Module V

Part I

Introduction to Multithreading

Multi programming

- In a multi-programmed system, as soon as one job goes for an I/O task, the Operating System interrupts that job, chooses another job from the job pool (waiting queue), gives CPU to this new job and starts its execution.
- The previous job keeps doing its I/O operation while this new job does CPU bound tasks.
- Now say the second job also goes for an I/O task, the CPU chooses a third job and starts executing it.
- As soon as a job completes its I/O operation and comes back for CPU tasks, the CPU is allocated to it.
- In this way, no CPU time is wasted by the system waiting for the I/O task to be completed.
- Therefore, the ultimate goal of multi programming is to keep the CPU busy as long as there are processes ready to execute.

Multiprocessing

- Multiprocessing is the use of two or more CPUs (processors) within a single Computer system.
- With the help of multiprocessing, many processes can be executed simultaneously.
- Say processes P1, P2, P3 and P4 are waiting for execution. Now in a single processor system, firstly one process will execute, then the other, then the other and so on.
- But with multiprocessing, each process can be assigned to a different processor for its execution.
- If its a dual-core processor (2 processors), two processes can be executed simultaneously and thus will be two times faster, similarly a quad core processor will be four times as fast as a single processor.

MULTITASKING

- Multitasking is a logical extension of multi programming.
- The major way in which multitasking differs from multi programming is that multi programming works solely on the concept of context switching whereas multitasking is based on time sharing alongside the concept of context switching.
- Thus the CPU makes the processes to share time slices between them and execute accordingly.
- As soon as time quantum of one process expires, another process begins its execution.

Multi threading

- Multi threading is an execution model that allows a single process to have multiple code segments (i.e., threads) running concurrently within the “context” of that process.
- We can think of threads as child processes that share the parent process resources but execute independently.

Example

- Say there is a web server which processes client requests. Now if it executes as a single threaded process, then it will not be able to process multiple requests at a time. Firstly one client will make its request and finish its execution and only then, the server will be able to process another client request. This is really costly, time consuming and tiring task. To avoid this, multi threading can be made use of.

Advantages of Multi threading

- Benefits of Multi threading include increased responsiveness.
- Since there are multiple threads in a program, so if one thread is taking too long to execute or if it gets blocked, the rest of the threads keep executing without any problem.
- Thus the whole program remains responsive to the user by means of remaining threads.
- Another advantage of multi threading is that it is less costly.
- Creating brand new processes and allocating resources is a time consuming task, but since threads share resources of the parent process, creating threads and switching between them is comparatively easy.
- Hence multi threading is the need of modern Operating Systems.

Thread in Java

- A thread is a lightweight subprocess, the smallest unit of processing.
- It is a separate path of execution.
- Threads are independent.
- If there occurs exception in one thread, it doesn't affect other threads.
- It uses a shared memory area.

Ways of Creating a thread in java

- There are two ways to create a thread:

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

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()**: Allocates a new Thread object
  - **Thread(Runnable target)**: Allocates a new Thread object
  - **Thread(Runnable target, String name)**: Allocates a new Thread object
  - **Thread(String name)**: Allocates a new Thread object

- **Commonly used methods of Thread class**:
  - **public void run():** is used to perform action for a thread.
  - **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
  - **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
  - **public void join():** waits for a thread to die.
  - **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
  - **public int getPriority():** returns the priority of the thread.
  - **public int setPriority(int priority):** changes the priority of the thread.
  - **public String getName():** returns the name of the thread.
  - **public void setName(String name):** changes the name of the thread.

- **public Thread currentThread():** returns the reference of currently executing thread.
- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.
- **public boolean isAlive():** tests if the thread is alive.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).

Runnable interface

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run()
- **public void run():** is used to perform action for a thread.

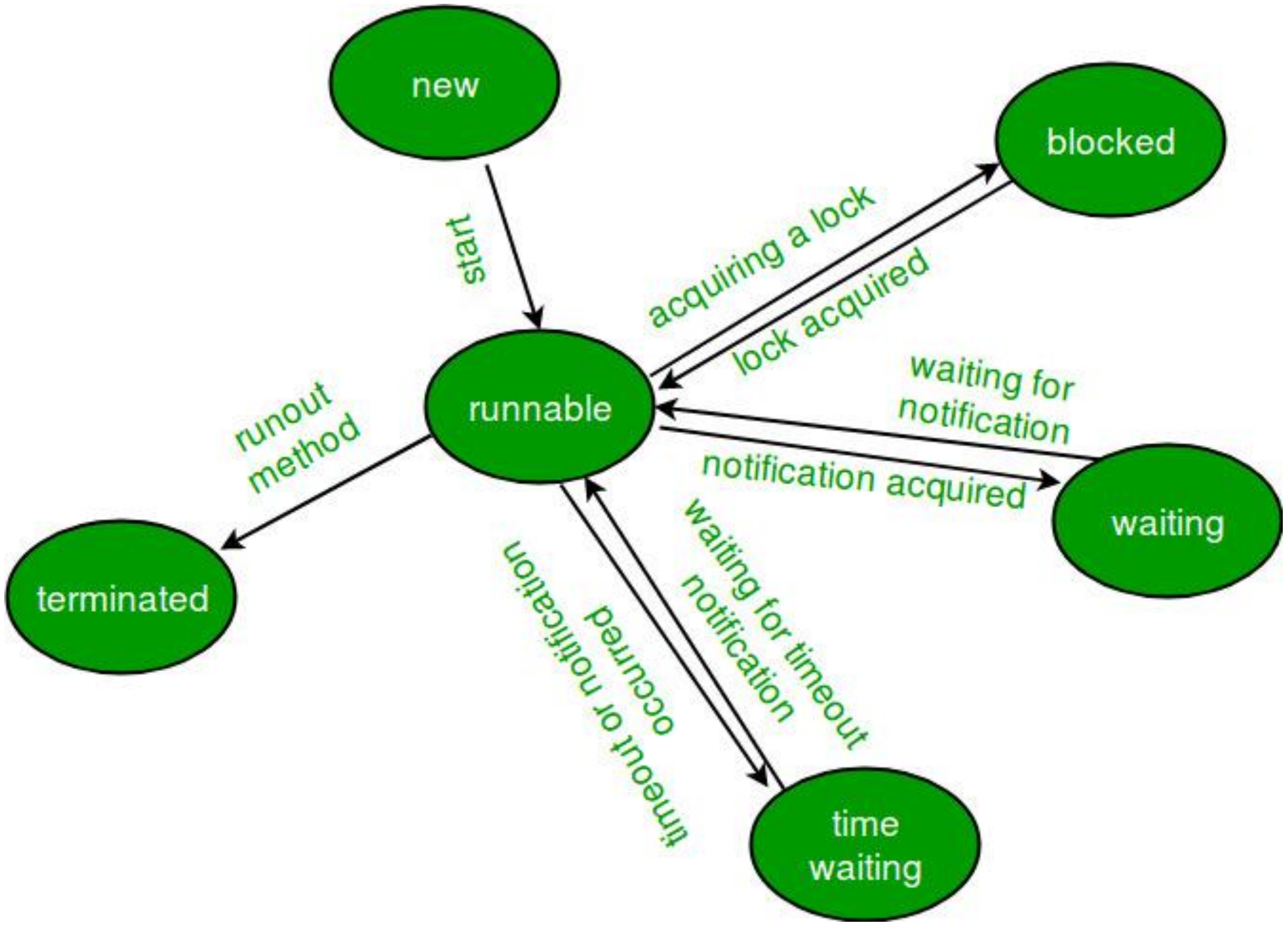
Starting a thread

- **start() method** of Thread class is used to start a newly created thread. It performs following tasks:
  - A new thread starts(with new callstack).
  - The thread moves from New state to the Runnable state.
  - When the thread gets a chance to execute, its target run() method will run.

Life cycle of a Thread (Thread States)

- A thread can be in one of the following states:
- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.
- A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

Thread states



Source:Google Images

Creating thread by extending thread class

```
public class MyThread extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
MyThread t1=new MyThread();
t1.start();
 }
}
```

Output:

```
thread is running...
```

Creating thread by implementing runnable interface

```
class MyThread implements Runnable{
public void run(){
System.out.println("thread is running..");
}
public static void main(String args[]){
MyThread m1=new MyThread();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output:

```
thread is running..
```

What if we call run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.
- So there will not be multiple threads

Can we start a thread twice

- No. After starting a thread, it can never be started again.
- If you does so, an *IllegalThreadStateException* is thrown.
- In such case, thread will run once but for second time, it will throw exception.

Thread Scheduler in Java

- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- **Only one thread at a time can run in a single process.**
- The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Sleep method in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.
- The Thread class provides two methods for sleeping a thread:
  - public static void sleep(long miliseconds)throws InterruptedException
  - public static void sleep(long miliseconds, int nanos)throws InterruptedException

EXAMPLE

```
public class Main extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{
            Thread.sleep(500);
        }
catch(InterruptedException e) {
            System.out.println(e);
        }
    System.out.println(i);
  }
 }
 public static void main(String args[]){
 Main t1=new Main();
  Main t2=new Main();
  t1.start();
  t2.start();
 } }
```

Output

```
1
1
2
2
3
3
4
4
```

The join() method

- The join() method waits for a thread to die.
- In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.
- Syntax:

```
public void join()throws InterruptedException
public void join(long milliseconds)throws InterruptedException
```

Example

```
class Main extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 Main t1=new Main();
 Main t2=new Main();
Main t3=new Main();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

OUTPUT

```
1
2
3
4
5
1
1
2
2
3
3
4
4
5
5
```

Java Thread yield() method

- The **yield()** method of thread class causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **yield()** basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.
- Syntax
  - **public static void** yield()

EXAMPLE

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i=0; i<5 ; i++)
            System.out.println(Thread.currentThread().getName()
                                + " in control");
    }
}
public class Main
{
    public static void main(String[]args)
    {
        MyThread t = new MyThread();
        t.start();

        for (int i=0; i<5; i++)
        {
            // Control passes to child thread
            Thread.yield();

            // After execution of child Thread
            // main thread takes over
            System.out.println(Thread.currentThread().getName()
                                + " in control");
        }
    }
}
```

OUTPUT

```
Thread-0 in control
Thread-0 in control
main in control
Thread-0 in control
Thread-0 in control
Thread-0 in control
main in control
main in control
main in control
```

Naming Thread and Current 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.
- By 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.

EXAMPLE

```
class MyThread extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  MyThread t1=new MyThread();
 MyThread t2=new MyThread();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());

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

  t1.setName("MyThread1");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

OUTPUT

```
Name of t1:Thread-0
Name of t2:Thread-1
After changing name of t1:MyThread1
running...
running...
```

Naming –using constructor

```
class Main extends Thread{
    Main(String s)
    {
        super(s);
    }
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Main t1 =new Main("MyThread");
t1.start();
System.out.println(t1.getName());
 }
}
```

Output

```
MyThread
thread is running…
```

Naming –using constructor

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

public static void main(String args[]){
Main m1=new Main();
Thread t1 =new Thread(m1,"MyThread");
t1.start();
System.out.println(t1.getName());
 }
}
```

OUTPUT

```
MyThread
thread is running…
```

Priority of a Thread

- Each thread have a priority. Priorities are represented by a number between 1 and 10.
- In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling).
- **But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.**
- 3 constants defined in Thread class
  - public static int MIN_PRIORITY
  - public static int NORM_PRIORITY
  - public static int MAX_PRIORITY
- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

EXAMPLE

```
class Main extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());


  }
 public static void main(String args[]){
  Main m1=new Main();
  Main m2=new Main();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

OUTPUT

```
running thread name is:Thread-0
running thread priority is:1
 running thread name is:Thread-1
running thread priority is:10
```

Daemon Thread in Java

- **Daemon thread in java** is a service provider thread that provides services to the user thread.
- Its life depend on the user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- It provides services to user threads for background supporting tasks.
- It has no role in life than to serve user threads.
- It is a low priority thread.
- Example:Garbage Collector