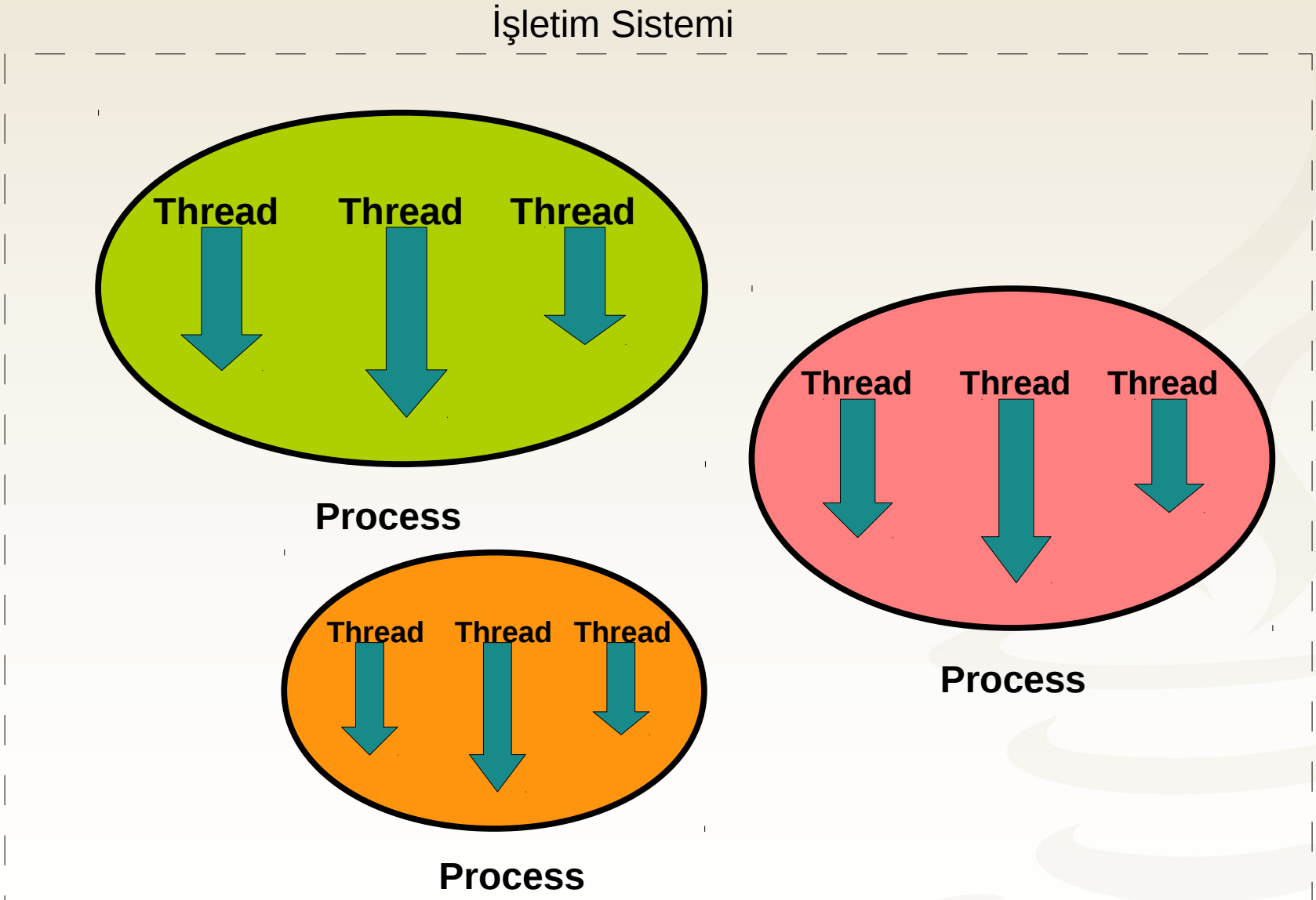
JAVA Eğitimleri

Java Threads & Concurrency

Multitasking/Multiprocessing Nedir?

- İşletim sisteminde aynı anda **pek çok process** yer alır
- Her process içinde de bir veya daha fazla task(**thread**) yer alabilir
- Thread, herhangi bir process içerisinde belirli bir görevi (**task**) yerine getiren işlem parçacığıdır
- Task**'ın tamamlanması ile birlikte** Thread'de sonlanır

Multitasking/Multiprocessing Nedir?



Thread Scheduling İşlemi

JAVA Eğitimleri

- Sistem, periyodik olarak **her process'e belirli bir süre** CPU kullanım izni verir
- Process'de içindeki her thread'e belirli bir süre **taskı tamamlaması** için imkan tanır
- Sistemlerde Thread'ler arası CPU kullanımı değişik yöntemlerle yapılabilir
- **Preemptive bir sistem**de her bir Thread'e çalışması için **belirli bir zaman aralığı** verilir
- **Kooperatif bir sistem**de ise her bir thread **gönüllülük usulü**ne göre kontrolü istediği zaman diğer bir thread'a devreder

Thread Scheduling İşlemi

- JVM'de bir schedule mekanizması **preemptive usule göre** çalışır
- JVM her bir thread'in **belirli zaman aralıklarında çalışması**nı sağlar
- Bu zaman aralığı dolduğu vakit thread'in çalışması askıya alınır, **context switch** yapılır ve diğer bir thread'a geçilir

Task & Runnable Arayüzü

```
public class Sayac implements Runnable {

   private int baslangic;
   private int bitis;
   private int artis;

   public Sayac(int bas, int bit, int art) {
      this.baslangic = bas;
      this.bitis = bit;
      this.artis = art;

   }
   public void run() {
      for(int i = baslangic; i <= bitis; i+=artis){
         System.out.println(i + " :" +
               Thread.currentThread().getName());
      }
   }
}
```

Her task Runnable interface'ini implement eder

İhtiyaç duyulan bilgi task'a Constructor aracılığı ile iletilir

Run metodu İçerisinde task'ın Yerine getireceği Görev implement edilir

Run metodunun sonlanması thread'in de sonlanması demektir

Thread Başlatılması

```
Sayac tek = new Sayac(1,50,2);
Sayac cift = new Sayac(0,50,2);

Thread t1 = new Thread(cift);
Thread t2 = new Thread(tek);

t1.start();

t2.start();

0 :Thread-0
2 :Thread-0
1 :Thread-1
3 :Thread-1
4 :Thread-0
6 :Thread-0
5 :Thread-1
8 :Thread-0
7 :Thread-1
...
```

Task'lar yaratıldıktan sonra her birisi İçin ayrı bir Thread instance'ı oluşturulur. Task'lar bu instance'lara parametre olarak verilirler

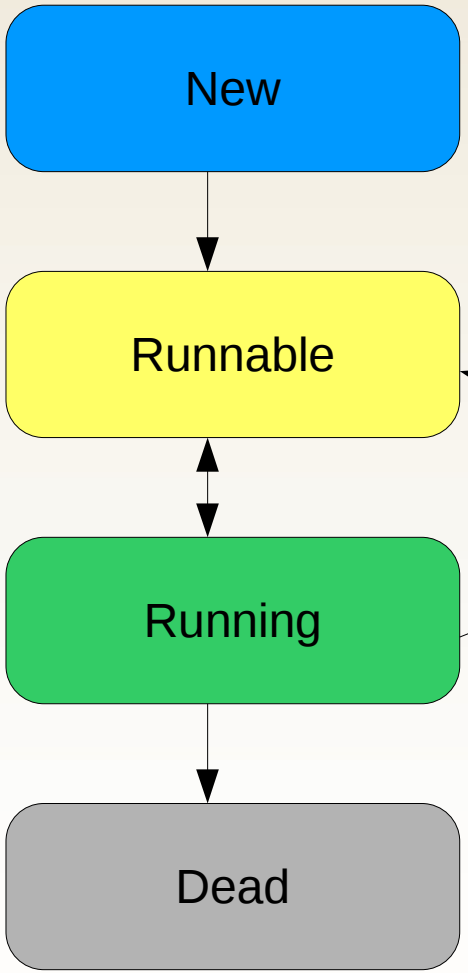
Thread.start() metodu ile thread çalışmaya başlar

Başlatılan bir thread'i durdurmak(stop) mümkün değildir

Thread'in sonlanması için run metodundan çıkılması gerekir

JVM scheduler sistemdeki thread'lere belirli zaman aralıklarında CPU cycle'ı kullanmaları için fırsat tanır

Thread Yaşam Döngüsü



new Thread() ifadesi ile yeni bir Thread yaratılır
Thread doğrudan runnable state'ine geçmiştir

Schduler, uygun gördüğü zaman aralıklarında Thread'i çalıştırır

Run metodu sürdüğü müddetçe thread Runnable ile running stateleri arasında gidip gelir

Run metodunun sona ermesi ile thread instance'ı da ölmüş olur

Ölü thread instance'ını tekrar kullanmak mümkün değildir

Waiting/Blocked

Çalışan thread belirli bir süre çalıştıktan Sonra scheduler tarafından beklemeye Alınabilir

Ya da thread işine devam edebilmesi Bir resource'un hazır olmasını bekleyebilir

Thread Sleep İşlemi

```
public void run() {

   for(int i = baslangic; i <= bitis; i+=artis) {
      System.out.println(i + " :" +
            Thread.currentThread().getName());
      try {
         Thread.sleep(100);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

Thread belirtilen msec kadar süre uyku moduna geçecektir, Scheduler başka bir thread nesnenin cpu cycle'ı kullanmasını sağlar

Bu süre zarfında eğer thread instance'ın sahip olduğu monitor(lock) var ise bunu tutmaya devam eder

Daemon Thread

- **JVM**, main metodu sonlandığı vakit terminate olur, ancak **herhangi bir normal thread varsa** bu thread(lerin) sonlanmasını **bekler**
- Eğer bütün **normal thread**'ler sonlanmış ise geri kalan daemon threadlerin **sona ermesi beklenmeden** uygulama sonlandırılır
- **Daemon thread'ler** uygulama çalıştığı müddetçe **arka plan**da belirtilen bir görevi yerine getirmeye çalışır
- **Thread.setDaemon(true)** ile thread nesnesinin, **çalışmaya başlamadan evvel** daemon olduğu belirtilmelidir

Daemon Thread

```
Thread t1 = new Thread(new Sayac(0,50,2));
t1.setDaemon(true);

Thread t2 = new Thread(new Sayac(1,50,2));
t2.setDaemon(true);

t1.start();
t2.start();
```

```
0 :Thread-0
2 :Thread-0
JVM exit
```

Her iki thread nesnesi de daemon thread olduğu için, thread nesneleri task'larını bitirmeye fırsat bulamadan main Thread sonlanacağı için uygulama sona erer.

Thread & Exception

```
public void run() {


   for(int i = baslangic; i <= bitis; i+=artis) {
      System.out.println(i + " :" +
      Thread.currentThread().getName());
   }

   if(true)
         throw new RuntimeException("
            sayac içerisinde bir hata meydana geldi!");

}
```

Eğer bir **exception run() metodunun dışına çıkmış ise** bu exception console'a kadar taşınacaktır

Thread & Exception

```
public class SayacExceptionHandler
                    implements UncaughtExceptionHandler {

   @Override
   public void uncaughtException(Thread t, Throwable e) {
      System.out.println(
         "Hatanın meydana geldiği thread :" + t.getName());
      System.out.println("Hata mesajı :" + e.getMessage());
   }
}
```

Herhangi bir exception run metodundan fırlatıldığı vakit thread nesnesi ölmeden evvel **uncaughtException( )** metodu çağrılır

Thread & Exception

```
Thread t1 = new Thread(new Sayac(0,50,2));

t1.setUncaughtExceptionHandler(new SayacExceptionHandler());

t1.start();

...
46 :Thread-0
48 :Thread-0
50 :Thread-0
Hatanın meydana geldiği thread :Thread-0
Hata mesajı :sayac içerisinde bir hata meydana geldi!
```

setUncaughtExceptionHandler() metodu ile thread nesnesine bir **exception handler** bağlanmış olur

Race Condition ve Kaynak Paylaşımı

JAVA Eğitimleri

- Eşzamanlı programlamada **iki veya daha fazla task**'ın aynı anda **tek bir kaynağa erişim**i söz konusu olabilir
- İki farklı thread'in ortak bir kaynak üzerinde değişik sırada işlem gerçekleştirmesi farklı sonuçlar üretebilir
- Buna **race condition** adı verilir
- Race condition'ları engellemek için ortak kaynaklara **sıralı biçimde erişmek** gerekir

Race Condition ve Kaynak Paylaşımı

Transaction 1

Read bakiye

Bakiye = bakiye + 100

Update bakiye

Transaction 2

Read bakiye

Bakiye = bakiye + 200

Update bakiye

Thread 1 read bakiye	0 TL
Thread 1 update bakiye	100 TL
Thread 2 read bakiye	100 TL
Thread 2 update bakiye	300 TL

1

Thread 1 read bakiye	0 TL
Thread 2 read bakiye	0 TL
Thread 1 update bakiye	100 TL
Thread 2 update bakiye	200 TL

2

Thread 2 read bakiye	0 TL
Thread 1 read bakiye	0 TL
Thread 2 update bakiye	200 TL
Thread 1 update bakiye	100 TL

3

Banka Hesabı
Bakiye:0TL

Critical Section Nedir?

Resimde her iki tren hattı'nın da **kesiştiği kısım ortak kaynak**tır ve critical section olarak adlandırılabilir. Kritik bölgenin iki hat tarafından **sıralı biçimde kullanılması** gerekmektedir

Lock/Mutex/Monitor ve Semaphore Kavramları

- Ortak kaynaklara erişimin sıraya sokulması kilit(**lock**) mekanizması ile sağlanır
- Ortak kaynağa erişimin sıralı hale getirilmesi için ilgili kod bloğunun etrafına sadece **tek bir taskın erişimini düzenleyen bir ifade** konur
- Bu ifade "**mutual exclusion**" sağladığı için yaygın ismi "**mutex**"dir
- Java'daki **bütün nesneler ve sınıflar** built-in kilit yapısına sahiptir
- Bu yapıya "**monitor**" veya **semaphore** adı verilir

Syncronized Block

```
public class Banka {

    private Hesap hesap = new Hesap();

    public void paraEkle(int miktar) {

        synchronized (hesap) {
            int bakiye = hesap.getBakiye();
            bakiye += miktar;
            hesap.setBakiye(bakiye);
        }

    }
}
```

Ortak kaynağa erişmeden evvel bu kaynak üzerinde bir lock atılır

Synchronized ifadesi blok içine erişmeden evvel thread'in Hesap nesnesine ait monitor'ü kilitlemesini sağlar

Kilit alınamaz ise thread kilit alınana değin bloğu çalıştıramayacaktır

Synchronized Metot

```
public class Hesap {
    private int bakiye;

    public int getBakiye() {
        return bakiye;
    }

    public void setBakiye(int bakiye) {
        this.bakiye = bakiye;
    }

    public synchronized void paraEkle(int miktar) {
        int bakiye = this.getBakiye();
        bakiye += miktar;
        this.setBakiye(bakiye);
    }
}
```

Metotlara erişmeden evvel metodun çağrıldığı nesnenin üzerinde kilit alınmaya çalışılır

Synchronized anahtar kelimesinin metot üzerinde kullanılması ile aşağıdaki gibi bir blok şeklinde kullanılması aynı şeydir

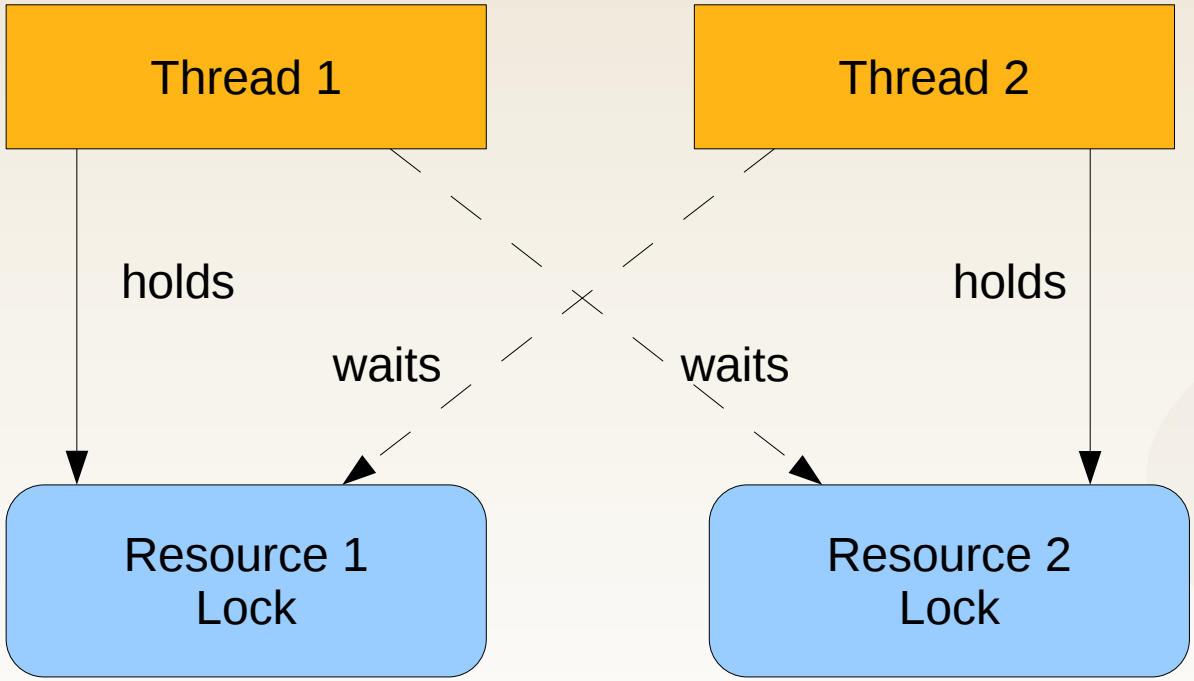
```
public void paraEkle(int miktar) {

    synchronized (this) {
        int bakiye = this.getBakiye();
        bakiye += miktar;
        this.setBakiye(bakiye);
    }

}
```

Statik metotlarda kullanıldıkları takdirde sınıf düzeyinde kilit atılmış olunur

Deadlock Nedir?



Thread 1 işini tamamlamak için resource 2'ye ihityaç duyuyor, ancak Resource 2 o an da Thread 2 tarafından tutuluyor. Bu nedenle Thread 1 beklemek zorunda kalıyor. Aynı şekilde Thread 2 işini tamamlamak için Resource 1'e ihtiyaç duyuyor, ancak o da Thread 1 tarafından tutulduğu İçin Thread 2'de beklemek zorunda kalıyor.

Deadlock, iki thread arasında doğrudan veya dolaylı biçimde ortaya çıkabilir.

Thread Dump

- CPU'daki yığılmayı, geç cevapları, deadlock problemlerini vs tespit etmek için JVM'deki **thread'lerin durumunu gösteren bir snapshot**'tur
- Farklı yöntemlerle elde edilebilir
  - JDK_HOME/bin/jcmd ile java process'inin PID'i tespit edilebilir
  - JDK_HOME/bin/jstack -l <PID> ile snapshot elde edilebilir
  - Kill -3 <PID> ile snapshot uygulamanın std err çıktısına yazdırılabilir

İletişim

www.harezmi.com.tr

www.java-egitimleri.com

info@harezmi.com.tr

info@java-egitimleri.com

@HarezmiBilisim

@JavaEgitimleri