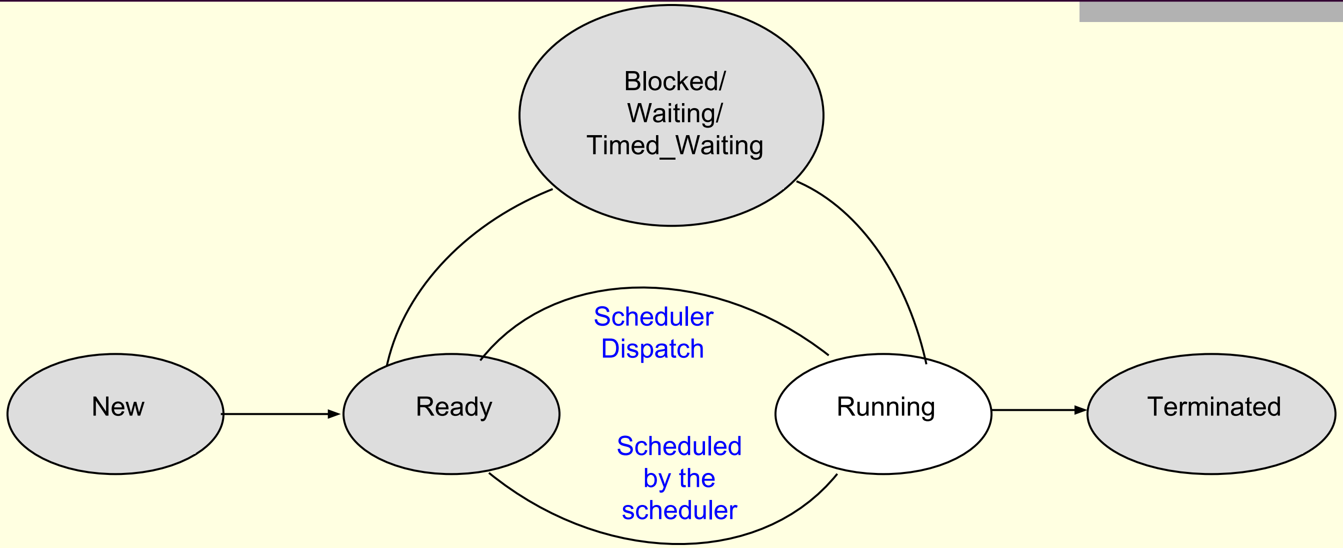
Multithreading

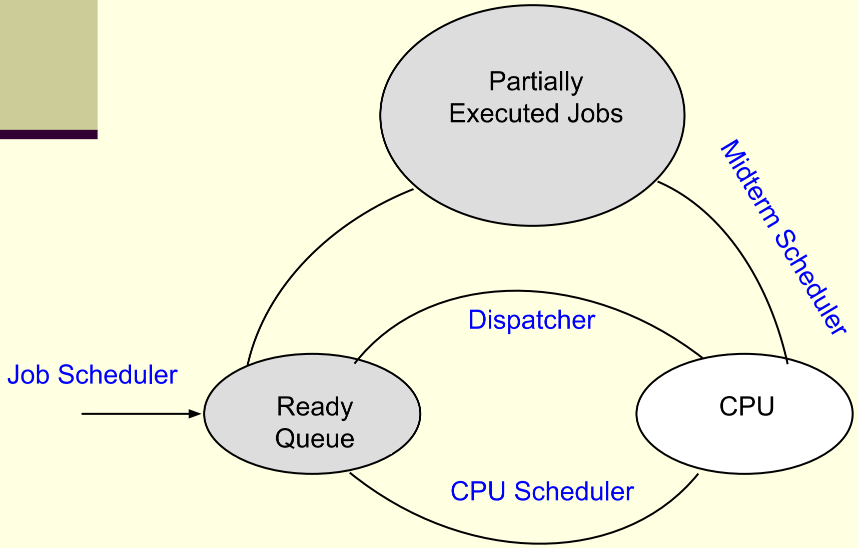
By

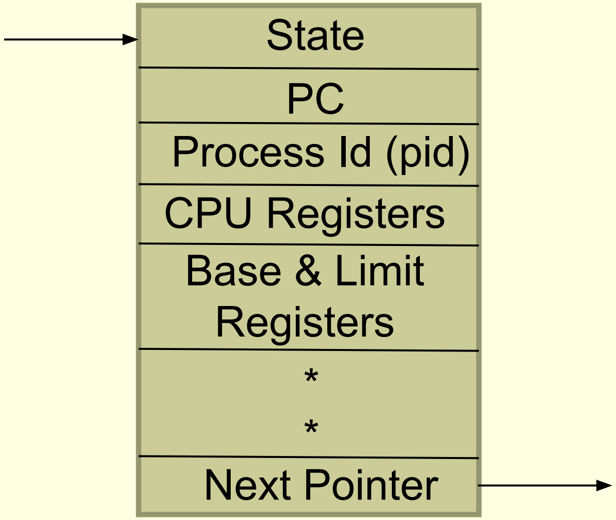
Sunirmal Khatua

University of Calcutta

Multiprocessing







State

PC

Process Id (pid)

CPU Registers

Base & Limit
Registers

*

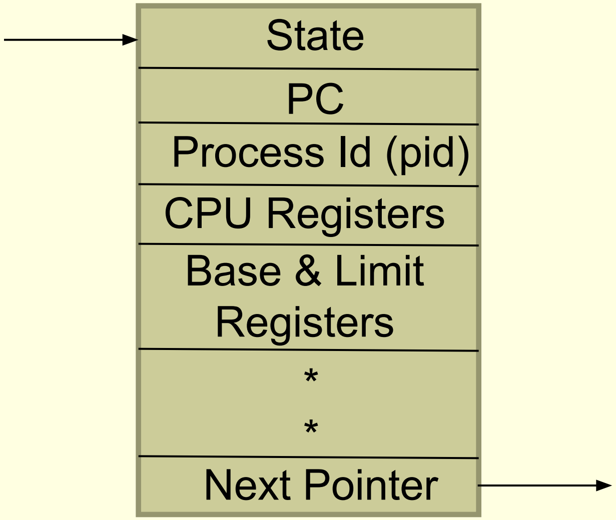
*

Next Pointer

Multithreading

- ❑ *Threads* are separate tasks running within a Program
- ❑ *Multitasking* allows an OS to run two or more programs simultaneously.
- ❑ A thread based multitasking is called *Multithreading*
- ❑ Threads are also called Lightweight Process

Process Control Block (PCB)



State

PC

Process Id (pid)

CPU Registers

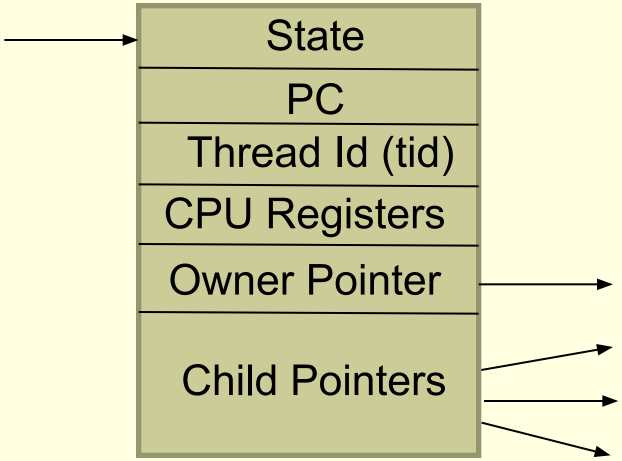
Base & Limit Registers

*

*

Next Pointer

Thread Control Block (TCB)



State

PC

Thread Id (tid)

CPU Registers

Owner Pointer

Child Pointers

Some Processes

```
public class Account {
      private int balance;
      public int getBalance() {
            return balance;
      }
      public void setBalance(int balance) {
            this.balance = balance;
      }
}


public class Depositor {
      private Account account;
      public Depositor(Account account) {
            this.account = account;
      }
      public void deposit(int amount) {
            int balance = account.getBalance();
            balance = balance + amount;
            account.setBalance(balance);
      }
      public static void main(String[] args) {
            Account account = getAccount();
            Depositor depositor = new Depositor(account);
            depositor.deposit(500);
      }
}
```

```
public class Withdrawer {
      private Account account;
      public Withdrawer(Account account) {
            this.account = account;
      }

      public void withdraw(int amount) {
            int balance = account.getBalance();
            balance = balance - amount;
            account.setBalance(balance);
      }

      public static void main(String[] args) {
            Account account = getAccount();
            account.setBalance(1000);
            Withdrawer withdrawer = new Withdrawer(accoun
            withdrawer.withdraw(500);
      }
}
```

Processes Synchronization

- ❑ Critical Section(CS) is a part of the program that accesses a shared resource.
- ❑ Critical Sections have to be executed in a Mutually Exclusive manner

```
public class Depositor {
      private Account account;
      public Depositor(Account account) {
            this.account = account;
      }
      public void deposit(int amount) {
            int balance = account.getBalance();
            balance = balance + amount;
            account.setBalance(balance);
      }
      public static void main(String[] args) {
            Account account = getAccount();
            Depositor depositor =
                        new Depositor(account);
            depositor.deposit(500);
      }
}
```

```
public class Withdrawer {
      private Account account;
      public Withdrawer(Account account) {
            this.account = account;
      }

      public void withdraw(int amount) {
            int balance = account.getBalance();
            balance = balance - amount;
            account.setBalance(balance);
      }

      public static void main(String[] args) {
            Account account = getAccount();
            account.setBalance(1000);
            Withdrawer withdrawer =
                          new Withdrawer(account);
            withdrawer.withdraw(500);
      }
}
```

Processes Synchronization

- Critical Section(CS) is a part of the program that accesses a shared resource.
- Critical Sections have to be executed in a Mutually Exclusive manner
- Solutions to a CS problem must satisfy the following:
  - Mutual Exclusion
  - Progress
  - Bounded Wait

2-process solution to CS

Solution to a CS problem must satisfy: 1.Mutual Exclusion 2. Progress 3. Bounded Wait

Suppose, there are 2 process Pi and Pj and a global variable turn initialized to either i or j

Code for Pi

```
 <non CS Code>
 while (tuen==j) do nop;
       CS
 jurn=j
 <non CS Code>
```

Code for Pj

```
 <non CS Code>
 while (tuen==i) do nop;
       CS
 jurn=I
 <non CS Code>
```

```
<non CS Code>
flag[i] = true;
turn = j
while (flag[j] ==true  &&  turn==j) do nop;
      CS
flag[i] = false
<non CS Code>
```

```
<non CS Code>
flag[j] = true;
turn = i;
while (flag[i] ==true  &&  turn==i) do nop;
      CS
flag[j] = false
<non CS Code>
```

n-process solution to CS

Bakery Algorithm is used to solve n–process CS problem

Code for Pi

```
<non CS Code>
choosing[i] = true;
      number[i] = max(number[0], number[1],…, number[n-1]) + 1
choosing[i] = false;
for(j=0; j<n; j++){
    while(choosing[j]==true) do nop;
    while (number[j] != 0 && (number[j], j) < (number[i], i)) do nop;
      CS
number[i] = 0
<non CS Code>
```

Semapore-based solution to CS

- Semaphore is a process synchronization tool
- A semaphore S is a variable that apart from initialization (S=1 for binary semaphore) can only be accessed from the following 2 atomic operations

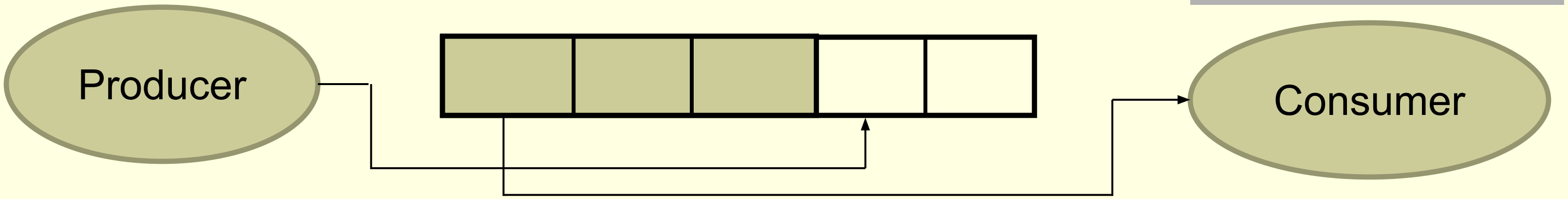
```
wait(S) {
    while(S<=0) do nop;
    S = S - 1;
}
```

```
signal(S) {
      S = S + 1;
}
```

Code for Pi

```
<non CS Code>
wait(S);
      CS
signal(S);
<non CS Code>
```

Producer – Consumer (PC) Problem

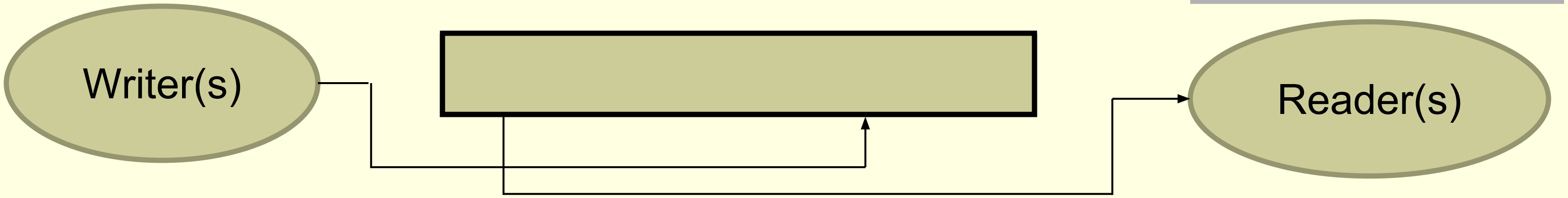


One binary semaphore : mutex = 1
Two count semaphore : empty = n and full = 0

```
<non CS Code>
wait(empty);
wait(mutex)
     <produce the next item>
signal(mutex);
signal(full)
<non CS Code>
```

```
<non CS Code>
wait(full);
wait(mutex)
     <consume the next item>
signal(mutex);
signal(empty)
<non CS Code>
```

Readers – Writers (RW) Problem



Two binary semaphore : mutex = 1   and write = 1

```
<non CS Code>
wait(write)
    <write to buffer>
signal(write);
<non CS Code>
```

```
<non CS Code>
wait(mutex)
    readCnt = readCnt + 1;
    if(readCnt == 1)
        wait(write)
signal(mutex);
    <read from buffer>
wait(mutex)
    readCnt = readCnt - 1;
    if(readCnt == 0)
        signal(write)
signal(mutex);
<non CS Code>
```

Dining Philosophers (DP) Problem

One binary semaphore for each Chopstick : chopstick[i] = 1 for i = 0 to n

```
while(true){
   wait(chopstick[i]);
   wait(chopstick[(i+1) % n);
         <Eat>
   signal(chopstick[i]);
   signal(chopstick[(i+1) % n);
   <Think>
}
```

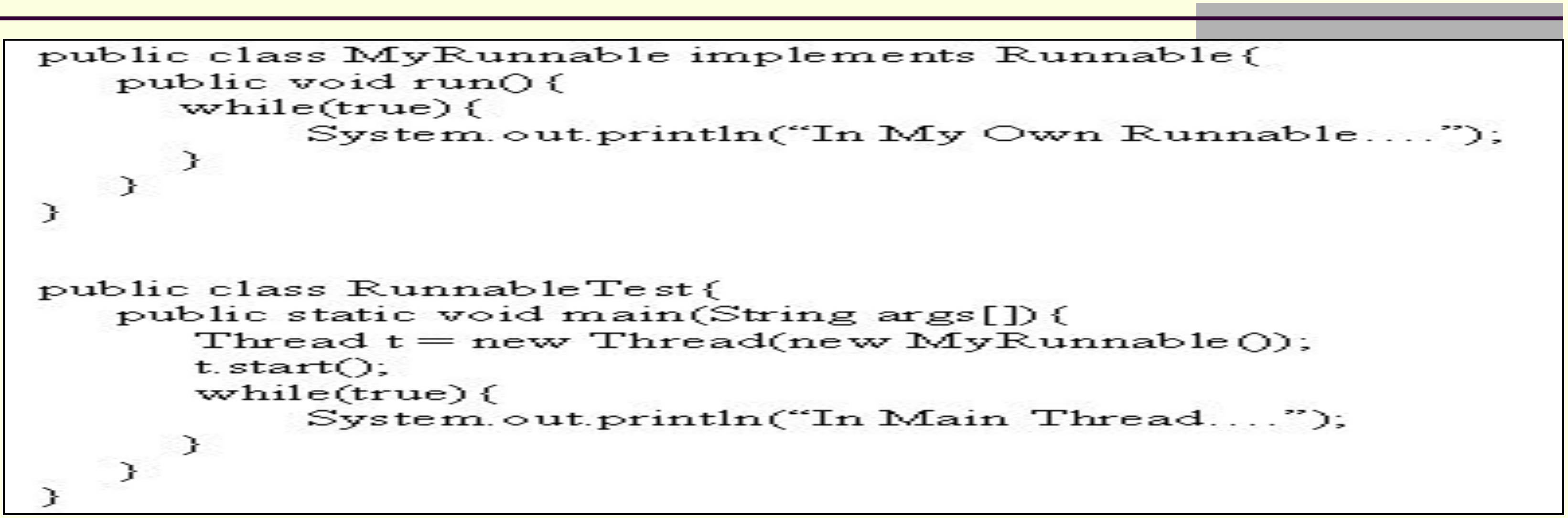
Solution to Deadlock in DP Problem:

1. Allow (n-1) philosophers with n chopsticks
2. Change the order to taking chopsticks for even and odd position philosophers : Even position take left chopstick first and odd position take right chopstick first.
3. Allow a philosopher to take a chopstick when both are available

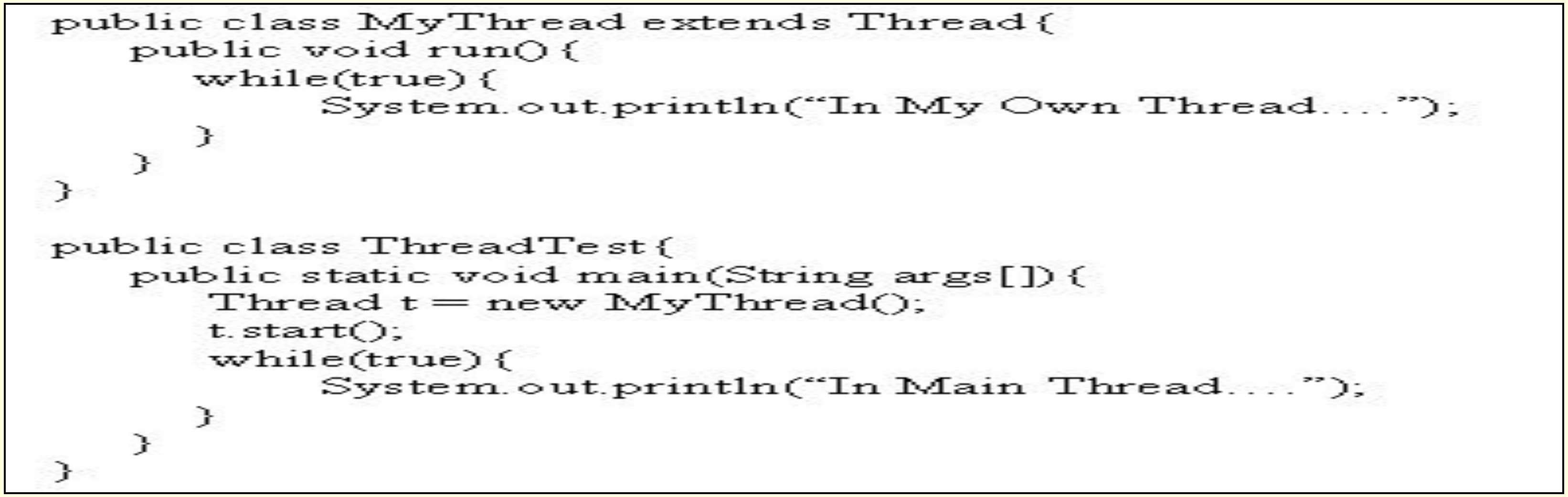
Creating Threads

- Java provides two ways of Creating Threads:
  - Implementing the *Runnable* Interface
  - Extending the *Thread* Class
- Implementing Runnable Interface
  - Create a Class that implements Runnable Interface
  - Override the run() method
  - Instantiate a Thread Object passing the Runnable Object to the constructor
  - Call the start() method on the Thread Object
- Extending Thread Class
  - Create a Class that extends Thread Class
  - Override the run() method
  - Instantiate the Thread Object Directly
  - Call the start() method on the Thread Object

Creating Threads (Cont.)

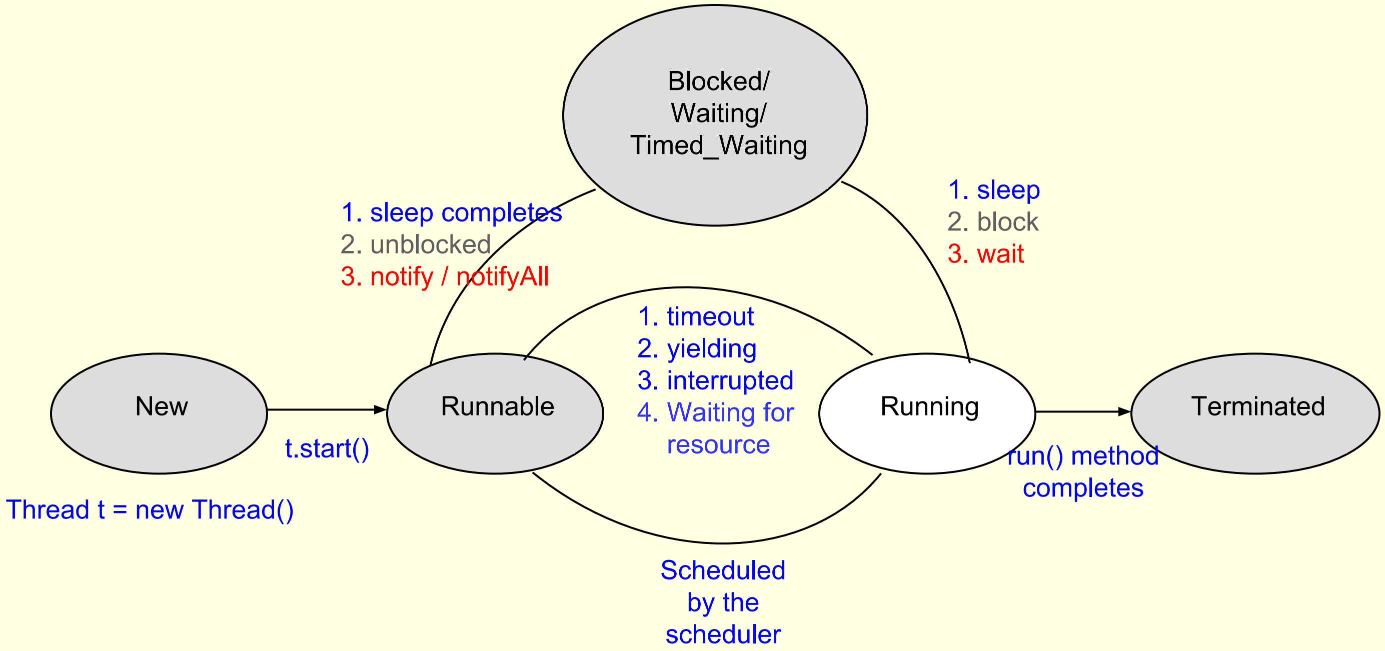


```
public class MyRunnable implements Runnable{
    public void run() {
       while(true) {
             System.out.println(“In My Own Runnable....”);
       }
    }
}

public class RunnableTest{
    public static void main(String args[]) {
        Thread t = new Thread(new MyRunnable());
        t.start();
        while(true) {
             System.out.println(“In Main Thread....”);
       }
    }
}
```



```
public class MyThread extends Thread{
    public void run() {
       while(true) {
             System.out.println(“In My Own Thread....”);
       }
    }
}

public class ThreadTest{
    public static void main(String args[]) {
        Thread t = new MyThread();
        t.start();
        while(true) {
             System.out.println(“In Main Thread....”);
       }
    }
}
```

Life Cycle of a Thread



Creating Multiple Threads

- Once a thread is started you can’t start the thread again

```
Thread t = new MyThread();
t.start();
t.start(); ✗
```

- You have to create a new Thread instance & start it again. You can assign a unique name to a particular instance.

```
Thread t = new MyThread();
t.setName(“Thread1”);
t.start();
t = new MyThread();
t.setName(“Thread2”);
t.start();
```

- You can get the corresponding thread instance from within the thread through *Thread.currentThread()*
- Once a thread ends you can’t start it again. Every time you have to create a new Instance & start it

Checking a Thread for Completion

- isAlive()
  - Method to check whether a thread reach dead state or not

- join()
  - Method to make the current thread wait for another to complete. If the other thread have already completed, the method has no effect.

  - t.join() is equivalent to

    ```
    while(t.isAlive());
    ```

Daemon Vs Non-Daemon Thread

- Daemon Thread:
  - Infrastructure Threads
  - Run in background
  - Generally controls non-daemon threads
  - Examples – Garbage Collector Thread
- Non-Daemon Thread:
  - Task specific Threads
  - Run in foregrounds
  - Example – Main Thread
- You can set the daemon-ness by calling setDaemon() method prior to start the thread
- The JVM terminates when all the non-daemon threads complete.

Concurrency Control with Thread

- What happens if two or more threads accesses a shared resource at the same time?
  - May Results in Inconsistent State
- Solution
  - Thread Synchronization
    - Method Synchronization
    - Block Synchronization

Inter-thread Communication

- Threads may be inter-dependent □ One thread depends on another thread to complete an operation
  - Consider a Producer Thread producing some value to a Queue which is consumed by a Consumer Thread. What happen if a consumer try to consume something that is still not produced?
- Java solves Inter-thread Communication through thee methods provided in Object Class
  - wait() □ the calling thread gives up the monitor(lock) & go to the WAITING state until some other thread call notify() method on the same object.
  - notify() □ Move one of the waiting thread on the same object to the RUNNABLE state
  - notifyAll() □Move all the waiting thread on the same object to the RUNNABLE state