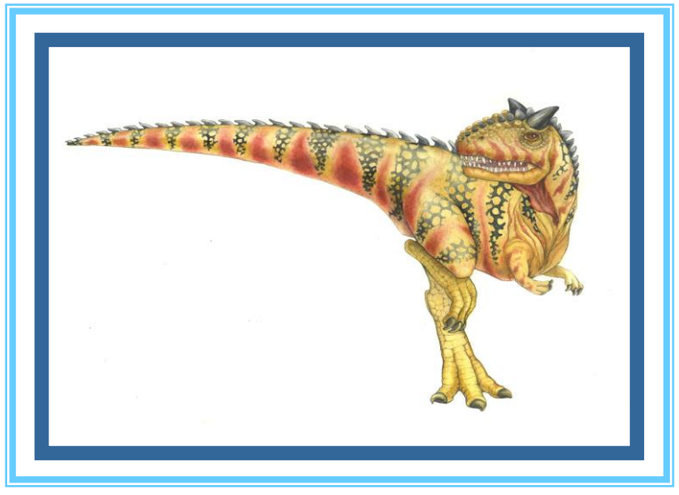
Bölüm 6: Proses Senkronizasyonu



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

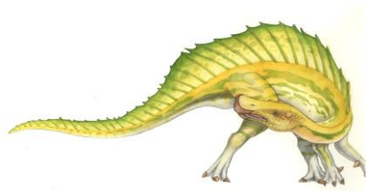
- Proses Senkronizasyonu kavramını sunmak
- Paylaşılan verinin tutarlılığını sağlamak için kullanılabilecek Kritik Bölge Problemi çözümlerini tanıtmak
- Kritik Bölge Problemlerine ilişkin yazılımsal ve donanımsal çözümleri sunmak
- Birkaç klasik proses senkronizasyonu problemlerini incelemek
- Atomik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

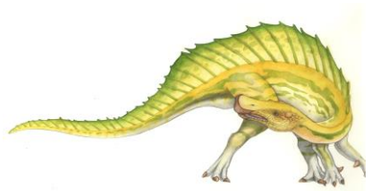
- Paylaşılan verilere eşzamanlı erişim veya paralel olarak erişim, veri tutarsızlıklarına neden olabilir.

- 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.

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 ilerlemelidir.

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

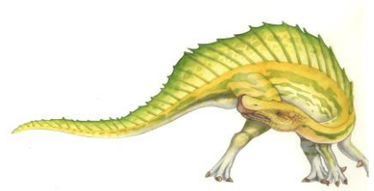
- Kesintisiz çekirdekte prosesler CPU'da işlemlerini gerçekleştirirken herhangi bir kesintiye uğramadan sonuna kadar devam ettirirler. Bir proses işini bitirdikten sonra CPU'dan ayrılır. Buda aslında bu durumda yarış durumunun engellenmesi demektir.

- 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`**)

Donanım Senkronizasyonu

- Bir çok sistem kritik bölge kodu için donanım desteği sağlar.
- Aşağıdaki yaklaşımlarda **kilitleme** fikri esas alınmıştır
- 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.
- Modern makineler özel atomik donanım komutlarını destekler.
    - Atomic = kesintisiz (non-interruptable)
  - Test and Run
  - Compare and Swap

Kilitlenme ile Kritik Bölge Probleminin Çözümü

```
do {
    kilitle
            kritik bölge
    kilidi aç
            geri kalan kısım
} while (TRUE);
```

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 kilitli, 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);
```

TestandSet() ile Sınırlı Beklemeli Karşılıklı Dışlama

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // kritik bölge
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // geri kalan kısım
} 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 ($i + 1, i + 2, \ldots, n - 1, 0, \ldots, i - 1$).

MUTEX kilidi

- Donanımsal çözümler karmaşıktır ve genellikle uygulama programcıları tarafından erişilememektedir.
- İşletim Sistemi tasarımcıları, kritik bölge problemini çözmek için yazılım araçları oluştururlar
- 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
- 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);
```