Bölüm 6: İşlem Senkronizasyonu (2)

İşlem Senkronizasyonu

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

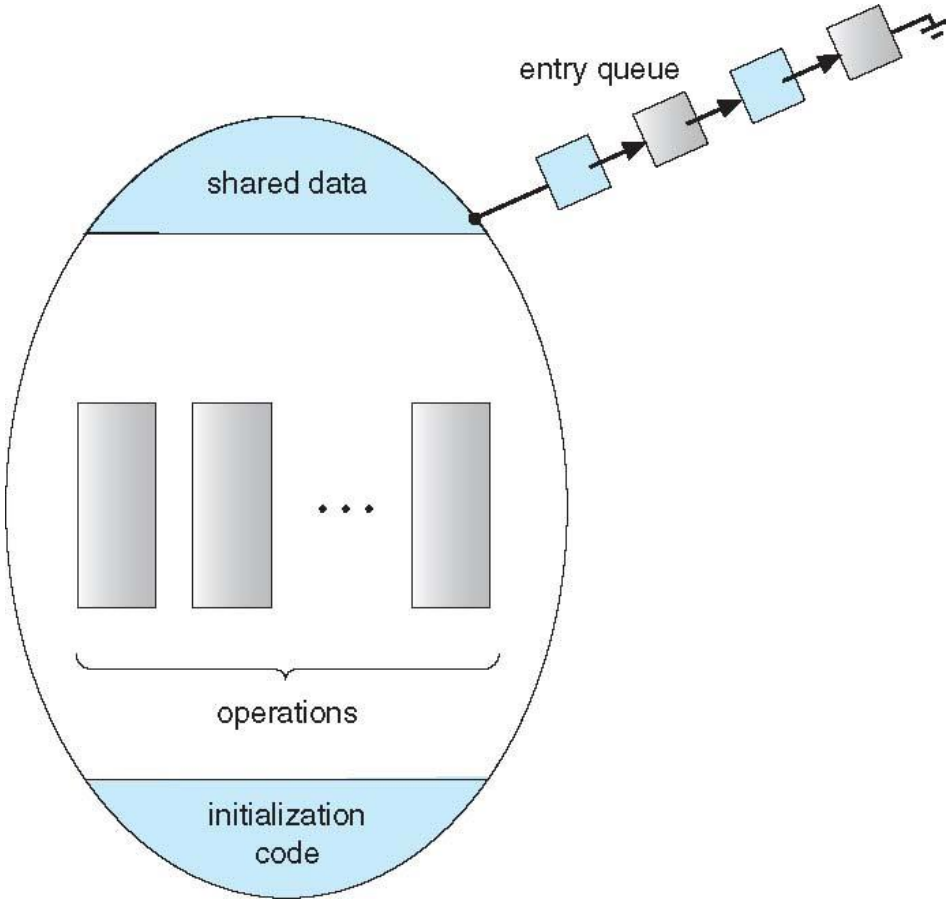
Monitörler

- İşlem senkronizasyonu için rahat ve etkili bir mekanizma sunan üst seviye soyutlama
- Pek çok programlama dili (Örnek: C# ve Java) monitör kavramını gerçekleştirmiştir
- Belirli bir zamanda, sadece bir işlem monitor içerisinde aktif olabilir
- Monitörler içerisinde ortak değişkenler ve fonksiyonlar bulunur
- Ortak verilere erişim kontrol altındaki fonksiyonlar aracılığıyla sağlanabilir

Monitör Kullanımı

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Monitörün Şekilsel Gösterimi

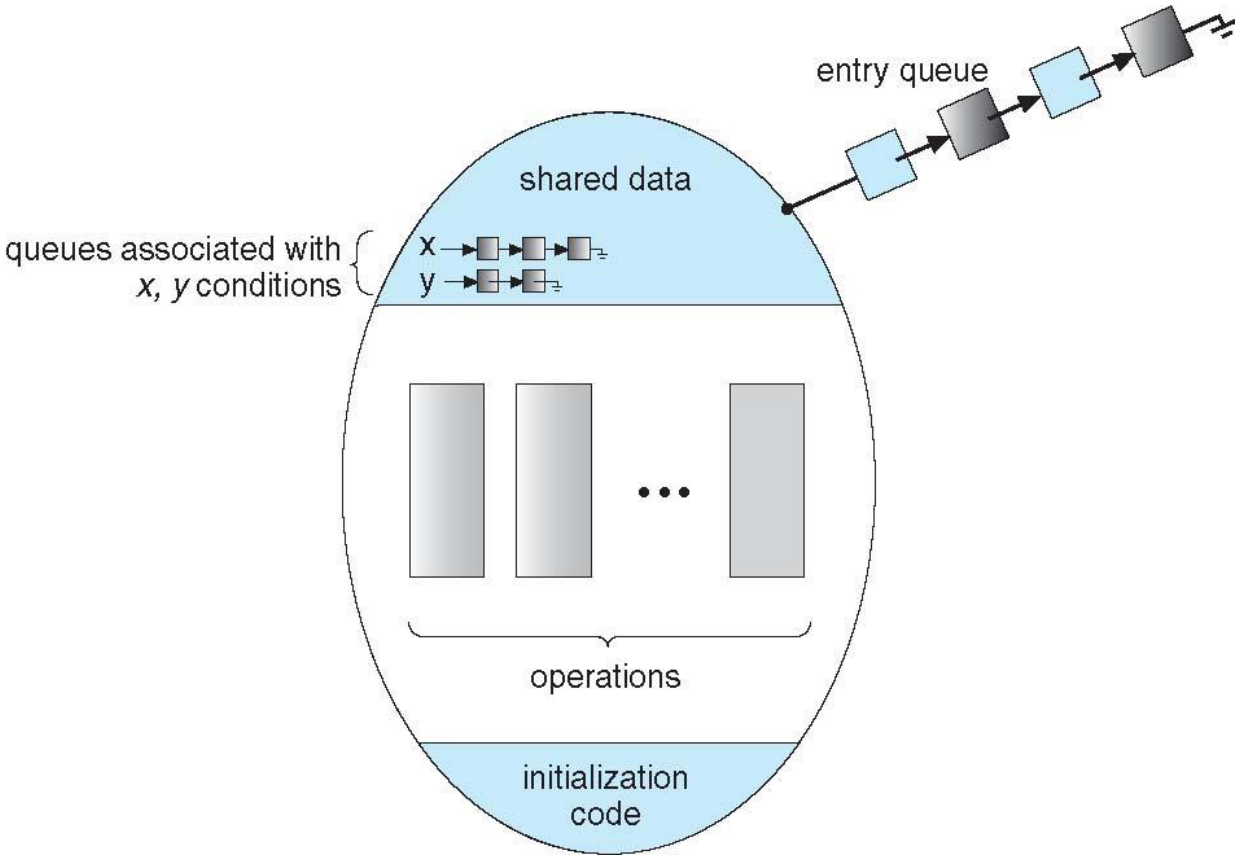


Koşul Değişkenleri

- Koşul değişkenleri (condition variable) monitörleri daha güçlü hale getiren özel değişkenlerdir
- Koşul değişkeni tanımlama: **Condition x, y**;

- Bir koşul değişkeni üzerinde iki metot tanımlıdır:
  - **x.wait ()** – bu metotu çağıran işlem askıya alınır
  - **x.signal ()** – wait metodunu çağırmış işlemlerden (eğer varsa) birini çalışmaya devam ettirir

Koşul Değişkenleri Olan Bir Monitör



Yemek Yiyen Filozoflar - Monitör Çözümü

- Her bir *i* filozofu **takeForks(i)** ve **returnForks(i)** metotlarını aşağıdaki sırada çağırır:

dp.takeForks (i) // çubukları al

YEMEK YE (EAT)

dp.returnForks (i) // çubukları bırak

Yemek Yiyen Filozoflar - Monitör Çözümü

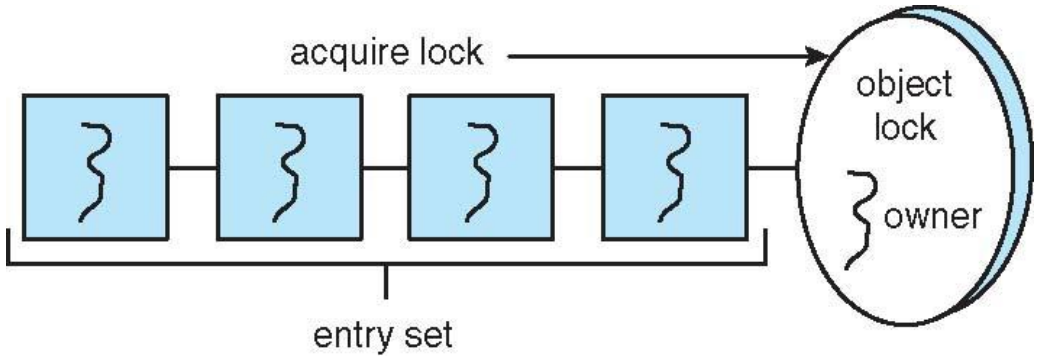
```
monitor DiningPhilosophers
{
    enum State {THINKING, HUNGRY, EATING};
    State[] states = new State[5];
    Condition[] self = new Condition[5];

    public DiningPhilosophers {
       for (int i = 0; i < 5; i++)
          state[i] = State.THINKING;
    }

    public void takeForks(int i) {
       state[i] = State.HUNGRY;
       test(i);
       if (state[i] != State.EATING)
          self[i].wait;
    }

    public void returnForks(int i) {
       state[i] = State.THINKING;
       // test left and right neighbors
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

    private void test(int i) {
       if ( (state[(i + 4) % 5] != State.EATING) &&
          (state[i] == State.HUNGRY) &&
          (state[(i + 1) % 5] != State.EATING) ) {
             state[i] = State.EATING;
             self[i].signal;
       }
    }
}
```

Java Senkronizasyonu

- Java dil seviyesinde senksronizasyon sağlar
- Her Java nesnesi ilişkili bir kilite (lock) sahiptir
- Bu kilit **synchronized** bir metot çağırıldığında elde edilir.
- **synchronized** metottan çıkıldığındabu kilit iade edilir
- Böylece bir nesne üzerindeki senksronize metotlardan, aynı anda sadece biri çağırılabilir. Diğer iş parçacıkları bu metot bitmeden önce senkronize (synchronized) metotlardan birini çalıştıramaz
- Nesne kilitini elde etmek isteyen iş parçacıkları, nesne kilidi için tanımlı **giriş kümesine (entry set)** eklenirler.

Java Senkronizasyonu – Giriş Kümesi

- Her bir nesne **giriş kümesi (entry set)** ile ilişkilendirilir.



Java Senkronizasyonu - Bounded Buffer

```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
      // buffer is initially empty
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
    }

    public synchronized void insert(E item) {
       // Figure 6.29
    }

    public synchronized E remove() {
       // Figure 6.29
    }
}
```

BoundedBuffer – Yanlış insert/remove

- Senkronize insert() ve remove() metotları – Yanlış!

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE)
       Thread.yield();

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
}


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

    while (count == 0)
       Thread.yield();

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    return item;
}
```

insert/remove Neden Yanlış?

- Meşgul bekleme (busy waiting) yok, ancak canlı-kilite (livelock) neden olabilir
  - Üretici, tampon bellek dolu olduğunda veya
  - Tüketici, tampon bellek boş olduğunda
- Canlı-kilit, kilitlenmeye (deadlock) benzer. Her ikisi de iki veya daha fazla iş parçacığının çalışmaya devam etmesine engel olur
- Kilitlenme, bir küme iş parcağındaki tüm iş parçacıklarının, aynı kümedeki bir iş parçacığının sonuç üretmesini bekleyerek kilitlenmesidir
- Canlı kilit (livelock) ise bir iş paracığının devamlı olarak çalışmak istemesi ama bunu başaramamasıdır

Canlı Kilit Senaryosu

- Hatırlatma: JVM iş parçacıklarını öncelik tabanlı olarak (priority based) zamanlar
- Daha çok öncelikli iş parçacıklarına, daha az öncelikli iş parçacıklarına göre daha fazla iltimas gösterir
- Eğer üretici (tüketiciye göre) daha fazla öncelikli ise ve tampon bellek doluysa
  - Üretici, while döngüsüne girdiğinde, count değişkeni değeri azalana kadar yield komutunu çalıştırır
  - Ancak üretici daha az öncelikli olduğu için, JVM kontrlü üretici yerine yeniden tüketiciye verip
  - Tüketiciye hiç bir zaman kontrolü devretmeyebilir
- Bu durumda üretici canlı kilit durumunda, tüketicinin tampon belleği boşaltmasını bekleyecektir

insert/remove Neden Yanlış?

- Üretici ve tüketici tarafından paylaşılan count değişkeni, synchronized anahtar kelimesi sayesinde yarışma durumuna (race condition) neden olmamakta – değişkene erişim kontrollü
- Ancak synchronized, kilitlenmeye (deadlock) neden olabilir
  - Tampon belleğin dolu olduğunu ve tüketicinin de uyuduğunu varsayın
  - Üretici insert() metotunu çağırırsa, kilit müsait olduğundan, nesne kilitine sahip olacaktır
  - Ancak tampon belleğin dolu olduğunu gördüğünde yield() metotunu çağırıp CPU kontrolünü elinden bırakacaktır
  - Ancak nesne kilidine halen sahiptir
  - Tüketici, zamanı geldiğinde remove() metotunu çağırdığında ise, nesne kilidi sahipli olduğundan bloklanacaktır
  - Böylece hem üretici hem de tüketici çalışmaya devam edemeyecektir: kilitlenme

Java Senkronizasyonu - wait/notify()

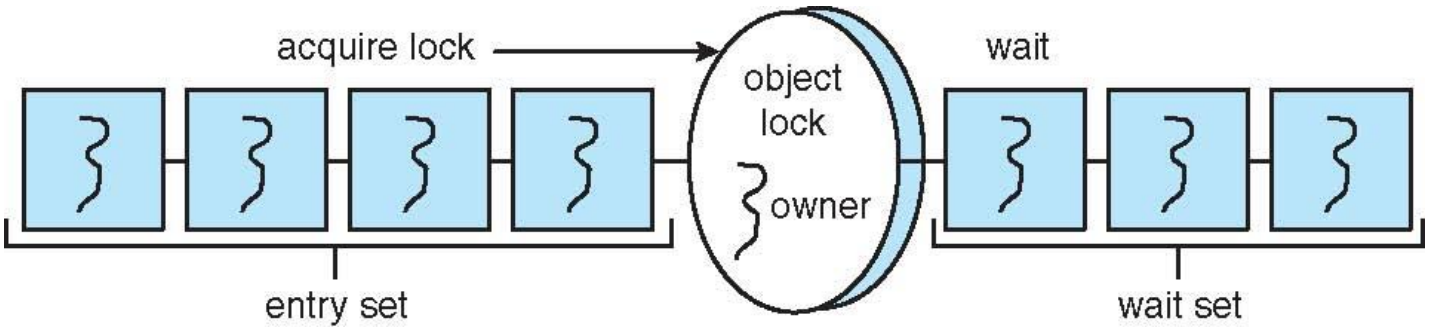
- wait() ve notify() metotları bu tür problemlerı çözmemizi sağlar
- Bir iş parçacığı **wait()** metotunu çağırdığında:

  1. İş parçacığı nesne kilidini serbest bırakır;
  2. İş parçacığının durumu Blocked'a çevrilir;
  3. İş parçacığı nesne içi kullanılan bekleme kümesine (wait set) alınır.

- Bir iş parçacığı **notify()** metotunu çağırdığında:

  1. Bekleme kümesinden herhangi bir iş parçacığı (T) seçilir;
  2. T, bekleme kümesinden, giriş kümesine (entry set) alınır;
  3. T iş parçacığının durumu Runnable durumuna getirilir.

Giriş ve Bekleme Kümeleri



Wait/notify() Örneği – insert()

- Senkronize insert() metotu – Doğru!

```
// Producers call this method
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;

    notify();
}
```

Wait/notify() Örneği – remove()

- Senkronize remove() metotu – Doğru!

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

    while (count == 0) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    notify();

    return item;
}
```

Notify() ve notifyAll()

- **notify()** çağrısı bekleme kümesinden herhangi bir iş parçacığını seçer
- Seçilen iş parçacığının, beklenilen/istenilen iş parçacığı olmama ihtimali vardır
- **notifyAll()** çağrısı bekleme kümesindeki tüm iş parçacıklarını giriş kümesine taşır
- Genel olarak, **notifyAll(), notify()**'a göre daha konservatif (korumalı) bir metottur.

notifyAll Örneği

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public synchronized void doWork(int myNumber) {
    while (turn != myNumber) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    // Do some work for awhile . . .

   /**
    * Finished working. Now indicate to the
    * next waiting thread that it is their
    * turn to do some work.
    */
    turn = (turn + 1) % 5;

    notify(); <-------------------
}
```

notify() doğru iş parcağını çalıştırmayabilir!

Okuyucular-Yazıcılar - Veritabanı

```
public class Database implements ReadWriteLock
{
    private int readerCount;
    private boolean dbWriting;

    public Database() {
       readerCount = 0;
       dbWriting = false;
    }

    public synchronized void acquireReadLock() {
       // Figure 6.34
    }

    public synchronized void releaseReadLock() {
       // Figure 6.34
    }

    public synchronized void acquireWriteLock() {
       // Figure 6.35
    }

    public synchronized void releaseWriteLock() {
       // Figure 6.35
    }
}
```

Okuyucular-Yazıcılar – Okuyucu Metotları

```
public synchronized void acquireReadLock() {
    while (dbWriting == true) {
       try {
          wait();
       }
       catch(InterruptedException e) { }
    }

    ++readerCount;
}

public synchronized void releaseReadLock() {
    --readerCount;

   /**
    * The last reader indicates that
    * the database is no longer being read.
    */
    if (readerCount == 0)
       notify();
}
```

Okuyucular-Yazıcılar – Yazıcı Metotları

```
public synchronized void acquireWriteLock() {
    while (readerCount > 0 || dbWriting == true) {
       try {
          wait();
       }
       catch(InterruptedException e) { }
    }

   /**
    * Once there are no readers or a writer,
    * indicate that the database is being written.
    */
    dbWriting = true;
}

public synchronized void releaseWriteLock() {
    dbWriting = false;

    notifyAll();
}
```

Blok Senkronizasyonu

- Blok senskronizasyonu, tüm metodu senkronize yapmak yerine, kod bloklarını senkronize yapabilmemizi sağlar

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();

    synchronized(mutexLock) {
       criticalSection();
    }

    remainderSection();
}
```

Blok Senskronizasyonu – wait/notify

- wait()/notify() kullanarak blok senkronizasyonu

```
Object mutexLock = new Object();
. . .
synchronized(mutexLock) {
    try {
       mutexLock.wait();
    }
    catch (InterruptedException ie) { }
}

synchronized(mutexLock) {
    mutexLock.notify();
}
```

Java 5'deki Aynı Anda Kullanım Özellikleri

- Java 5'e kadar, tek aynı anda kullanım (concurrency) özellikleri synchronized/wait/notify mekanizmalarıydı.
- Java 5 ile birlikte, API'ye yeni özellikler de eklendi:
  - Yeniden Girişli Kilitler (reentrant locks)
  - Semaforlar (semaphores)
  - Koşul değişkenleri (condition variables)

Yeniden Girişli Kilitler

- Synchronized mekanizmasina benzer
- Ancak ek özellikleri vardır – Örneğin adillik (fairness) parametresi
- Adillik parametresi en çok bekleyen iş parçacığına kilitin verilmesini sağlar
- lock() çağırıldığında eğer iş parçacığı kilite zaten sahipse (reenterant!) veya kilite sahip başka bir iş parçacığı yoksa, çalışmaya devam eder
- Kilit başkasına aitse kilidin serbest bırakılmasını bekler

```
Lock key = new ReentrantLock();

key.lock();
try {
   // critical section
}
finally {
   key.unlock();
}
```

Semaforlar

- Java 5'te sayaç semaforu (counting semaphore) gerçekleştirilmiştir

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   // critical section
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Koşul Değişkenleri

- wait/notify mekanizmasına benzer bir mekanizma sunar
- Öncelikle yeniden girişli bir kilit (reentrant lock) oluşturulmalıdır
- Daha sonra bu kilit üzerinden **newCondition()** metotu çağırılarak yeni bir koşul değişkeni oluşturulabilir

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- Koşul değişkeni oluşturulduktan sonra bu koşul değişkeni üzerinden **await()** ve **signal()** metotları çağırılabilir

Koşul Değişkenleri - Örnek

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public void doWork(int myNumber) {
   lock.lock();

   try {
      /**
       * If it's not my turn, then wait
       * until I'm signaled
       */
      if (myNumber != turn)
        condVars[myNumber].await();

      // Do some work for awhile . . .

      /**
       * Finished working. Now indicate to the
       * next waiting thread that it is their
       * turn to do some work.
       */

      turn = (turn + 1) % 5;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```

Senkronizasyon Örnekleri

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Senkronizasyonu

- İşlem senkronizasyonu için pek çok mekanizmayı desteklemektedir
- **Semaforlar**
- **Uyarlanabilir muteksler** (adaptive mutexes) etkin bir şekilde kısa kod bölümlerini korumak için kullanılır
- **Koşul değişkenleri** (condition variables) ve **okuyucu yazıcı kilitleri** (readers-writers) daha uzun kod bölümlerini korumak gerektiğinde kullanılır
- **Turnstiles:** uyarlanabilir muteksleri ve koşul değişkenlerini bekleyen iş parçacıklarını sıraya sokmak için kullanılan kuyruklardır

Windows XP Senkronizasyonu

- Tek işlemcili sistemlerde, genel kaynaklara erişimi korumak için **kesinti maskelerini** (interrupt masks) kullanır
- **Spin kilitleri** (spinlocks): çok işlemcili sistemlerde senkronizasyonu sağlayan bir mekanizmadır
- **Dağıtıcı nesneler** (dispatcher objects): muteks veya semafor gibi davranabilirler
- Dağıtıcı nesneler **olayları** (events) da sağlayabilir
  - Olaylar aslında koşullu değişkenlerin davranışına benzer

Linux Senkronizasyonu

- Linux:
  - Çekirdeğin 2.6 verisyonuna kadar, kısa kritik bölümler için kesintiler kapatılmaktaydı (iptal edilmekteydi)
  - Versiyon 2.6 ve sonrasında sistem tamamen kesilebilir (preemptive) hale getirildi
- Linux'un sağladığı mekanizmalar:
  - **semaforlar**
  - **spin kilitleri** (spin locks)

Pthreads Senkronizasyonu

- Pthreads API'si işletim sistemi bağımlı
- Aşağıdaki özellikleri sağlar:
  - **mutex kilitler** (mutex locks)
  - **koşul değişkenleri** (condition variables)
- Taşınabilir olmayan (non-portable) eklentiler:
  - **okuma yazma kilitleri** (read-write locks)
  - **spin kilitleri**

Atomik İşlemler

- Sistem Modeli
- Log-tabanlı Log-based Kurtarma
- Kontrol Noktaları (checkpoints)
- Aynı Anda Çalışan Atomik İşlemler

Atomik Faaliyetli Hafıza (Transactional Memory)

- **Atomik Faaliyet (transaction):** Tek bir mantıksal fonksiyonu atomik olarak gerçekleştiren komut kümesidir
- **Hafıza Atomik Faaliyeti (memory transaction):** atomik olan bir dizi okuma-yazma işlemi.
- Soldaki kod parçasını, sağdaki ile değiştiriyoruz:

```
update () {
   acquire();
   /* modify shared data */
   release();
}
```

```
update () {
   atomic {
      /* modify shared data */
   }
}
```

- **atomic{S}** satırı **S** in bir faaliyet olarak gerçekleşmesini garantiler

Sistem Modeli

- İşlemlerin tek mantıksal iş birimi olarak çalışmasını ya da hiç çalışmamasını garanti eder
- Veritabanı sistemleri ile ilişkilidir
- Amaç bilgisayar sistemi hatalarına rağmen atomikliği garantilemektir
- **Atomik Faaliyet (transaction)** - Tek bir mantıksal fonksiyonu atomik olarak gerçekleştiren komut kümesidir
  - Kalıcı kayıt birimine (disk) olan değişikliklerle ilgileniliyor
  - Atomik faaliyet bir dizi **okuma** ve **yazma** işleminden oluşuyor
  - **commit** (atomik faaliyet başarılı) işlemiyle veya **abort** (atomik faaliyet başarısız) işlemiyle sonlanıyor
  - **Roll back** - atomik faaliyet başarısız olduğunda gerçekleşen değişiklikler geri alınır

Kayıt Birimi Çeşitleri

- Geçici kayıt birimi (volatile storage) – information stored here does not survive system crashes
  - Example: ana hafıza, önbellek
- Kalıcı kayıt birimi (nonvolatile storage) – bilgi genellikle sistem çakılmalarından başarıyla kurtulur
  - Örnek: disk and kaset (tape)
- Güvenilir kayıt birimi (stable storage) – bilgi hiç bir zaman kaybolmaz
  - Aslında mümkün değil. RAID cihazları ile veya Not actually possible, so approximated via kopyalayarak çoğaltma (replication) yoluya yaklaşık olarak elde edilir

Amaç, atomik faaliyetlerin atomikliğini sistem sorunlarının kalıcı kayıt biriminde bilgi kaybına neden olduğu durumlarda sağlamaktır

Kayıt (Log) Tabanlı Kurtarma

- Bir atomik faaliyet tarafından yapılan bütün değişiklikler hakkında bilgiler güvenilir kayıt birimine kaydedilir
- En çok kullanılan: **write-ahead logging**
  - Her bir kayıt atomik faaliyetin tek bir yazma işlemini açıklar
    - Atomik faaliyet adı
    - Veri elemanı adı
    - Eski değer
    - Yeni değer
  - <T_i starts> T_i başladığında kayıda geçilir
  - <T_i commits> T_i teslim edildiğinde (commit) kayıda geçilir
- Veri üzerindeki asıl işleme başlanmadan önce kayıt girişi yapılmalıdır

Kayıt Tabanlı Kurtarma Algoritması

- Kayıt birimini kullanarak, sistem herhangi bir geçici kayıt birimi hatasını çözebilir
  - **Undo(T_i)** T_i tarafından gerçekleştirilen her tür veri güncellemesini geri alabilir
  - **Redo(T_i)** T_i içerisindeki tüm veri güncellemelerini yeniden gerçekleştirebilir
- Undo(T_i) ve redo(T_i) must be **eş kuvvetli (idempotent)** olmalıdır
  - Çoklu çalıştırmalar tek bir çalıştırmayla aynı sonucu üretmelidir
- Eğer sistem çökerse, güncellenen tüm verileri kayıtları kullanarak eski haline getir
  - Eğer kayıt <T_i commits> olmaksızın <T_i starts> içeriyorsa, **undo(T_i)**
  - Eğer kayıt <T_i starts> ve <T_i commits> içeriyorsa, **redo(T_i):** atomik faaliyet yeniden gerçekleştirilmeli

Kontrol Noktaları (Checkpoints)

- Kayıtlar çok artarsa, kurtarma çok zaman alabilir
- Kontrol noktaları kayıt sayısını azaltır ve kurtarma zamanını kısaltır
- Kontrol noktası mekanizması:
  1. Geçici kayıt birimindeki tüm kayıtları güvenilir kayıt birimine aktar
  2. Değişiklik olmuş tüm veriyi geçici kayıt biriminden güvenilir kayıt birimine aktar
  3. <checkpoint> isimli kaydı güvenilir kayıt birimine aktar
- Bu noktada, kurtarma sadece sadece en yakın kontrol noktasından hemen önce başlayan Ti atomik faaliyetinden başlar ve ondan sonraki tğm atomik faaliyetleri kapsar.
- Diğer tüm atomik faaliyetler çoktan güvenilir kayıt birimindedir

Atomik Faaliyetlerin Aynı Anda Kullanımı

- Atomik faaliyetlerin aynı anda kullanımı ile sırasıyla kullanımı arasında sonuç olarak fark olmamalıdır – **serileştirilebilirlik** (serializability)
- Tüm atomik faaliyetler kritik kısım içerisinde gerçekleştirilebilir
  - Verimsiz ve çok fazla kısıtlanmış bir çözüm
- **Aynı anda kullanım kontrol algoritmaları** (concurrency-control algorithms) serileştirilebilirliği sağlar

Serileştirilebilirlik

- A ve B adında iki veriyi ele alalım
- T_0 ve T_1 adında iki atomik faaliyeti ele alalım
- T_0 ve T_1 'i atomik olarak çalıştıralım
- Çalışma sırasına tarife (schedule) denir
- Atomik olarak çalıştırılan atomik faaliyet sırasına seri tarife (serial schedule) denir
- Bir sistemde N atomik faaliyet bulunursa, bu sistemde N! adet geçerli seri tarife bulunur

Seri Tarife

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Seri Olmayan Tarife

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Kilitleme Protokolü

- Serileştirilebilirliği her bir veriye bir kilit atayarak sağlamak
  - Erişim kontrolü için kilitleme protokolünü takip et
- Kilitler
  - **Paylaşımlı (Shared)** – T_i , Q verisi üzerinde S paylaşımlı kilidine sahip. T_i , Q verisini okuyabiliyor ama Q'ya yazamıyor
  - **Dışlayıcı (Exclusive)** – Ti , Q verisi üzerinde X dışlayıcı kilidine sahip. T_i , Q verisini okuyabiliyor ve Q'ya yazabiliyor
- Q üzerindeki her atomik faaliyet, bu faaliyete uygun kilidi elde etmeyi gerektirir
- Eğer kilit başka bir atomik faaliyet tarafından kullanılıyorsa, yeni istek beklemelidir
  - Okuyucular-yazıcılar algoritmasına benzer şekilde

İki Aşamalı Kilitleme Protokolü

- Two-phase Locking Protocol
- Tüm atomik faaliyetler kilitleme ve kilit açma işlemlerini iki aşamada gerçekleştirirler
  - Growing – kilidi elde etmek
  - Shrinking – kilidi iade etmek
- Kilitlenmeyi engellemez

Zaman-etiketi Tabanlı Protokoller

- Timestamp-based Protocols
- Atomik faaliyetler arasındaki srıa baştan belirlenir – **zaman etiketi sıralaması (timestamp-ordering)**
- T_i atomik faaliyeti başlamadan önce bu faaliyete $TS(T_i)$ zaman etiketi verilir
  - Eğer Ti, T_j 'den önce sisteme girmişse: $TS(T_i) < TS(T_j)$
  - TS sistem saatinden veya sisteme atomik faaliyetler geldikçe arttırılan bir sayaçtan elde edilebilir
- Zaman etiketleri serileştirilebilirlik sırasını belirler
  - Eğer $TS(T_i) < TS(T_j)$ ise, sistem serileştirilmiş tarifede T_i 'nin T_j'den önce gözükmesini sağlar