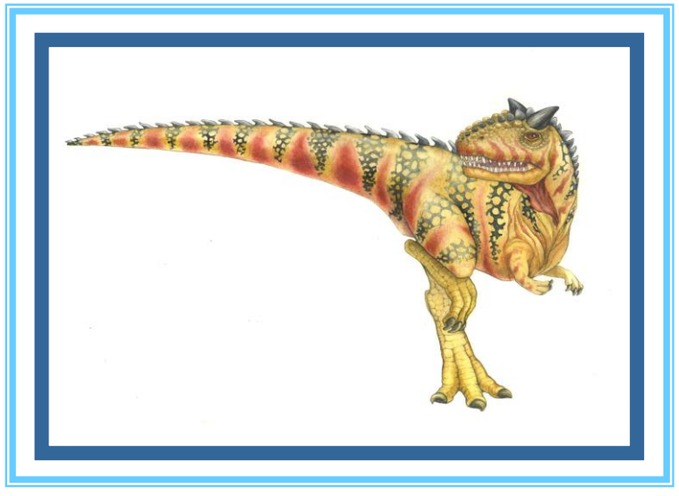
Bölüm 6: Proses Senkronizasyonu-2



Semafor

- Mutex kilidine benzer fakat daha sağlam yapıya sahip ve biraz daha karmaşık senkronizasyon aracıdır.
- *S* Semaforu – bir tamsayı değişkenidir
- S üzerinde İki standart işlem değişiklik yapabilir : wait() ve signal()
  - P()- ***proberen, test için*** ve V()- ***verhogen artım için***, biçiminde adlandırılırlar.
- Sadece iki bölünmez (atomik) işlem üzerinden erişilebilir.
  - wait (S) {

```
        while S <= 0
                    ; // bekleme
          S--;
    }
```

  - signal (S) {

```
    S++;
  }
```

Genel Senkronizasyon Aracı Olarak Semafor

- **Sayma (Counting)** semaforu – tamsayı değeri sınırsız bir etki alanı içinde değişebilir.
- **İkili (Binary)** semafor – Tamsayı değeri yalnızca 0 ve 1 olarak değişebilir; uygulaması daha basit olabilir.
  - **mutex kilidi** ile benzer.
- Birçok senkronizasyon problemini çözebiliriz;
- Karşılıklı dışlama sağlamak için ikili semaforlar kullanılabilir.
- Sayma semaforları, sonlu sayıda örnekten oluşan belirli bir kaynağa erişimi kontrol etmek için kullanılabilir.
- Semafor mevcut kaynakların sayısına göre başlatılır.
- Kaynak kullanmak isteyen her proses, semafor üzerinde bir wait () işlemi gerçekleştirir.
- Bir proses bir kaynağı bıraktığında, bir signal () işlemi gerçekleştirir
- Semafor için sayım 0'a gittiğinde, tüm kaynaklar kullanılmıştır. Bundan sonra, bir kaynak kullanmak isteyen prosesler sayım 0'dan büyük oluncaya kadar engellenir.

Genel Senkronizasyon Aracı Olarak Semafor

- Karşılıklı dışlama şartını sağlar.

```
Semaphore mutex;    //  1 e kurulur
do {
      wait (mutex);
         // Krtik Bölge
      signal (mutex);
               // geri kalan bölge
} while (TRUE);
```

- Birçok probleme de çözüm olabilir. P1 prosesin S1 işlemi bittikten sonra P2 prosesinin S2 işlemini yapmasını istiyoruz. Çözüm;(synch paylaşılmış semafor ve 0 dan başlatılır)

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Semafor Uygulanması

- Herhangi iki prosesin aynı anda, aynı semafor üzerinde wait() ve signal() fonksiyonlarını çalıştıramayacağı garanti altına alınmalıdır.
- Yoksa, uygulama wait ve signal kodunun kritik bölüme yerleştirildiği kritik bölüm problemi haline gelir
  - Şimdi kritik bölge uygulamasında **busy waiting (meşgul bekleme)** olabilir.
    - Fakat uygulama kodu kısadır.
    - Kritik bölge nadiren meşgul olursa, kısa süreli meşgul beklemeler olur.
- Uygulamalar kritik bölgelerde çok fazla zaman harcayabilir bu nedenle bunun iyi bir çözüm olmadığını unutmayın.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
int value;
struct process *list;
} semaphore;
```

Yoğun Beklemesiz Semafor Uygulanması (Devam)

- Bekleme uygulaması :

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0)//pozitif değilse beklemeli
        {
                add this process to S->list;
                block();
        }
}
```

- Sinyal uygulaması :

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Kilitlenme ve Açlık

- **Deadlock (Kilitlenme)**–iki yada daha çok prosesin belirsiz bir süre beklemesidir,
- S ve Q iki semafor ve 1 ‘ den başlatılmış

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- P0'un wait (S) ve ardından P1 wait (Q) işlemini gerçekleştirdiğini varsayalım. P0 wait (Q) uyguladığında, P1'in signal (Q) çalıştırana kadar beklemesi gerekir. Benzer şekilde, P1 wait (S) uyguladığında, P0'un signal (S) çalıştırana kadar beklemesi gerekir. Bu signal () işlemleri gerçekleştirilemediğinden, P0 ve P1 kilitlendi.

- **Starvation (Açlık)** – Belirsiz engelleme
  - Askıya alınmış bir process asla semafor kuyruğundan silinmez.
- **Priority Inversion (Öncelik Değişimi)** –Yüksek öncelikli bir prosesin ihtiyaç duyduğu bir kaynağı daha düşük öncelikli bir process kilitli tutuyorsa planlama problemi meydana gelir.
  - **priority-inheritance protocol (Önceliğin Kalıtımı Protokolü)** aracılığıyla çözülmüştür.

Senkronizasyonun Klasik Problemleri

- Yeni önerilen senkronizasyon planlarını sınamak için kullanılan klasik problemler.
  - Bounded-Buffer Problem (Sınırlı Tampon Problemi)
  - Readers and Writers Problem (Okuyucu ve Yazıcı Problemi)
  - Dining-Philosophers Problem (Yemek Yiyen Filozoflar Problemi)

Sınırlı Buffer (Tampon) Problemi

```
public class BoundedBuffer<E> implements Buffer<E>
{
private static final int BUFFER SIZE = 5;
private E[] buffer;
private int in, out;
private Semaphore mutex;
private Semaphore empty;  // boş bufferların sayısı
private Semaphore full;  //dolu bufferların sayısı
public BoundedBuffer() {
// buffer is initially empty
in = 0; out = 0;
mutex = new Semaphore(1);
empty = new Semaphore(BUFFER SIZE);
full = new Semaphore(0);
buffer = (E[]) new Object[BUFFER SIZE];
}
public void insert(E item) {
// Figure X }
public E remove() {
// Figure Y }
}
```

Sınırlı Buffer Problemi (Devam)

- Üretici process'in yapısı (X The insert() method)

```
public void insert(E item) {
wait (empty);
wait (mutex);
// add an item to the buffer
buffer[in] = item;
in = (in + 1) % BUFFER SIZE;
signal (mutex);
signal (full);
}
```

Sınırlı Buffer Problemi (Devam)

- Tüketici process'in yapısı (Figure Y The remove() method)

```
public E remove() {
E item;
wait (full);
wait (mutex);
// remove an item from the buffer
item = buffer[out];
out = (out + 1) % BUFFER SIZE;
 signal (mutex);
signal (empty);
return item;
}
```

Bounded-buffer producer

```
import java.util.Date;

public class Producer implements Runnable
{
   private Buffer<Date> buffer;

   public Producer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // produce an item & enter it into the buffer
         message = new Date();
         buffer.insert(message);
      }
   }
}
```

Bounded-buffer consumer

```
import java.util.Date;

public class Consumer implements Runnable
{
   private Buffer<Date> buffer;

   public Consumer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // consume an item from the buffer
         message = (Date)buffer.remove();
      }
   }
}
```

Bounded-buffer factory

```
import java.util.Date;

public class Factory
{
    public static void main(String args[]) {
       Buffer<Date> buffer = new BoundedBuffer<Date>();

       // Create the producer and consumer threads
       Thread producer = new Thread(new Producer(buffer));
       Thread consumer = new Thread(new Consumer(buffer));

       producer.start();
       consumer.start();
    }
}
```

Okuyucular-Yazıcılar Problemi

- Bir veri tabanı eşzamanlı processler dizisi arasında paylaşılmıştır.
  - Okuyucular – veri tabanı yalnızca okuyabilirler; herhangi bir güncelleme yapmazlar.
  - Yazıcılar – hem okuyabilir hem yazabilir.

- Problem –
  - Aynı anda birden fazla okuyucuya izin verilir.
  - Tek bir anda yalnızca tek bir yazıcı paylaşılmış veriye ulaşabilir. Başka yazıcı veya okuyucu erişemez

- Birkaç varyasyon ile okuyucular ve yazıcılar işlem görür

- Paylaşılmış veri
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore rw_mutex initialized to 1
  - Integer readcount initialized to 0

Okuyucular-Yazıcılar Probleminin Çeşitleri

- *Birinci* varyasyon – bir yazar zaten paylaşılan nesneyi kullanma iznini almadıkça hiçbir okuyucunun bekletilmemesini gerektirir.

- *İkinci* varyasyon – İkinci okuyucu-yazar sorunu, bir kez bir yazar hazır olduğunda, bu yazar en kısa sürede yazmasını gerektirir. Başka bir deyişle, bir yazar nesneye erişmeyi bekliyorsa, yeni okuyucu okumaya başlayamaz.

- İlk durumda, yazarlar açlıktan ölebilir; İkinci durumda okurlar açlıktan ölebilir.

Okuyucular-Yazıcılar Problemi (Devam)

- Ayrıca, bir yazar signal(rw_mutex) çalıştırdığında, bekleyen okuyucuların veya bekleyen tek bir yazıcının yürütülmesine yeniden başlayabiliriz.
- Seçim çizelgeleyici tarafından yapılır.
- Yazıcı process yapısı

```
do {
        wait (rw_mutex) ;
                //    writing is performed
        signal (rw_mutex) ;
} while (TRUE);
```

Okuyucular-Yazıcılar Problemi (Devam)

- Okuyucu process'in yapısı

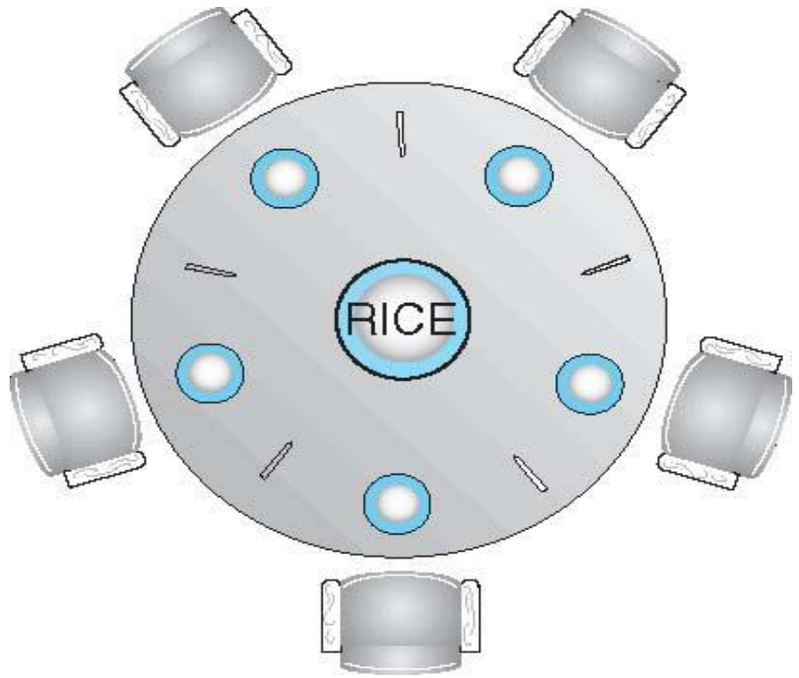
```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)  //ilk okuyucu için (yazıcı yoksa giriş yap)
            wait (rw_mutex) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
            signal (rw_mutex) ;// son okuyucu için (yazıcılar giriş yapabilir)
        signal (mutex) ;
} while (TRUE);
```

Yemek Yiyen Filozoflar Problemi

- Filozoflar yaşamlarını yemek yiyerek ve düşünerek geçirirler.
- Komşularıyla etkileşime geçmeden, arasıra kasesindeki yemeği yemek için 2 yemek çubuğunu almaya çalışıyor (her seferinde bir tane)
  - Yemek için ikisine de ihtiyaç duyar, işini bitirdiğinde ikisini de serbest bırakır.
- 5 filozof durumunda
  - Paylaşılmış veri
    - Bir kase pirinç (veri seti)
    - Basit bir çözüm, her bir çubuğu semafor ile temsil etmektir. Semafor chopstick [5] 1 ile başlatılır.
  - Wait() ile çubuk alınır signal() ile bırakılır



Yemek Yiyen Filozoflar Problem Alg.

- Filozof *i* nin yapısı:

```
Semaphore chopStick[] = new Semaphore[5];
for(int i = 0; i < 5; i++)
chopStick[i] = new Semaphore(1);
do  {
        wait ( chopstick[i] );
         wait ( chopStick[ (i + 1) % 5] );
                    //  eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
                  //  think
} while (TRUE);
```

- Bu algoritmadaki problem nedir?
- Bu çözüm, hiçbir komşunun aynı anda yemediğini garanti etmesine rağmen, yine de bir ölümcül kilitlenme oluşturabileceği için reddedilmelidir.
- Beş filozofun hepsinin aynı anda acıktığını ve her birinin sol çubuğunu kaptığını varsayalım. Çubuk elemanları artık 0'a eşit olacak. Her filozof sağa doğru çubuğu tutmaya çalışınca, sonsuza dek bekleyecekler.

Yemek Yiyen Filozoflar Problem Algoritması

- Kilitlenme sorununun çeşitli olası çözümleri aşağıda dır:
- En fazla 4 filozofun masada aynı anda oturmasına izin verin.
- Bir filozofun çatalları sadece ikisi de mevcutsa almasına izin verin (toplama, kritik bölgede yapılmalıdır)
- Asimetrik bir çözüm kullanın - tek sayıdaki bir filozof önce sol çubuğu ve ardından sağdaki çubuğu alır. Çift numaralı filozof önce sağ çubuğu ve sonra da sol çubuğu alır.

Semaforların Sorunları

- Semafor işlemlerinin yanlış kullanımı:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - wait (mutex) ya da signal (mutex) (ya da her ikisi de) ihmal etme
- Deadlock (Kilitlenme) ve starvation(açlık)

Bölüm 6 Sonu