

# Bölüm 6: İşlem Senkronizasyonu (1)





# İşlem Senkronizasyonu

---

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





# Hedefler

---

- Kritik-kısım problemini tanıtmak
- Kritik-kısım problemine hem yazılım hem de donanım tabanlı çözümler sunmak
- Kritik-kısım problemi çözümlerinin paylaşılan verinin tutarlılığını nasıl sağladığını irdelemek
- Atomik işlem kavramını tanıtmak ve atomikliği sağlayan mekanizmaları açıklamak





# Arkaplan Bilgisi

---

- 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 hücrecini 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 - Örnek

- `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}





# 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-kısıma 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-kısıma erişimi, aynı anda bir işlemin erişebileceği şekilde sınırlandırmalıyız





# 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





# Tipik Bir İşlemin Yapısı

---

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





# Peterson'un Çözümü

---

- İki işlem çözümü
- LOAD ve STORE koutlarının atomik olduğunu varsayın
- 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<sub>i</sub>** işleminin hazır olduğunu gösteriyor





# P<sub>i</sub> İşlemi için Algoritma

```
while (true) {
```

```
    flag[i] = true;
```

```
    turn = j;
```

```
    while (flag[j] && turn == j);
```

```
    critical section
```

```
    flag[i] = false;
```

```
    remainder section
```

```
}
```

Dikkat:  $j = 1 - i$





# 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  
}
```





# 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 hücresinin değiş tokuş eder





# 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);
    }
}
```







# GetAndSet 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 Komutu 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 standard işlem : **acquire()** ve **release()**
  - 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
```





# Java ile Semafor Kullanımı (1)

---

```
public class SemaphoreFactory
{
    public static void main(String args[]) {
        Semaphore sem = new Semaphore(1);
        Thread[] bees = new Thread[5];

        for (int i = 0; i < 5; i++)
            bees[i] = new Thread(new Worker(sem));
        for (int i = 0; i < 5; i++)
            bees[i].start();
    }
}
```





# Java ile Semafor Kullanımı (2)

---

```
public class Worker implements Runnable
{
    private Semaphore sem;

    public Worker(Semaphore sem) {
        this.sem = sem;
    }

    public void run() {
        while (true) {
            sem.acquire();
            criticalSection();
            sem.release();
            remainderSection();
        }
    }
}
```





# 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  
        wakeup(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

$P_0$	$P_1$
<code>S.acquire();</code>	<code>Q.acquire();</code>
<code>Q.acquire();</code>	<code>S.acquire();</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>S.release();</code>	<code>Q.release();</code>
<code>Q.release();</code>	<code>S.release();</code>

- **Açlık (Starvation)** – sınırsız bloklanma. Semafor bekleme listesinde bekleyen bir işlemin hiçbir zaman listeden silinmemesi. Örnek: listenin LIFO (last-in first-out) sırasıyla çalışması





# 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 takip ediyor
- **empty** semaforu, başlangıç değeri  $N$  – boş tampon belleklerin sayısını takip ediyor





# Sınırlı Tampon Bellek - 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();
    }
}
```





# Sınırlı Tampon Bellek

```
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;
    private Semaphore full;

    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 6.10
    }

    public E remove() {
        // Figure 6.11
    }
}
```





# Sınırlı Tampon Bellek - insert()

---

```
// Producers call this method
public void insert(E item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}
```

Figure 6.10 The insert() method.





# Sınırlı Tampon Bellek - remove()

---

```
// Consumers call this method
public E remove() {
    E item;

    full.acquire();
    mutex.acquire();

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```







# Sınırlı Tampon Bellek - 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);
        }
    }
}
```





# Sınırlı Tampon Bellek - 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();
        }
    }
}
```





# 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.





# Okuyucular-Yazıcılar Problemi (2)

---

- Varyasyon 1 için Java'da yazılmış bir çözüm
- Paylaşılan veri
  - Veri kümesi
  - **readerCount** tamsayısı, başlangıç değeri 0, okuyucu sayısı
  - **db** semaforu, başlangıç değeri 1, ortak veriye erişimi sınırlandırır
  - **mutex** semaforu, başlangıç değeri 1, readerCount'a erişimi sınırlandırır ve okuyucuları kendi içinde sıraya sokar





# Okuyucular-Yazıcılar – Kilit Arayüzü

---

- Okuma-yazma kilitleri için arayüz

```
public interface ReadWriteLock
{
    public void acquireReadLock();
    public void acquireWriteLock();
    public void releaseReadLock();
    public void releaseWriteLock();
}
```





# Okuyucular-Yazıcılar - Yazıcı

---

```
public class Writer implements Runnable
{
    private ReadWriteLock db;

    public Writer(ReadWriteLock db) {
        this.db = db;
    }

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

            db.acquireWriteLock();

            // now write to write to the database
            SleepUtilities.nap();

            db.releaseWriteLock();
        }
    }
}
```





# Okuyucular-Yazıcılar - Okuyucu

---

```
public class Reader implements Runnable
{
    private ReadWriteLock db;

    public Reader(ReadWriteLock db) {
        this.db = db;
    }

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

            db.acquireReadLock();

            // now read from the database
            SleepUtilities.nap();

            db.releaseReadLock();
        }
    }
}
```





# Okuyucular-Yazıcılar - Veritabanı

---

```
public class Database implements ReadWriteLock
{
    private int readerCount;
    private Semaphore mutex;
    private Semaphore db;

    public Database() {
        readerCount = 0;
        mutex = new Semaphore(1);
        db = new Semaphore(1);
    }

    public void acquireReadLock() {
        // Figure 6.19
    }

    public void releaseReadLock() {
        // Figure 6.19
    }

    public void acquireWriteLock() {
        // Figure 6.20
    }

    public void releaseWriteLock() {
        // Figure 6.20
    }
}
```







# Okuyucular-Yazıcılar - Okuyucu Metotları

---

```
public void acquireReadLock() {
    mutex.acquire();

    /**
     * The first reader indicates that
     * the database is being read.
     */
    ++readerCount;
    if (readerCount == 1)
        db.acquire();

    mutex.release();
}

public void releaseReadLock() {
    mutex.acquire();

    /**
     * The last reader indicates that
     * the database is no longer being read.
     */
    --readerCount;
    if (readerCount == 0)
        db.release();

    mutex.release();
}
```





# Okuyucular-Yazıcılar - Yazıcı Metotları

---

```
public void acquireWriteLock() {  
    db.acquire();  
}  
  
public void releaseWriteLock() {  
    db.release();  
}
```

Bu çözüm açlığa neden olur mu?

- Varyasyon 1: yazıcılar aç kalabilir
- Varyasyon 2: okuyucular aç kalabilir





# Yemek Yiyen Filozoflar Problemi



## ■ Paylaşılan veri

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





# Yemek Yiyen Filozoflar – Filozof i

---

- *i*'ninci filozofun yapısı:

```
while (true) {  
    // get left chopstick  
    chopStick[i].acquire();  
    // get right chopstick  
    chopStick[(i + 1) % 5].acquire();  
  
    eating();  
  
    // return left chopstick  
    chopStick[i].release();  
    // return right chopstick  
    chopStick[(i + 1) % 5].release();  
  
    thinking();  
}
```





# 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 anlatılacak
- 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

