MULTI-THREADING

WHAT IS MULTI_PROGRAMMING:

- Running more than one program that is running multiple programs on a single machine or a computer is known as multi-programming.
- The idea of multiprogramming started from the utilisation of the CPU when it is idle as the CPU works for just few time in the whole hour.
- There are different form of multi-programming.

➔ **Multi-user**: more than one user using the machine / running their programs simultaneously.

- for connecting more than one user to single computer the **DUMMY TERMINALS** were used.
- here the **ROUND ROBIN** fashion was introduced as the programs were executed simultaneously.
- **UNIX and LINUX** are famous operating systems for the multi-user environment.
- Multi-user machines were known as **TIME_SHARING** machines.

➔ **Multi-Tasking**: single user runs multiple tasks simultaneously.

- here the CPU runs the programs alternatively on high rate.
- **WINDOWS and MacOS(OS X)** operating system supports this type of environment.

➔ **MULTI_THREADING**: it is a type of multi-threading where there are different tasks going on under a single application.

- threads are light weighted compared to the task.
- CPU runs the threads alternatively where the user fells the threads running all together.
- examples: animation, application, gaming, websites, webserver

CONTROL FLOW OF A PROGRAM

- A single program in java contains one control flow.
- Entry point of the program is main method which executes at first.
- Example program:

```
class Test

{
      static void display( )
      {
            s.o.p("HELLO");
      }
      p.s.v main(…)
      {
            display( );
            s.o.p("WORLD");
      }
}
```

- The second method which executes is the calling method.
- The third method which Is executed is the called method.
- The above program is example explaining that the main method is the only thread which controls the flow of the program.
- It is just like getting a reference of a page while reading a book.
- If there is an infinite loop in the display method then the flow of program does not executes further so there needs to be two flow of controls to execute the program.

MULTITHREADING USING THREAD CLASSES

- Java provides **thread class** and **runnable interface** to achieve multithreading.
- Thread class contains the actual mechanism for multithreading.
- In java a class can extend from only one class.

- Runnable interface is used to extends class from some other class
- To achieve multithreading there need to be a class extending another class.
- Example program using thread class:

```
class MyThread extends Thread
{
     public void run( )
     {
          Int i=1;
          while(ture)
          {
               s.o.p(i+”hello”);
               i++;
          }
     }
}
class Test
{
     p.s.v main( )
     {
          MyThread t=new MyThread();
          t.start( );
          int i=1;
          while(true);
          {
               s.o.p(i+”world”);
               i++;
          }
     }
}
```

- Run is the starting point of a program for multithreading.
- There should be a class having all the features for multithreading like the one in the above program.
- the object of thread class is created in the main method.
- The start method call the thread class and runs the method which is a built in method of thread class.

➢ Example program using runnable interface:

```
class My implements Runnable
{
      public void run( );
      {
            Int i=1
            while(true)
            {

                  s.o.p(i+"hello");
                  i++;
            }
      }
}
class Test
{
      p.s.v main( )
      {
            My m=new My( );
            Thread t=new Thread(m);
            t.start( );
            int i=1;
            while(true)
            {
                  s.o.p(i+"world");
                  i++;
            }
      }
}
```

Interfaces are implemented.

➢ The class becomes abstract if it does not implements all the features of interface.

➢ In the above program it gives the example that the object also runs when the thread runs.

STATES OF A THREAD

➢ The first state of the thread is new it stores the object of the thread.

➢ To run the object of thread the start method is called.

➢ When start method is called then it is entered into the ready state where it is ready to run.

- Then it enters into the running state.
- After completing the task it will enter into the terminated state.
- A thread which is terminated is just like a thread which is killed.
- Therefore the different states of thread are

  NEW → READY → RUNNING → TERMINATED
- While running the thread may also enter into different states like:
  - **WAIT STATE:** waiting for acquiring some resource or made to wait by some other thread.
  - **TIME WAIT STATE:** to make the thread to delay for some time using the sleep method, it is also known as sleep state
  - **WAIT AND NOTIFY:** where the thread is to be in the waiting state to get to its chance till it gets notified.
  - **BLOCKED STATE:** it is just like entering into the monitor where the thread is being locked for some time, it is similar to waiting state.

THREAD PRIORITIES

- JAVA supports thread priorities from 1-10.
- Execution of threads depends upon scheduler.
- If a thread is having higher priority then it should get some preference over other threads.
- In java there are different levels of priority that are:
  - **MIN_PRIORITY=1.**
  - **NORM_PRIORITY=5.**
  - **MAX_PRIORITY=10.**

for example:

In MS Word one thread takes the input from the keyboard, another thread checks for the spellings which works simultaneously, another thread which works to auto save the document.
In the above the first priority is given to the thread which takes the input

- The priority of default thread is always 5.
- The higher priority is given to the thread which gets the input or the data.
- The thread with higher priority gets the more CPU time.
- Multithreading features are provided by the operating systems but in java JVM have its own scheduler.

THREAD CLASS

- Object of the thread class can be created.
- Whenever a thread is created it gets some IDE.
- Threads can be identified by mentioning their names.
- There are different constructors to give the thread classes:

→ Thread( )

- it is a default class.

→ Thread(Runnable r)

- the thread contains the runnable interface.

→ Thread(Runnable r, String name)

- the thread class have its own name with runnable interface.

→ Thread(ThreadGroup g, String name)

- thread group to manage various threads together.

For example:

Different types of balloons or balls in an animation having their own thread classes is the example for the thread group.

➔ Thread(String name)

- the thread class have its own name.

➢ There are various getter and setter methods:

➔ long getId( )

- to get the id of particular thread.

➔ String getName( )

- to get the name of the thread mentioned.

➔ int getPriority( )

- to know the current priority of the thread.

➔ Thread.state getState( )

- to get to know the state of the thread .

➔ ThreadGroup getthreadGroup( )

- to know the group to which the thread belongs.

➔ void setName(String name)

- to set the Name of the thread class.

➔ void setPriority(int p)

- to set the priority of the thread class.

➔ void setDaemon(Boolean d)

- to set a background thread with least priority with no user interaction.

For example:

The garbage collector in the JVM which have the least priority.

➢ There are different methods for the enquirying the thread classes:

➔ boolean isAlive( )

- to check if the thread is alive or terminated.

➔ boolean isDaemon( )

- to check if the thread is acting as a daemon or not.

➔ boolean isInterrupted( )

- to check whether the thread is interrupted by some other methods or not.

➢ There are different instance methods for the thread classes:

➔ void interrupt( )

- it is used to interrupt the thread mostly in the state of waiting or sleeping.

➔ void join( )

- instead of terminating the thread is been kept waiting to join with other thread until it is completed using the join method.

➔ void join(long millis)

- it is used to join the main function with other for a period of time.

➔ void run( )

- it has the actual functionality of thread and it can be overided

➔ void start( )

➢ There are different static methods for the thread classes:

➢ These can be called by just using the class name.

➔ Int activeCount( )

- it gives the number of threads active in a particular group.

➔ Thread currentThread( )

- to get the reference of the current or the running thread.

➔ void yield( )

- To give the indication for the thread with higher priority to wait for sometime to let the threads with low mpriority to finish their job.

➔ void dumpStack( )

- to know the contents in the stack or to know the depth of the stack.

SYNCHRONIZATION

➢ **Definition:** it is the coordination or understanding between two entities.

➢ In multithreading it is usually used between threads.

➢ Terms important to learn synchronization in multithreading:

➔ **Resource Sharing:**

- it is like more than one thread accessing same resource like file, network connection, data objects etc
- if there is an object in the heap then it can be accessed by multiple threads.
- then than object becomes the shared resource.

➔ **Critical Section:**

- the lines of code in a thread which are accessing the shared resource/object is the critical section.
- it is said to be a piece of code which is critical.

➔ **Mutual Exclusion:**

- the mutual exclusion states that the accessing of one thread prevents the accessing of another.

➔ **Locking/Mutex:**

- Mutex is the variable used for locking the threads.
- if the mutex variable is set to zero then it is free or it is not occupied.
- when the time period of one thread is finished then the another thread cannot access the object as the mutex will not remain zero.
- thread two can access the object if and only if the mutex is zero again.
- for every shared resource there should be a lock which is being applied by the thread itself.
- here the threads are responsible for mutual exclusion.
- mutex was not useful as the threads would be overlapping each other if the mutex was not being locked by the first one.

➔ **Semaphore:**

- semaphore was like an operating system before the introducing of java to control the coordination of threads as they should not overlap.
- it was supported by UNIX operating system.
- the semaphore creates the scenario where the thread which have been occupying the object would signal the other after its work is finished.
- it creates a block queue where the upcoming thread is to be in waiting state.
- the methods used here are wait( ) and signal( ).
- here the operating system have the mutual exclusion.

➔ **Monitor:**

- here the object itself takes the responsibility for the mutual exclusion.
- it can be achieved using object orientation.
- the complete mechanism is inside the object itself.
- the read and write method, the data and the block queue belongs to the shared object itself as it can be accessed by any of the threads at a particular time.
- Here java makes sure than one thread is accessed at a time.
- example program:

```
class MyData
{
    void display (String str)
    {
    synchronize(this)
    {
        for(int
i=0;i<str.length();i++)
        {
            s.o.p(str.charAt(i));
        }
    }
    }
}
class MyThread1 extends Thread
{
    Mydata d;
    MyThread1(MyData dat){d=dat;}
    public void run()
    {
        d.display("Hello World");
    }
}
class MyThread2 extends Thread
{
    Mydata data;
    MyThread2(MyData dat)
{data=dat;}
    public void run()
    {
        d.display("Welcome");
    }
}
class Test
{
    p.s.v.main(…)
    {
        MyData d=new Mydata( );
        Mythread1 t1=new
MyThread1(d);
        MyThread2 t2=new
MyThread(d);
        t1.start( );
```

- the classes Mythread1 and Mythread2 access the data from the display class which is the shared object.
- the synchronize method is used to execute the classes separately in the above example.
- synchronize method prints the data separately from both thread in other words it prevents the data to be overlapped.
- the for loop inside the synchronize prints one method at a time.
- The other method for synchronization is to write the synchronize at the signature method of the display class so the whole method is being synchronized.

➔ **Inter-thread communication:**

- it is the communication between the synchronized threads.
- it is the communication between a single producer thread and a consumer thread.
- the inter-thread communication refers to the synchronization between the producer thread and the consumer thread to access the write and read method simultaneously.
- to achieve the communication the flag=t and flag=f, are used.

➔ **Race condition:**

- here there is a single producer thread and multiple consumer threads.

- here all consumers do not execute at once they do in a round robin fashion.
- when the count is zero then it is the producers turn.
- and when the count is not zero then it is the consumers turn.
- since there are more than one consumer any of the consumer can access it.
- the notify method can open any thread her as they may not be in an order.
- The race condition states that
- If one thread is accessing the shared resource and all other are blocked then once the thread has finished.
- working it will inform the thread and any of the threads may access the object just like in a race.
- So, in the above the count method is used to control the race.
- the race condition can be avoided by the inter-thread condition.
- example program:

```
class MyData
{
    int value=0;
    boolean flag=true;
    synchronised void set(int v)
    {
        while(flag!=true)
        wait( );
        value=v;
        flag=false;
        notify( );
    }
    synchronized int get( )
    {
        int x=0;
        while(flag!=false)
        wait( )
        X=value;
        flag=true;
        notify;
        return x;
    }
}

class Producer extends Thread
{
    MyData d;
    Producer(Mydata dat){d=dat;}

    public void run( )
    {
        Int i=1;
        while(ture)
        {
            d.set(i);
            s.o.p("producer: "+i);
            i++;
        }
    }
}

class consumer extends thread
{
```

In the above example:

- the inter-thread communication part is achieve by the programmer itself.
- the producer will be writing in the shared object and the consumer will be reading from the shared object.
- the shared object, have three things one is the data or the value given, second is the set method to write the data and the third is the get method to read the data.
- as the set and get methods are synchronised only one thread is allowed to enter.
- If the flag is true then it is producers turn and if the flag is false then it is the consumers turn
- wait( ) and notify( ) is to bring in the communication between the threads.