İşlem Senkronizasyonu

Dr. Günay TEMÜR

Bölüm 6: İşlem Senkronizasyonu

- Kritik-kısım Problemi
- Peterson Çözümü
- Senkronizasyon Donanımı
- Semaforlar
- Senkronizasyonun Klasik Problemleri
- Monitörler
- Senkronizasyon Örnekleri
- Atomik İşlemler

BackGround

- Paylaşılan veriye aynı anda erişim veride tutarsızlıklara neden olabilir
- Verinin tutarlılığını korumak, veriye ortak erişen işlemlerin veriye erişimlerini sıraya sokan bir mekanizmayı gerektirir
- Üretici-tüketici probleminde tüm tampon belleği dolduracak bir çözüm sunmak istediğimizi varsayalım
- Bunu dolu bellek hücrelerini saymakta kullanacağımız count adında bir tamsayı sayaç ile sağlayabiliriz
  - Başlangıçta count sıfıra eşitlenecektir
  - Üretici yeni bir tampon bellek hücresini doldurduğunda count bir artacaktır
  - Tüketici bir tampon bellek hücresindeki veriyi tükettiğinde ise count bir azalacaktır

Üretici

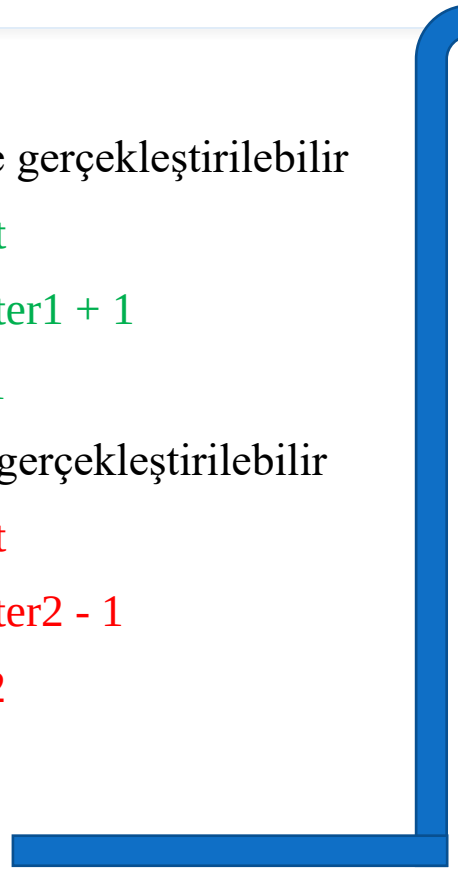
```
while (count == BUFFER.SIZE); // do nothing

// add an item to the buffer

buffer[in] = item;

in = (in + 1) % BUFFER.SIZE;

++count;
```

Tüketici

```
while (count == 0); // do nothing
// remove an item from the
buffer item = buffer[out];
out = (out + 1) % BUFFER.SIZE;
--count;
```

Yarışma Durumu (Race Condition)

- count++ şu şekilde gerçekleştirilebilir
  - register1 = count
  - register1 = register1 + 1
  - count = register1
- count-- şu şekilde gerçekleştirilebilir
  - register2 = count
  - register2 = register2 - 1
  - count = register2



- "count = 5" iken aşağıdaki işlemlerin gerçekleştiğini varsayın:
  - T0: üreticinin çalıştırdığı kod: register1 = count {register1 = 5}
  - T1: üreticinin çalıştırdığı kod: register1 = register1 + 1 {register1 = 6}
  - T2: tüketicinin çalıştırdığı kod: register2 = count {register2 = 5}
  - T3: tüketicinin çalıştırdığı kod: register2 = register2 - 1 {register2 = 4}
  - T4: üreticinin çalıştırdığı kod: count = register1 {count = 6 }
  - T5: tüketicinin çalıştırdığı kod: count = register2 {count = 4}
  - Alternatif:
  - T4: tüketicinin çalıştırdığı kod: count = register2 {count = 4}
  - T5: üreticinin çalıştırdığı kod: count = register1 {count = 6 }

Yarışma Durumu - Tanım

- Pek çok işlemin aynı anda bir veriye erişmek ve onu değiştirmek istediği durumlarda işlemlerin çalışması sonucu elde edilen sonucun işlemlerin veriye eriştiği sıraya bağlı olduğu durumlara yarışma durumları (race condition) denir.
- Yarışma durumunda tutarlı sonuç elde etmek için, count değişkenine aynı anda sadece bir işlemin erişmesini sağlamalıyız. Bu da işlemlerin senkronizasyonu ile mümkündür.
- Yarışma durumları işletim sistemlerinde çok karşılaşılan bir durumdur.
- Bunun nedeni kaynakların (örn: hafıza, I/O cihazları) pek çok bileşen tarafından paylaşılıyor olmasıdır.
- Çok çekirdekli işlemcilerin ve iş parçacıklarının kullanımı da durumu giderek daha karmaşık hale getirmektedir.

Kritik - Kısım Problemi

- n tane işlemin olduğu bir sistem düşünelim
- Her bir işlemin bir kısım kodunun aşağıdaki işlemlerden birini yapan bir kritik-kısma sahip olduğunu düşünün
  - Ortak bir değişkenin değerini değiştiren
  - Ortak bir tabloyu güncelleyen
  - Ortak kullanılan bir dosyayı güncelleyen
- Böyle bir sistemin tutarlı sonuç üretmesi için kritik - kısma erişimi, aynı anda bir işlemin erişebileceği şekilde sınırlandırmalıyız

Tipik Bir İşlemin Yapısı

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

```
while (true) {
        while (turn == j);
                critical section
        turn = j;
                remainder section
}
```

Kritik – Kısım Problemine Çözüm

❑ Kritik-kısım problemine önerilen çözüm aşağıdaki kriterleri sağlamalıdır:

1. Karşılıklı Dışlama (mutual exclusion) – Eğer işlem P_i kritik kısımda çalışıyorsa, diğer işlemler kritik kısımda çalışamaz.
2. İlerleme (progress) – Eğer kritik kısımda çalışan bir işlem yoksa ve bazı işlemler kritik kısımda çalışmak istiyorsa, bu işlemlerden birini seçip çalıştırmak sonsuza kadar ertelenmemelidir.
3. Sınırlı Bekleme (bounded waiting) - Kritik kısma girmek isteyen bir işlemin bekleme süresi sınırlandırılmalıdır. O işlem beklerken, diğer işlemlerden en fazla belirlenen sayıda işlem kritik kısma girmelidir. Ardından bekleyen işlemin kritik kısma girmesine izin verilmelidir.

❑ Her bir işlemin sıfır dışında bir hızda çalıştığı varsayılmaktadır

❑ Bu N işlemin bağıl hızları hakkında herhangi bir varsayımımız yoktur

Peterson’un Çözümü

- İki işlem çözümü
- LOAD ve STORE komutlarının atomik olduğunu varsayın. (Atomik = kesilmeyen (non-interruptable))
- Bu iki işlem iki değişken paylaşır:
  - int turn;
  - boolean flag[2]
- turn değişkeni kritik kısma giriş sırasının kimde olduğunu belirtiyor
- flag dizisi bir işlemin kritik kısma girişe hazır olup olmadığını belirtiyor.

  flag[i] = true P_i işleminin hazır olduğunu gösteriyor.

P_i işlemi için Algoritma

```
while (true) {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = false;
                remainder section
}
```

Dikkat: j = 1 - i

Senkronizasyon Donanımı

- Kritik-kısım problemi için pek çok sistem, donanım desteği sunmaktadır
- Tek işlemcili sistemler – geçici olarak kesintileri (interrupts) iptal edebilirler.
  - O an çalışan kod, bölünmeden çalışmaya devam edebilir.
- Genel olarak çok işlemcili bilgisayarlarda verimli değildir - işlemciler arasında mesajlaşma gerektirir.
  - Bu özelliği kullanan işletim sistemleri ölçeklenebilir değildir.
- Modern makinalar özel atomik donanım komutları sağlarlar (Atomik = kesilmeyen (non-interruptable))
  - Hafıza hücresinin değerini değiştirir veya test eder
  - Veya iki hafıza hücresini değiş tokuş eder

Kilitleri Kullanarak Kritik-Kısım Problemi Çözümü

- Yazılım tabanlı çözümlerin (Peterson'un algoritması gibi) modern bilgisayar mimarilerinde çalışmasının garantisi yoktur.
- Genel olarak kritik-kısım problemini çözmek için küçük bir araca ihtiyacımız olduğunu söyleyebiliriz: kilit (lock)

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

Donanım Çözümleri için Veri Yapısı

```
public class HardwareData
{
    private boolean value = false;

    public HardwareData(boolean value) {
      this.value = value;
    }

    public boolean get() {
      return value;
    }

    public void set(boolean newValue) {
      value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
      boolean oldValue = this.get();
      this.set(newValue);

      return oldValue;
    }

   public void swap(HardwareData other) {
     boolean temp = this.get();

     this.set(other.get());
     other.set(temp);
    }
}
```

Get and Set Komutu ile Çözüm

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
    while (lock.getAndSet(true))
       Thread.yield();

    // critical section
    lock.set(false);
    // remainder section
}
```

Swap Komut ile Çözüm

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
       lock.swap(key);
    }
    while (key.get() == true);

    // critical section
    lock.set(false);
    // remainder section
}
```

Semafor (Semaphore)

- Meşgul bekleme (busy waiting) gerektirmeyen senkronizasyon aracı
- Semafor S – tamsayı değişken
- S üzerinde iki standart işlem : acquire()- (wait()) ve release() (signal())
  - Orijinal olarak P() ve V()
- Daha az karmaşık
- Sadece iki atomik işlem ile erişilebiliyor

```
acquire() {
        while (value <= 0); // no-op
        value--;
}
        release(){
                value++;
        }
```

Semafor Genel Senkronizasyon Aracı

- Sayaç semaforu (counting semaphore) – tamsayı değeri sınırsız bir değer aralığına sahiptir.
- İkili semafor (binary semaphore) – tamsayı değeri sadece 0 ya da 1 değerlerini alabilir; gerçekleştirimi daha basit olabilir.
  - mutex lock olarak da bilinir:

```
Semaphore sem = new Semaphore(1);
sem.acquire();
        //critical section
sem.release();
        //remainder section
```

Semafor Gerçekleştirimi

- acquire() ve release() komutlarını iki ayrı işlemin aynı anda çalıştırması engellenmelidir.
- Mevcut gerçekleştirim meşgul bekleme (busy waiting) yapıyor
- Uygulamaların kritik kısımda çok fazla zaman harcayabileceğine dikkat edin
- Böyle bir durumda meşgul bekleme yapan semafor uygun bir gerçekleştirim değildir

Meşgul Bekleme Yapmayan Semafor Gerçekleştirimi (1)

- Her bir semafor ile bir bekleme listesi ilişkilendirilir
- Bekleme listesindeki her bir kayıt aşağıdaki verileri içerir:
  - değer (tamsayı tipinde)
  - listedeki sonraki kayıda işaretçi
- İki işlem:
  - block – bu komutu çalıştıran işlemi uygun bekleme listesine yerleştirir
  - wakeup – bekleme listesinde bulunan bir işlemi listeden siler ve bekleme kuyruğuna (ready queue) yerleştirir

Meşgul Bekleme Yapmayan Semafor Gerçekleştirimi (2)

- acquire() gerçekleştirimi:

```
acquire() {
        value--;
        if (value < 0) {
                add this process to list
                block;
        }
}
```

- release() gerçekleştirimi

```
release() {
        value++;
        if (value <= 0) {
                remove a process P from list
                wekaup(P);
        }
}
```

Deadlock and Starvation

- Kilitlenme (Deadlock) – iki veya daha fazla işlem, sadece bekleyen bir işlemin neden olabileceği bir olayı sonsuza kadar bekliyor.
- S ve Q ilk değeri 1 olarak belirlenen iki semafor

P_0
```
S.acquire();
Q.acquire();
      .
      .
      .
S.release();
Q.release();
```

P_1
```
Q.acquire();
S.acquire();
      .
      .
      .
Q.release();
S.release();
```

- Açlık (Starvation) – sınırsız bloklanma. Semafor bekleme listesinde bekleyen bir işlemin hiçbir zaman listeden silinmemesi.

Klasik Senkronizasyon Problemleri

- Sınırlı Tampon Bellek Problemi (Bounded-Buffer Problem)
- Okuyucular -Yazıcılar Problemi (Readers-Writers Problem)
- Yemek Yiyen Filozoflar Problemi (Dining-Philosophers Problem)

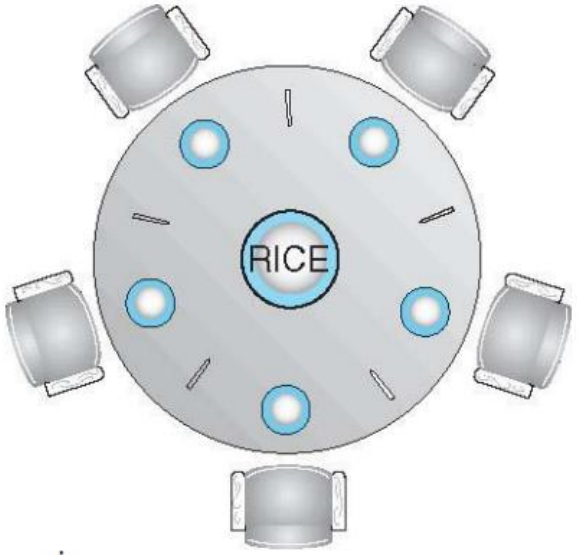
Sınırlı Tampon Bellek Problemi

- N tampon bellek, her biri bir şey tutabiliyor
- mutex semaforu, başlangıç değeri 1 – karşılıklı dışlamayı (mutual exclusion) sağlıyor
- full semaforu, başlangıç değeri 0 – dolu tampon belleklerin sayısını takıp ediyor
- empty semaforu, başlangıç değeri N – boş tampon belleklerin sayısını takip ediyor

Okuyucular-Yazıcılar Problemi (1)

- Bir veri kümesi, aynı anda çalışan birden fazla işlem arasında paylaştırılıyor
  - Okuyucular – sadece veriyi okuyorlar, veriyi güncellemiyorlar
  - Yazıcılar – hem okuyup hem de yazabiliyorlar
- Problem – aynı anda birden fazla okuyucuya okumak için izin vermek. Aynı anda sadece bir yazıcının paylaşılan veri kümesine erişimine izin vermek
- **Varyasyon 1:** (ilk okuyucular – yazıcılar problemi): Bir yazıcı veriye erişim hakkını çoktan kazanmış olmadığı sürece, hiçbir okuyucu beklemez
- **Varyasyon 2**: Bir yazıcı paylaşılan veriye erişmek istiyorsa, hiçbir yeni okuyucu paylaşılan veriye erişemez. Yazıcılar öncelikli.

Yemek Yiyen Filozoflar Problemi



- Paylaşılan veri
  - bir kase pilav (veri kümesi)
  - chopStick [5] semaforları, ilk değerleri 1 – yemek çubuğu

Semafor Çözümü Problemleri

- Tüm filozoflar acıkıp aynı anda sol çubuğu alırsa ne olur?
- Kilitlenmeyi (deadlock) önlemek için getirilebilecek kısıtlamalar:
  - Aynı anda en fazla 4 filozof yemeğe başlayabilir
  - Bir filozof, sadece iki çubuk birden hazırsa, çubukları alabilir (kritik kısımda gerçekleşmeli)
  - Asimetrik bir çözüm kullanmak: Tek numaralı filozoflar önce sol çubuğu daha sonra sağ çubuğu alırken, çift numaralı filozoflar önce sağ çubuğu daha sonra sol çubuğu alır
- Monitörler ile problemin kilitlenmeyen çözümü (henüz anlatılmadı)
- Kilitlenmeye (deadlock) neden olmayan bir çözümün açlığa (starvation) neden olmayacağı garanti değil !
- Semaforlar zamanlama hatalarına çok açıktır ve bu tür hatalar çok nadir ortaya çıkabileceğinden hataların ayıklanması zordur

BİTTİ