmasını (görünür hale getirilmesini) zorlayan bir komuttur.
- Bellek bariyeri, sistem sonraki bir bellek okuma/yazma işleminden önce tüm bellek okuma/yazma işlemlerinin tamamlanmasını sağlar.

Thread 1

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

Thread 2

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

- İş Parçacığı 1: flag değerinin x değerinden önce okunması garanti edilir.
- İş Parçacığı 2: x atamasının flag atamasından önce olması garanti edilir.

Donanım Çözümleri

- Bir sözcüğün içeriğini test edip değiştirmemize veya iki kelimenin içeriğini atomik olarak (kesintisiz olarak) değiştirmemize izin veren özel donanım talimatları.
  - Test Et ve Ayarla talimatı (test-and-set)
  - Karşılaştır ve Değiştir talimatı (compare-and-swap)

TSL Komutu

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

TSL Komutu

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

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

XCHG Komutu

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

XCHG Komutu

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

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

Uyuma ve Uyandırma

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

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

Üretici Tüketici Problemi

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

- Basit bir çözüm

Ölümcül Yarış Durumu - Producer

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

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

Ölümcül Yarış Durumu - Consumer

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

Veri Kaybı Sorunu

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

Semafor

- Dijkstra tarafından önerilen yeni bir değişken türü
- Atomik eylem, tek ve bölünmez
- Down – wait (P)
  - semafor değeri kontrol edilir,
    - 0 ise meşgul bekler,
    - değilse değeri azalt ve devam et
- Up - signal (V)
  - semafor değeri arttırılır,
  - semaforda bekleyen süreçler devam eder,
  - kaynak sayısının bir işareti olarak düşünülebilir

Üretici-Tüketici Sorununa Çözüm

- **full**: dolu yuvaların sayısı, başlangıç değeri 0
- **empty**: boş yuvaların sayısı, başlangıç değeri N
- **mutex**: arabelleğe (buffer) aynı anda erişimi engeller, başlangıç değeri 0 (ikili semafor)

- Senkronizasyon/karşılıklı dışlama

Semafor Kullanımı - Producer

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

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

Semafor Kullanımı - Consumer

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

Mutex Kilitleri

- En basit senkronizasyon muteks kilidi kullanmak
  - Kilidin mevcut olup olmadığını gösteren Boolean değişken
- Kritik bir bölümü şu şekilde korunur
  - Önce bir kilit edin (acquire)
  - Ardından kilidi serbest bırak (release)
- acquire() ve release() çağrıları atomik olmalıdır
  - Genellikle XCHG gibi atomik donanım komutları ile uygulanır.
- Ancak bu çözüm meşgul beklemeyi (**busy waiting**) gerektirir
- Bu nedenle bu kilit, **spinlock** olarak adlandırılır

mutex_lock ve mutex_unlock

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

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

Pthreads Mutex Çağrıları

Çağrı	Tanım
Pthread_mutex_init	Mutex oluştur
Pthread_mutex_destroy	Mutex'i kaldır
Pthread_mutex_lock	Kilit al ya da bloke et
Pthread_mutex_trylock	Kilit al ya da hata ver
Pthread_mutex_unlock	Kilidi kaldır

Pthreads Condition Çağrıları

Çağrı	Tanım
Pthread_cond_init	Koşul değişkeni oluşturur
Pthread_cond_destroy	Koşul değişkenini yok eder
Pthread_cond_wait	Sinyal bekler
Pthread_cond_signal	Başka bir iş parçacığına sinyal gönderir
Pthread_cond_broadcast	Birden fazla iş parçacığına sinyal gönderir

Pthreads Mutex - Producer

```
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

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

Pthreads Mutex - Consumer

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

Gözleyici (Monitors)

- Süreç senkronizasyonu için yüksek seviyeli soyutlama sağlar
- Muteks ve koşul değişkenlerini kullanarak işleri karıştırmak kolay.
  - Semaforlu üretici tüketici kodundaki iki down fonksiyonunun yer değiştirmesi kilitlenmeye neden olur
- Gözleyici, karşılıklı dışlama ve bloke etme mekanizmasını uygulayan yapı.
  - Gruplandırılmış {prosedürler, veri yapıları ve değişkenlerden} oluşur
- Bir süreç, gözleyicinin içindeki prosedürleri çağırabilir, ancak içindeki öğelere doğrudan erişemez.
  - Aynı anda gözleyici içinde bir prosedür çalışabilir

Gözleyici

```
monitor example
    integer i;
    condition c;
    procedure producer();
    .
    .
    end;
    procedure consumer();
    .
    .
    end;
end monitor;
```

Gözleyici

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

Gözleyici

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

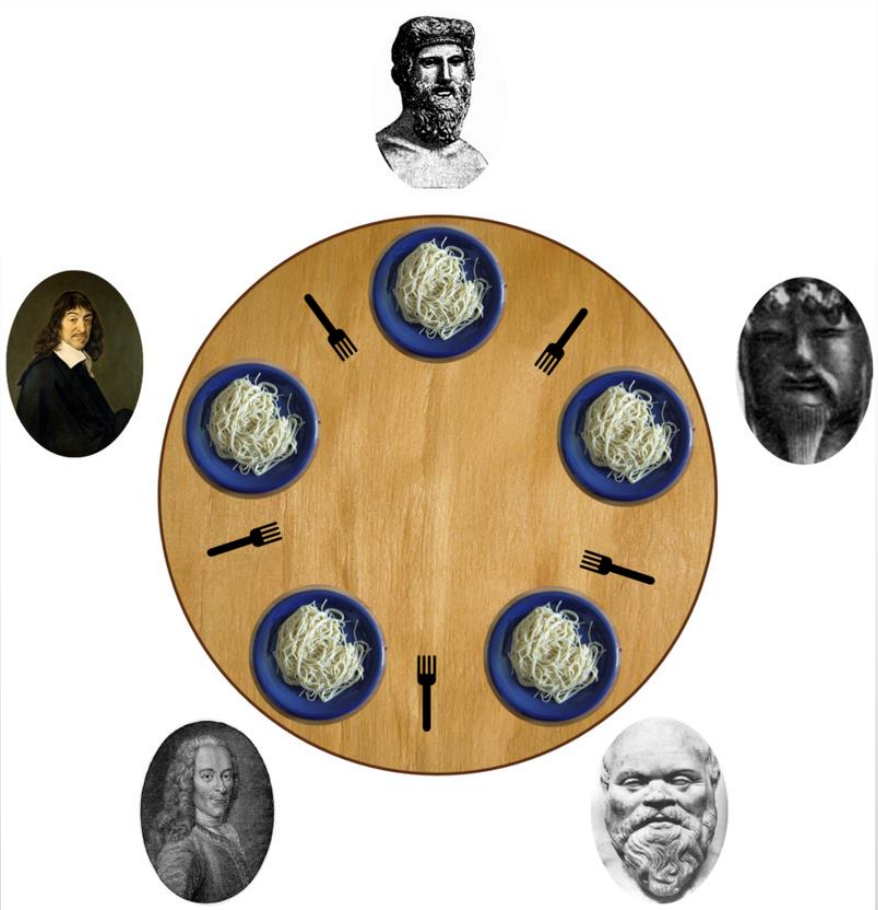
Süreçler Arası İletişim Problemleri

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

- Okur-Yazar problemi
  - Bir veritabanına erişimi modeller

Dining Philosophers Problemi

- .



Dining Philosophers

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

Dining Philosophers - loop

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

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

Dining Philosophers – take forks

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

Dining Philosophers – put forks

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

Dining Philosophers – test state

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

Okur-Yazar Problemi - writer

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

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

Okur-Yazar Problemi - reader

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

SON