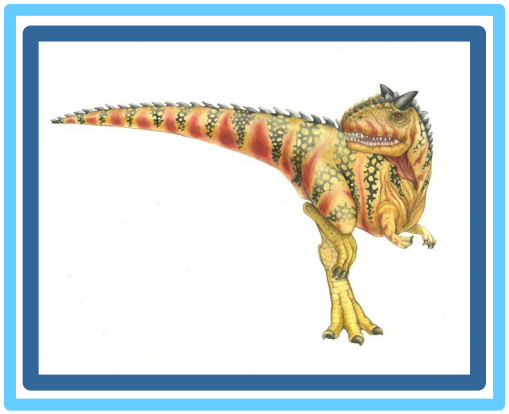
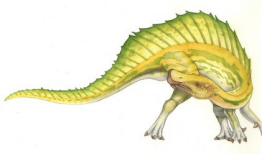
CENG322 İşletim Sistemleri Senkronizasyonu

Işıl ÖZ'ün notu:
Slaytlarımız A. Silberschatz, P.B. Galvin, G. Gagne'den uyarlanmıştır: İşletim Sistemi Kavramları kitap slaytları.



Arka plan

- Süreçler eşzamanlı olarak yürütülebilir
  - Yürütmeyi kısmen tamamlayarak herhangi bir zamanda kesintiye uğrayabilir
- Paylaşılan verilere eş zamanlı erişim veri tutarsızlığına neden olabilir
- Veri tutarlılığının korunması, işbirliği yapan süreçlerin düzenli bir şekilde yürütülmesini sağlayacak mekanizmalar gerektirir
- Sınırlı Tampon probleminde, üretici ve tüketici tarafından eş zamanlı olarak güncellenen ve yarış koşuluna yol açan bir sayaç kullanımı ile ilgili sorun



Yapımcı

```
while (true) {
    /* bir sonraki üretimde bir öğe üretir */

    while (sayaç == BUFFER_SIZE)
        ; /* hiçbir şey yapmayın
    */ buffer[in] =
    next_produced; in = (in + 1)
    % BUFFER_SIZE;
    karşı++;
}
```

Tüketici

```
while (true) {
    while (sayaç == 0)
        ; /* hiçbir şey yapmayın */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        karşı--;
    /* öğeyi bir sonraki tüketilende tüket */
}
```

Yarış Durumu

sayaç++ r1 = sayaç olarak

uygulanabilir

r1 = r1 + 1

sayaç = r1

sayaç-- r2 = sayaç olarak

uygulanabilir

r2 = r2 - 1

sayaç = r2

Başlangıçta "sayaç = 5" ile bu yürütme serpiştirmesini düşünün:

S0: üretici yürütme r1 = sayaç (yükleme) {r1 = 5}

S1: üretici r1 = r1 + 1 yürütür {r1 = 6}

S2: tüketici yürütme r2 = sayaç (sakla) {r2 = 5}

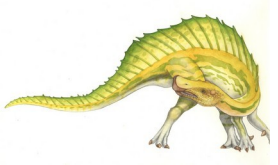
S3: tüketici r2 = r2 - 1 yürütür {r2 = 4}

S4: üretici yürütme sayacı = r1 {sayı = 6 }

S5: tüketici yürütme sayacı = r2 {sayı = 4}

Kritik Kesit Sorunu

- ***n*** süreçten oluşan bir sistem düşünün ***{p0, p1, ...*** pn-1}
- Her sürecin **kritik** kod bölümü vardır
  - İşlem ortak değişkenleri değiştirmek, tabloyu güncellemek, dosya yazmak vb. olabilir.
  - Bir süreç kritik bölümdeyken, başka hiçbir süreç onun kritik bölümünde olamaz
- ***Kritik bölüm problemi,*** bunu çözmek için protokol tasarlamaktır
- Her süreç **giriş bölümünde** kritik bölüme girmek için izin istemelidir, kritik bölümü **çıkış bölümü ve** ardından **kalan bölümü** takip edebilir



Kritik Bölüm

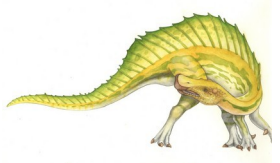
- P_i sürecinin genel yapısı

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

Kritik Kesit Problemi (Devam)

Kritik bölüm sorununun çözümü için gerekenler

1. **Karşılıklı Dışlama** - P_i süreci kendi kritik bölümünde çalışıyorsa, başka hiçbir süreç kendi kritik bölümlerinde çalışamaz
   - Kritik bölümde aynı anda yalnızca bir iş parçacığı
2. **İlerleme** - Kritik bölümünde hiçbir süreç yürütülmüyorsa ve kritik bölümüne girmek isteyen bazı süreçler varsa, kritik bölüme bir sonraki girecek sürecin seçimi süresiz olarak ertelenemez
   - Aynı anda birden fazla talep varsa, birinin devam etmesine izin verilmelidir
3. **Sınırlı Bekleme** - Bir süreç kendi kritik bölümüne girmek için talepte bulunduktan sonra ve bu talep kabul edilmeden önce diğer süreçlerin kendi kritik bölümlerine girmelerine izin verilen sayı için bir sınır olmalıdır
   - Sonunda her bekleyen iş parçacığının girmesine izin vermelidir

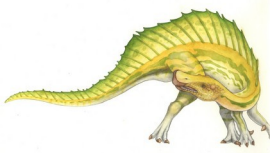


Kritik Bölüm Örneği

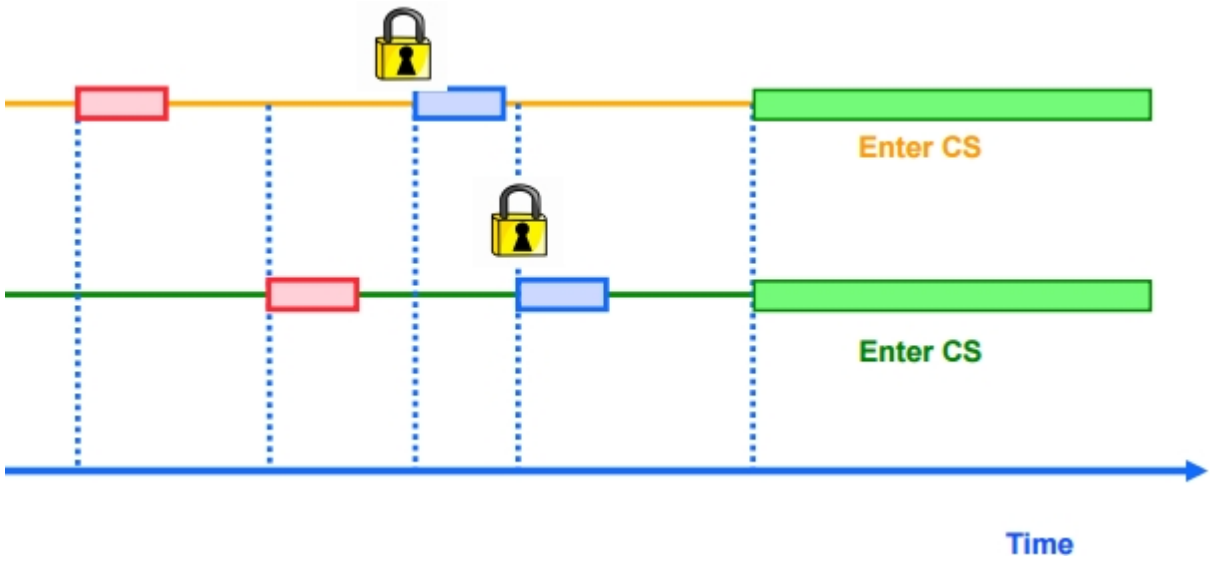
- İki iş parçacığı bellekteki hesap bakiyesini paylaşır
- Her biri ortak kodu çalıştırır, deposit()
- void deposit (int amount) {
- bakiye = bakiye + tutar;
- }
- Montaj talimatları dizisine derleyin
- yük R1, denge
- R1, miktar ekleyin
- R1 deposu, bakiye

Deneme 1: Paylaşılan Kilit Değişkeni

```
boolean lock = false; // paylaşılan

değişken void deposit(int amount) {
    while (lock == true) {} /* bekle */
    lock = true; /* kilidi alır */ balance
    += amount; // kritik bölüm lock =
    false; /* kilidi serbest bırak */
 }
```

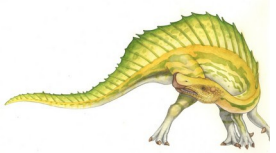


Deneme 1: Karşılıklı Dışlama




```
boolean lock = false; // shared variable
void deposit(int amount)
  {
  while( lock == true ) {} /* wait */ ;
  lock = true;
  balance += amount; // critical section
  lock = false;
  }
```

Başarısız oldu çünkü iki iş parçacığı kilit değişkenini aynı anda okudu ve her ikisi de sıranın kendisinde olduğunu düşündü
kritik bölüm Sırayla!



Deneme 2: Alternatif 2 İplik

```
int turn = 0; // shared

void deposit(int amount) {
    while (turn == 1-tid) {} /* bekle */
    bakiye += miktar; // kritik bölüm dönüş
    = 1-tid;
}
```

Deneme 2: İlerleme

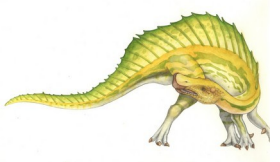

```
int turn = 0; // shared variable
void deposit( int amount )
  {
  while( turn == 1-tid ) {} /* wait */ ;

  balance += amount; // critical section

  turn = 1-tid;
  }
```

Diğer iş parçacığının CS ile ilgilenip ilgilenmediğini bilmemiz gerekir ve ilgilenmeyen bir iş parçacığını beklememeliyiz

Başkalarının ihtiyaçlarını bilmemiz gerekir!

Deneme 3: Diğerlerini Kontrol Edin

```
boolean lock[2] = {false, false}; // shared

void deposit(int amount) {
    lock[tid] = true;
    while (lock[1-tid]) {} /* bekle */
   bakiye += miktar; // kritik bÖlÜm
    lock[tid] = false;
}
```



Deneme 3: Sınırlandırılmış Bekleme

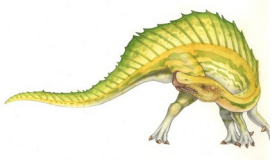
```
boolean lock[2] = {false, false} // shared
void deposit( int amount )
  {
  lock[tid] = true;

  while( lock[1-tid] == true ) {} /* wait */;

  balance += amount; // critical section

  lock[tid] = false;
  }
```

Her iş parçacığı diğerini bekler

Çıkmaz!

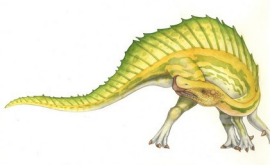
Peterson'ın Algoritması

```
int turn = 0; // paylaşılan
boolean lock[2] = {false, false}; // paylaşılan

void deposit(int amount) {
    lock[tid] = true;
    turn = 1-tid;
    while (lock[1-tid] && turn == 1-tid) {}/* bekle */
    balance += amount; // kritik bÖlÜm
    lock[tid] = false;
}
```



Peterson'ın Algoritma Sezgisi

- Karşılıklı hariç tutma: Sadece ve sadece aşağıdaki durumlarda kritik bölüme girin
  - Diğer konu girmek istemiyor
  - Diğer konu girmek istiyor, ama sıra sizde
- İlerleme kaydedildi: Her iki iş parçacığı da while() döngüsünde sonsuza kadar bekleyemez
  - Diğer süreç girmek istemezse tamamlanır
  - Diğer süreç (eşleştirme sırası) sonunda bitecektir
- Sınırlandırılmış bekleme
  - Her süreç en fazla bir kritik bölüm bekler

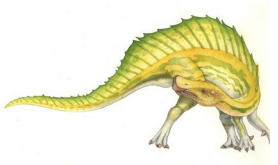


Peterson Algoritması (*Pi İşlem*i)

```
while (true){

    bayrak[i] = true;
    dönüş = j;
    while (bayrak[j] && dönüş = =
    j)
       ;

        /* kritik bölüm */

    bayrak[i] = yanlış;

    /* kalan bölüm */

}
```

Peterson'ın Çözümü ve Modern Mimari

- Bir algoritmayı göstermek için yararlı olsa da, Peterson'ın Çözümünün modern mimarilerde çalışması garanti edilmez.
  - Performansı artırmak için, işlemciler ve/veya derleyiciler bağımlılıkları olmayan işlemleri yeniden sıralayabilir
- Neden çalışmayacağını anlamak, yarış koşullarını daha iyi anlamak için yararlıdır.
- Tek iş parçacıklı için sonuç her zaman aynı olacağından bu sorun değildir.
- Çok iş parçacıklı için yeniden sıralama tutarsız veya beklenmedik sonuçlar üretebilir!



Modern Mimari Örneği

- İki iş parçacığı verileri paylaşır:

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

- Konu 1 gerçekleştirir

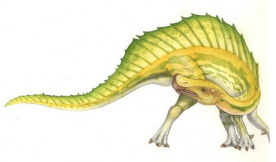
```
while (!flag)
 ;
x yazdır
```

- Konu 2 gerçekleştirir

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

- Beklenen çıktı nedir?

100



Modern Mimari Örneği (Devam)

- Ancak, `flag` ve `x` değişkenleri birbirinden bağımsız olduğundan, talimatlar:

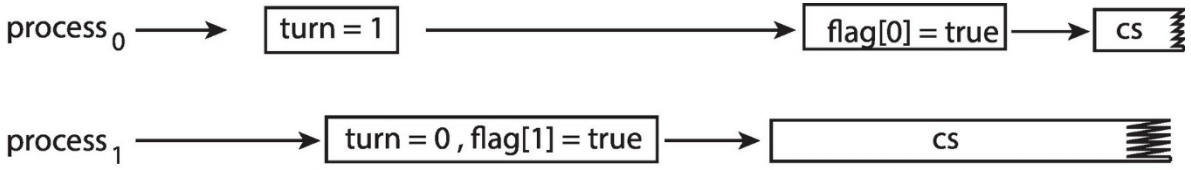
```
flag = true;
x = 100;
```

İplik 2 için yeniden sıralanabilir

- Bu gerçekleşirse, çıkış 0 olabilir!

Peterson'ın Çözümü Tekrar Ziyaret Edildi

- Peterson'ın Çözümünde talimatların yeniden sıralanmasının etkileri



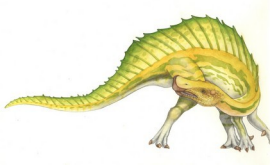

```
while (true){

      bayrak[i] = true;
      dönüş = j;
      while (bayrak[j] && dönüş = = j)
        ;

         /* kritik bölüm */

      flag[i] = false;

      /* kalan bölüm */

}
```

Bu, her iki sürecin de aynı anda kritik bölümlerinde olmasını sağlar!

- Peterson'ın çözümünün modern bilgisayar mimarisinde doğru çalışmasını sağlamak için **Bellek Bariyeri** kullanmalıyız.

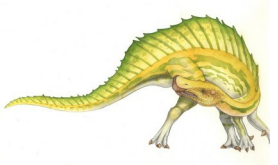
Hafıza Bariyeri

- **Bellek modeli,** bir bilgisayar mimarisinin uygulama programlarına verdiği bellek garantileridir.
- Bellek modelleri şunlardan biri olabilir:
  - **Güçlü sıralı** - bir işlemcinin bellek modifikasyonunun diğer tüm işlemciler tarafından hemen görülebildiği durum.
  - **Zayıf sıralı** - bir işlemcinin bellek modifikasyonu diğer tüm işlemciler tarafından hemen görülemeyebilir.
- **Bellek bariyeri,** bellekteki herhangi bir değişikliğin diğer tüm işlemcilere yayılmasını (görünür hale gelmesini) zorlayan bir talimattır.



Hafıza Bariyeri Talimatları

- Bir bellek bariyeri talimatı gerçekleştirildiğinde, sistem sonraki herhangi bir yükleme veya saklama işlemi gerçekleştirilmeden önce tüm yükleme ve saklama işlemlerinin tamamlanmasını sağlar.
- Bu nedenle, talimatlar yeniden sıralansa bile, bellek bariyeri, depolama işlemlerinin bellekte tamamlanmasını ve gelecekteki yükleme veya depolama işlemleri gerçekleştirilmeden önce diğer işlemciler tarafından görülebilmesini sağlar.



Bellek Bariyeri Örneği

- İş Parçacığı 1'in 100 çıktısı vermesini sağlamak için aşağıdaki talimatlara bir bellek engeli ekleyebiliriz:
- Konu 1 şimdi gerçekleştiriliyor

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

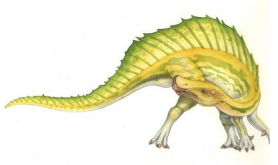
- Konu 2 şimdi gerçekleştiriliyor

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

- Thread 1 için `flag` değerinin `x değerinden` önce yükleneceği garanti edilir.
- Thread 2 için `x'`e yapılan atamanın atama `bayrağından` önce gerçekleşmesini sağlarız.

Donanım Talimatları

- Bir sözcüğün içeriğini *test etmemize ve değiştirmemize* ya da iki sözcüğün içeriğini atomik olarak (kesintisiz) *değiştirmemize* olanak *tanıyan* özel donanım talimatları
  - **Test Et ve Ayarla** talimatı
  - **Karşılaştır ve Değiştir** talimatı

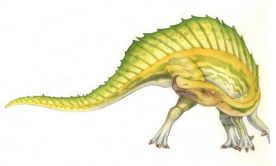


test_and_set Talimatı

- Tanım

```
boolean test_and_set (boolean *target)
   {
          boolean rv = *hedef;
          *hedef = true;
          return rv:
   }
```

- Özellikler
  - Atomik olarak yürütülür
  - Geçilen parametrenin orijinal değerini döndürür
  - Geçirilen parametrenin yeni değerini `true` olarak ayarlar



Çözüm test_and_set() fonksiyonunu kullanma

- Paylaşılan boolean değişkeni **`lock`**, **`false`** olarak başlatılır
- Çözüm:

```
yap {
        while (test_and_set(&lock))
        ; /* hiÇbir Şey yapmayın */

            /* kritik bÖlÜm */ lock =

   false;
                /* kalan bÖlÜm */
 } while (true);
```

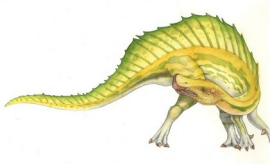
- Karşılıklı dışlama sorununu çözer.

compare_and_swap Komutu

- Tanım

```
int compare_and_swap(int *value, int expected, int new_value)
 {
   int temp = *değer;
   eğer (*değer == beklenen)
       *değer = yeni_değer;
    return temp;
 }
```

- Mülkler
  - Atomik olarak yürütülür
  - Geçilen parametre **değerinin** orijinal değerini döndürür
  - Değişken **değerini** aktarılan **yeni_değer** parametresinin değerine ayarlar, ancak yalnızca ***değer == beklenen** doğruysa. Yani, takas yalnızca bu koşul altında gerçekleşir.



compare_and_swap kullanarak çözüm

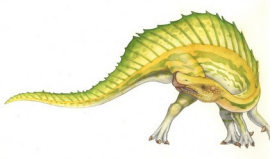
- Paylaşılan tamsayı **`kilidi`** 0 olarak başlatıldı;
- Çözüm:

```
while (true){
        while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* hiÇbir Şey yapmayın */

        /* kritik bÖlÜm */ lock

        = 0;

    /* kalan bÖlÜm */
}
```

- Karşılıklı dışlama sorununu çözer.

Karşılaştır ve değiştir ile sınırlı bekleme

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* kritik bölüm */ j =
   (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   eğer (j == i)
      kilit = 0;
   başka
      waiting[j] = false;
   /* kalan bölüm */
}
```



Atomik Değişkenler

- Tipik olarak, karşılaştırma ve takas gibi talimatlar diğer senkronizasyon araçları için yapı taşı olarak kullanılır.
- Bir araç, tamsayılar ve booleanlar gibi temel veri türlerinde **atomik** (kesintisiz) güncellemeler sağlayan bir *atomik* **değişkendir**.
- Örneğin:
  - **`Sekans`** atomik bir değişken olsun
  - **`increment()`** atomik değişken üzerinde bir işlem olsun **`sıra`**
  - Komuta:

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

`dizinin` kesintisiz olarak artırılmasını sağlar:

Atomik Değişkenler

- `increment()` işlevi aşağıdaki gibi uygulanabilir:

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