Bölüm 6: Senkronizasyon

İşletim Sistemleri

Senkronizasyon

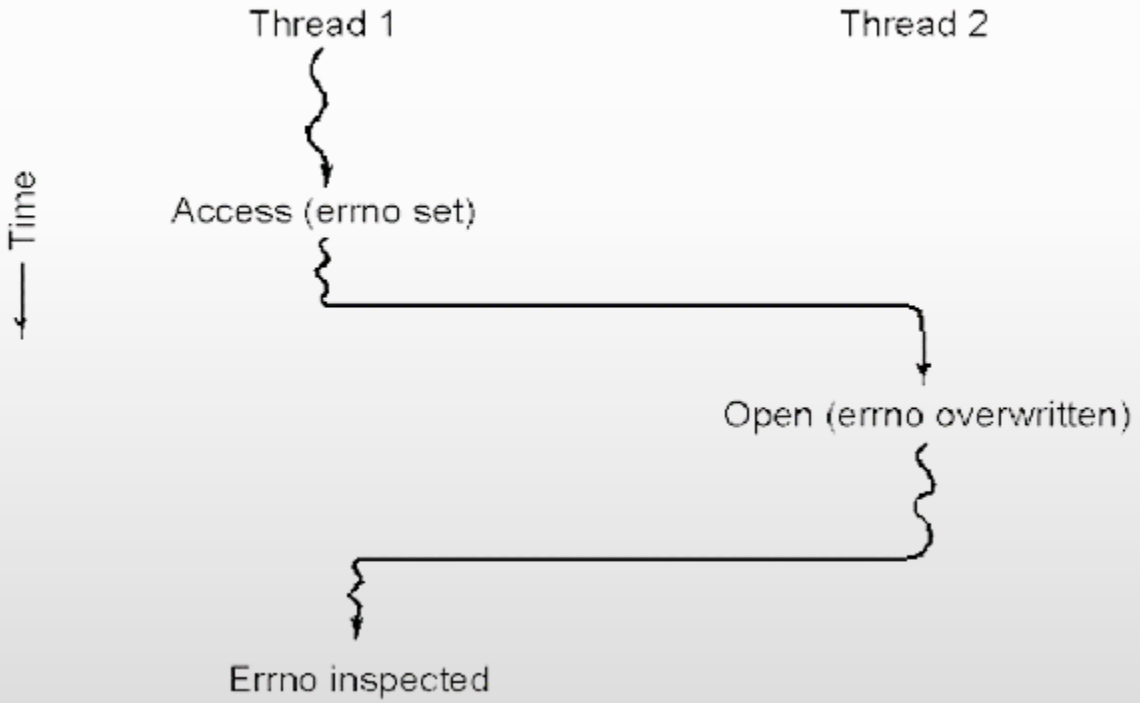
- Süreçler arasında düzen ve uyum sağlama sürecidir.
- Süreçler aynı anda yürütülebilir.
  - Bir süreç herhangi bir anda kesintiye uğrayabilir.
  - Yapılan iş yarım kalmış olabilir.
- Paylaşılan verilere eşzamanlı erişim, veri tutarsızlığına neden olabilir.
- Veri tutarlılığı için, süreçleri kontrol eden mekanizmalar gerekir.

Yaşanacak Problemler

- Yeniden girilemez (*not re-entrant*) kütüphane fonksiyonları.
  - Bir iş parçacığı aldığı mesajı tampon belleğe koyduğunda,
  - Yeni bir iş parçacığı bellekte mesajın üzerine yazabilir.
- Bellekten tahsis edilen yerler geçici olarak tutarsız durumda olabilir.
  - Yeni iş parçacığı yanlış işaretçi almış olabilir.
- İki veya daha fazla iş parçacığı birbirlerini bekleyebilir.
- Bir iş parçacığı diğer iş parçacıkları yüzünden hiç çalıştırılmayabilir.

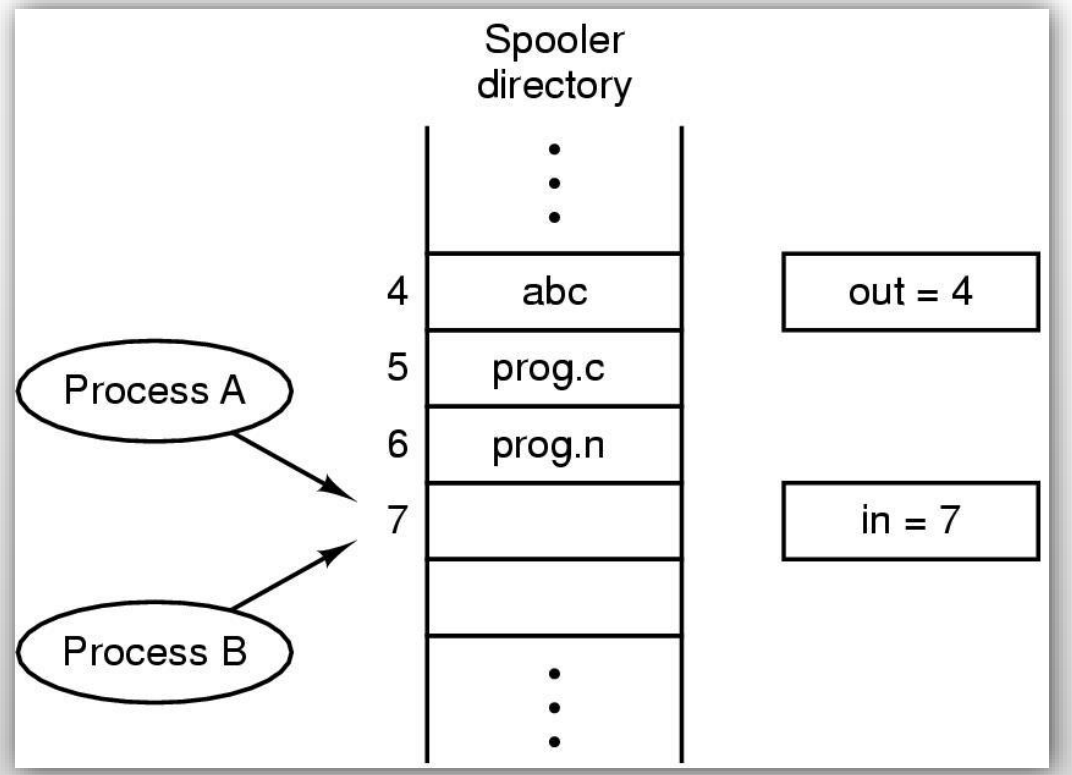
Veri Tutarsızlığı

- İş parçacıkları global bir değişkene kontrolsüz erişebilir.



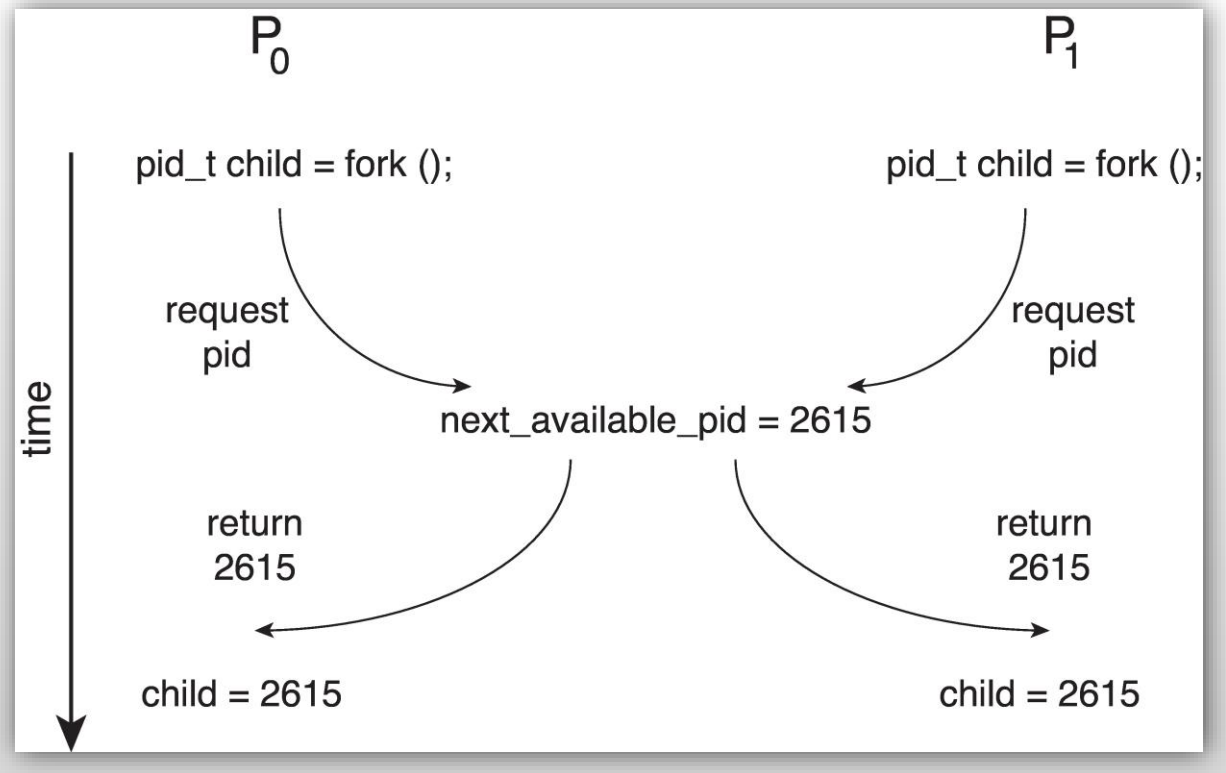
Süreçler Arası İletişim Problemleri

- **Yarış durumu**: iki süreç aynı bellek alanına aynı anda erişmek isterse.



Yarış Durumu

- P0 ve P1, *next_available_pid* değişkenine aynı anda erişirse, aynı pid iki farklı sürece atanabilir!



Kritik Bölge Problemi

- n adet süreç {p_0, p_1, … p_{n-1}} olsun.
- Her sürecin kritik kod bölgesi vardır.
- Süreç, bu bölgede
  - bir global değişkene değer atıyor,
  - bir tabloyu güncelliyor,
  - bir dosyaya yazıyor olabilir.
- Bir süreç kritik bölgede iken, diğerleri kritik bölgede olmamalı.
- Her süreç, kritik bölgeye girmek için izin istemeli.

Kritik Bölge Problemi

- **Karşılıklı Dışlama:** P süreci kritik bölgede yürütülürken, diğer süreçler kritik bölgede yürütülemez.
- **İlerleme:** Kritik bölgede yürütülen bir süreç yoksa ve kritik bölgeye girmek isteyen bir süreç varsa, bu süreç süresiz olarak beklememeli.
- **Sınırlı Bekleme:** Bir süreç, kritik bölgeye girmek istedikten sonra, diğer süreçler sınırlı bir sayıda kritik bölgeye girebilmeli.

Önerilen Çözümler

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

Kesmeleri Devre Dışı Bırakma

- Süreç, kritik bölgeye girmeden önce kesmeleri devre dışı bırakır, kritik bölgeye girer, kritik bölgeden çıktığında kesmeleri tekrar etkinleştirir.
- Problemler
  - Süreç, kesmeleri tekrar etkinleştiremezse sistem çöker.
  - Kesme devre dışı bırakıldığında, diğer süreçler CPU kullanamaz.
  - Çoklu çekirdekli sistemler için çözüm olmaz.
  - İşletim sisteminin kendisi için yararlı, ancak kullanıcılar için değil.
  - Kritik bölgeye giren süreç çok uzun sürebilir.
  - Bazı süreçlere hiç sıra gelmeyebilir (*starvation*).

Kilit Değişkeni

- Bir yazılım çözümü – Tüm süreçler bir kilidi paylaşır.
  - Kilit 0 ise, 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);
   /* kritik bölge */
   turn = j;
   /* devam */
}
```

Yarış Durumu

- Birden fazla süreç, aynı verilere eriştiğinde,
  - Nihai sonucun hangi sürecin ne zaman çalıştığına bağlı olması.
- **Karşılıklı dışlama**
  - Birden fazla sürecin aynı verilere aynı anda erişmesi engellenir.
- **Kritik bölge**
  - Programın ortak paylaşılan verilere erişim yaptığı kod bölümü.

Karşılıklı Dışlama

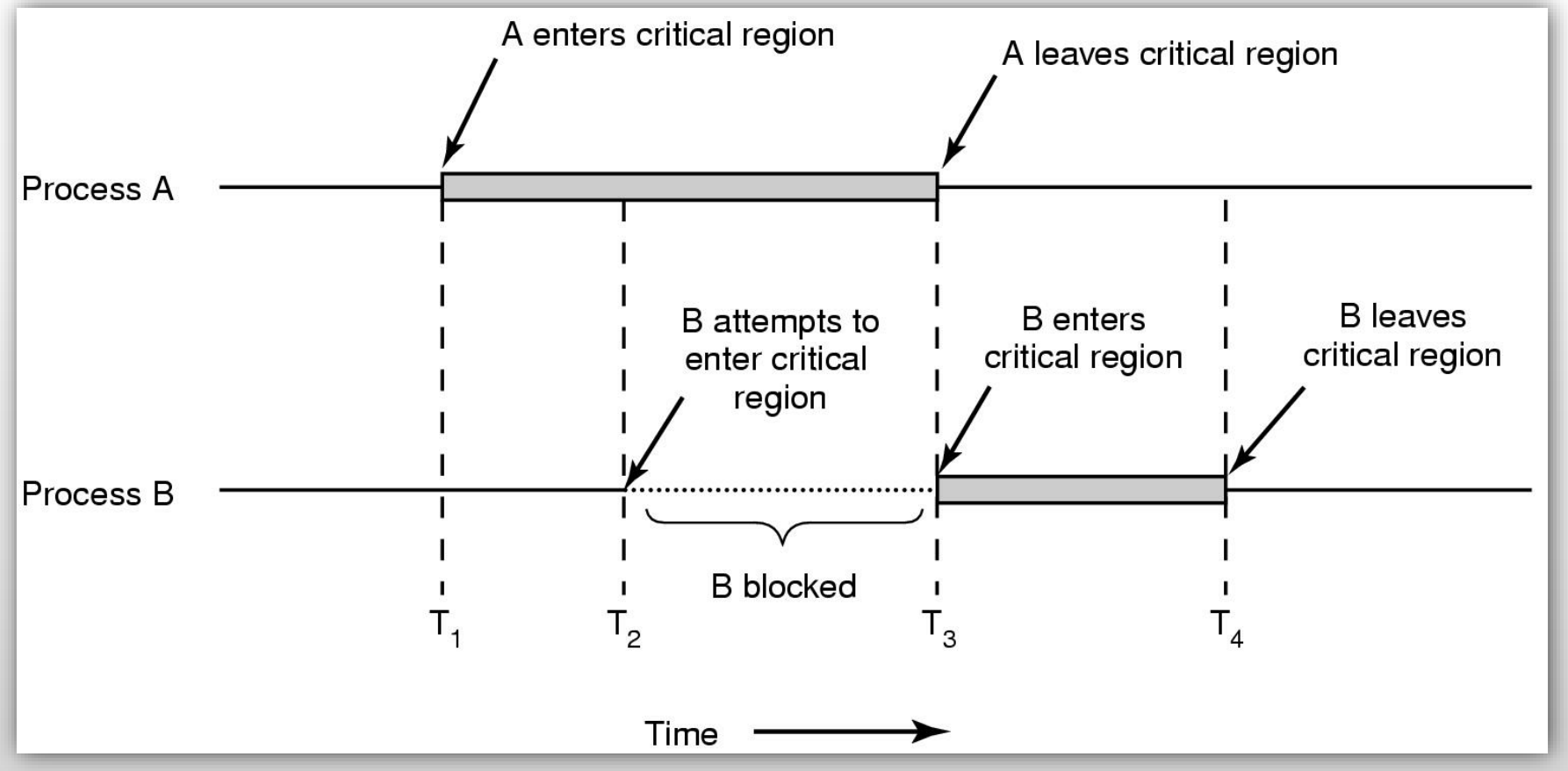
Karşılıklı dışlama için dört koşul;

- İki süreç aynı anda kritik bölgede olmamalı.
- İşlemci hızı ve sayısı hakkında varsayım yapılmamalı.
- Kritik bölgenin dışında çalışan bir süreç, diğerlerini engellememeli.
- Hiçbir süreç kritik bölgeye girmek için sonsuza dek beklememeli.

Kritik Bölge

```
do {
   /* giriş */
   /* kritik bölge */
   /* çıkış */
   /* devam */
} while (true);
```

Kritik Bölge Kullanarak Karşılıklı Dışlama



Sıkı Değişim (Strict Alternation)

- **Önce ben, sonra sen!**
- Tüm süreçler CPU'yu kullanabildiğinden 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

- Meşgul Bekleme (*Busy waiting*)
  - Bir değişken bir değere ulaşana kadar sürekli test etme.
- Döndürme Kilidi (*Spin lock*)
  - Meşgul beklemeyi kullanan bir kilit.

Peterson'un Çözümü

- Hangi sürecin kritik bölgeye gireceğini belirlemek için,
  - *turn* ve *flag[2]* olmak üzere iki değişken kullanır.
- *flag*, süreç tarafından kritik bölgeye girme niyetini belirtir.
- *turn*, sıradaki süreci belirtir.
- İki sürecin aynı anda kritik bölgeye girmesini önlemek için,
  - *Meşgul bekleme döngüsü* ve *bir dizi koşul* kullanır.
- Karşılıklı dışlamayı sağlar.
- Süreçlerin sonsuz bir bekleme döngüsüne girmesini engeller.
- Meşgul bekleme döngüsü yüksek miktarda CPU zamanı tüketir!

Peterson'un Çözümü

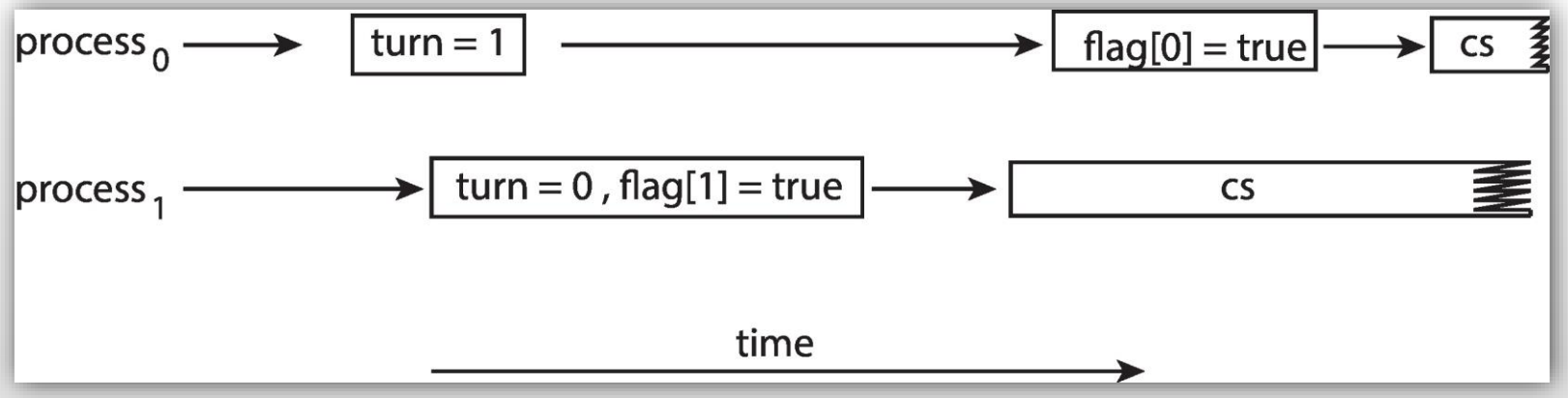
```
while (true) {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
   /* kritik bölge */
   flag[i] = false;
   /* devam */
}
```

Peterson'un Çözümü

```
final int N = 2; // Number of threads
volatile boolean[] flag = new boolean[N];
volatile int turn = 0;
int counter;
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
  counter++; // Critical region
  System.out.print("Counter:"+ counter + "i:"+ i + "j:" + j);
  flag[i] = false;
}
```

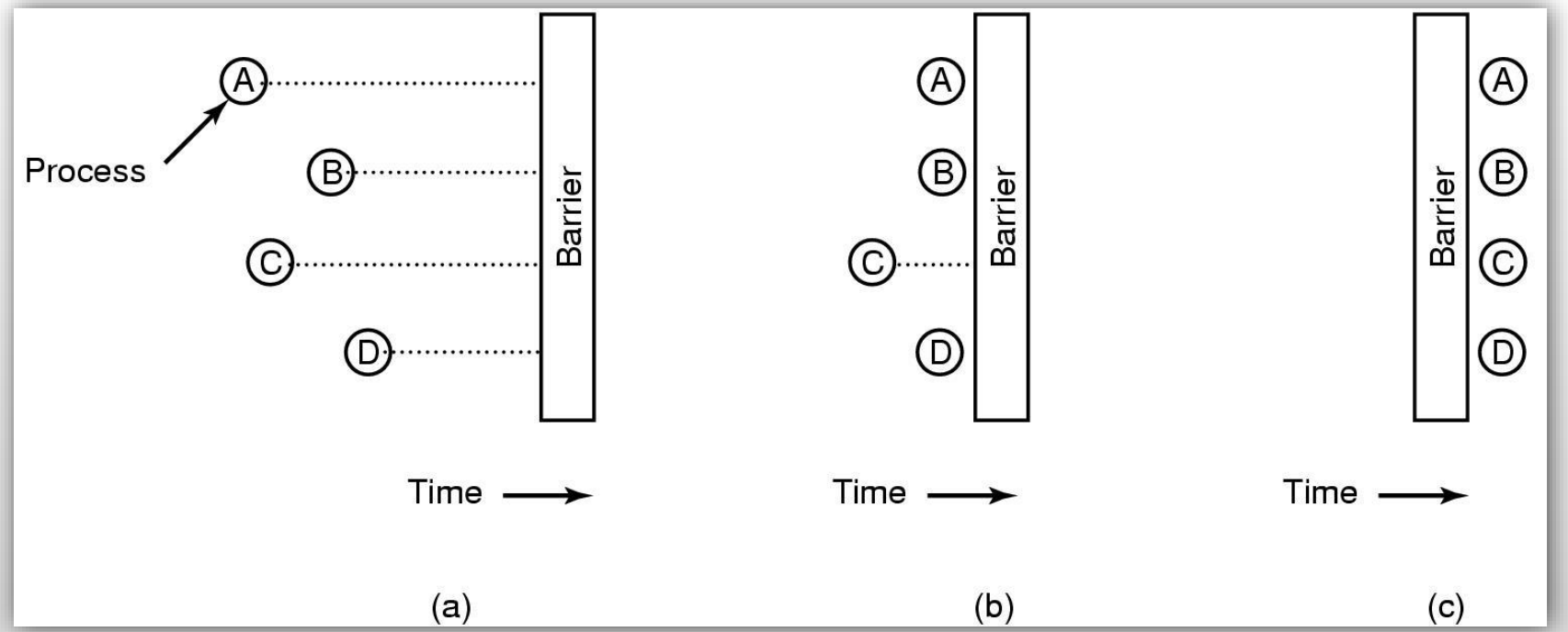
Modern Mimaride Peterson’un Çözümü

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



Bariyer (Barriers)

- Süreç gruplarını senkronize eder.
- Genellikle bilimsel hesaplamalarda kullanılır.



Bariyer (Barriers)

- Değişikliğin diğer süreçlere yayılmasını (görünür hale getirilmesini) zorlar.
- Sistem, 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'e atamanın *flag* atamasından önce olması garanti edilir.

Donanım Çözümleri

- Bir sözcüğün (*word*) içeriği test edilip değiştirilir, veya
- Sözcüğün içeriği atomik (*kesintisiz, tek adımda*) olarak değiştirilir.
- **Test Et ve Ata komutu** (*test-and-set*)
- **Karşılaştır ve Değiştir komutu** (*compare-and-swap*)

TSL Komutu

- **TSL** (*Test and Set Lock*),
  - Paylaşılan kaynaklara erişimi senkronize eder.
  - Basit ve verimli bir mekanizma sağlar.
  - Veri tutarsızlığı ve yarış koşulları riskini azaltır.
- Donanım düzeyinde atomik işlemler kullanarak hız ve verim sağlar.
- Aygıt sürücüleri gibi kritik bölgelere erişimi senkronize eder.
- 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*),
  - İki işlenenin (*operand*) içeriğini atomik olarak değiştirir,
  - Değişimin tek bir adımda tamamlanmasını sağlar.
- Kilit, semafor gibi senkronizasyon mekanizmalarını için kullanılır.
- Çok iş parçacıklı ortamda süreçler arası senkronizasyon için kullanılır.

XCHG Komutu

- XCHG A,B; a ve b değerlerini yer değiştirir.

```
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, düşük öncelikli süreci engeller.
  - Kilit'ten dolayı meşgul beklemede CPU'yu boşa harcar.
  - Düşük öncelikli süreç kritik bölge dışına çıkamaz.
  - Ölümcül kitlenmeye (*deadlock*) neden olabilir.
  - Öncelikleri değiştirme (*priority inversion*) çözüm olabilir.
- Meşgul beklemek yerine bloke etme,
  - Önce uyandır, sonra uyut (*wake up, sleep*).

Üretici Tüketici Problemi

- İki süreç,
  - Sabit boyutlu bir arabelleği paylaşır.
  - Üretici arabelleğe veri yazar.
  - Tüketici arabellekten veri okur.

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

- *count* değişkeni iki iş parçacığı tarafından paylaşılmakta.
- Eşzamanlılıktan kaynaklı sorun:
  - Tüketici *count* değişkenini 0 olarak okuduğunda,
  - Zamanında uykuya geçmediğinde,
  - Sinyal kaybolacaktır.

Semafor

- **Dijkstra** tarafından önerilen bir değişken türü.
- Atomik bir eylem, tek adımda tamamlanır, dolayısıyla bölünmez.
- **down – wait** (*P*)
  - Semafor değeri kontrol edilir,
    - 0 ise meşgul bekler,
    - Değilse değeri azaltır, devam eder.
- **up - signal** (*V*)
  - Semafor değeri arttırılır,
  - Semafor 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 (binary) semafor*).
- Karşılıklı dışlama ile senkronizasyon sağlar.

Semafor Kullanımı - Producer

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

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 mekanizması.
  - Kilidin mevcut olup olmadığını gösteren *Boolean* bir değişken.
- Kritik bölge nasıl korunur?
  - Önce bir kilit alınır (*acquire*).
  - Ardından kilit serbest bırakılır (*release*).
- *acquire*() ve *release*() çağrıları atomik olmalıdır.
  - *XCHG* gibi atomik donanım komutları kullanılır.
- Bu çözüm meşgul beklemeyi (**busy waiting**) gerektirir.
- Bu nedenle, **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 oluşturur.
Pthread_mutex_destroy	Mutex'i kaldırır.
Pthread_mutex_lock	Kilidi alır ya da bloke eder.
Pthread_mutex_trylock	Kilidi alır ya da hata verir.
Pthread_mutex_unlock	Kilidi kaldırır.

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;
void *producer(void *ptr) { /* produce data */
  for (int 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 (int 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 değişkenleri dikkatli kullanılmalı.
- Üretici-tüketici probleminde kullanılan *down* fonksiyonları,
  - Yer değiştirmesi halinde kitlenmeye (*deadlock*) neden olur.
- Gözleyici, karşılıklı dışlama ve bloke etme mekanizmasını kullanır.
  - Gruplandırılmış {prosedürler, veri yapıları ve değişkenlerden} oluşur.
- Bir süreç, gözleyicinin içindeki prosedürleri çağırabilir,
  - Ancak; içindeki öğelere doğrudan erişemez.
  - Aynı anda gözleyici içinde sadece bir prosedür çalışabilir.

Gözleyici

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

Gözleyici

- Karşılıklı dışlama, programcıya değil, derleyiciye bırakılır.
- *Gözleyicide* aynı anda yalnızca bir işlem olabilir.
  - Gözleyicide bir prosedür çağrıldığında,
  - İlk iş başka bir prosedürün çalışıp çalışmadığı kontrol edilir.
  - Bu durumda, çağrı askıya alınır.
- Bloke etme;
  - Koşul değişkenleri (*conditions*) kullanılarak.
  - Bekle, sinyal gönder işlemleri (*wait, signal*) kullanılarak.

Gözleyici

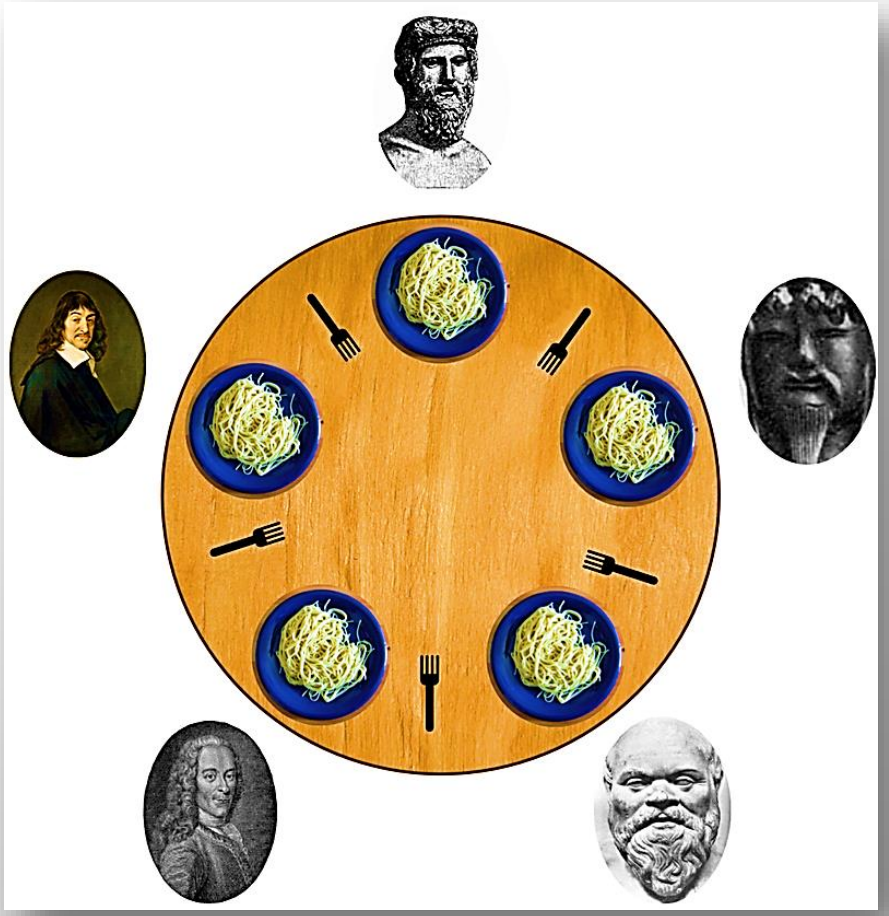
- Gözleyici, arabellek dolu olduğunda,
  - Bir koşul değişkeninde (dolu) sinyali yayınlayarak,
  - *Üretici* sürecin bloke olmasını sağlar.
  - *Tüketici* sürecin gözleyiciye girmesine izin verir.
  - Bu işlem, *üretici* süreci uyandıracak bir sinyal üretir.
  - Sinyali alan süreç, sinyali işler ve gözleyiciden çıkar.

Süreçler Arası İletişim Problemleri

- **Dining philosopher** problemi
  - Bir filozof ya yer, ya düşünür.
  - Acıkırsa, iki çatal alıp yemeye çalışır.
- **Okur-Yazar** problemi
  - Veritabanına erişimi modeller.

Dining Philosophers Problemi

- .



Dining Philosophers

```
while(true) {
  think(); // Initially, thinking
  // 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, 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, from 0 to N-1 */
  down(&mutex); /* enter critical region */
  state[i] = HUNGRY; /* 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, 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, 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' */
    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' */
    if (--rc == 0) up(&db); /* if this is the last reader ... */
    up(&mutex); /* release exclusive access to 'rc' */
    use_data_read(); /* noncritical region */
  }
}
```

SON