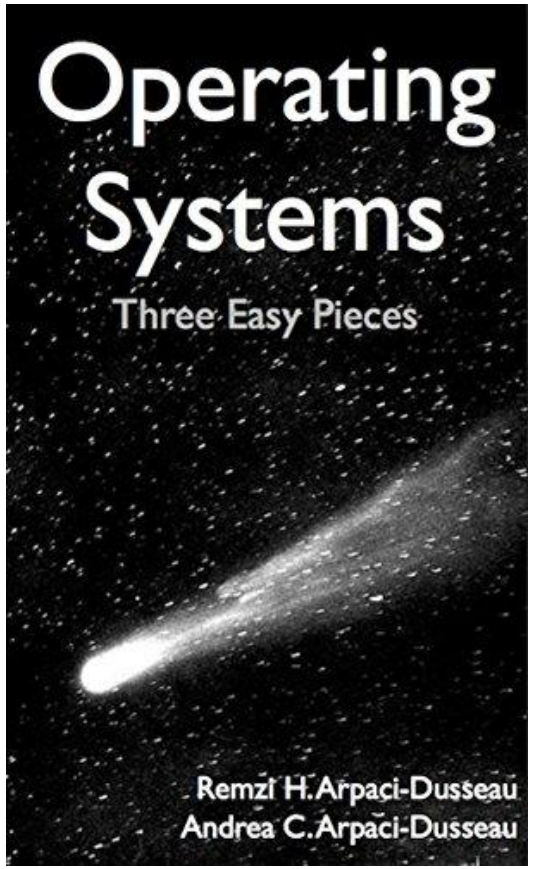
İşletim Sistemleri



9. Ders

Prof. Dr. Kemal Bıçakcı

Alıştırma Sorusu

- Aşağıdaki Ata Çocuğu Bekliyor (Parent Waiting Child) uygulaması kodu ilgili olarak aşağıdakilerden hangisi doğrudur?

a. Kod doğru çalışıyor.

b. Ata sonsuza kadar uyuyabileceği için kod istenildiği gibi çalışmaz.

c. Çocuk sonsuza kadar uyuyabileceği için kod istenildiği gibi çalışmaz.

d. Ata, Çocuktan daha erken sonlanabileceği için kod istenildiği gibi çalışmaz.

```
1       void thr_exit() {
2               Pthread_mutex_lock(&m);
3               Pthread_cond_signal(&c);
4               Pthread_mutex_unlock(&m);
5       }
6
7       void thr_join() {
8               Pthread_mutex_lock(&m);
9               Pthread_cond_wait(&c, &m);
10              Pthread_mutex_unlock(&m);
11      }
```

Doğru Çözüm

```
1         void thr_exit() {
2                 Pthread_mutex_lock(&m);
3                 done = 1;
4                 Pthread_cond_signal(&c);
5                 Pthread_mutex_unlock(&m);
6         }
7
8
9         void thr_join() {
10                Pthread_mutex_lock(&m);
11                while (done == 0)
12                        Pthread_cond_wait(&c, &m);
13                Pthread_mutex_unlock(&m);
14        }
```

Düzeltme: İngilizce «state variable» ve «condition variable» terimlerinin Türkçe karşılıkları?

32. Yaygın Eşzamanlılık Problemleri

Operating System: Three Easy Pieces

Yaygın Eşzamanlılık Problemleri

- Bazı akademik çalışmalar, yaygın eşzamanlılık hata türlerini incelemiştir.
- Bu bölümde, gerçek kod tabanlarında bulunan bazı örnek eşzamanlılık problemlerine kısaca göz atacağız.

Ne Tür Hatalar Var?

- Dört ana açık kaynaklı uygulamaya odaklanıyoruz
  - MySQL, Apache, Mozilla, OpenOffice.

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
Open Office	Office Suite	6	2
Total		**74**	**31**

Modern Uygulamalardaki Hatalar

Kilitlenme Olmayan Hatalar (Non-Deadlock Bugs)

- Eşzamanlılık hatalarının çoğu bu türdendir.
- Kilitlenme olmayan hataların iki ana türü:
  - Atomiklik ihlali
  - Sıra ihlali

Atomiklik İhlali Hataları

- Birden çok bellek erişimi esnasında istenen seri hale getirilebilirlik (serializability) ihlal edilmiştir.
- MySQL'de bulunan bir basit örnek:
  - İki farklı iş parçacığı `struct thd` yapısındaki `proc_info` alanına erişirler.

```
1       Thread1::
2       if(thd->proc_info){
3          …
4          fputs(thd->proc_info , …);
5          …
6       }
7
8       Thread2::
9       thd->proc_info = NULL;
```

Atomiklik İhlali Hataları (Devam)

- **Çözüm**: Paylaşılan değişkenlerin çevresine kilit (`lock-unlock`) eklemek

```
1   pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3   Thread1::
4   pthread_mutex_lock(&lock);
5   if(thd->proc_info){
6       …
7       fputs(thd->proc_info , …);
8       …
9   }
10  pthread_mutex_unlock(&lock);
11
12  Thread2::
13  pthread_mutex_lock(&lock);
14  thd->proc_info = NULL;
15  pthread_mutex_unlock(&lock);
```

Sıra İhlali Hataları

- İki bellek erişimi arasındaki istenen sıra korunmaz.
  - A her zaman B'den önce çalıştırılmalıdır, ancak bu sıra zorunlu hale getirilmez.
- **Örnek**:
  - Thread2'deki kod, `mThread` değişkeninin ilklendirildiğini (ve `NULL` olmadığını) varsaymaktadır (fakat bu varsayım doğru değildir).

```
1    Thread1::
2    void init(){
3         mThread = PR_CreateThread(mMain, …);
4    }
5
6    Thread2::
7    void mMain(…){
8         mState = mThread->State
9    }
```

Sıra İhlali Hataları (Devam)

- **Çözüm**: koşul değişkenleri (condition variables) ile sıra ihlalini engelle.

```
1    pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2    pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3    int mtInit = 0;
4
5    Thread 1::
6    void init(){
7        …
8        mThread = PR_CreateThread(mMain,…);
9
10       // signal that the thread has been created.
11       pthread_mutex_lock(&mtLock);
12       mtInit = 1;
13       pthread_cond_signal(&mtCond);
14       pthread_mutex_unlock(&mtLock);
15       …
16   }
17
18   Thread2::
19   void mMain(…){
20       …
```

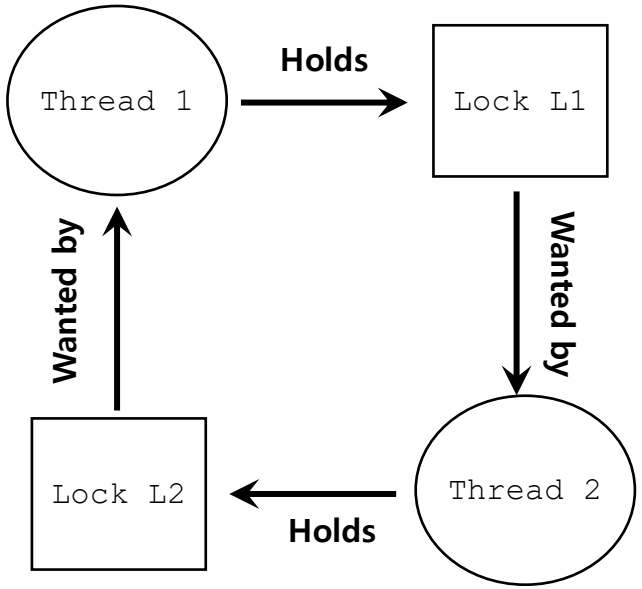
Sıra İhlali Hataları (Devam)

```
21        // wait for the thread to be initialized …
22        pthread_mutex_lock(&mtLock);
23        while(mtInit == 0)
24                pthread_cond_wait(&mtCond, &mtLock);
25        pthread_mutex_unlock(&mtLock);
26
27        mState = mThread->State;
28        …
29  }
```

Kilitlenme Hataları (Deadlock Bugs)

- Bir döngünün (cycle) varlığı:
  - `Thread1, L1` kilidini almış ve `L2` kilidini bekliyor.
  - `Thread2, L2` kilidini almış ve `L1` kilidini bekliyor.

```
Thread 1:          Thread 2:
lock(L1);          lock(L2);
lock(L2);          lock(L1);
```



Niye Kilitlenme Olur?

- Sebep 1:
  - Büyük kod tabanlarında, bileşenler arasında karmaşık bağımlılıklar vardır.

- Sebep 2: Kapsüllemenin (encapsulation) sonucu ortaya çıkar:
  - Implementasyon ayrıntılarının gizlenmesi yolu ile yazılımın modüler bir şekilde oluşturulması genelde iyi bir tercihdir.
  - Öte yandan, bu tür bir modülerlik, kilitlenme problemine yol açabilir.

Niye Kilitlenme Olur? (Devam)

- **Örnek**: Java Vector sınıfını ve `AddAll()` metodunu düşünelim:

```
1    Vector v1,v2;
2    v1.AddAll(v2);
```

- Toplama işlemi öncesinde her iki vektöre (`v1` ve `v2`) ait **kilitlerin** elde edilmesi gerekir.
  - Çağrılan metot kilitleri herhangi bir sırada elde eder.
  - Bir iş parçacığı `v1.AddAll(v2)` çağrısı yaparken diğer biri aynı anda `v2.AddAll(v1)` çağrısı yaparsa kilitlenme oluşabilir.

Kilitlenme Koşulları

- Bir kilitlenme olması için dört farklı koşulun aynı anda sağlanması gerekir.

Koşul	Açıklama
Karşılıklı Dışlama (Mutual Exclusion)	İş parçacıkları, ihtiyaç duydukları kaynakların özel ve tek olacak şekilde kontrolünü talep eder.
Tut-ve-bekle (Hold-and-wait)	İş parçacıkları, ek kaynakları beklerken kendilerine tahsis edilen kaynakları ise tutmaktadırlar.
Boşaltılamama (No preemption)	Kaynaklar, onları tutan iş parçacıklarından zorla geri alınamaz.
Dairesel Bekleme (Circular wait)	Her iş parçacığının zincirdeki bir sonraki iş parçacığı tarafından talep edilen bir kaynağı tuttuğu dairesel bir zincir vardır.

- Eğer bu dört koşuldan herhangi biri sağlanmıyorsa, kilitlenme olamaz.

Önleme:
Dairesel Bekleme (Koşulunu Bozma)

- Kilit elde edilirken hep bir sıra takip edilmesi.
- Bu yaklaşım, genel kilitleme stratejilerinin dikkatli bir şekilde tasarlanmasını gerektirir.
- Örnek:
  - Sistemde iki adet kilit bulunmaktadır (L1 ve L2).
  - L1'i her zaman L2'den önce elde ederek kilitlenmeyi önleyebiliriz.

Önleme: Tut-ve-bekle

- Tüm kilitleri bir anda ve atomik olarak elde et.

```
1    lock(prevention);
2    lock(L1);
3    lock(L2);
4    …
5    unlock(prevention);
```

- Bu kod, kilitleme ortasında zamansız bir iş parçacığının araya girmeyeceğini garanti eder.
- Sorun:
  - Bir çağrı yaparken tam olarak hangi kilitlerin elde edilmesi gerektiğini bilmemiz ve bunları önceden edinmemiz gerekir.
  - Eşzamanlılık azalır.

Önleme - Boşaltılamama

- Çoklu kilit edinimi genellikle sorunludur çünkü bir kilidi beklerken diğerini tutuyoruz.
- `trylock()`
  - Kilitlenme içermeyen, sıralı ve hatasız bir kilit edinme protokolü oluşturmak için kullanılır.
  - Kilidi elde et (mümkünse) veya -1 değeri döndür (daha sonra tekrar dene).

```
1    top:
2         lock(L1);
3         if( tryLock(L2) == -1 ){
4                   unlock(L1);
5                   goto top;
6         }
```

Önleme – Boşaltılamama (Devam)

- Livelock (sonsuz döngü):
  - Her iki sistemin de kod dizisini tekrar tekrar çalıştırması ve ilerleme kaydedememesi ihtimal dahilindedir.
- Çözüm:
  - Yeniden denemeden önce rastgele bir gecikme eklemek.

Önleme – Karşılıklı Dışlama

- Karşılıklı dışlama gerektirmeyen bir çözüm tasarlamak
  - Örneğin donanım buyruklarını doğrudan kullanarak.
  - Kilitleme gerektirmeyen bir şekilde eşzamanlı veri yapıları oluşturabiliriz.

```
1    int CompareAndSwap(int *address, int expected, int new){
2        if(*address == expected){
3                *address = new;
4                return 1; // success
5        }
6        return 0;
7    }
```

Önleme – Karşılıklı Dışlama (Devam)

- Bir değeri atomik olarak belirli bir miktarda artırmak istediğimizi varsayalım:

```
1    void AtomicIncrement(int *value, int amount){
2         do{
3                   int old = *value;
4         }while( CompareAndSwap(value, old, old+amount)==0);
5    }
```

- Değeri artırma gerçekleşene kadar `compare-and-swap` buyruğunu devamlı şekilde çalıştırıyoruz.
  - Kilit kullanılmıyor
  - Bu sebeple kilitlenme olmuyor
  - Ama hala **livelock** olabilir

Önleme – Karşılıklı Dışlama (Devam)

- Daha karmaşık bir örnek: listeye eleman ekleme

```
1    void insert(int value){
2         node_t * n = malloc(sizeof(node_t));
3         assert( n != NULL );
4         n->value = value ;
5         n->next  = head;
6         head     = n;
7    }
```

- Eğer aynı anda birden fazla iş parçacığı bu kodu çalıştırırsa, yarış durumu ortaya çıkar.

Önleme – Karşılıklı Dışlama (Devam)

- **Çözüm-1** (Kilit kullanımı):

```
1    void insert(int value){
2         node_t * n = malloc(sizeof(node_t));
3         assert( n != NULL );
4         n->value = value ;
5         lock(listlock); // begin critical section
6         n->next  = head;
7         head     = n;
8         unlock(listlock) ;  //end critical section
9    }
```

- **Çözüm-2** (`compare-and-swap` buyruğu kullanımı):

```
1    void insert(int value) {
2         node_t *n = malloc(sizeof(node_t));
3         assert(n != NULL);
4         n->value = value;
5         do {
6                   n->next = head;
7         } while (CompareAndSwap(&head, n->next, n));
8    }
```

Zamanlayıcı ile Kilitlenmeden Kaçınma

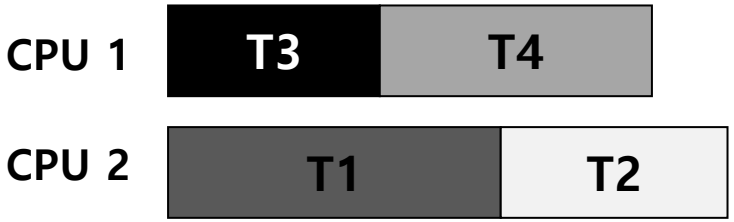
- Bazı senaryolarda kilitlenmeden kaçınma yöntemi tercih edilir.
- Global bilgi gereklidir:
  - Çeşitli iş parçacıkları çalıştırılırken hangi kilitleri elde ederler?
  - Daha sonra söz konusu iş parçacıklarının kilitlenme olmamasını garanti edecek şekilde zaman planlaması yapılır.

Örnek (1)

- İki işlemci ve dört iş parçacığı olsun.
  - İş parçacıklarının kilit isteme durumları:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

  - Akıllı bir zamanlayıcı, T1 ve T2 aynı anda çalıştırılmadığı sürece hiçbir kilitlenmenin olmayacağını hesaplayabilir.

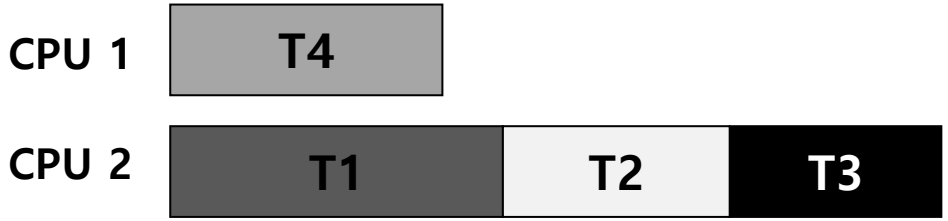


Örnek (2)

- Daha yoğun kilit talebi:

	T1	T2	T3	T4
L1	Yes	Yes	Yes	No
L2	Yes	Yes	Yes	No

- Kilitleme ihtimalini ortadan kaldıran bir zamanlama:



- İşleri tamamlamak için gereken toplam süre önemli ölçüde uzar.

Banker Algoritması?

The Banker's Algorithm for a Single Resource

Bu tahsis durumlarından hangisi güvenlidir?

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

The Banker's Algorithm for a Single Resource

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

(a) Safe. (b) Safe. (c) Unsafe.

Algıla ve Kurtar

- Kimi zaman kilitlenme oluşmasına izin vermek ve kilitlenme olursa ardından işlem yapmak yeterli bir çözüm olur.
  - Örnek: Bir işletim sistemi donarsa, onu yeniden başlatırsınız.
- Birçok veritabanı sistemi, kilitlenme algılama ve kurtarma tekniklerini kullanır.
  - Bir kilitlenme detektörü periyodik olarak çalışır.
  - Bir kaynak grafiği oluşturulur ve döngü olup olmadığı kontrol edilir.
  - Kilitlenme durumunda, sistemin yeniden başlatılması gerekir.

33. Olay Tabanlı Eşzamanlılık (Gelişmiş)

Event-based Concurrency (Advanced)

Operating System: Three Easy Pieces

Olay Tabanlı Eşzamanlılık

- Farklı bir eşzamanlı programlama stilidir.
  - GUI tabanlı uygulamalarda, bazı İnternet sunucusu türlerinde kullanılır.
- Olay tabanlı eşzamanlılığın ele aldığı sorun iki yönlüdür:
  - Çoklu iş parçacıklı uygulamalarda eşzamanlılığı doğru şekilde yönetmek zordur. Kilitlenme ve benzer sorunlar ortaya çıkabilir.
  - Geliştiricinin, belirli bir anda neyin zamanlandığı üzerinde çok az kontrolü vardır veya hiç kontrolü yoktur.

Bir Sunucu için 3 Alternatif

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Temel Fikir: Bir Olay Döngüsü

- Yaklaşım:
  - Bir şeyin (bir “olay”) gerçekleşmesini bekle.
  - Olduğunda, ne tür bir olay olduğunu kontrol et.
  - Gereken az miktarda işi yap.
- Örnek:

```
1    while(1){
2         events = getEvents();
3         for( e in events )
4                   processEvent(e); // event handler
5    }
```

Olay tabanlı bir sunucu (Sözde kod)

How exactly does an event-based server determine which events are taking place?

Olay Tabanlı Programlama - Zorluklar

- Olay tabanlı bir sunucu hangi olayların gerçekleştiğini nasıl anlar?
  - İşletim Sisteminden destek alır (**`select()`** veya **`poll()`** sistem çağrıları)
- Bir olay, sistem çağrısı yapmanızı gerektiriyorsa ne olur?
  - Çözüm-1: Bloklamayan Sistem Çağrısı + Asenkron I/O (peryodik sorgu)
  - Çözüm-2: Bloklamayan Sistem Çağrısı + Asenkron I/O (kesme, «Unix signals»)
- Durum yönetimini (State management) nasıl yapacağız?
- Birden fazla işlemci varsa? (iş parçacıkları yine de gerekli olabilir)
- Tüm sistem aktiviteleri bloklamayan yapıda olabilir mi?
  - Örnek: sayfa hatası (page fault)

Unix Sinyalleri (Unix Signals)

- Bir işlemle iletişim kurmanın basit bir yoludur.
- Farklı türler*: HUP* (hang up), *INT*(interrupt), *SEGV*(segmentation violation)

```
#include <stdio.h>
#include <signal.h>
void handle(int arg) {
        printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
        signal(SIGHUP, handle);
        while (1)
                ; // doin' nothin' except catchin' some sigs
        return 0;
}
```

Sonsuz bir döngüye girmiş basit bir program

Unix Sinyalleri (Devam)

- kill komutu aracıyla bir işleme sinyaller gönderebiliriz.
- Bu komut, programdaki ana `while` döngüsünü kesecek ve `handle()` işleyici kodunu çalıştıracaktır.

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```