CHAPTER - 8

MULTITHREADED PROGRAMMING

8. Intoduction

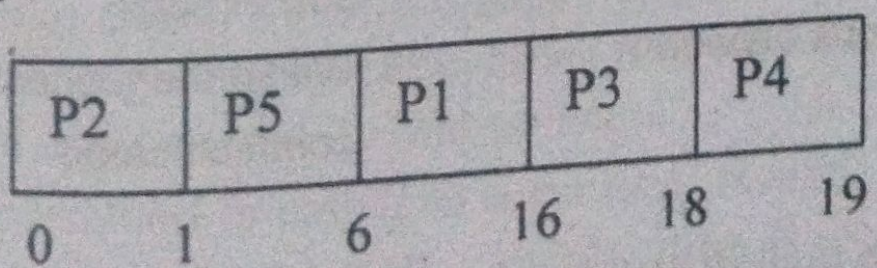
A Multithreaded program contains two or more parts that can run concurrently. Each part of such program is called a thread, and each thread defines a specific path of execution. Multithreading is a specialized form of Multitasking.

8.1. Multitasking:

All modern operating systems support the concept of multitasking. Multitasking is the feature that allows the computer to run two or more programs concurrently. CPU scheduling deals with the problem of deciding for which of the process, the ready queue is to be allocated. It follows CPU (Processor) scheduling algorithms.

a) Priority Scheduling:

Process	Burst time	Priority
P1	10	2
P2	01	4
P3	02	2
P4	01	1
P5	05	3

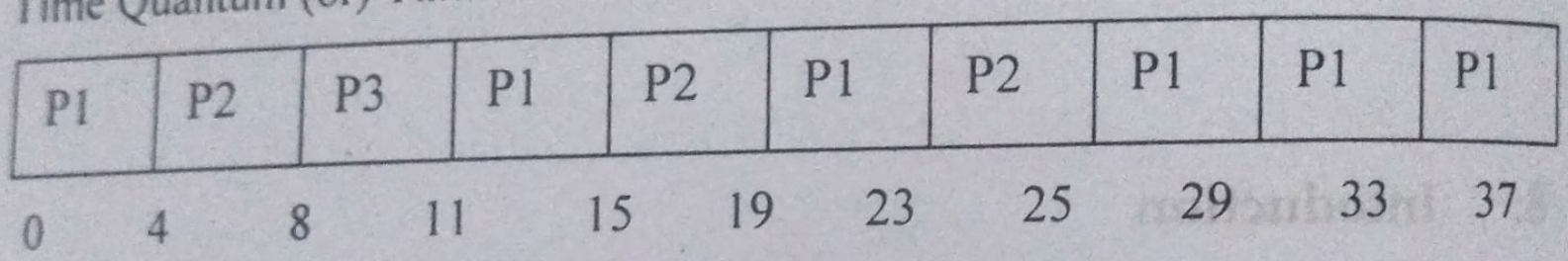


The priority scheduling follows on Sun Solaris systems. But it doesn't work on the present operating systems like **windows NT** etc., The thread with a piece of system code is called the thread scheduler. It determines which thread is running actually in the CPU.

The Solaris java platform runs a thread of given priority to completion or until a higher priority thread becomes ready at that point preemption occurs.

b) Round-Robin Scheduling:

Process	Burst time
P1	24
P2	10
P3	03
Time Quantum (or) Time Sliced :	04



Round Robin scheduling is available on Windows 95 and Windows NT is time sliced.

Java supports thread implementation depending on round robin scheduling. Here, each thread is given a limited amount of time called time quantum. While executing the process when that time quantum expires the thread is made to wait state and all threads that are equal priority get their chances to use their quantum in round robin fashion.

8.2. Difference between Multiprocessing and Multithreading

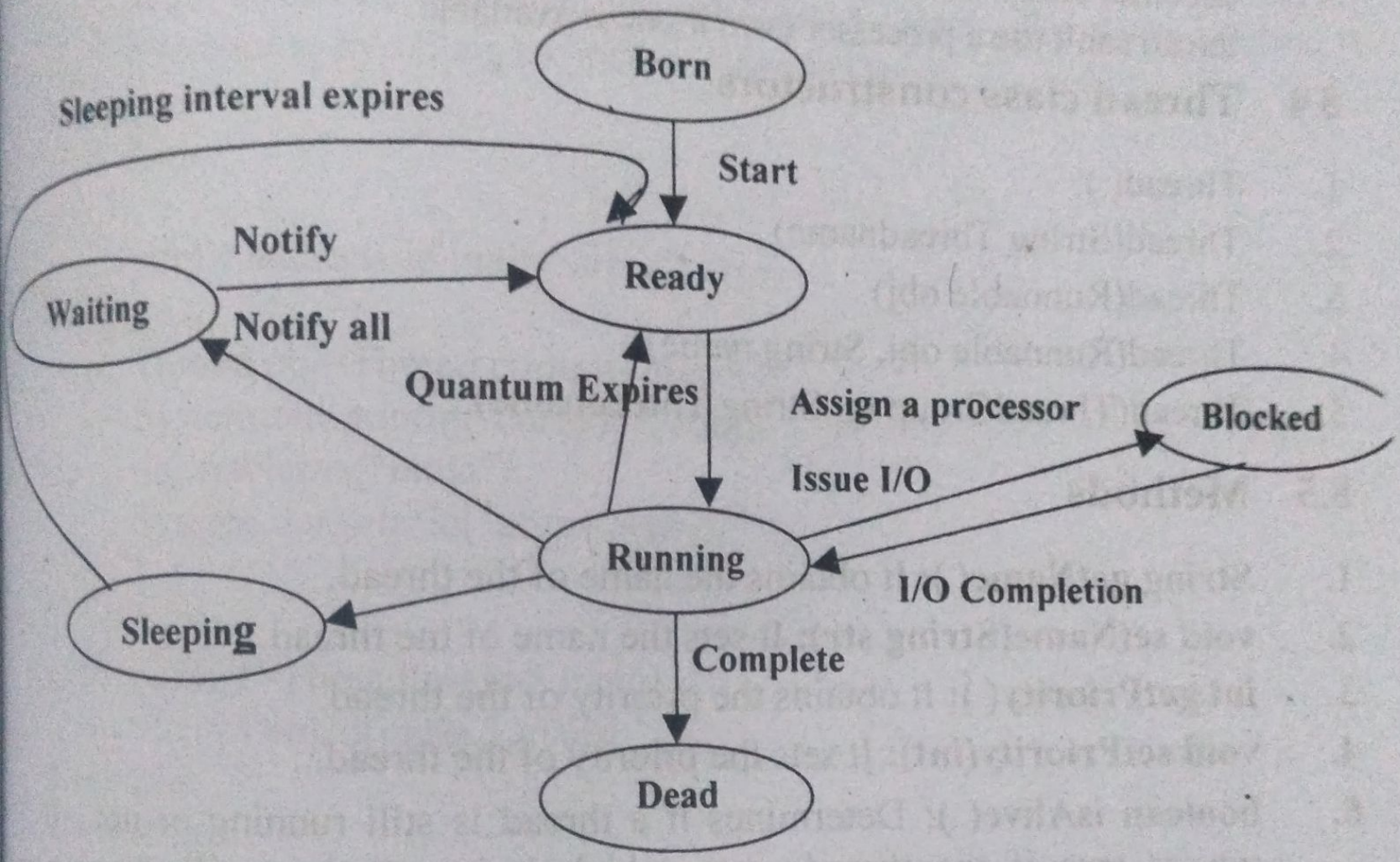
Multiprocessing	Multithreading
1. More than one process running simultaneously	1. More than one thread running simultaneously
2. Its part of a program	2. Its part of a program
3. It's a heavy wait process	3. It's a light wait process
4. Process is divided into threads	4. Threads are divided into sub threads
5. There is no communications between processors directly	5. Within the process threads are communicated

Note: 1. Java provides built-in support for multithreaded programming
2. Java multithreading is a platform independent
3. Elimination of main loop/polling is the main benefit of multithreaded programming.

8.3. Thread States (Life-Cycle of a Thread)

The life cycle of a thread contains several states. At any time the thread is falls into any one of the states.

At any time a thread is said to be in one or more states.



- The thread that was just created is in the born state.
- The thread remains in this state until the threads start method is called. This causes the thread to enter the ready state.
- The highest priority ready thread enter the running state when the system assigns a processor to the thread i.e., the thread begins executing.
- When a running thread calls wait the thread enters into a waiting state for the particular object on which wait was called. Every thread in the waiting state for a given object becomes ready on a call to notify all by another thread associated with that object.
- When a sleep method is called in a running thread, that thread enters into the suspended (sleep) state. A sleeping thread becomes ready after the designated sleep time expires. A sleeping thread cannot use a processor even if one is available.

- A thread enters the dead state when its run( ) method complete (or) terminates for any reason. A dead thread is eventually be disposed of by the system.
- One common way for a running thread to enter the blocked state is when the thread issues an input or output request. In this case a blocked thread becomes ready when the input or output waits for completes. A blocked thread can't use a processor even if one is available.

8.4 Thread class constructors

1. Thread( )
2. Thread(String Threadname)
3. Thread(Runnable obj)
4. Thread(Runnable obj, String name)
5. Thread(ThreadGroup tg, String Threadname)

8.5 Methods

1. **String getName( ):** It obtains the name of the thread.
2. **void setName(String str):** It sets the name of the thread.
3. **int getPriority( ):** It obtains the priority of the thread.
4. **void setPriority(int):** It sets the priority of the thread.
5. **boolean isAlive( ):** Determines if a thread is still running or not. It returns true if the thread upon which it is called is still running. Otherwise, it returns false.
6. **void sleep(long milliseconds):** Suspend a thread for a period of time.
7. **void start( ):** Start a thread by calling its run method.
8. **void run( ):** It defines the code that constitutes the new thread. When ever the start( ) method executes it automatically call of run( ) method.
9. **void join( ):** This method waits until the thread on which it is called terminates.
10. **Thread currentThread( ):** It returns a reference to the currently executing thread.
11. **void yield( ):** A thread can call the yield method to give other threads a chance to execute.
12. **ThreadGroup getThreadGroup( ):** It returns the name of the thread group which contain currently running thread. ThreadGroup represents set of Threads. It has one constructor method as **ThreadGroup(String name).**

8.6 Main Thread:

When a java program starts up, one thread begins running immediately. This one is main thread, which is created by virtual machine. It is executed in the program for the first time. It must be the last thread to finish execution. When the main thread stops the program terminates.

Even though the main thread is automatically created when the program starts, it can be controlled through Thread object. For this we obtain a reference to by calling the method "currentThread( )".

Example:

```
class demo
{
    public static void main(String[] args)
    {
    Thread obj=Thread.currentThread();
    System.out.println("current="+obj);
    obj.setName("india");
    System.out.println("after="+obj);
    }
}
```

O/P: current=Thread[main,5,main]
after=Thread[india,5,main]

Example:

```
class demo
{
    public static void main(String[] args)
    {
    Thread obj=Thread.currentThread();
    try
    {
    for(int n=5;n>0;n--)
    {
    System.out.println(n);
    obj.sleep(1000);
    }
    }
    catch(InterruptedException e)
    {
    System.out.println("exception");
    }
    }
}
```

8.7 Creating a new Thread

A new thread can be created in two ways

1. implements the Runnable interface
2. extends the Thread class itself

1. **implements the Runnable interface:** Runnable is a system defined interface. Once we create a class, it implements the Runnable interface forms a thread class. To implement Runnable, a class need only implement a single method called **run( ).**

Example:

```
class new1 implements Runnable
{
    Thread t;
    public new1()
    {
    t=new Thread(this,"child thread");
    System.out.println("sub thread="+t);
    t.start();
    }
    public void run()
    {
    try
    {
    for(int n=5;n>0;n--)
    {
    System.out.println("child="+n);
    Thread.sleep(500);
    }
    }
    catch(InterruptedException e)
    {
```

```
    System.out.println("child exception");
    }
    System.out.println("child exit");
    }
}
class demo1
{
    public static void main(String[] args)
    {
    new new1();
    try
    {
    for(int n=5;n>0;n--)
    {
    System.out.println("main="+n);
    Thread.sleep(1000);
    }
    }
    catch(InterruptedException e)
    {
    System.out.println("main exception");
    }
    System.out.println("main exit");
    }
}
```

2. **extending the Thread class:** The second way to create a thread is to create a new class that extends **Thread,** and then create an instance of that class. The extending class must override the run( ) method. It must also call start( ) to begin execution of the new thread.

Example:

```
class newthread extends Thread
{

    public newthread()
    {
    super("child thread");
    System.out.println("sub thread="+this);
    start();
```

```
        }
        public void run()
        {
        try
        {
        for(int n=5;n>0;n--)
        {
        System.out.println("child="+n);
        Thread.sleep(500);
        }
        }
        catch(InterruptedException e)
        {
        System.out.println("child exception");
        }
        System.out.println("child exit");
        }
}
class demo2
{
        public static void main(String[] args)
        {
        new newthread();
        try
        {
        for(int n=5;n>0;n--)
        {
        System.out.println("main="+n);
        Thread.sleep(1000);
        }
        }
        catch(InterruptedException e)
        {
        System.out.println("main exception");
        }
        System.out.println("main exit");
        }
}
```


Example:

```
class a extends Thread
{
        public void run()
        {
        for(int n=0;n<5;n++)
        {
        if(n==3)
        yield();
        System.out.println("child 1="+n);
        }
        }
}
class b extends Thread
{
        public void run()
        {
        for(int n=0;n<5;n++)
        {
        try
        {
        sleep(1000);
        }
        catch(Exception e)
        {
        }
        System.out.println("child 2="+n);
        }
        }
}
class c extends Thread
{
        public void run()
        {
        for(int n=0;n<5;n++)
        {
        if(n==2)
```

```
        stop();
        System.out.println("child 3="+n);
        }
        }
}
class demo3
{
        public static void main(String[] args)
        {
        a obj=new a();
        b obj1=new b();
        c obj2=new c();
        obj.start();
        obj1.start();
        obj2.start();
        }
}
```

Example:

```
class b implements Runnable
{
        Thread t;
        b(String tname)
        {
        t=new Thread(this,tname);
        t.start();
        }
        public void run()
        {
        try
        {
        for(int n=0;n<5;n++)
        {
        System.out.println("child 1="+n);
        //sleep(1000); can't work in the case implementing interface
        Thread.sleep(1000);
        }
```



```
        }
        catch(InterruptedException e)
        {
        System.out.println("child exception");
        }
        }
}
class demo4
{
        public static void main(String[] args)
        {
        b obj1=new b("one");
        b obj2=new b("two");
        System.out.println("b is alive:"+obj1.t.isAlive());
        System.out.println("c is alive:"+obj2.t.isAlive());
        try
        {
        obj1.t.join();
        obj2.t.join();
        }
        catch(InterruptedException e)
        {
        System.out.println("child exception");
        }
        System.out.println("b is alive:"+obj1.t.isAlive());
        System.out.println("c is alive:"+obj2.t.isAlive());
        }
}
```

8.8 Thread Priorities

Every thread in Java is assigned a priority value, when more than one thread is computing for CPU time. Generally highest priority thread is running before the lowest priority thread. It is also possible to set priority to each thread by the user using setPriority( ) method. System defined thread priorities are

Thread.MIN_PRIORITY	1
Thread.MAX_PRIORITY	10
Thread.NORM_PRIORITY	5

The default priority is Thread.NORM_PRIORITY

Example:

```
class Demo extends Thread
{
    public void run()
    {
        for(int i=1;i<=3;i++)
        System.out.println(getName()+" "+i);
    }
}
class ThprDemo
{
    public static void main(String[] args)
    {
        Demo obj=new Demo();
        Demo obj1=new Demo();
        Demo obj2=new Demo();
        obj.setPriority(Thread.MAX_PRIORITY);
        obj1.setPriority(Thread.MIN_PRIORITY);
        obj2.setPriority(Thread.NORM_PRIORITY);
        obj.start();
        obj1.start();
        obj2.start();
    }
}
```

O/P:
```
Thread-1 1
Thread-1 2
Thread-1 3
Thread-3 1
Thread-3 2
Thread-3 3
Thread-2 1
Thread-2 2
Thread-2 3
```


8.9 Synchronization

When two or more threads need access to a shared resource, they need someway to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called Synchronization. Java uses the concept of monitor (also called Semaphore) for Synchronization.

A monitor is an object that is used as a mutually exclusive lock, or mutex. The monitor allows one thread at a time to execute a Synchronized method on the object.

8.10 Daemon threads

A "daemon" thread is one that is supposed to provide a general service in the background as long as the program is running, but is not part of the essence of the program. Thus, when all of the non-daemon threads complete, the program is terminated. Conversely, if there are any nondaemon threads still running, the program doesn't terminate. (There is, for instance, a thread that runs **main( )**.)

You can find out if a thread is a daemon by calling isDaemon( ), and you can turn the "daemonhood" of a thread on and off with setDaemon( ). If a thread is a daemon, then any threads it creates will automatically be daemons.

The following example demonstrates daemon threads:

```
//::Daemons.java
import java.io.*;
class Daemon extends Thread {
private static final int SIZE = 10;
private Thread[] t = new Thread[SIZE];
public Daemon() {
setDaemon(true);
start();
}
public void run() {
for(int i = 0; i < SIZE; i++)
t[i] = new DaemonSpawn(i);
for(int i = 0; i < SIZE; i++)
System.out.println(
"t[" + i + "].isDaemon() = "
+ t[i].isDaemon());
```

```
while(true)
yield();
}
}
class DaemonSpawn extends Thread {
public DaemonSpawn(int i) {
System.out.println(
"DaemonSpawn " + i + " started");
start();
}
public void run() {
while(true)
yield();
}
}
public class Daemons {
public static void main(String[] args)
throws IOException {
Thread d = new Daemon();
System.out.println(
"d.isDaemon() = " + d.isDaemon());
// Allow the daemon threads to
// finish their startup processes:
System.out.println("Press any key");
System.in.read();
}
}
```