MULTITREADING AND MULTITASKING

In programming, there are two main ways **to improve the throughput of a program:**

i) **by using multi-threading**

ii) **by using multitasking**

Both these methods take advantage of parallelism to efficiently utilize the power of CPU and improve the throughput of program.

difference between multithreading and multi-tasking

1. The basic difference between multitasking and multithreading is that in multitasking, the **system allows executing multiple programs and tasks at the same time**, whereas, in multithreading, **the system executes multiple threads of the same or different processes at the same time.**
2. Multi-threading is more granular than multi-tasking. In multi-tasking, **CPU switches between multiple programs to complete their execution in real time**, while in multi- threading CPU switches between multiple threads of the same program. Switching between multiple processes has more context switching cost than switching between multiple threads of the same program.
3. Processes are heavyweight as compared to threads. They require their own address space, which means multi-tasking is heavy compared to multithreading.
4. **Multitasking allocates separate memory and resources for each process**/program whereas, in **multithreading threads belonging to the same process shares the same memory and resources as that of the process**.

Comparison between multithreading and multi-tasking

Parameter	Multi tasking	Multi threading
Basic	Multitasking lets CPU to execute multiple tasks at the same time.	Multithreading lets CPU to execute multiple threads of a process simultaneously.
Switching	In multitasking, CPU switches between programs frequently.	In multithreading, CPU switches between the threads frequently.
Memory and Resource	In multitasking, system has to allocate separate memory and resources to each program that CPU is executing.	In multithreading, system has to allocate memory to a process, multiple threads of that process shares the same memory and resources allocated to the process.

Multitasking

Multitasking is when a single CPU performs several tasks (program, process, task, threads) at the same time. To perform multitasking, the CPU switches among these tasks very frequently so that user can interact with each program simultaneously.

In a multitasking operating system, several users can **share the system** simultaneously. CPU rapidly switches among the tasks, so a little time is needed to switch from one user to the next user. This puts an impression on a user that entire computer system is dedicated to him.

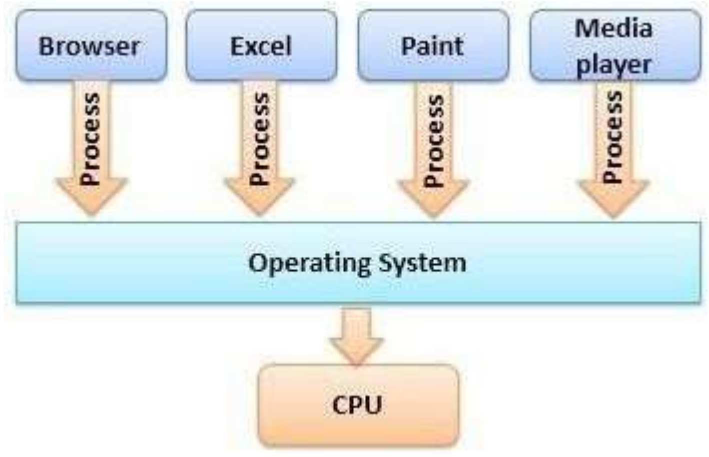

Figure: Multitasking

When several users are sharing a multitasking operating system, CPU scheduling and multiprogramming makes it possible for each user to have at least a small portion of Multitasking OS and let each user have at least one program in the memory for execution.

Multi threading

Multithreading is different from multitasking in a sense that multitasking allows multiple tasks at the same time, whereas, the Multithreading allows multiple threads of a single task (program, process) to be processed by CPU at the same time.

A thread is a basic execution unit which has its own program counter, set of the register and stack. But it shares the code, data, and file of the process to which it belongs. A process can have multiple threads simultaneously, and the CPU switches among these threads so fre- quently making an impression on the user that all threads are running simultaneously.

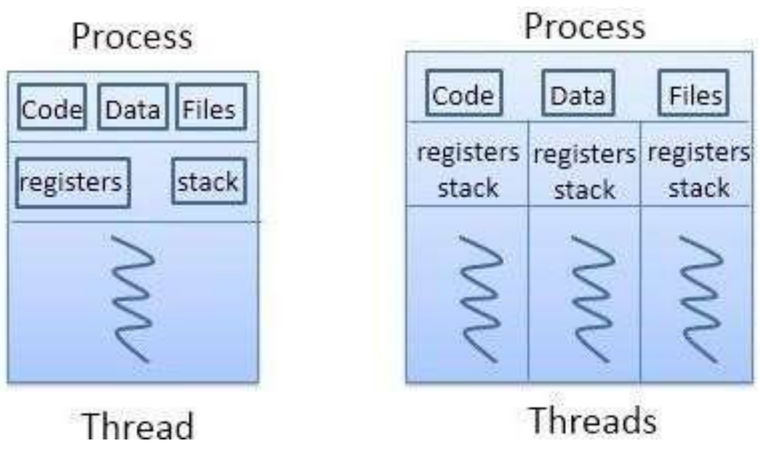

Figure: Multithreading

Benefits of Multithreading

- Multithreading increases the **responsiveness** of system as, if one thread of the application is not responding, the other would respond in that sense the user would not have to sit idle.

- Multithreading allows **resource sharing** as threads belonging to the same process can share code and data of the process and it allows a process to have multiple threads at the same time active in same address space.
- Creating a different process is costlier as the system has to allocate different memory and resources to each process, but creating threads is easy as it does not require allocating separate memory and resources for threads of the same process.

THREAD LIFE CYCLE

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1) New
2) Runnable
3) Blocked
4) Waiting
5) Timed Waiting
6) Terminated

The following figure represents various states of a thread at any instant of tim

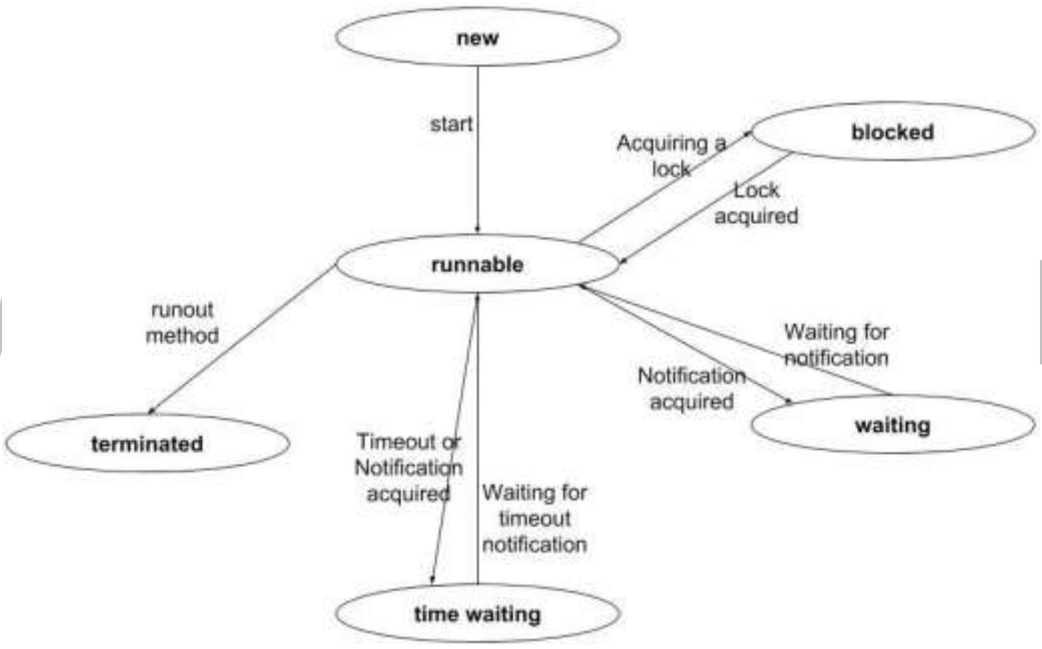

Figure: Life Cycle of a thread

1. New Thread:

2. Runnable State:

- A thread that is ready to run is moved to runnable state.
- In this state, a thread might actually be running or it might be ready run at any instant of time.
- It is the responsibility of the thread scheduler to give the thread, time to run.
- A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

***3.* Blocked/Waiting state:**

- When a thread is temporarily inactive, then it's in one of the following states:
  - Blocked
  - Waiting
- For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/ waiting thread.
- A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.
- A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the threads which is blocked for that section and moves it to the runnable state. A thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.
- If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

***4.* Timed Waiting:**

- A thread lies in timed waiting state when it calls a method with a time out parameter.
- A thread lies in this state until the timeout is completed or until a notification is received.
- For example, when a thread calls sleep or a conditional wait, it is moved to timed waiting state.

***5.* Terminated State:**

- A thread terminates because of either of the following reasons:

  - Because it exits normally. This happens when the code of thread has entirely executed by the program.
  - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.
- A thread that lies in terminated state does no longer consumes any cycles of CPU.

Creating threads

- Threading is a facility to allow multiple tasks to run concurrently within a single process. Threads are independent, concurrent execution through a program, and each thread has its own stack.

In Java, There are two ways to create a thread:

1) By extending Thread class.
2) By implementing Runnable interface.

Java Thread Benefits

1. Java Threads are lightweight compared to processes as they take less time and re-source to create a thread.
2. Threads share their parent process data and code
3. Context switching between threads is usually less expensive than between process-es.
4. Thread intercommunication is relatively easy than process communication.

THREAD CLASS

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Commonly used methods of thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.

1. **public void join(long miliseconds):** waits for a thread to die for the specified mili-seconds.
2. **public int getPriority():** returns the priority of the thread.

3. **public int setPriority(int priority):** changes the priority of the thread.
4. **public string getname():** returns the name of the thread.
5. **public void setname(string name):** changes the name of the thread.
6. ***public thread currentthread():*** returns the reference of currently executing thread.

7. **public int getid():** returns the id of the thread.
8. **public thread.state getstate():** returns the state of the thread.
9. **public boolean isalive():** tests if the thread is alive.
10. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
11. **public void suspend():** is used to suspend the thread(depricated).
12. **public void resume():** is used to resume the suspended thread(depricated).
13. **public void stop():** is used to stop the thread(depricated).
14. **public boolean isdaemon():** tests if the thread is a daemon thread.
15. **public void setdaemon(boolean b):** marks the thread as daemon or user thread.
16. **public void interrupt():** interrupts the thread.
17. **public boolean isinterrupted():** tests if the thread has been interrupted.
18. **public static boolean interrupted():** tests if the current thread has been interrupted.

naming thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. But we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

public string getname(): is used to return the name of a thread.

public void setname(string name): is used to change the name of a thread.

extending thread

The first way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

Sample java program that creates a new thread by extending Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread
{
   NewThread()
   { // Create a new, second thread super("Demo
      Thread"); System.out.println("Child thread: "
      + this); start(); // Start the thread
   }
   // This is the entry point for the second thread.
   public void run()
   {
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println("Child interrupted.");
      }
      System.out.println("Child thread is exiting");
   }
}
public class ExtendThread
{
   public static void main(String args[])
   {
      new NewThread(); // create a new thread
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println("Main Thread: " + i);
            Thread.sleep(1000);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread is exiting.");
   }
```

```
}
```

Sample Output:

```
(output may vary based on processor speed and task load)
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Child thread is exiting.
Main Thread: 2
Main Thread: 1
Main thread is exiting.
```

The child thread is created by instantiating an object of NewThread, which is derived from Thread. The call to super( ) is inside NewThread. This invokes the following form of the Thread constructor:

```
public Thread(String threadName)
```

Here, threadName specifies the name of the thread.

implementing runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- Runnable abstracts a unit of executable code. We can construct a thread on any object that implements Runnable.
- To implement Runnable, a class need only implement a single method called run( ), which is declared as:

```
public void run( )
```

- Inside run( ), we will define the code that constitutes the new thread. The run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within the program. This thread will end when run( ) returns.
- After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class.
- After the new thread is created, it will not start running until we call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ).

- The start( ) method is shown as:

```
void start( )
```

Sample java program that creates a new thread by implementing Runnable:

```
// Create a second thread
class NewThread implements Runnable
{
   Thread t;
   NewThread()
   {
      // Create a new, second thread
      t = new Thread(this, “Demo Thread”);
      System.out.println(“Child thread: “ + t);
      t.start(); // Start the thread
    }
// This is the entry point for the second thread.
   public void run()
   {
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println(“Child Thread: “ + i);
            Thread.sleep(500);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println(“Child interrupted.”);
      }
      System.out.println(“Child thread is exiting.”);
   }
}
public class ThreadDemo
{
   public static void main(String args[])
```

```
    {
      new NewThread(); // create a new thread
      try
      {
        for(int i = 5; i > 0; i--)
        {
          System.out.println("Main Thread: " + i);
          Thread.sleep(1000);
        }
      }
      catch (InterruptedException e)
      {
        System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread is exiting.");
    }
  }
```

Inside NewThread's constructor, a new Thread object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing this as the first argument indicates that we want the new thread to call the run( ) method on this object. Next, start( ) is called, which starts the thread of execution beginning at the run( ) method. This causes the child thread's for loop to begin. After calling start( ), NewThread's constructor returns to main(). When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.

Sample Output:

(output may vary based on processor speed and task load)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Child thread is exiting.
```

```
Main Thread: 2
Main Thread: 1
Main thread is exiting.
```

In a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang.” The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread.

Choosing an approach

The Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run(). This is, of course, the same method required when we implement Runnable. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if we will not be overriding any of Thread’s other methods, it is probably best simply to implement Runnable.

Creating Multiple threads

The following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable
{
  String name; // name of thread
  Thread t;
  NewThread(String threadname)
  {
    name = threadname;
    t = new Thread(this, name);
    System.out.println(“New thread: “ + t);
    t.start(); // Start the thread
  }
  // This is the entry point for thread.
  public void run()
  {
    try
    {
      for(int i = 5; i > 0; i--)
      {
        System.out.println(name + “: “ + i);
```

```
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(name + “Interrupted”);
        }
        System.out.println(name + “ exiting.”);
    }
}
public class MultiThreadDemo
{
    public static void main(String args[])
    {
        new NewThread(“One”); // start threads
        new NewThread(“Two”);
        new NewThread(“Three”);
        try
        {
            // wait for other threads to end
            Thread.sleep(10000);
        }
        catch (InterruptedException e)
        {
            System.out.println(“Main thread Interrupted”);
        }
        System.out.println(“Main thread exiting.”);
    }
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
 Two: 5
```

```
Three: 5
One: 4
Two: 4
 Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One     exiting.
Two     exiting.
Three exiting.
Main thread exiting.
```

As we can see, once started, all three child threads share the CPU. The call to sleep(10000) in main(). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

using isalive( ) and join( )

We want the main thread to finish last. In the preceding examples, this is accomplished by calling sleep( ) within main( ), with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended?

Two ways exist to determine whether a thread has finished or not.

- First, we can call isAlive( ) on the thread. This method is defined by Thread.

Syntax:

```
final boolean isAlive( )
```

The isAlive( ) method returns true, if the thread upon which it is called is still running. It returns false, otherwise.

- Second, we can use join() to wait for a thread to finish.

Syntax:

```
final void join( ) throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it.

Sample Java program using join() to wait for threads to finish.

```
class NewThread implements Runnable
```

```
{
   String name; // name of thread
   Thread t;
   NewThread(String threadname)
   {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start(); // Start the thread
   }
   // This is the entry point for thread.
   public void run()
   {
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println(name + " interrupted.");
      }
      System.out.println(name + " is exiting.");
   }
}
public class DemoJoin
{
   public static void main(String args[])
   {
      NewThread   ob1   =   new   NewThread("One");
      NewThread   ob2   =   new   NewThread("Two");
      NewThread ob3 = new NewThread("Three");
```

```
        System.out.println("Thread  One  is  alive:  "  +  ob1.t.isAlive());
        System.out.println("Thread  Two  is  alive:  "  +  ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        // wait for threads to finish
        try
        {
          System.out.println("Waiting for threads to finish.");
          ob1.t.join();
          ob2.t.join();
          ob3.t.join();
        }
        catch (InterruptedException e)
        {
          System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        System.out.println("Main thread is exiting.");
      }
    }
```

sample output:

```
(output may vary based on processor speed and task load)
New thread: Thread[One,5,main]
New thread:Thread[Two,5,main]
One: 5
New thread: Thread[Three,5,main]
Two: 5
Thread One is alive: true Thread
Two is alive: true Thread Three
is alive: true Waiting for threads
to finish. Three: 5
One: 4
Two: 4
Three: 4
```

```
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
One     is     exiting.
Two     is     exiting.
Three is exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread is exiting.
```