Multithreaded Programming in Java

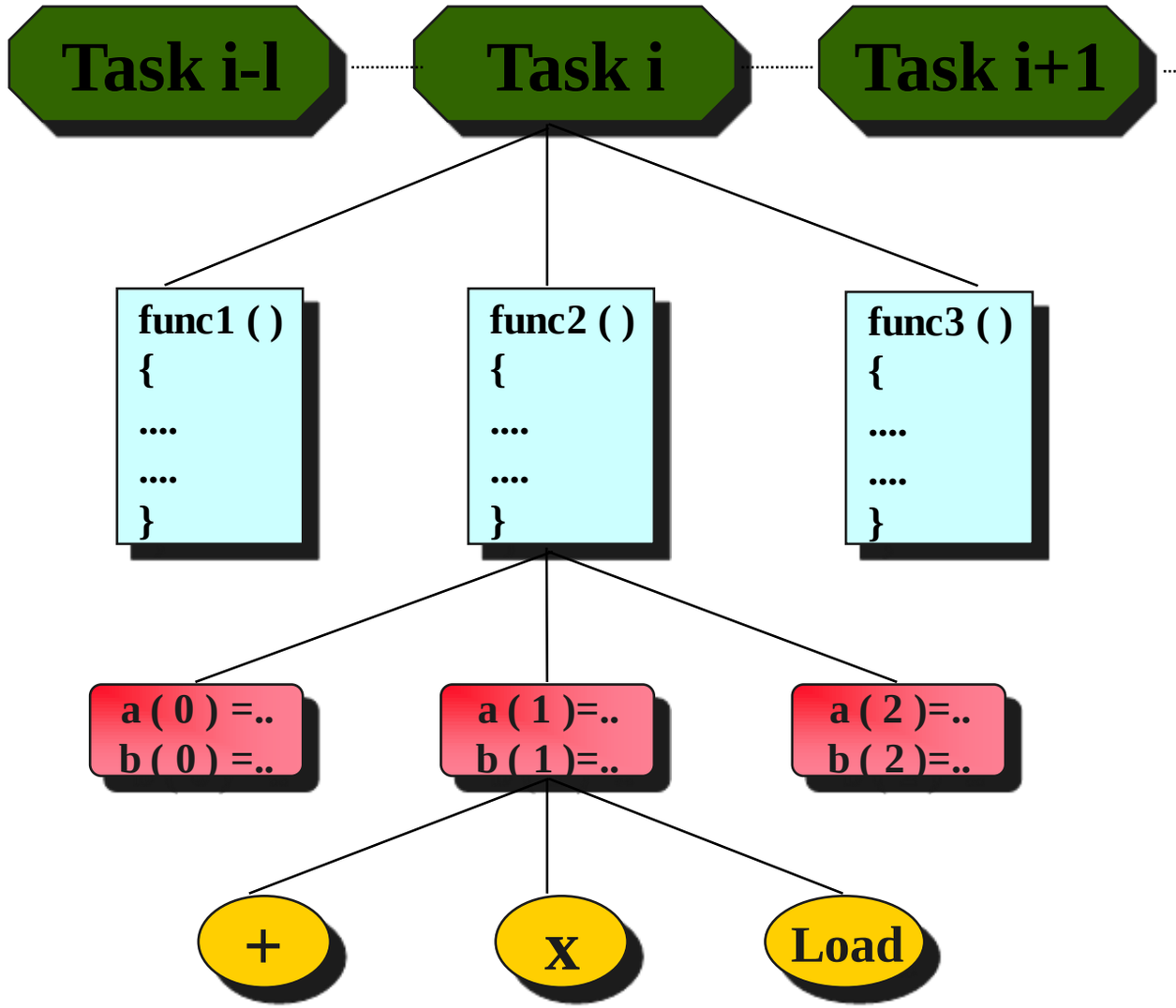
Levels of Parallelism

Sockets

Threads

Compilers

CPU



Code-Granularity

Code Item

Large grain (task level)

Program

Medium grain (control level)

Function (thread)

Fine grain (data level)

Loop (Compiler)

Very fine grain (multiple issue)

With hardware

Multitasking

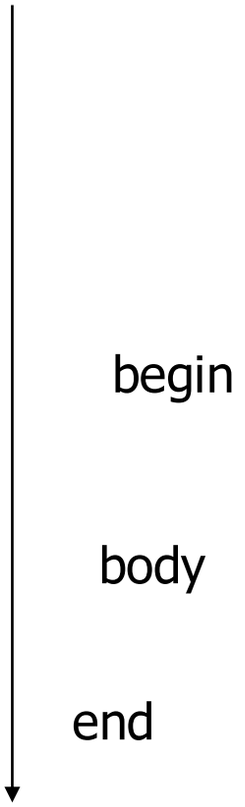
- Process based multitasking
  - Several independent process executing simultaneously
  - Os based concept-advatages for os and not programs
  - Download a file,run java program,open browser

Multithreading

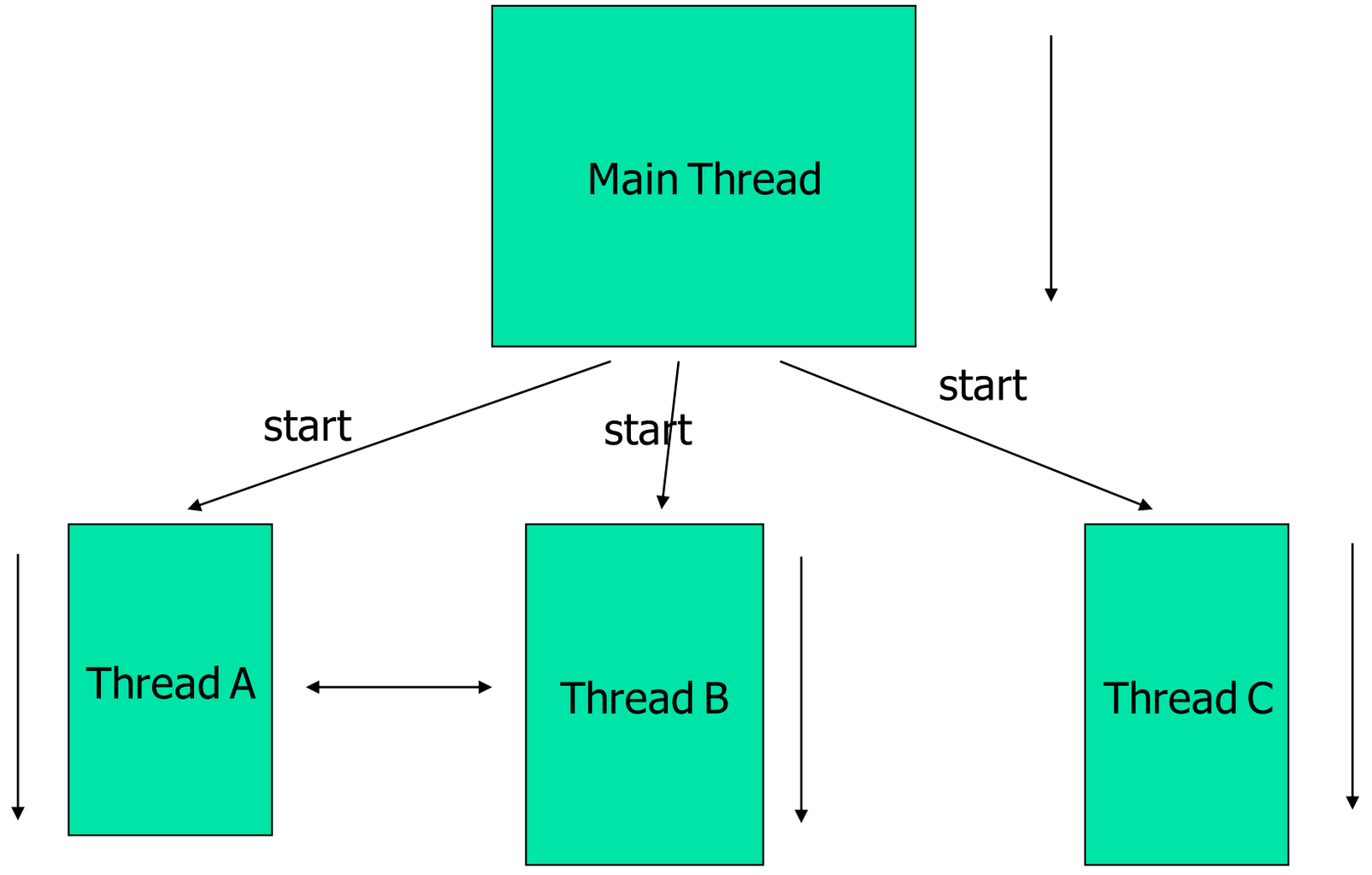
- Thread based multitasking
  - Same program-simultaneously executing multiple task where each task is a piece of independent code
  - Programmatic level
  - Word-edit,print,checking
  - Games-different objects on a platform
    - Road rash-different bikes/objects
    - Youtube –download 4 files

A single threaded program

```
class ABC
{
....
    public void main(..)
    {
    ...
    ..
    }
}
```



A Multithreaded Program



Threads may switch or exchange data/results

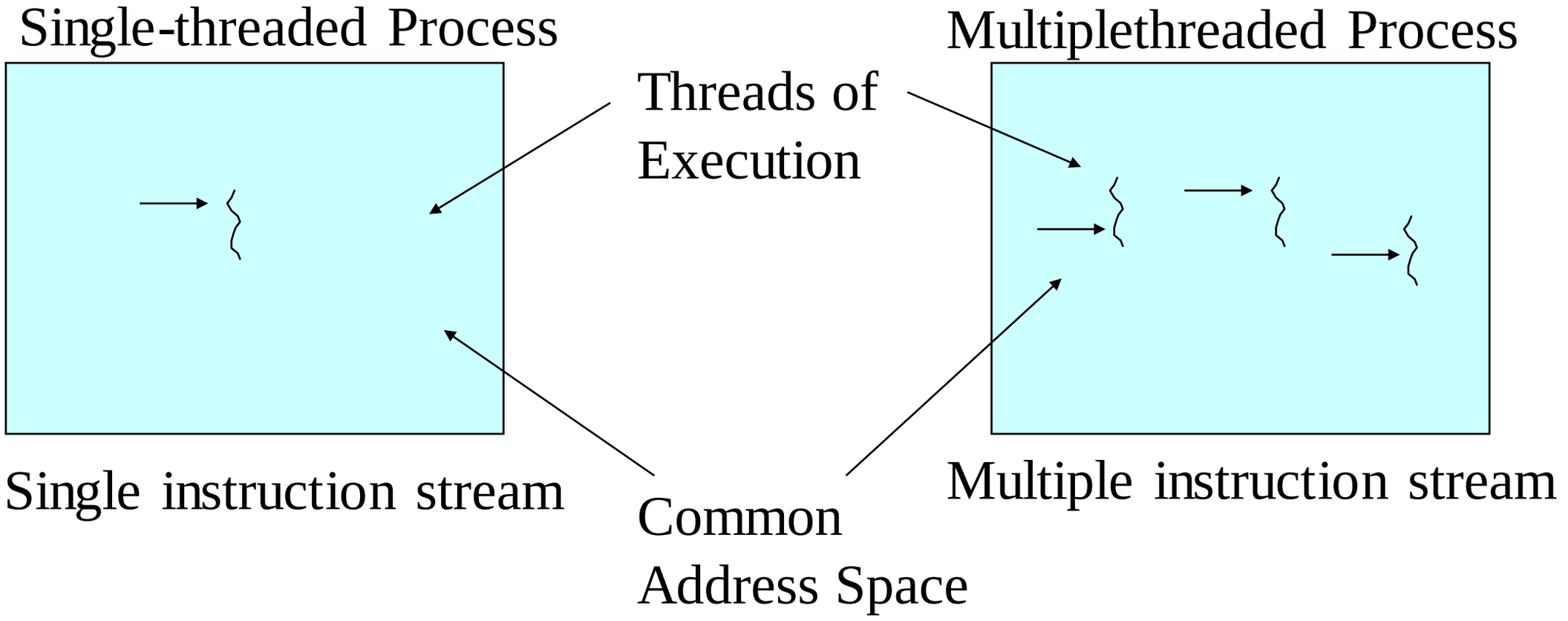
A thread provides the mechanism for running a task.

Each task is **an instance of the Runnable interface**, also called a runnable object.

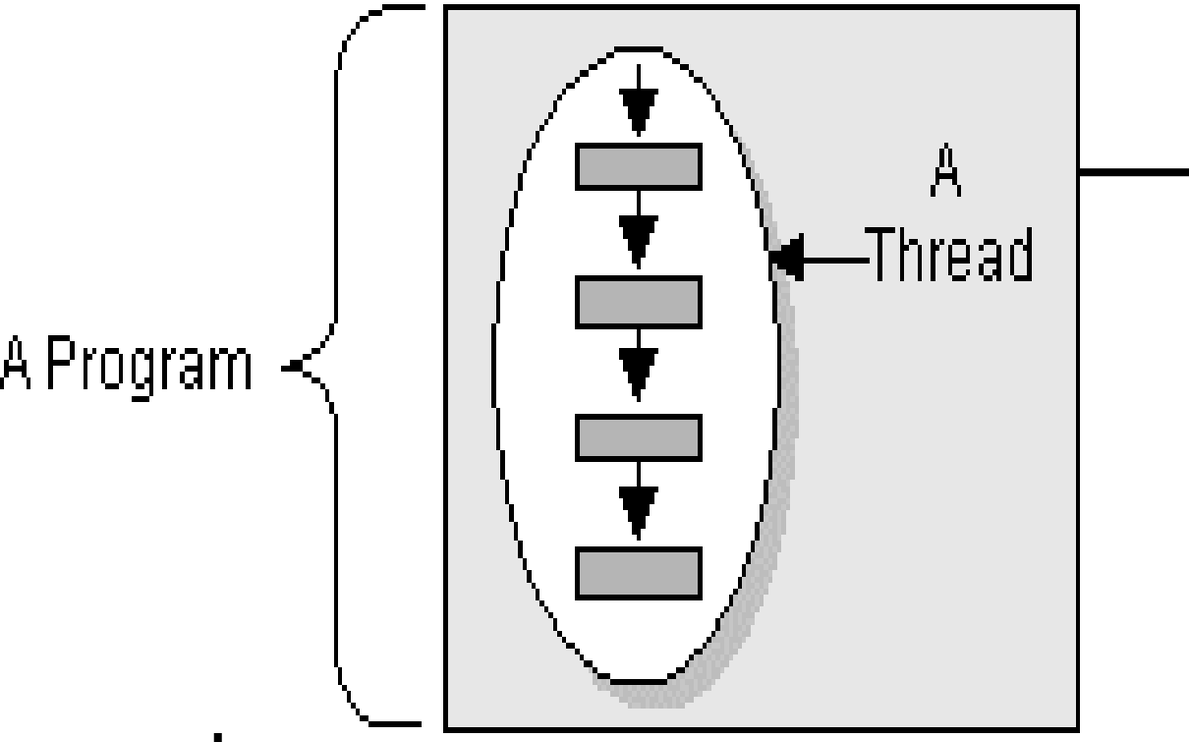
A **thread is essentially an object that facilitates the execution of a task**

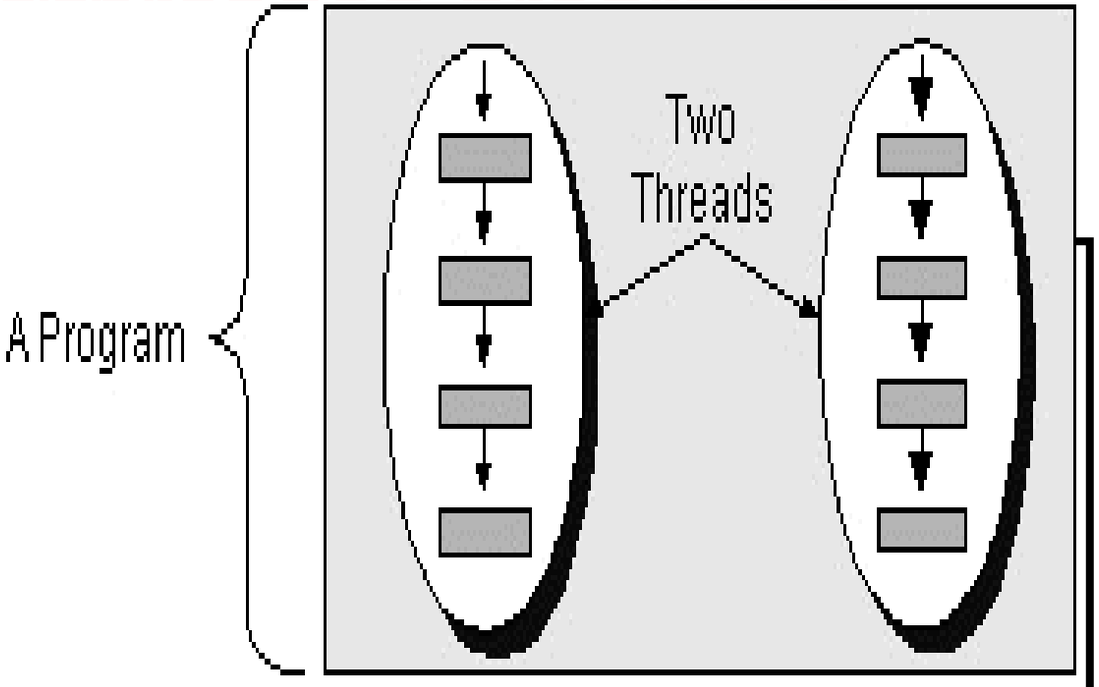
Single and Multithreaded Processes

threads are light-weight processes within a process



single-threaded vs multithreaded programs

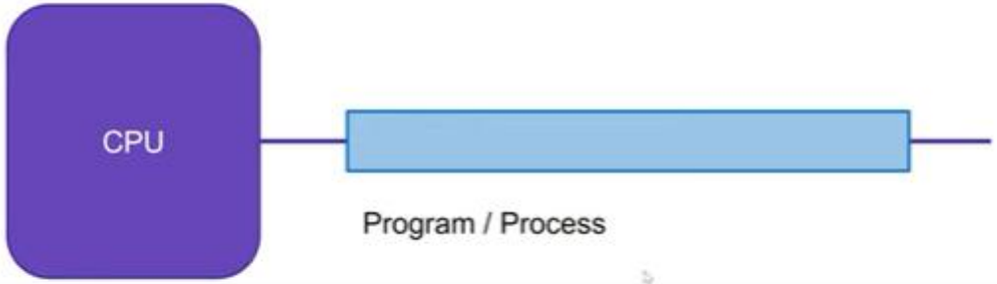





```
{ A(); A1();  A2();   A3();
        B1();  B2();  }
```

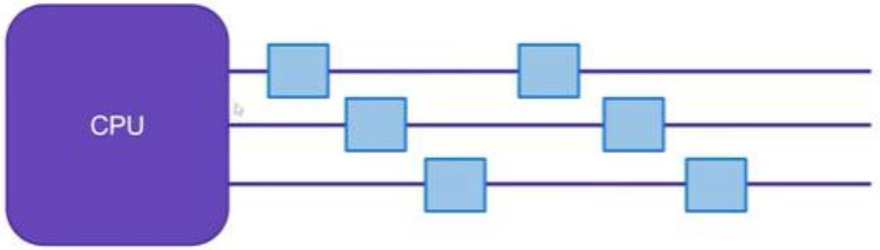
```
{  A();
newThreads {
{ A1(); A2(); A3() };
  {B1(); B2() }
  }
}
```

Singletasking in Early Computing



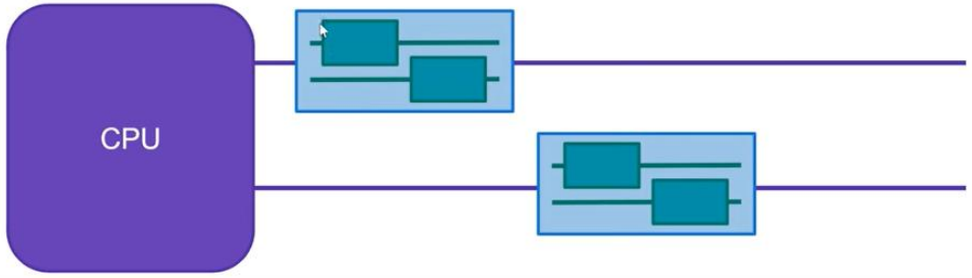
One CPU / Computer can run
one program (process)
at a time.

Multitasking in Early Computing



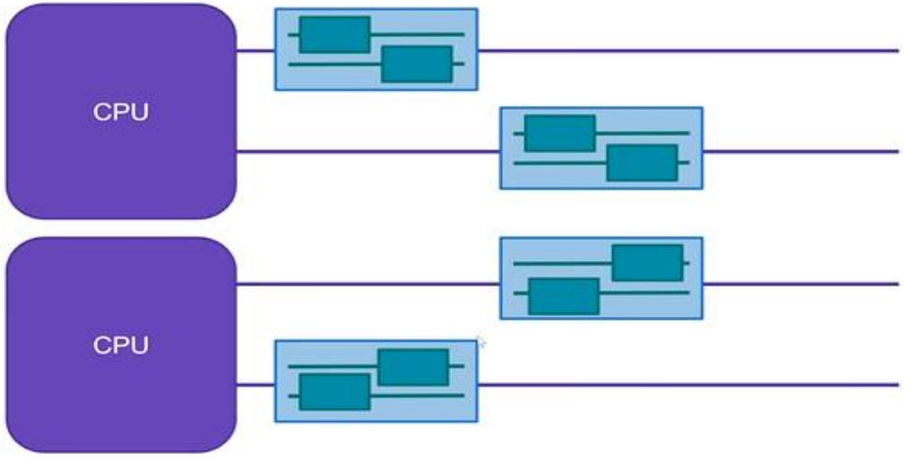
One CPU / Computer can run
Multiple programs (process)
at a time - by switching between
executing one program at a time for a
little time, and then switch to the next.

Multithreading

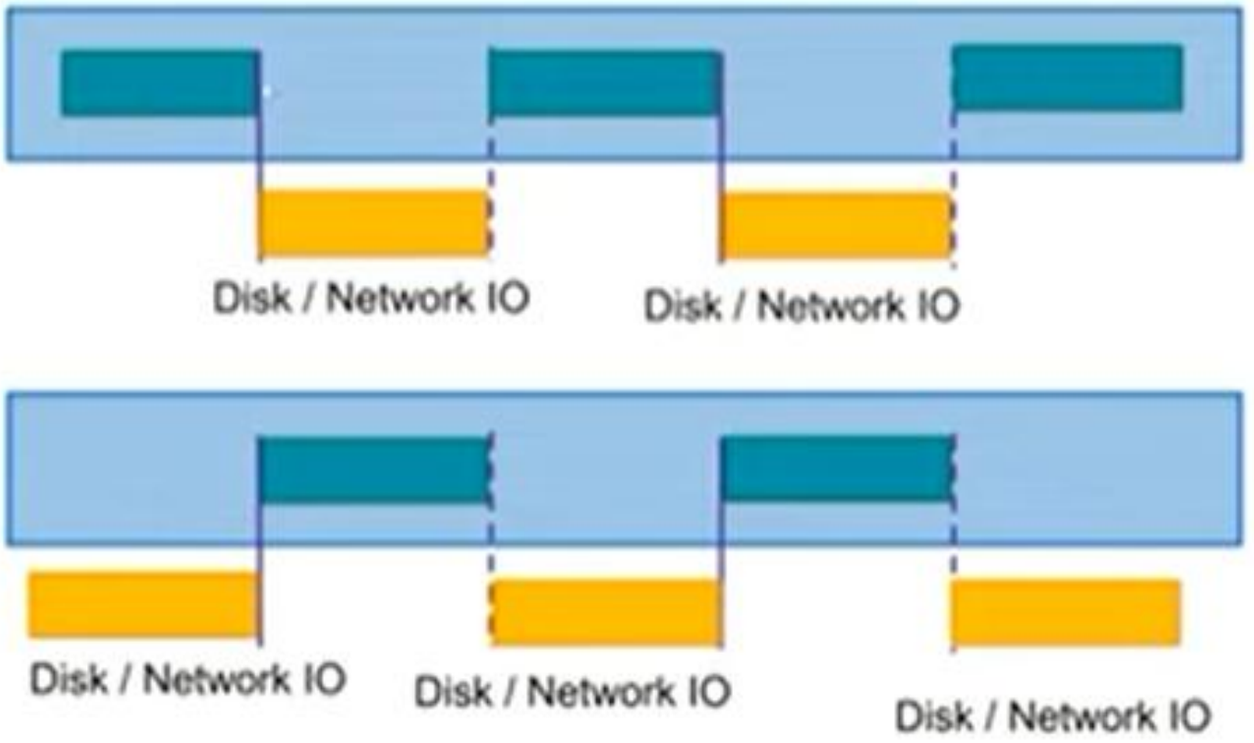


One CPU / Computer can run
Multiple programs (process)
at a time, with multiple threads of
execution inside.

Multithreading With Multiple CPUs



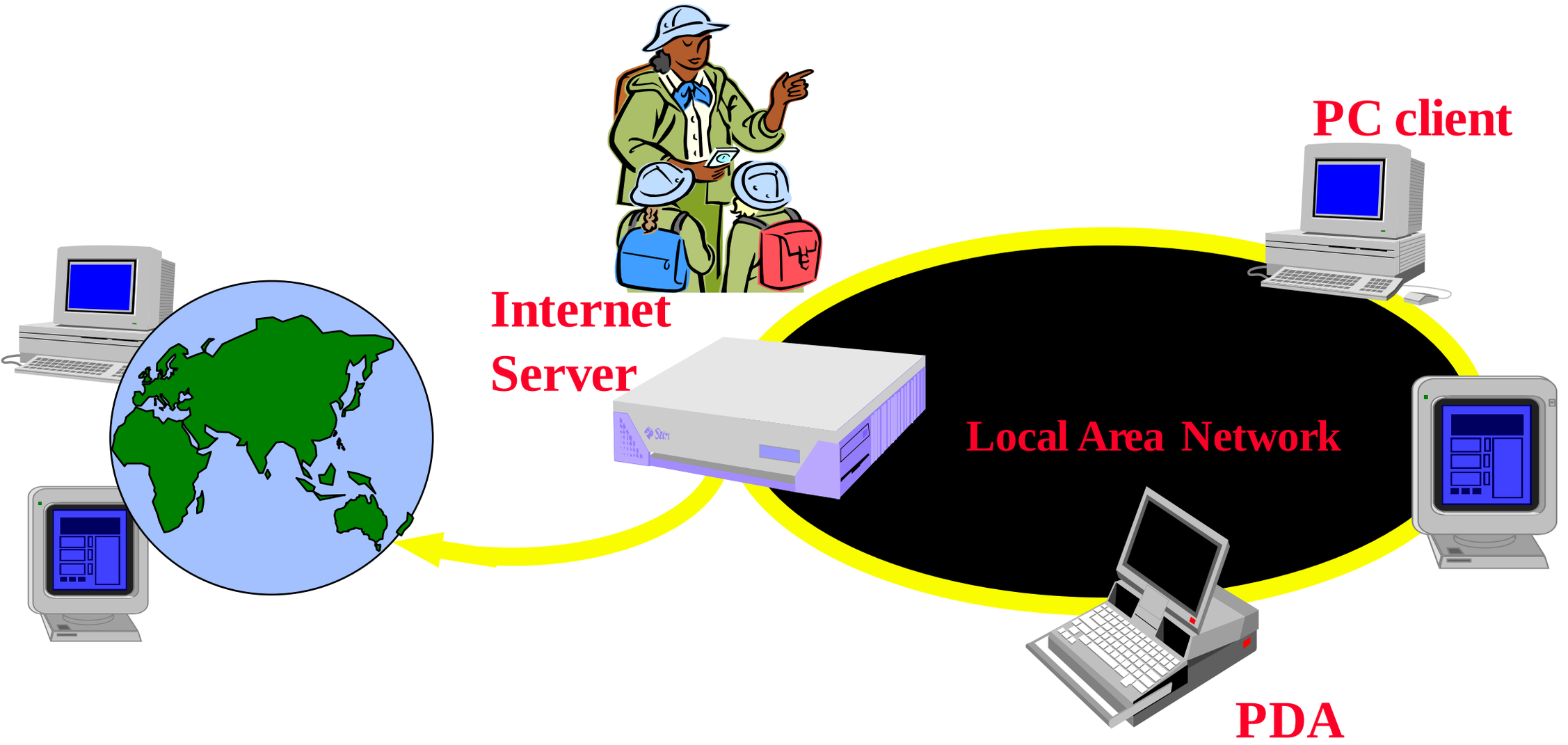
Why Multithreading?



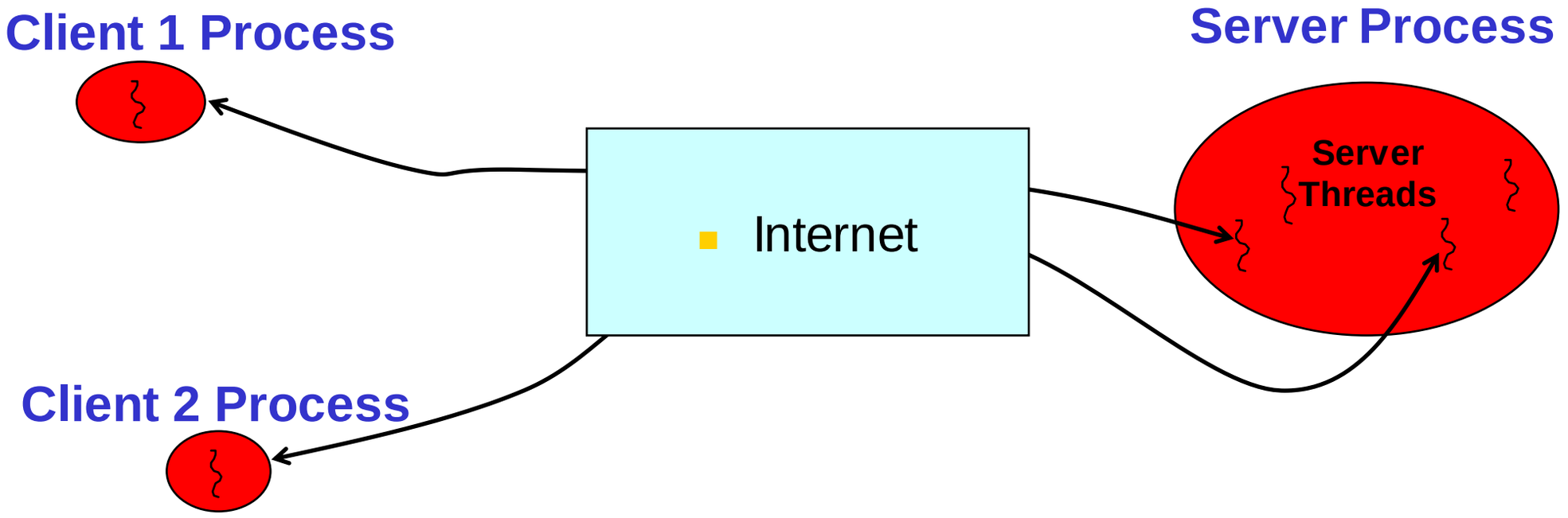
- Better IO Utilization

MULTI-THREADED APPLICATIONS

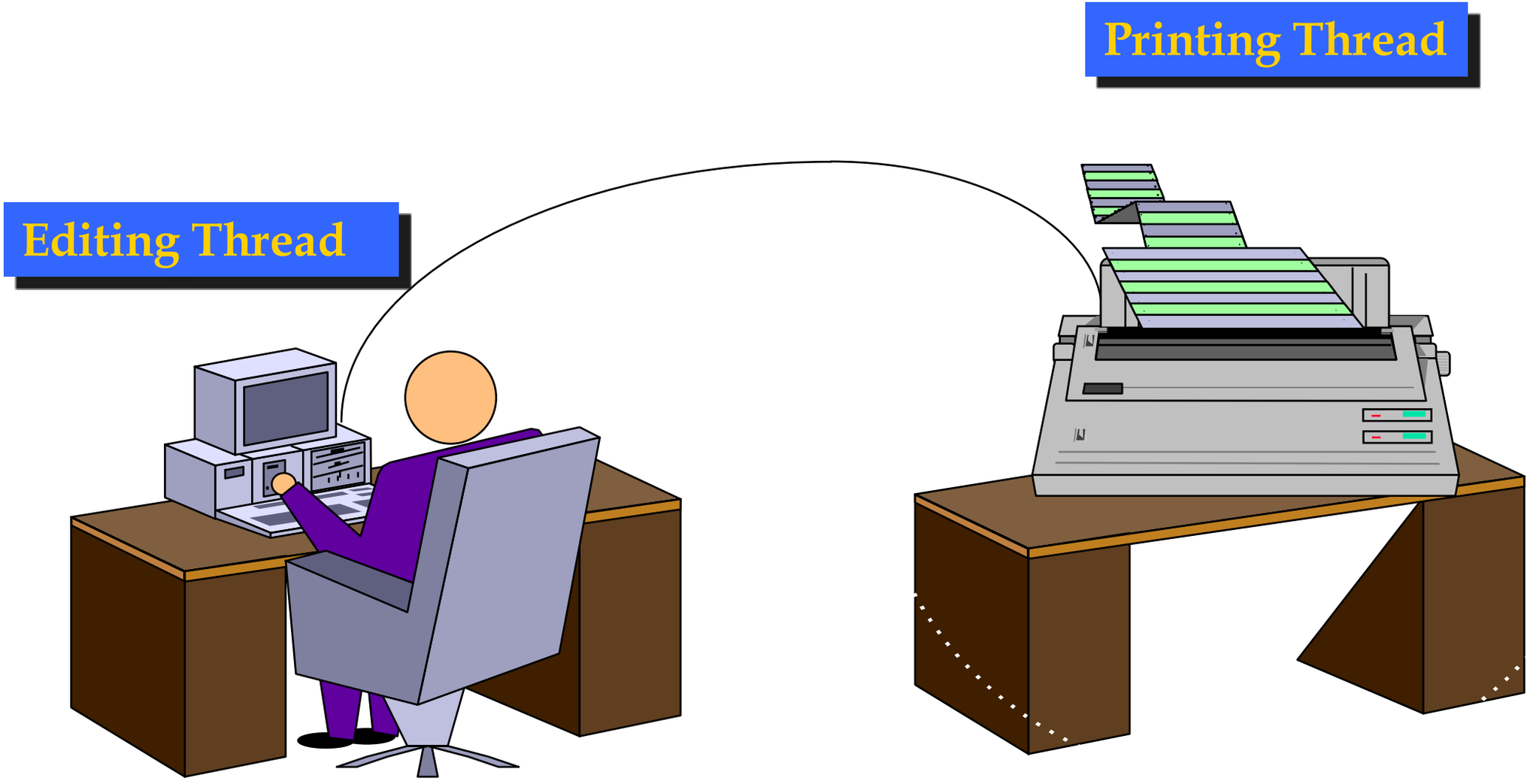
Web/Internet Applications: Serving Many Users Simultaneously



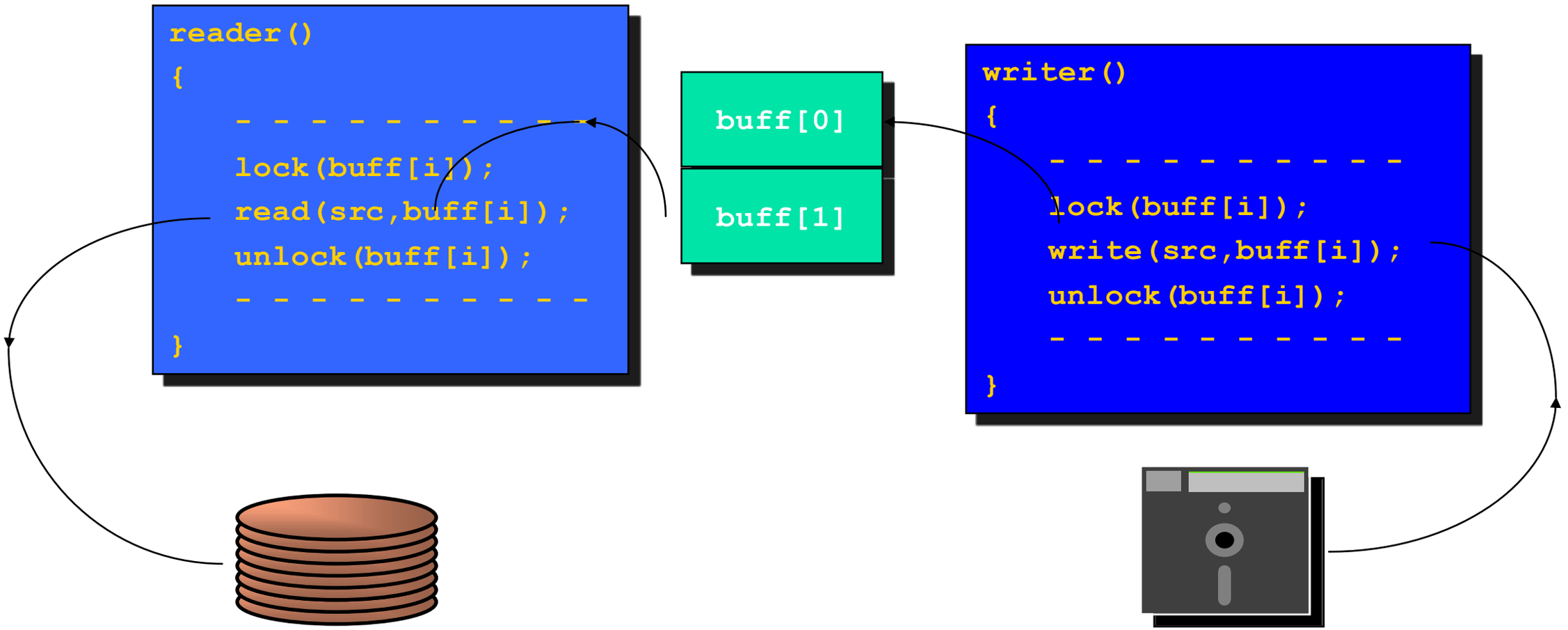
Multithreaded Server: For Serving Multiple Clients Concurrently



Modern Applications need Threads (ex1): Editing and Printing documents in background.



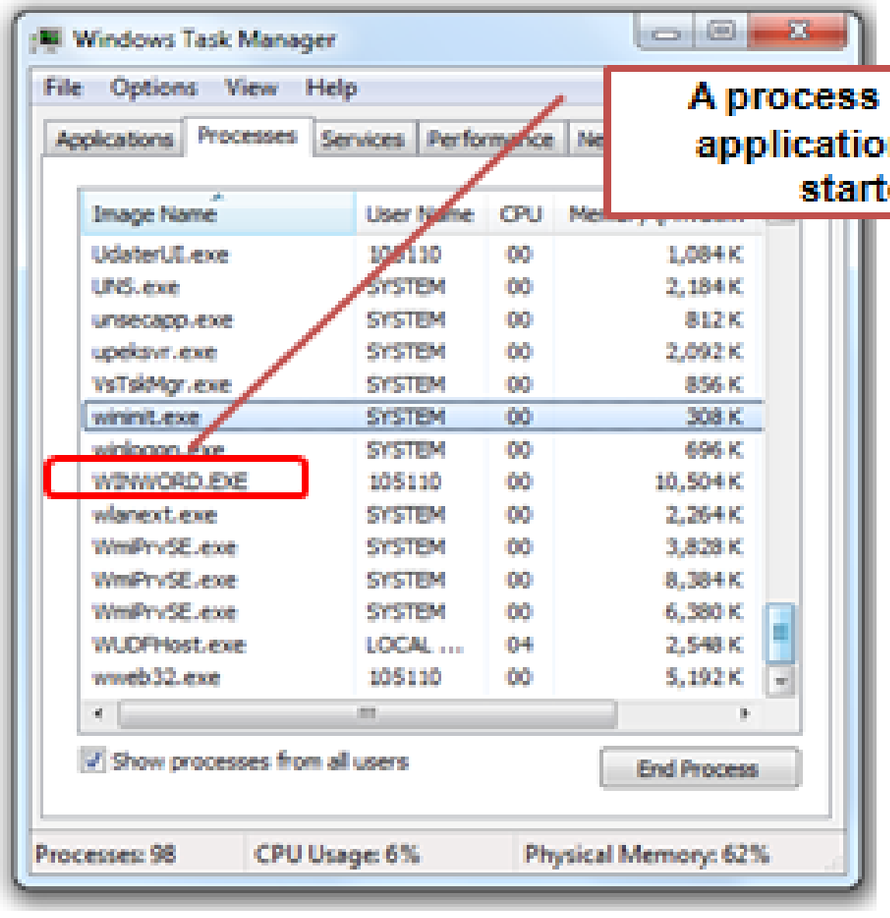
Multithreaded/Parallel File Copy



Cooperative Parallel Synchronized Threads

Example application-Process

Lets consider Microsoft word application to understand it better, what happens when you start an word application.



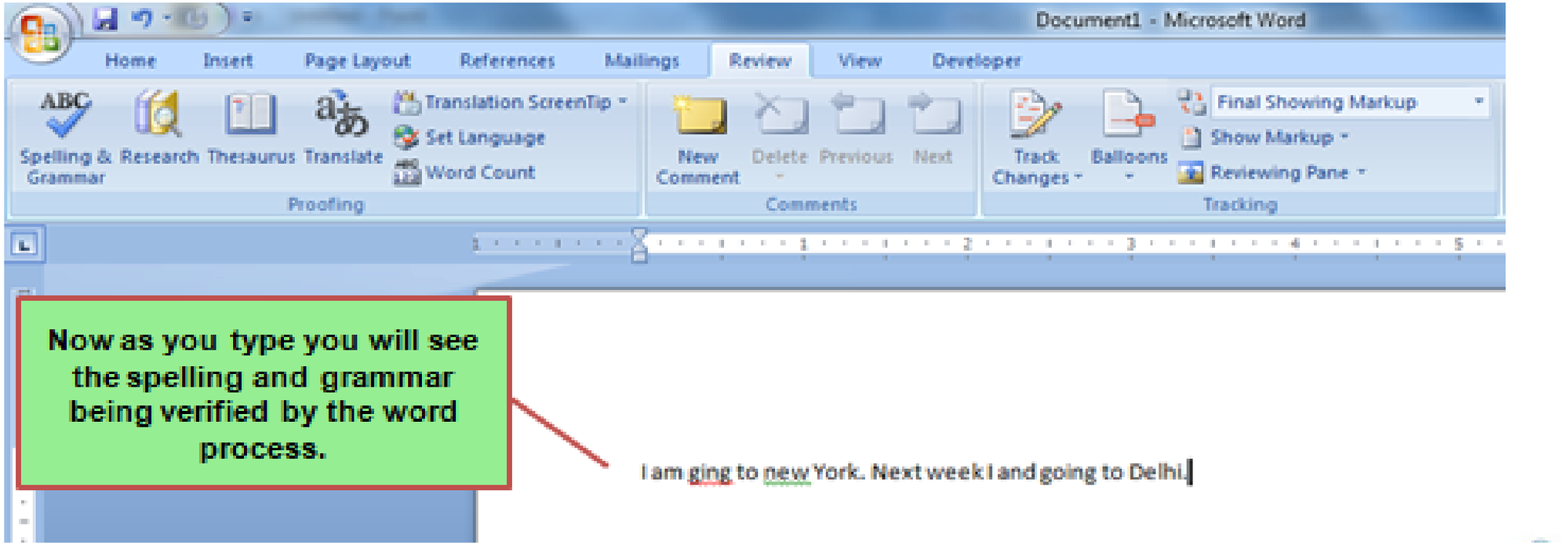
A process for word application will be started.

Now lets see what happens when the user starts using word application.

Threads in word application

The spell check has been implemented as a ***thread*** within the worde.exe process which runs continuously and verifies what you type.

Word.exe is the **process** and **spell check** is a **thread** running inside the process.



Virtual Cores...

Threads are a series of programed instructions that allow a CPU core to appear to be split into two cores. So for each core you have two threads:

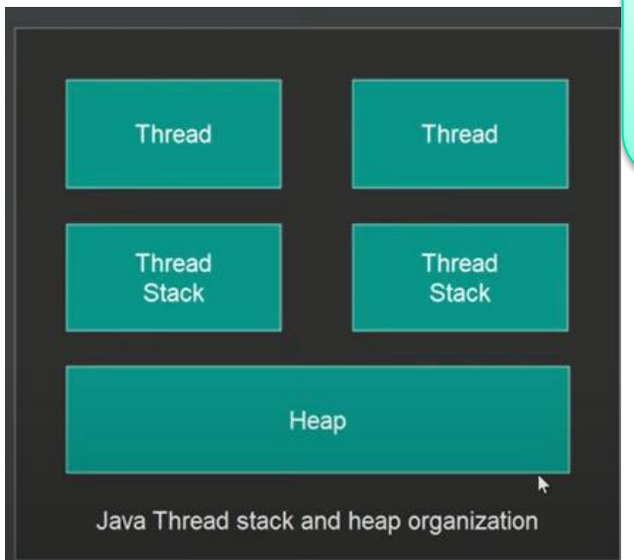
- 2 Cores / 4 Threads
- 4 Cores / 8 Threads
- 6 Cores / 12 Threads
- 32 Cores / 64 Threads

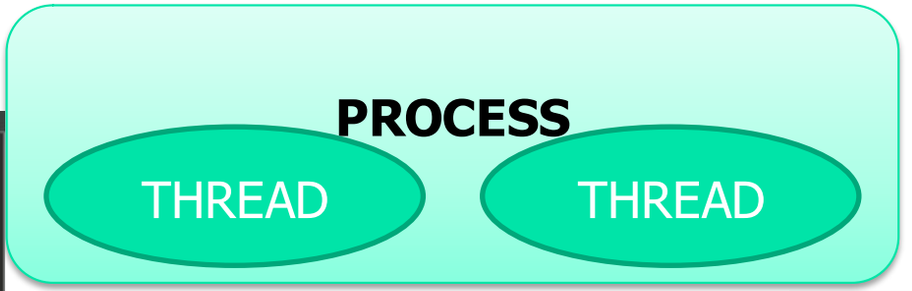
So how does it work?

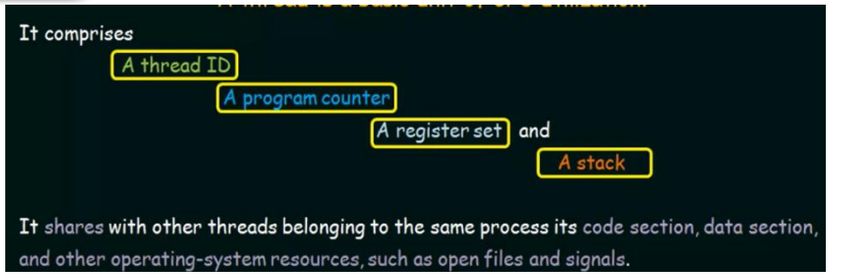
- When you take an action on your computer (such as open Photoshop), an process is started. That process creates a thread.
- For the remainder of your actions in Photoshop you will be calling on a variety of Threads to conduct your work in Photoshop.
- Process can use multiple threads depending on the program use are using and how it is written.
- For Specialized tasks the more threads you have the better. When you have multiple threads a single “process” can handle a variety of different tasks.

Process Vs Threads

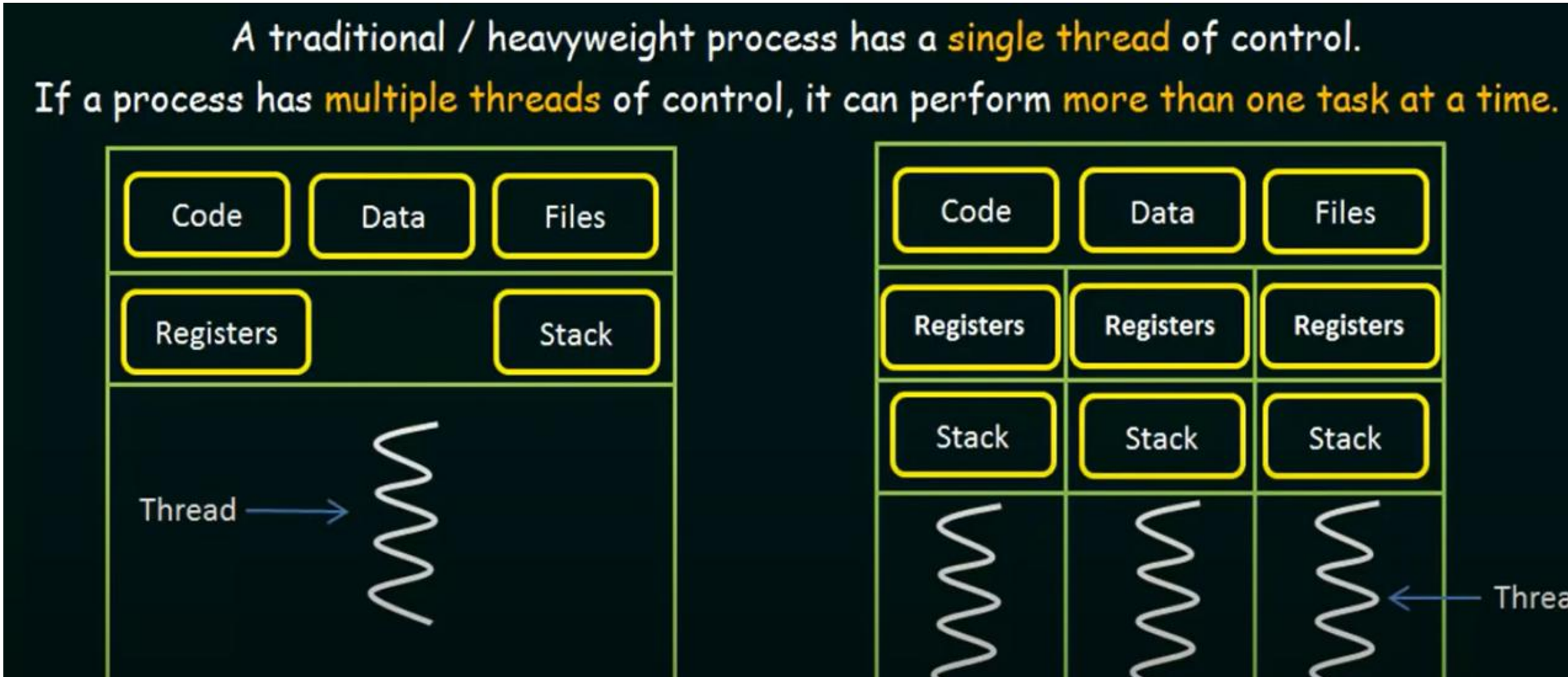
- Process→ executables which runs in separate memory space
- Threads → small process which shared memory space within a process



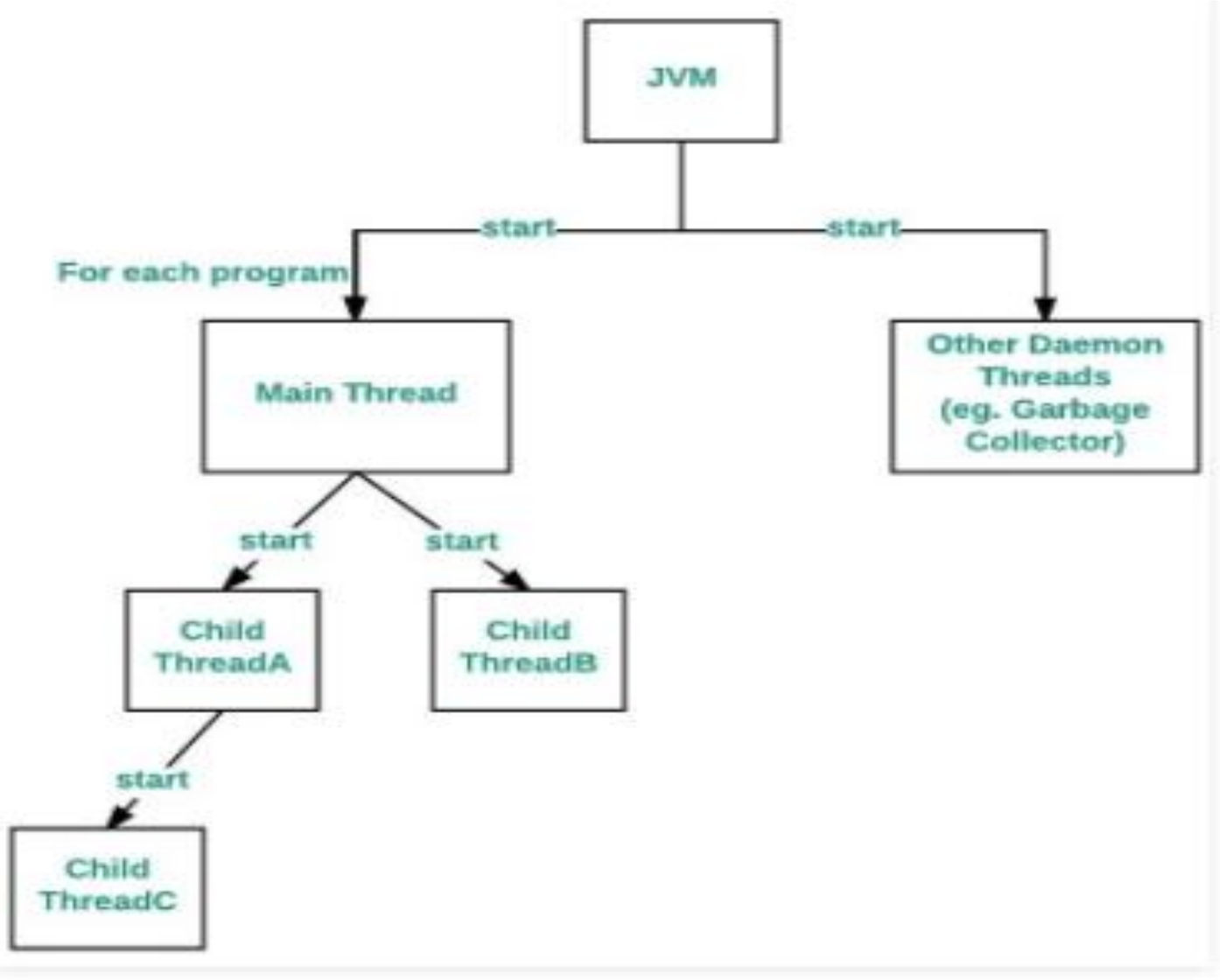




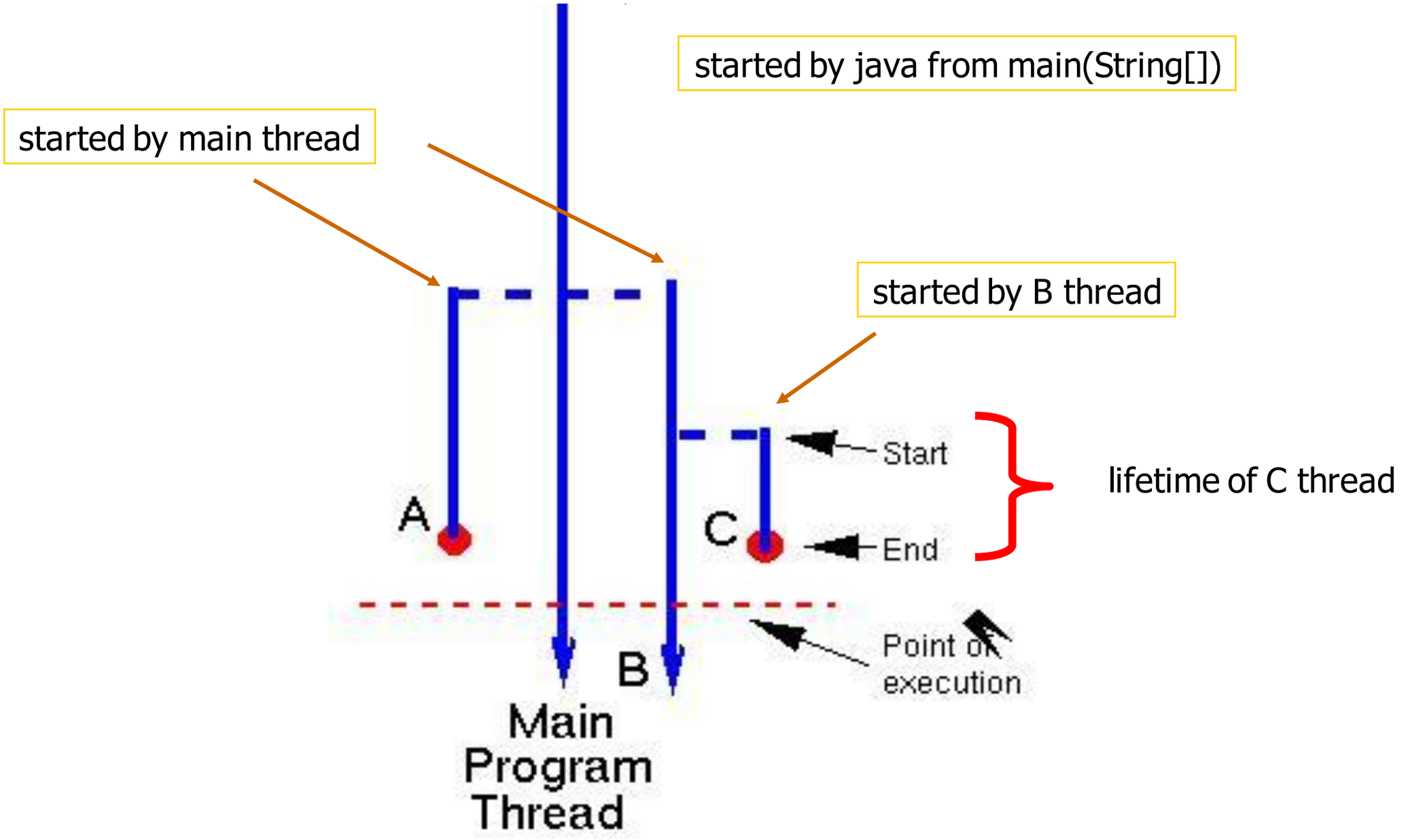
Threads


A traditional / heavyweight process has a single thread of control.
If a process has multiple threads of control, it can perform more than one task at a time.
Code
Data
Files
Registers
Stack
Thread
Code
Data
Files
Registers
Registers
Registers
Stack
Stack
Stack

Threads


JVM
start
start
For each program
Main Thread
Other Daemon Threads (eg. Garbage Collector)
start
start
Child ThreadA
Child ThreadB
start
Child ThreadC

Thread ecology in a java program



What are Threads?

- A piece of code that run in concurrent with other threads.
- Each thread is a statically ordered sequence of instructions.
- Threads are being extensively used express concurrency on both single and multiprocessors machines.
- Programming a task having multiple threads of control – Multithreading or Multithreaded Programming.

Java Threads

- Java has built in thread support for Multithreading
- Synchronization
- Thread Scheduling
- Inter-Thread Communication:
  - currentThread start setPriority
  - yield run getPriority
  - sleep stop suspend
  - resume
- Java Garbage Collector is a low-priority thread

Define and launch a java thread

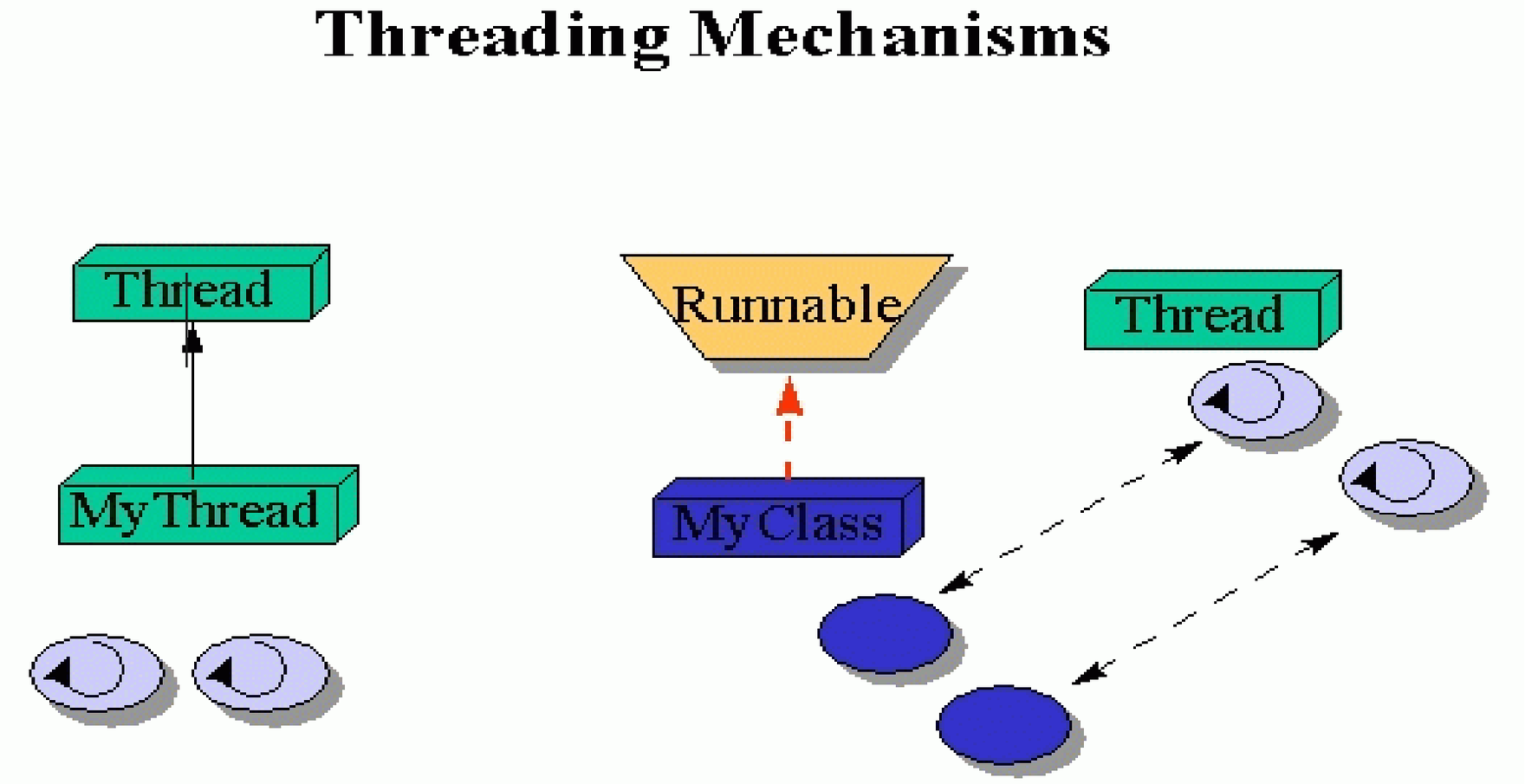
- Each Java Run time thread is encapsulated in a java.lang.Thread instance.
- Two ways to define a thread:
  1. Extend the Thread class
  2. Implement the Runnable interface :

```
package java.lang;
public interface Runnable {  public void run() ; }
```

- Steps for extending the Thread class:
  1. Subclass the Thread class;
  2. Override the default Thread method run(), which is the entry point of the thread, like the main(String[]) method in a java program.

Threading Mechanisms...

- Create a class that extends the Thread class
- Create a class that implements the Runnable interface



The Thread Class

```
public class Thread extends Object implements Runnable {
  public Thread();
  public Thread(String name);
  public Thread(Runnable target);
  public Thread(Runnable target,
                String name);
  public Thread(Runnable target,
         String name, long stackSize);

  public void run();
  public void start();
  ...
}
```

An Overview of the `Thread` Methods

- Thread-related methods
  - Constructors
    - **`Thread()`** – Creates a thread with an auto-numbered name of format **`Thread-1`**, **`Thread-2`**...
    - **`Thread( threadName )`** – Creates a thread with name
  - **`run`**
    - Does "work" of a thread – What does this mean?
    - Can be overridden in subclass of **`Thread`** or in **`Runnable`** object (more on interface **`Runnable`** elsewhere)
  - **`start`**
    - Launches thread, then returns to caller
    - Calls **`run`**
    - Error to call **`start`** twice for same thread

1st method: Extending Thread class

- Threads are implemented as objects that contains a method called run()

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

- Create a thread:

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

- Start Execution of threads:

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

An example

```
class MyThread extends Thread {

        public void run() {
                System.out.println(" this thread is running ... ");
        }
}

class ThreadEx1 {
        public static void main(String [] args  ) {
           MyThread t = new MyThread();
           t.start();
        }
}
```

2nd method: Threads by implementing Runnable interface

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

- Creating Object:

```
MyThread myObject = new MyThread();
```

- Creating Thread Object:
- A Runnable object can be wrapped up into a Thread object

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

- Start Execution:

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

An example

```
class MyThread implements Runnable  {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
}

class ThreadEx2 {
        public static void main(String [] args  ) {
                Thread t = new Thread(new MyThread());
                   t.start();
        }
}
```

Need for Runnable

```
// Example:
public class Print2Console extends Thread {
    public void run() {  // run() is to a thread what main() is to a java program
        for (int b = -128; b < 128; b++)  out.println(b);  }
    … // additional methods, fields …
     }
```

- Impement the Runnable interface if you need a parent class:

```
// by extending JTextArea we can reuse all existing code of JTextArea
public class Print2GUI extend JTextArea implement Runnable {
    public void run() {
        for (int b = -128; b < 128; b++)  append( Integer.toString(b) + "\n" );  }
     }
```

Thread Scheduling

Usually, in Java technology threads are *pre-emptive,* but not necessarily time-sliced (the process of giving each thread an equal amount of CPU time). It is a common mistake to believe that *pre-emptive* is another word for *does time-slicing*.

The model of a pre-emptive scheduler is that many threads might be runnable, but only one thread is running. This thread continues to run until it ceases to be runnable or until another thread of higher priority becomes runnable. In the latter case, the lower priority thread is *pre-empted* by the thread of higher priority, which gets a chance to run instead.

A thread might cease to be runnable (that is, become *blocked*) for a variety of reasons. The thread's code can execute a `Thread.sleep()` call, asking the thread to pause deliberately for a fixed period of time. The thread might have to wait to access a resource and cannot continue until that resource becomes available.

All threads that are runnable are kept in pools according to priority. When a blocked thread becomes runnable, it is placed back into the appropriate runnable pool. Threads from the highest priority non-empty pool are given CPU time.

A Program with Three Java Threads

- Write a program that creates 3 threads

Three threads example

```
class A extends Thread
{
    public void run()
     {
        for(int i=1;i<=5;i++)
          {
              System.out.println("\t From ThreadA: i= "+i);
          }

         System.out.println("Exit from A");
     }

}
```

```
class B extends Thread
{
    public void run()
     {

        for(int j=1;j<=5;j++)
          {
              System.out.println("\t From ThreadB:
  j= "+j);
          }

           System.out.println("Exit from B");
    }
```

```
class C extends Thread
{
    public void run()
     {

        for(int k=1;k<=5;k++)
          {
              System.out.println("\t From ThreadC:
  k= "+k);
          }

           System.out.println("Exit from C");
     }
```

```
class ThreadTest
{
   public static void main(String args[])

         {
               new A().start();
               new B().start();
               new C().start();

         }



}
```

Run 1

- From ThreadA: i= 1
  From ThreadA: i= 2
  From ThreadA: i= 3
  From ThreadA: i= 4
  From ThreadA: i= 5

Exit from A

  From ThreadC: k= 1
  From ThreadC: k= 2
  From ThreadC: k= 3
  From ThreadC: k= 4
  From ThreadC: k= 5

Exit from C

  From ThreadB: j= 1
  From ThreadB: j= 2
  From ThreadB: j= 3
  From ThreadB: j= 4
  From ThreadB: j= 5

Exit from B

Run2

- From ThreadA: i= 1
  From ThreadA: i= 2
  From ThreadA: i= 3
  From ThreadA: i= 4
  From ThreadA: i= 5
  From ThreadC: k= 1
  From ThreadC: k= 2
  From ThreadC: k= 3
  From ThreadC: k= 4
  From ThreadC: k= 5

Exit from C

  From ThreadB: j= 1
  From ThreadB: j= 2
  From ThreadB: j= 3
  From ThreadB: j= 4
  From ThreadB: j= 5

Exit from B

Exit from A

You decided to modify the application by using multiple threads to reduce the computation time. For this, accept the number of counters or threads at the beginning of the problem and get the string for each counter or thread. Create a thread by extending the Thread class and take the user entered string as input. Each thread calculates the character frequency for the word assigned to that thread. All the counts are stored locally in the thread and once all the threads are completed print the character frequency for each of the threads.

Create a class Main.

Input and Output format:
Refer to sample Input and Output for formatting specifications.

Sample input and output:
[All Texts in bold corresponds to the input and rest are output]

Enter Number of Counters :
2
Enter text for counter 1 :
FrequencyCounter
Enter text for counter 2 :
JavaTheCompleteReference
Counter 1 Result :
C:1 F:1 c:1 e:3 n:2 o:1 q:1 r:2 t:1 u:2 y:1
Counter 2 Result :
C:1 J:1 R:1 T:1 a:2 c:1 e:7 f:1 h:1 l:1 m:1 n:1 o:1 p:1 r:1 t:1 v:1

Life cycle of A Thread

States:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

Thread States: Life Cycle of a Thread

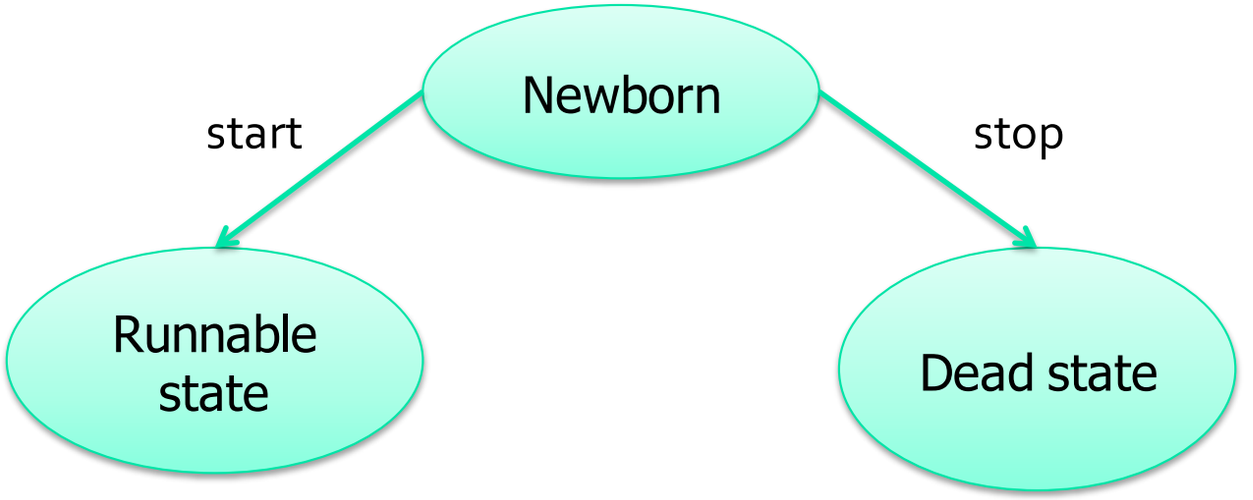
- Born state
  - Thread just created
  - When **`start`** called, enters ready state
- Ready state (runnable state)
  - when **start()** method is called on the thread object. A thread in runnable state is scheduled to run by JVM but it may not start running until it gets CPU cycle.
- Running state
  - System assigns processor to thread (thread begins executing)
  - When **`run`** completes or terminates, enters dead state
- Dead state
  - Thread marked to be removed by system
  - Entered when **`run`** terminates or throws uncaught exception

Newborn state:

- when we create a thread object , the thread is born
- not yet scheduled for running

Only following methods can be used:

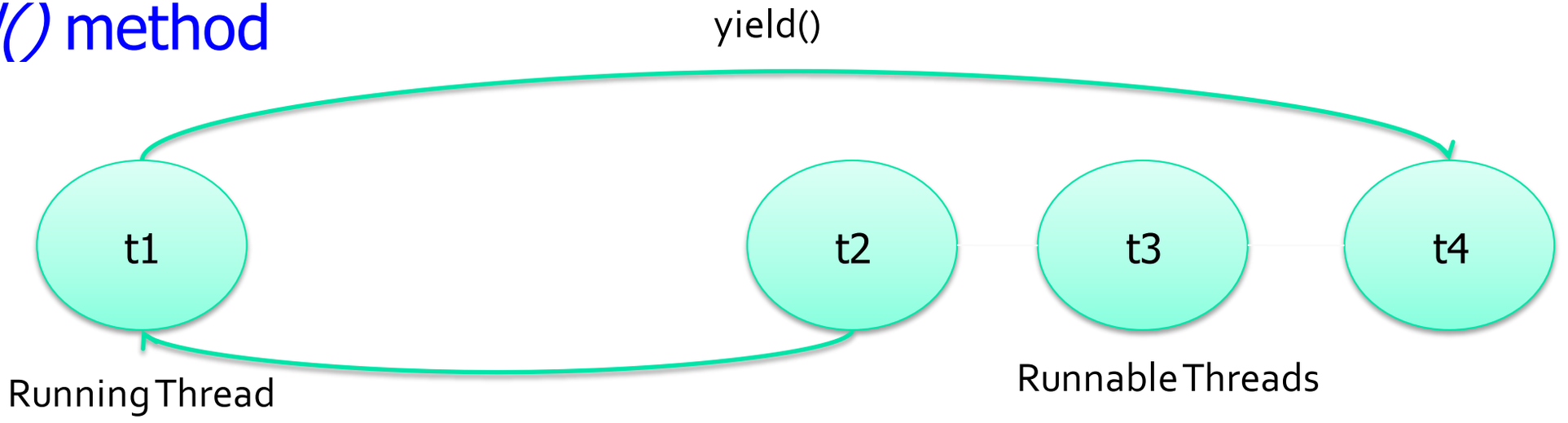
1. start()
2. stop()



if any other method is invoked at this stage, an exception will be thrown

Runnable state:

- Means thread is ready for execution and waiting for availability of the processor
- Threads will be in queue, processed based on priority
- Equal priority threads have been assigned time slots for execution → **time-slicing**
- Relinquish control from one thread to another of same priority can be given before its turn comes , by using *yield()* method

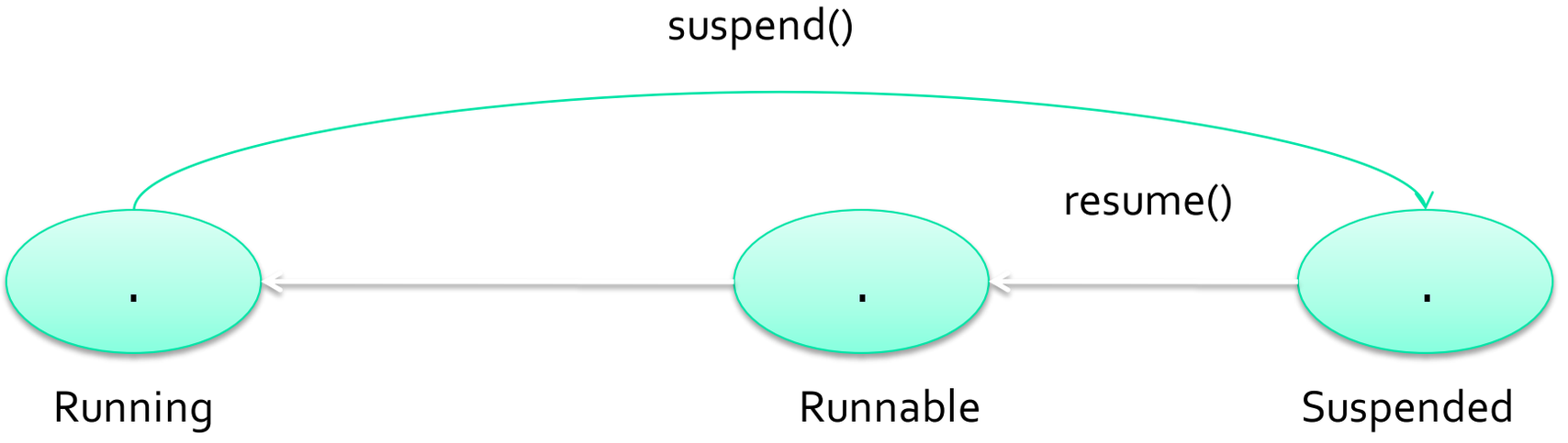


Running state:

- Means that the processor has given its time to the thread for its execution
- Thread runs until the control given to some other thread

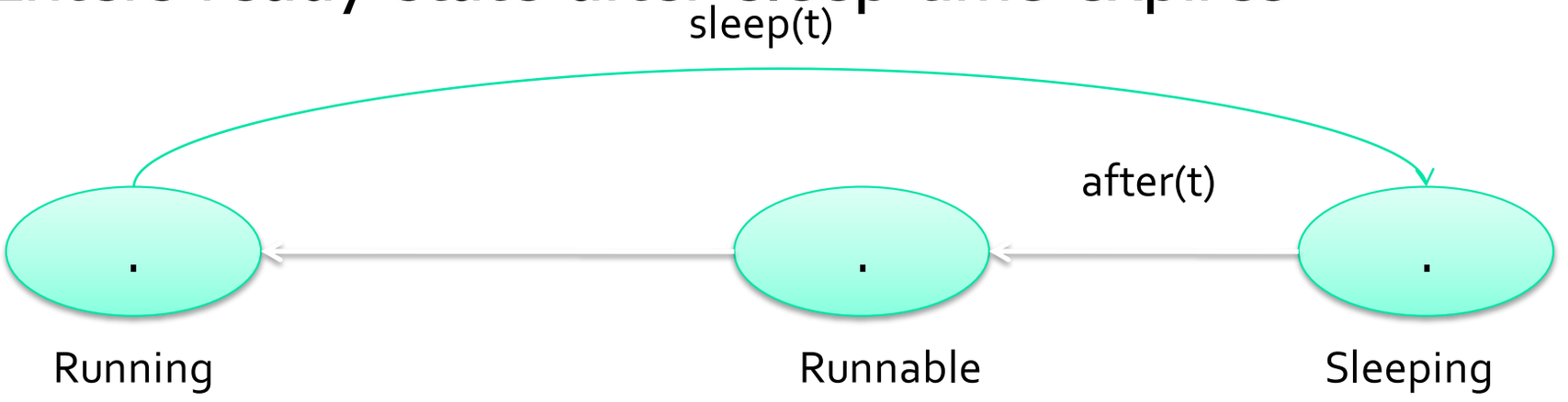
Situations when running thread may relinquish its control:

1. Suspended using suspend() method → suspend for some time due to certain reason, but not to kill the thread

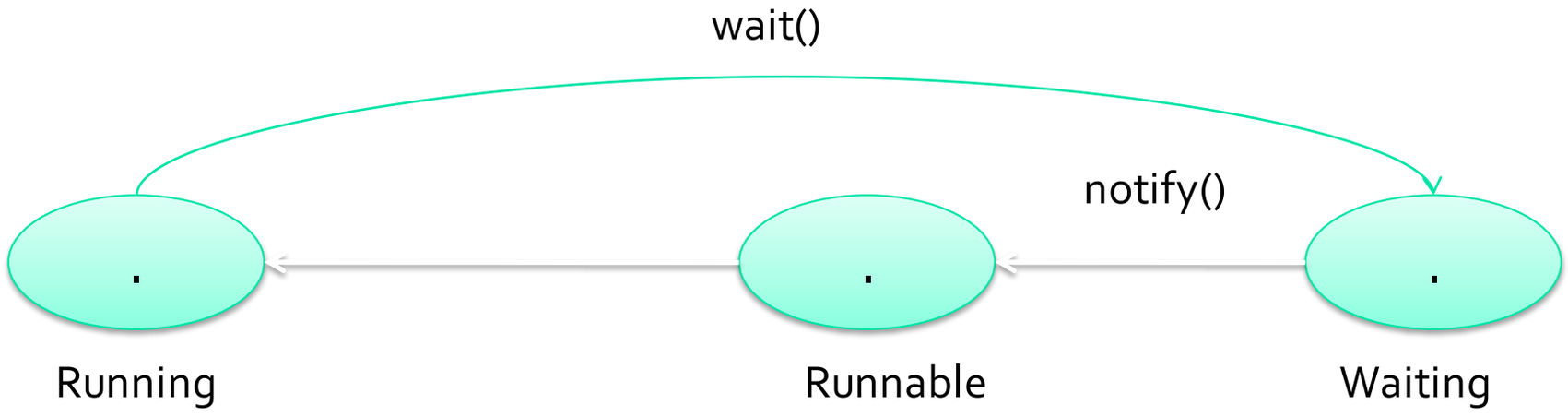


2. Sleep(time) thread is out of the queue this time period

- Entered when **`sleep`** method called
- Cannot use processor
- Enters ready state after sleep time expires



- 3. Wait until some event occurs
- Waiting state
  - Entered when **`wait`** called in an object thread is accessing
  - One waiting thread becomes ready when object calls **`notify`**
  - **`notifyAll`** - all waiting threads become ready



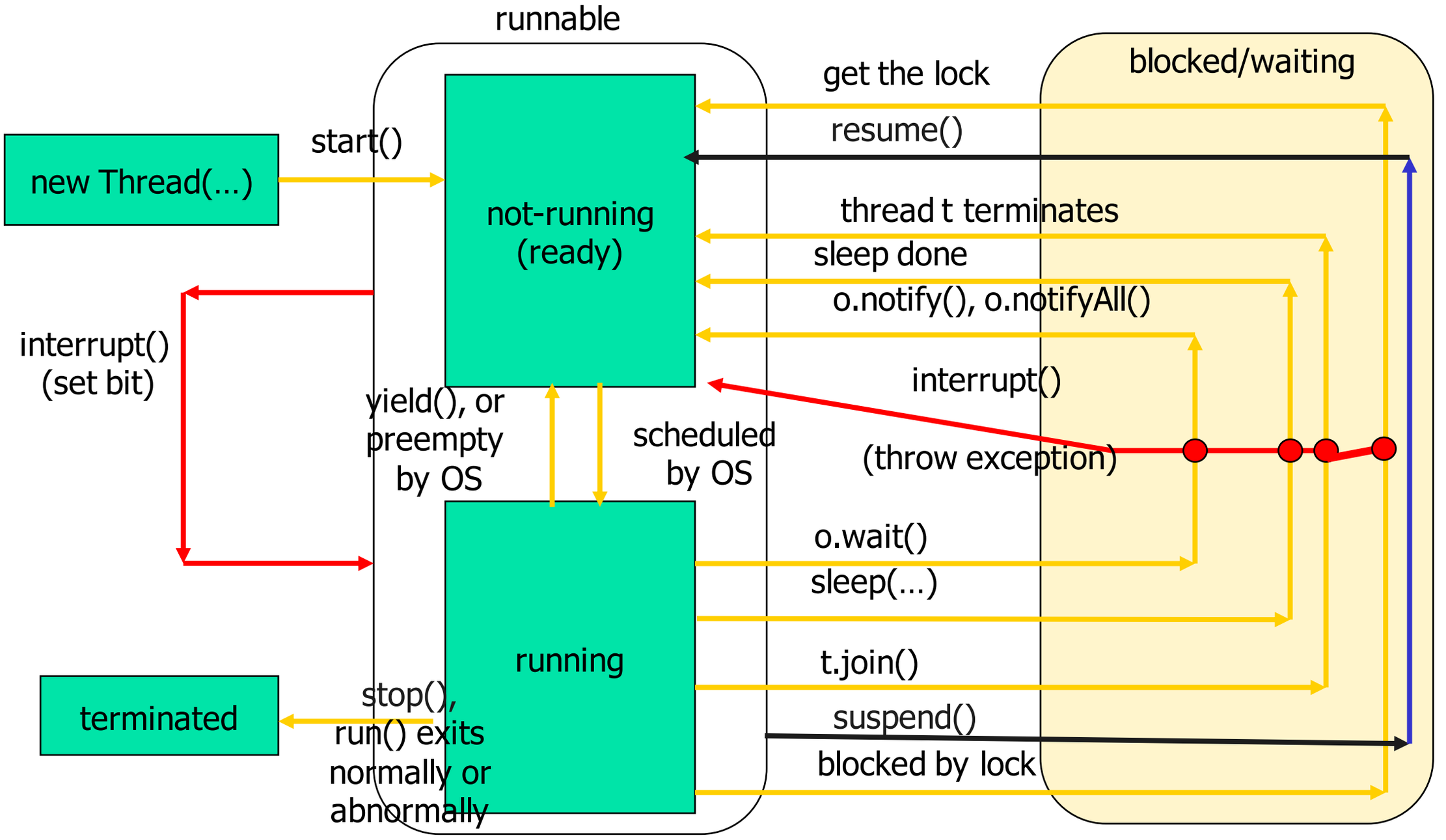
Blocked state/Waiting

- Blocked thread is considered "not runnable" but not dead and fully qualified to run again
  - Entered from running state
  - Blocked thread cannot use processor, even if available
  - Common reason for blocked state - waiting on I/O request
  - waiting to acquire a monitor lock to enter or re-enter a synchronized block/method
- suspend(),Sleep(),wait(),join()

Dead state

- Running thread ends when it completed executing its run() method
- Also using stop()
- All stages running,runnable,blocked

The life cycle of a Java thread



```
public class ThreadDemo extends Thread
 {
public void run()
{
 System.out.println("Thread is running !!");
 }
public static void main(String[] args)
{
ThreadDemo t1 = new ThreadDemo();
 ThreadDemo t2 = new ThreadDemo();
    System.out.println("T1 ==> " + t1.getState());
    System.out.println("T2 ==> " + t2.getState());
```

```
t1.start();
System.out.println("T1 ==> " + t1.getState());
System.out.println("T2 ==> " + t2.getState());
t2.start();
System.out.println("T1 ==> " + t1.getState());
System.out.println("T2 ==> " + t2.getState()); } }
```

- 1 ==> NEW
- T2 ==> NEW
- T1 ==> RUNNABLE
- T2 ==> NEW
- T1 ==> RUNNABLE
- T2 ==> RUNNABLE
- Thread is running !!
- Thread is running !!

isAlive()

- TwoThreadAlive tt = new TwoThreadAlive(); tt.setName("Thread"); System.out.println("before start(), tt.isAlive()=" + tt.isAlive());
- tt.start();
- System.out.println("just after start(), tt.isAlive()=" + tt.isAlive());

Thread Priority

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running. The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
  - Java allows users to change priority:
    - ThreadName.setPriority(intNumber)
      - MIN_PRIORITY = 1
      - NORM_PRIORITY=5
      - MAX_PRIORITY=10

Thread Priority Example

```
class A extends Thread
{
      public void run()
       {
           System.out.println("Thread A started");

           for(int i=1;i<=4;i++)
            {
                 System.out.println("\t From ThreadA: i= "+i);
            }

             System.out.println("Exit from A");
       }

}

class B extends Thread
{
      public void run()
       {
           System.out.println("Thread B started");

           for(int j=1;j<=4;j++)
            {
                 System.out.println("\t From ThreadB: j= "+j);
            }

             System.out.println("Exit from B");
       }
```

Thread Priority Example

```
class C extends Thread
{
        public void run()
          {
              System.out.println("Thread C started");

              for(int k=1;k<=4;k++)
                {
                     System.out.println("\t From ThreadC: k=
   "+k);
                }
                 System.out.println("Exit from C");
          }
```

```
class ThreadPriority
{
        public static void main(String args[])
         {
                  A threadA=new A();
                  B threadB=new B();
                  C threadC=new C();

                threadC.setPriority(Thread.MAX_PRIORITY);
                threadB.setPriority(threadA.getPriority()+1);
                threadA.setPriority(Thread.MIN_PRIORITY);

                System.out.println("Started Thread A");
                 threadA.start();

                System.out.println("Started Thread B");
                 threadB.start();

                System.out.println("Started Thread C");
                 threadC.start();

                 System.out.println("End of main thread");
         }
}
```