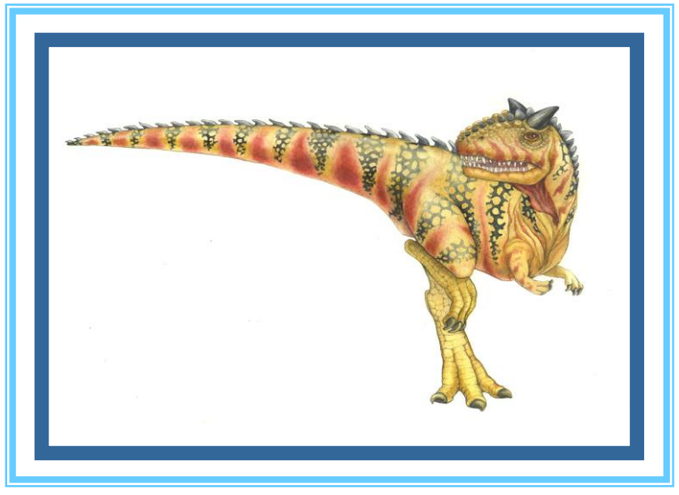
Bölüm 6: Proses Senkronizasyonu-3



Monitörler(İzleyiciler)

- Semaforlar kullanıldığında da senkronizasyon hataları olabilmektedir.
- Tüm process'lerin **kritik bölüme girmeden önce wait()**, **girdikten sonra ise signal() işlemlerini** yapmaları gereklidir.
- **Program geliştirici bu sıraya dikkat etmezse,iki veya daha fazla process aynı anda kritik bölüme girebilir.**
- Bu durumlar programcılar arasında yeterli işbirliği olmadığı durumlarda da olabilmektedir.
- Kritik bölüm problemine ilişkin tasarımda oluşan sorunlardan dolayı **kilitlenmeler** veya **eşzamanlı erişimden dolayı yanlış sonuçlar** ortaya çıkabilmektedir.

Semaforların Sorunları

- Semafor işlemlerinin yanlış kullanımı:
  - signal (mutex) …. Kritik bölge…. wait (mutex)
  - Örnekte **birden fazla process kritik bölüme aynı anda girebilir.**
  - wait (mutex) … wait (mutex)
  - Bu durumda da **deadlock oluşur.**
  - wait (mutex) ya da signal (mutex) (ya da her ikisi de) unutulursa karşılıklı dışlama yapılamaz veya deadlock oluşur.
  - Bu tür hataların tespit edilmesi tespit edilmesi kolay olmayabilmektedir (Her zaman gerçekleşmeyebilir)
- Deadlock (Kilitlenme) ve starvation(açlık) olabiliyor

Monitörler

- Programcıdan kaynaklanabilecek bu hataların giderilmesi için **(monitör)** kullanılır.
- Senkronizasyonda doğru programların yazılmasını kolaylaştırmak adına Brinch Hansen (1973) ve Hoare (1974) yüksek seviyeli senkronizasyon yapıları geliştirmişlerdir.
- **Monitör içinde tanımlanan bir fonksiyon**, sadece **monitor içinde tanımlanan değişkenlere** ve **kendi parametrelerine erişebilir.**
- Bir monitörde tanımlanan bir fonksiyon, yalnızca monitörde yerel olarak bildirilen değişkenlere (private) sadece public (fonksiyonlar) içindeki kod tarafından erişilebilirdirler.
- Monitör yapısı, aynı anda yalnızca bir işlemin monitör içinde etkin olmasını sağlar. Sonuç olarak, programcının bu senkronizasyon kısıtını açıkça kodlaması gerekmez. (Mutex kendiliğinden oluşur)

Monitörler

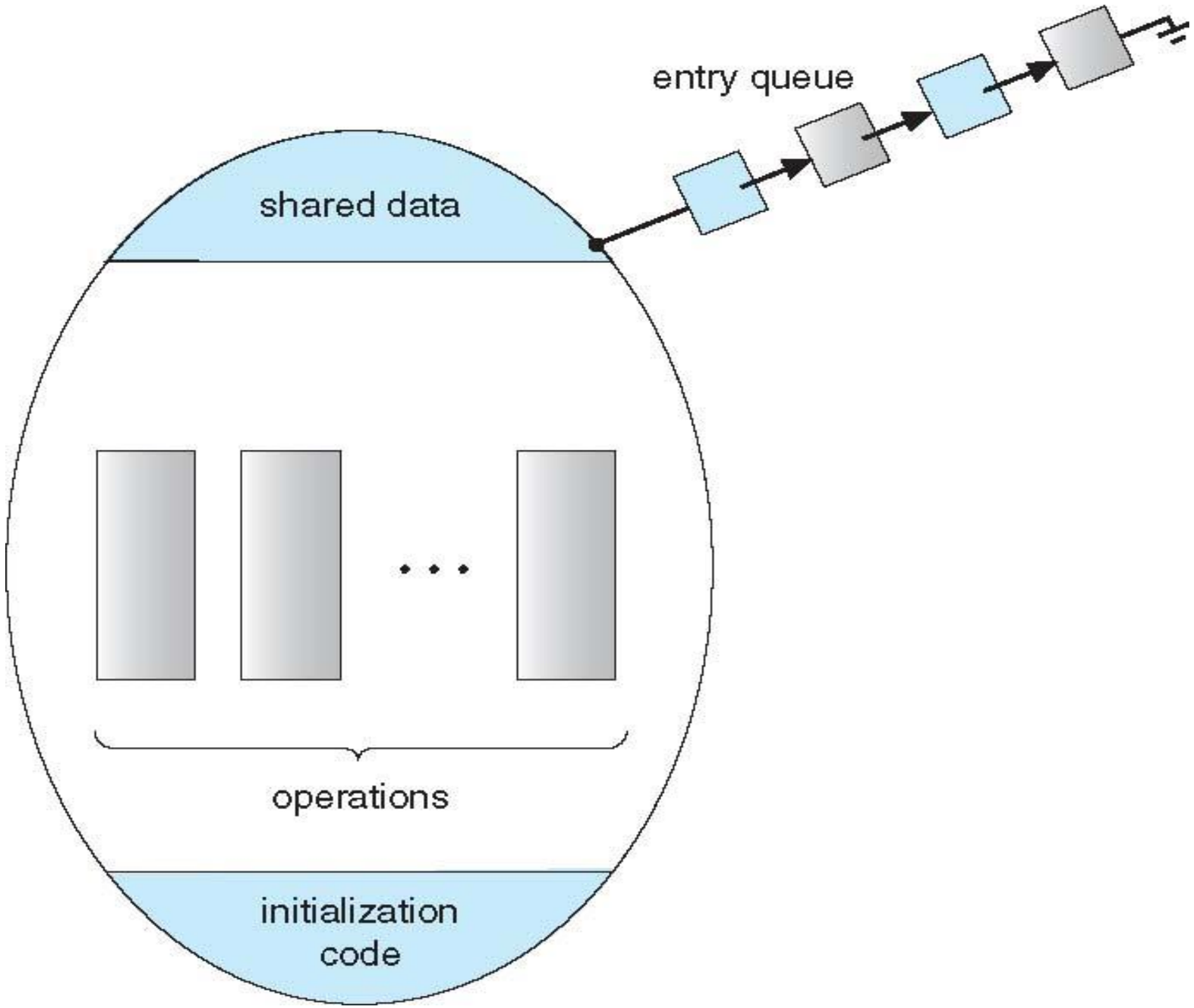
- Monitör programları paylaşılan nesneye ulaşmada meydana gelebilecek problemleri ortadan kaldırmaya yönelik geliştirilmiştir.

  - Paylaşılan nesneyi oluşturan veri,
  - Bu nesneye ulaşmak için kullanılan bir grup procedure (fonksiyonlar),
  - Nesneyi başlangıç konumuna getiren bir program parçasından

  oluşmaktadır.

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
    procedure Pn (…) {……}
    Initialization code (…) { … }
    }
}
```

Monitor'ün Şematik Görünümü



Durum Değişkenleri

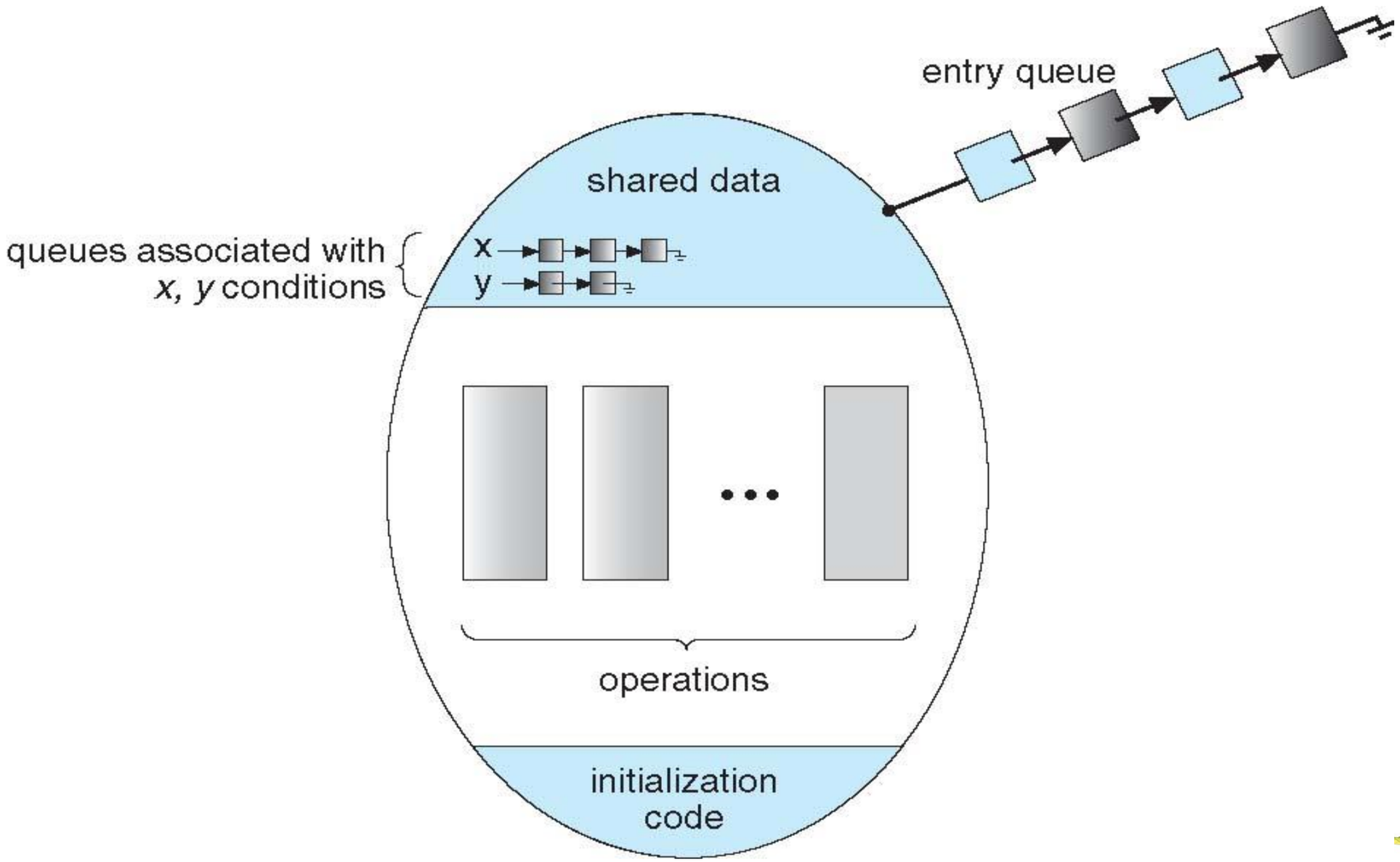
- Fakat bazı senkronizasyon şemalarını modellemek için yeterince güçlü değildir.
- Bu amaçla, ek senkronizasyon mekanizmaları tanımlamamız gerekir. Bu mekanizmalar durum (condition) yapısı tarafından sağlanır.
- Programcı işe veya **değişkene özel senkronizasyon** oluşturmak için durum değişkenleri oluşturabilir.

- condition x, y; //x ve y durumları

- Durum değişkenleri üzerinde iki işlem yapılabilir:
  - x.wait () –işlemi çağıran bir proses x.signal () işlevine kadar askıya alınır.
  - x.signal () – (varsa) x.wait () ile beklemeye alınmış proses lerden biri devam ettirilir.
    - x.wait () ile beklemeye alınmış değişken yoksa, değişken üzerinde hiçbir etkisi yoktur.

Durum Değişkenleri İle Monitör

- **x** ve **y** durum değişkenlerine bağlı process'lerin monitör içinde çalışması.



Durum Değişkenleri Seçimleri

- **x.signal()** işlemini **bir P process'i çağırmış olsun**. Aynı anda, **x** durumuna bağlı **x.wait()** ile **beklemekte olan bir Q process'i de olsun.** , daha sonra ne olmalı?
- Monitör içindeki P ve Q aynı anda aktif olamaz.
- **Q process'i** çalışmaya **başlarsa**, **P process'i beklemek zorundadır.**
- Seçenekler şunlardır :
- **Signal and wait: P process'i**, **Q process'inin monitör'den ayrılmasını** veya başka bir duruma geçmesini **bekler.**
- **Signal and continue:** Q process'i, **P process'inin monitör'den ayrılmasını** veya başka bir duruma geçmesini **bekler.**
- **İkisininde artıları eksileri vardır.**
- Birçok programlama dili, Java ve C # de dahil olmak üzere bu bölümde anlatıldığı şekilde monitör fikrinde birleşmişlerdir.

Yemek Yiyen Filozofların Çözümü

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

      public void pickup (int i) {
state[i] = State.HUNGRY;
test(i);
if (state[i] != State.EATING)
self[i].wait;
}
      public void putdown(int i) {
state[i] = State.THINKING;
// test left and right neighbors
test((i + 4) % 5);
test((i + 1) % 5);
}
```

Yemek Yiyen Filozofların Çözümü (Devam)

```
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;
}
}

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

}
```

Yemek Yiyen Filozofların Çözümü (Devam)

- Her filozof aşağıdaki sırayla pickup() ve putdown() fonksiyonlarını çağırır :

```
DiningPhilosophers.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

- Deadlock (kilitlenme) olmaz fakat starvation (açlık) mümkün.

Semafor Kullanarak Monitör Uygulama

- Değişkenler

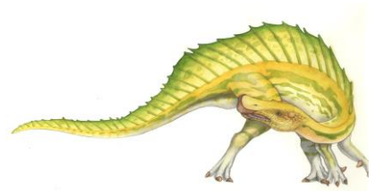
```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Her prosedür ***F*** bu şekilde değiştirilecektir.

```
wait(mutex);
    …
        body of F;

    …
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```

- Monitörün içinde mutual exclusion (karşılıklı dışlama) sağlanır.



Monitor Uygulama – Durum Değişkenleri

- Her koşul değişkeni ***x*** için şunlara sahibiz :

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- x.wait işlemi şu şekilde uygulanabilir :

```
x_count++;
if (next_count > 0)
      signal(next);
else
      signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Uygulama (Devam)

- x.signal işlemi şu şekilde uygulanabilir :

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Bir Monitör İçindeki Processleri Sürdürme

- X durum kuyruğunda birden fazla process mevcutsa ve x.signal() çalıştırıldıysa, hangisi sürdürülmelidir?

- FCFS (first come first serve -ilk gelen ilk işlem görür) genellikle yeterli değildir.

- **conditional-wait (koşullu bekleme)** x.wait(c) biçiminde form oluşturulur.
  - C, **priority number** (öncelik sayısı)dır.
  - Düşük numaralı proses (yüksek öncelik) bir sonraki işlem olarak çizelgelenir

Single Resource allocation

- Bir işlemin kaynağı kullanmayı planladığı maksimum adeti/süreyi belirten öncelik numaraları kullanarak rakip işlemler arasında tek bir kaynak tahsis edilir.

```
R.acquire(t);
    ...
  access the resource;
    ...
R.release;
```

- R, **`ResourceAllocator`** türünde bir örnektir.

Tek bir kaynağı tahsis etmek için bir Monitör

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
            if (busy)
                x.wait(time);
            busy = TRUE;
    }
    void release() {
            busy = FALSE;
            x.signal();
    }
initialization code() {
     busy = FALSE;
    }
}
```

Senkronizasyon Örnekleri

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Senkronizasyon

- Multitasking, multithreading (gerçek zamanlı threadler de dahil olmak üzere) ve multiprocessing desteklemek için çeşitli kilitler uygular.(uyarlanabilir muteksler (adaptive mutex locks), durum değişkenleri, semaforlar, okuyucu-yazıcı kilitleri , ve turnstiles)

- Kısa kod bölümlerinde verimlilik sağlamak için **adaptive mutexes (uyarlanabilir muteksler)** kullanır.
  - standart semafor spin-lock gibi başlar.
  - Eğer kilit tutulursa, diğer bir CPU'da çalışan thread spin atar.
  - Eğer kilit çalışma durumunda olmayan bir thread tarafından tutulursa , bloke etmiş olur ve kilidin serbest bırakılma sinyali beklenirken diğer thread ler uyur. Bu şekilde boşa spin atmazlar

- **condition variables (durum değişkenleri)** kullanılır

Solaris Senkronizasyon

- Uzun bölümlerde kod veriye erişim ihtiyacı duyduğunda **okuyucu-yazıcı kilitleri** kullanır. Okuyucu-yazar kilitleri, sık erişilen verilerin korunması için kullanılır, ancak genellikle salt okunur (read-only) bir şekilde erişilir.

- Uyumlu muteks veya okuyucu-yazar kilidi elde etmeyi bekleyen iş parçacıklarının listesini düzenlemek için **turnikeler** (**turnstiles**) kullanır.
  - Bir turnike, bir kilit üzerinde bloke edilen iş parçacıklarını içeren bir sıra yapısındadır.
- Turnikelerde, daha düşük öncelikli bir iş parçacığı şu anda daha yüksek öncelikli bir iş parçacığının engellediği bir kilidi bulundurursa, daha düşük öncelikli iş parçacığı, daha yüksek öncelikli iş parçacığının önceliğini geçici olarak kalıtım alır. Kilit serbest bırakıldıktan sonra iş parç. orijinal önceliğine geri dönecektir.

Windows XP Senkronizasyon

- Tek işlemcili sistemlerde global kaynaklara güvenli erişimi korumak için interrupt masks (kesme maskeleri) kullanır.
- Birden fazla işlemcili sistemlerde **spinlocks** kullanır.
  - Spinlocking-thread asla preempted (işlem önceliğine sahip) olmamalıdır.
- Ayrıca kullanıcı tarafında muteksler, semaforlar, olaylar ve zamanlayıcılar gibi davranan **dispatcher objects (dağıtıcı nesneler)** sağlar.
  - **Events (olaylar)**
    - Bir olay daha çok bir durum değişkeni gibi davranır.
  - Timers (zamanlayıcılar) süre aşıldığında (time expired) bir ya da daha fazla thread a bildirir.
  - Dispatcher nesneleri ya sinyal-edilmiş durumunda (nesne mevcut) veya sinyal-edilmemiş durumda (thread bloke edecek)

Linux Senkronizasyon

- Linux:
  - Kernel 2.6 versiyonundan önceki versiyonlarda, kritik bölgeleri tamamlamak için kesmeler devre-dışı bırakılır.
  - Versiyon 2.6 ve sonrası, tamamen kesintili
- Linux şunları sağlar :
  - semaforlar
  - spinlocks
  - Hem okuyucu-yazıcı sürümleri
  - Atomik tam sayılar
- Tek CPU'lu sistemlerde, spinlock'lar, kesintili-çekirdek özelliğini etkinleştirme ve devre dışı bırakma ile değiştirildi

Pthreads Senkronizasyon

- Pthreads API, OS-independent (işletim sisteminden bağımsız)'dır.
- Şunları destekler:
  - muteks kilitleri
  - koşul değişkenleri
- Taşınabilir olmayan uzantılar şunları içerir:
  - okuma-yazma kilitler
  - spinlocks

Alternatif Yaklaşımlar

- ***İşlemsel bellek (Transactional memory)***
- ***OpenMP (Open Multi-Processing)***
- **Fonksiyonel Programlama Dilleri (Örn: Java)**

İşlemsel bellek

- **Multicore sistemlerde**, **mutex lock**, **semafor** gibi mekanizmalarda **deadlock** gibi problemlerin **oluşma riski bulunmaktadır.**
- Bunun yanı sıra, **thread sayısı arttıkça deadlock problemlerinin ortaya çıkma olasılığı artmaktadır.**
- **Klasik mutex lock (veya semafor)** kullanılarak paylaşılmış veride güncelleme yapan **update()** fonksiyonu aşağıdaki gibi yazılabilir di!

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

İşlemsel bellek

- Klasik kilitleme yöntemlerine alternatif olarak **programlama dillerine yeni özellikler eklenmiştir.**
- Bellek işlemi (Memory trasnsaction), atomik olan okuma-yazma işlemleri dizisidir.
- Örneğin, **atomic(S)** kullanılarak **S işlemlerinin tümünün işlemsel (transactional) olarak gerçekleştirilmesi sağlanır.(yukarı grafikteki metot yerine aşağıdaki kod yazılır)**

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

- **Lock işlemine gerek kalmadan** ve **kilitlenme olmadan işlem tamamlanır.**

OpenMP (Open Multi-Processing)

- OpenMP, C, C++ ve Fortran için compiler direktiflerinden oluşan API'dir.
- OpenMP, paylaşılmış hafızada eşzamanlı çalışmayı destekler.
- OpenMP, **#pragma omp critical** komutu ile kritik bölümü belirler ve aynı anda sadece bir thread çalışmasına izin verir.

```
void update(int value)
{
   #pragma omp critical
   {
      counter += value;
   }
}
```

Fonksiyonel Programlama Dilleri

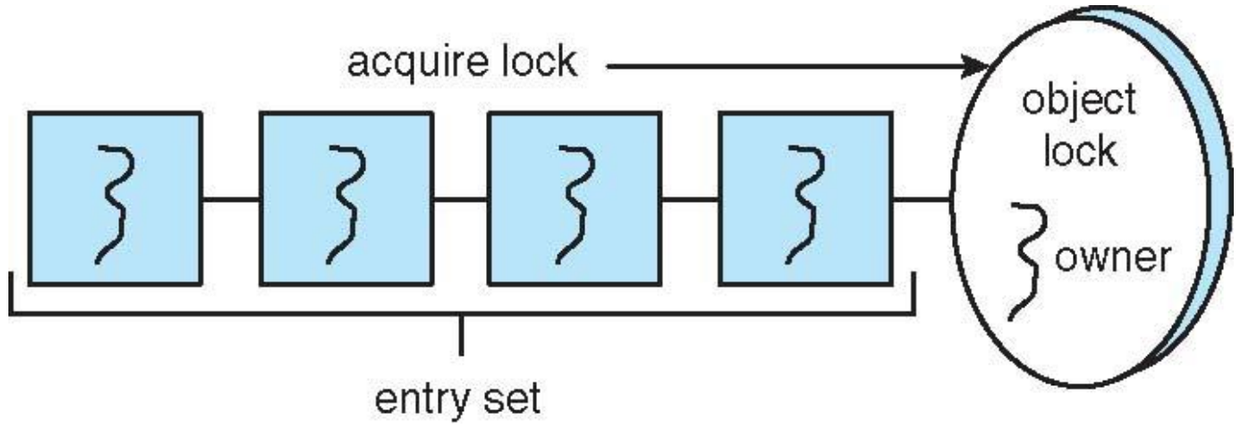
- Veri yarışlarını ele alma yaklaşımlarından dolayı Erlang ve Scala veya Java gibi Fonksiyonel dillere olan ilginin artması dikkat çekiyor.
- Java, dil seviyesinde senkronizasyon sağlar.
- Her bir Java nesnesinin ilişkili bir kilidi vardır.

Java Synchronization

- Java, dil seviyesinde senkronizasyon sağlar.
- Her bir Java nesnesinin ilişkili bir kilidi vardır.
- Bu kilit, senkronize edilmiş bir metot (synchronized method) çağırarak elde edilir.
- Senkronize edilmiş metotdan (synchronized method.) çıkılırken bu kilit serbest bırakılır.
- Nesne kilidini elde etmek için bekleyen iş parçacıkları, nesne kilidinin giriş kümesine yerleştirilir.

Java Synchronization

- Her nesnenin ilişkili giriş kümesi (**entry set**) vardır.



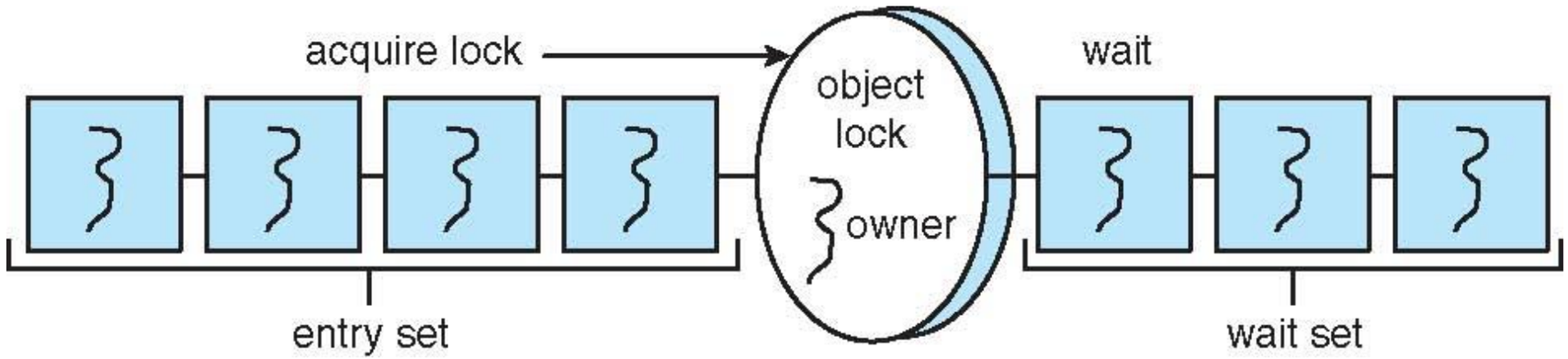
Java Synchronization wait/notify()

- Bir iş parçacığı **wait ()** işlevini çağırdığında:
  1. iş parçacığı, nesne kilidini serbest bırakır;
  2. İş parçacığının durumu bloke edildi (blocked) olarak ayarlanır;
  3. İş parçacığı, nesne için bekleme kümesine yerleştirilir.

- Bir iş parçacığı, **notify()** işlevini çağırdığında
  1. Bekleme kümesindeki keyfi bir İş parçacığı T seçilir;
  2. T bekleme durumundan giriş kümesine geçer;
  3. T durumu Runnable olarak ayarlanır.

Java Synchronization

- Entry and wait sets



Java Synchronization – wait/notify

- Synchronized insert() method – Correct!

```
// 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();
}
```

Java Synchronization – wait/notify

- Synchronized remove() method – Correct!

```
// 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;
}
```

Bölüm 6 Sonu