СПбГЭТУ «ЛЭТИ»
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

Parallel Computing

Concurrent Programming in Java

Zahra Dorostkar
530020
2020

content

- What is thread
- Advantages of using thread
- Thread life cycle diagram
- How to create thread
- Inheritance of thread class
- Implementing runnable interface
- ExecuterService
- Functions in thread class
- Daemon Thread
- Class Callable
- Synchronization
- Lock
- Synchronized
- Using Semaphore
- CountDownLatch
- CyclicBrarrier

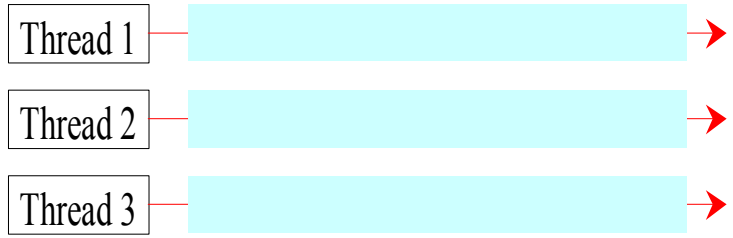
Thread : **Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process**

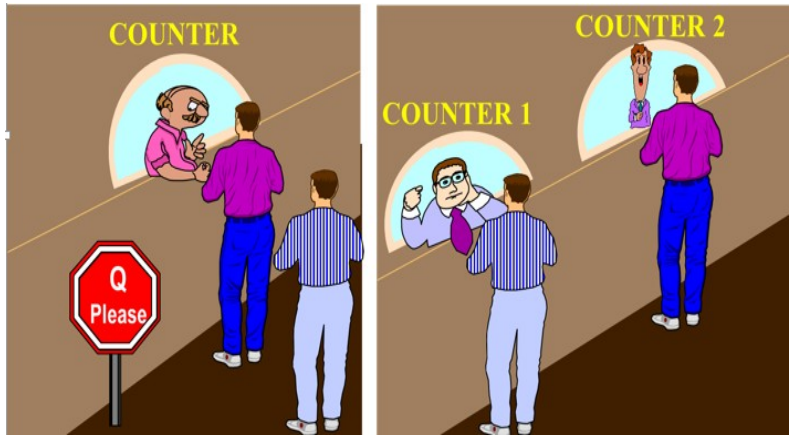
Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files

Thread using advantages :

Multiple threads on multiple CPUs :

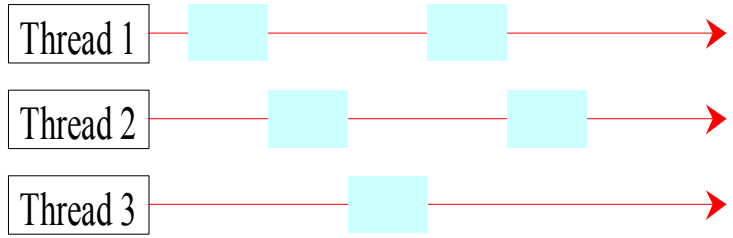
Increasing efficiency

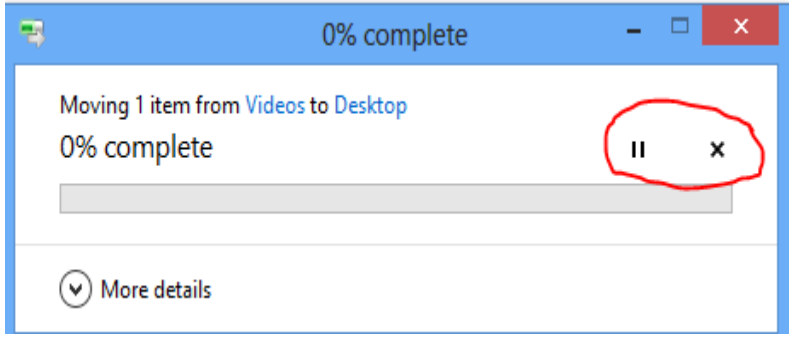




Multiple threads sharing a single CPU :

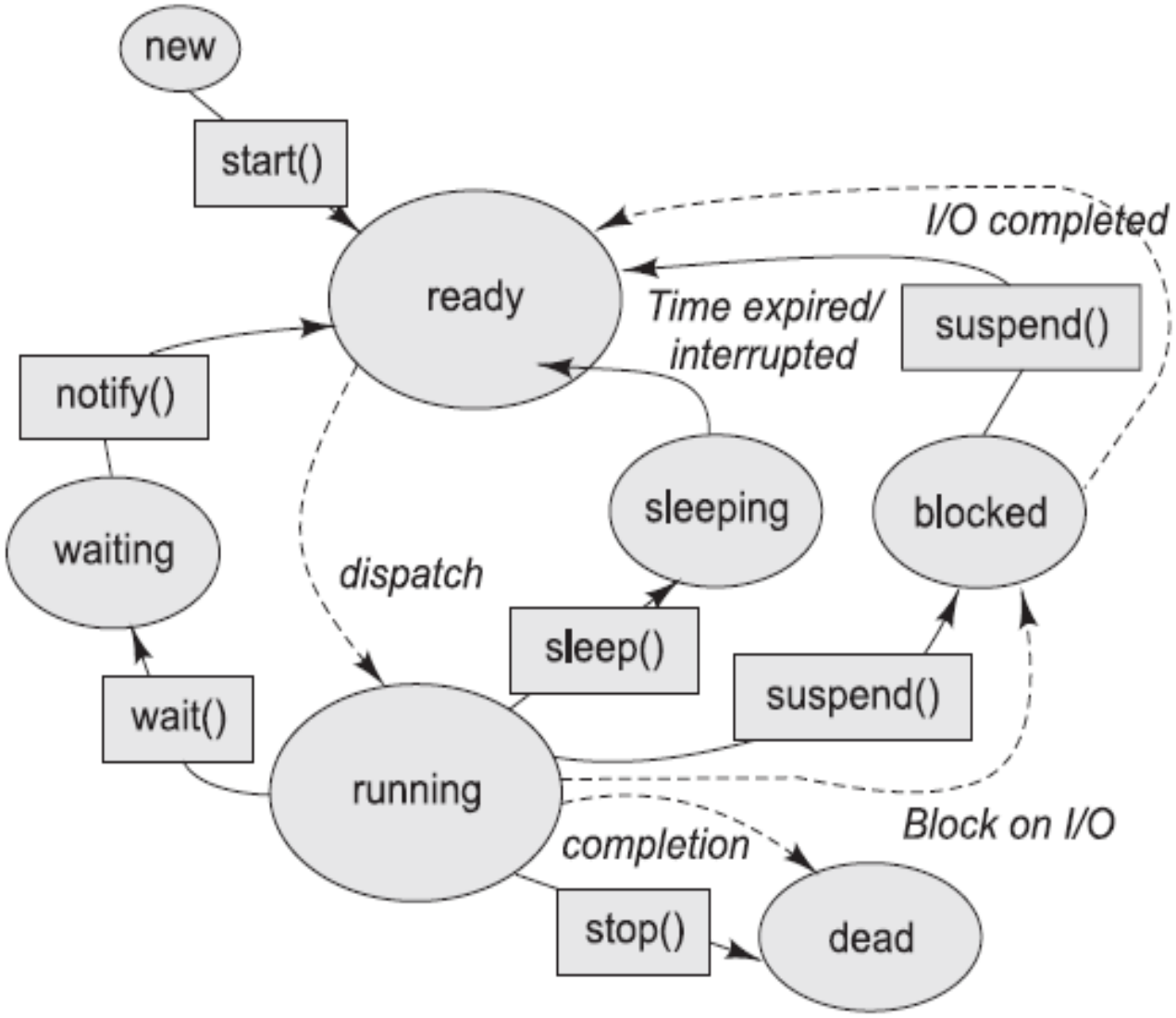
Increasing response time





Thread life cycle

A thread from start to end can have these states:



Ways of creating thread

1. Inheritance of thread class

2. Implementing Runnable interface

3. Using of ExecuterService

1. Inheritance of Thread class :

```
class MyThread extends Thread
    {
      public void run()
      {
            // thread body of execution
      }
     }
```

- Creating thread:

```
MyThread thr1 = new MyThread();
```

- Start Execution:

```
thr1.start();
```

2. Implementing Runnable interface :

```
class ClassName implements Runnable
{
   .....
   public void run()
   {
      // thread body of execution
   }
}
```

Creating Object:

```
ClassName myObject = new ClassName();
```

Creating Thread Object:

```
Thread thr1 = new Thread( myObject );
```

Start Execution:

```
thr1.start();
```

✓ Threads are on-time used.

3. ExecuterService :

import java.util.concurrent.*;

```
class ClassName implements Runnable
{
  .....
  public void run()
  {
     // thread body of execution
  }
}
```

Creating Object:

```
ClassName myObject = new ClassName();
```

Creating ExecutorService Object:**(there are three ways to creating ExecutorServises)**

```
I. ExecutorService exe1= Executors.newCachedThreadPool();
II.ExecutorService exe2=Executors.newFixedThreadPool(number of Threads);
III.ExecutorService exe3=Executors.newSingleThreadExecutor();
```

Start Execution:

```
exe1.execute();
```

Terminate Execution:

```
exe1.shutdown();
```

Some functions of Thread class

java.lang.Thread	
+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

Some functions of Thread class (cont.)

- Sleeping:
  - Thread.sleep(calculated in ms);
  - TimeUnit.DAYS.sleep(timeout);
  - TimeUnit.HOURS.sleep(timeout);
  - TimeUnit.MINUTES.sleep(timeout);
- setPriority :

```
Thread t =new Thread(Runnable Object);
```

  - t.setPriority(newPriority); // 1<=newPriority<=10 and 10 is max priority
- Yield :
  - Thread.yield();
- Therad name :
  - Thread.currentThread().getName() //Returns this thread's name

Daemon Thread

Daemon is a thread that immediately stops when the main thread stops no matter its job finished or not.

For creating daemon :

```
Thread t = new Thread();

t.setDaemon(True);
```

Class Callable

When the thread needs to return the result :

```
class ClassName implements Callable<T>
{
   .....
   public <T> call()
   {
      // thread body of execution
   }
}
```

Creating Callable Object:

```
ClassName myObject = new ClassName();
```

Creating Thread Object:

```
ExecutorService exe1= Executors.newCachedThreadPool();
```

Result :

Future<T> result = new Future<T>

Start Execution:

result = exe1.submit(myObject);

Using Result :

result.get();

Synchronization

1. Using Lock

2. Using “ Synchronized " keyword

3. Using Semaphore

1. Using Lock :

```
import java.util.concurrent.locks.*;

Lock lock = new ReentrantLock();
try {
                                    lock.lock();
                                        critical section
     } catch (Exception e) {}

finally {
lock.unlock();
          }
```

2. Using " Synchronized " keyword :

```
synchronized void f() { /* body */ }
```

is equivalent to

```
void f() { synchronized(this) { /* body */ } }

Class Mytask implements Runnable{


Puplic synchronized methode1(int sharedValue)
{
        sharedValue ++;
}

Public void run(){
       methode1(int sharedValue);
}

}
```

3. Using Semaphore :

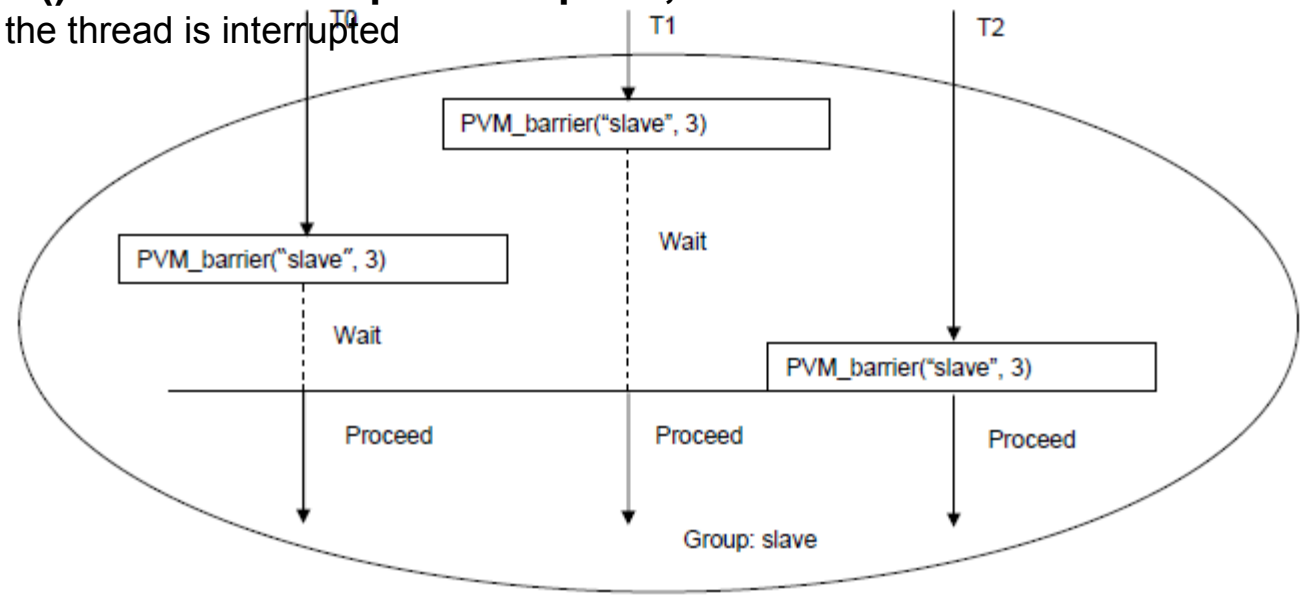
```
import java.util.concurrent.Semaphore;

Semaphore s= new Semaphore(n , True/False) //Creates a Semaphore with the given number of permits and the given fairness setting
Try{
                    s.aquire();
          Critical section
    } catch (Exception e) {}

s.release;
```

CountDownLatch

- ✓ They are used for making threads concurrent
- ✓ They can be used one time.
- ✓ They have two methods :

public void countDown(); //Decrements the count of the latch, releasing all waiting threads if the count reaches zero

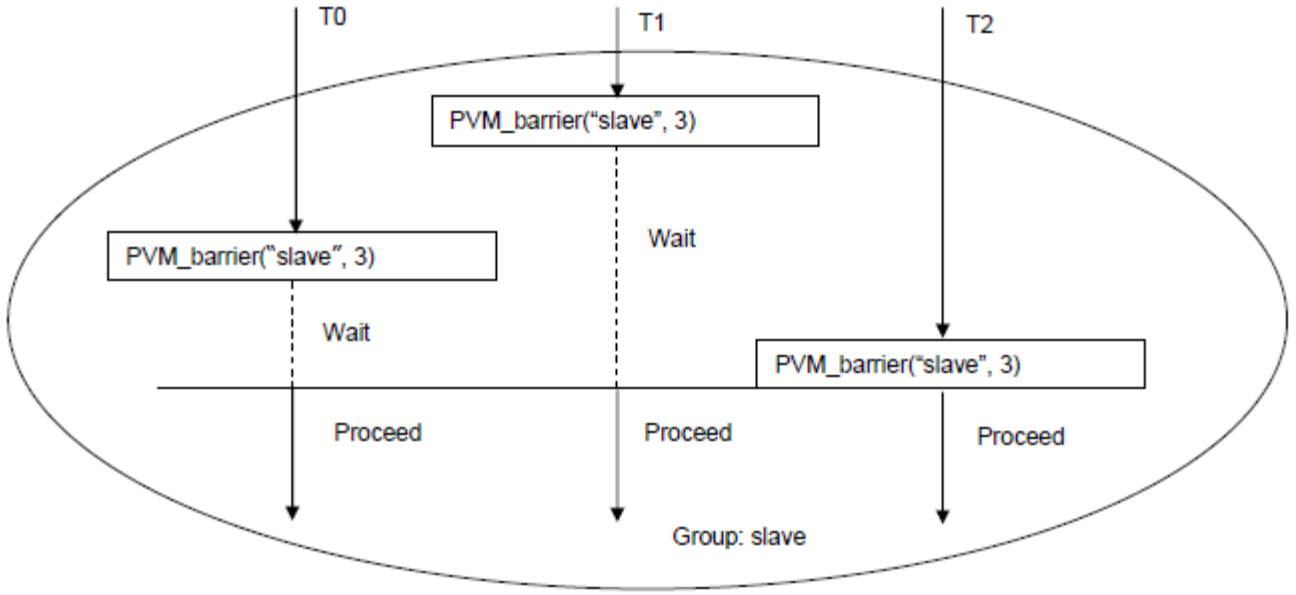
public void await() throws InterruptedException; //Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted



CyclicBrarrier

- ✓ They are used for making threads concurrent
- ✓ They can be used multiple times.
- ✓ They have this method:

public int await() throws InterruptedException; BrokenBarrierException //Waits until all parties have invoked await on this barrier



Thank you for your attention