Unit - IV

Multi-Threading

Uni Processing

- In the early days of computer only one program will occupy the memory. The second program must be in waiting.
- The second program will be entered whenever first program completed and released the resources voluntarily.
- In this procedure the main disadvantage is most of the times the CPU is kept as ideal. so that CPU resources are getting wasted.

Multi Programming

- Multi Programming means multiple programs are running at the same time on a single processor.
- So the advantage is the CPU can be utilized more efficiently than uni processing.

Multi Processing

- In this procedure more than one processor will execute the programs parallel.
- Because of two processors multiple (or) parallel execution is possible.

Multi Tasking

- Multi Tasking means multiple parts of the program is getting started at the same time.
- This multi tasking is possible on single processor and multiple processors also.
- In multi tasking for each and every process will be allocated a time period.
- Multi Tasking=Multi Programming + Time Sharing.

Multi Tasking

- There are two distinct types of multi tasking

  1) Process-based

  2) Thread-based

- A process is a program that is executing.
- Process-based multi tasking allows computer to run two or more programs concurrently.

  E.g., Process-based multitasking allows computer to run the java compiler at the same time we are using a text editor.

Thread-based Multi Tasking

- In thread-based multi tasking environment a single program can perform two or more tasks simultaneously.

E.g., A text Editor can format text at the same time it can perform printing.

Thread

- A thread is nothing but separate part of execution.
- Whenever thread is created it will not occupy any separate memory. But it will share the memory which is already allocated to a program.

Process-based vs Thread-based

- Processes are heavy weight tasks that require their own separate address spaces.
- Interprocess communication is expensive and limited.
- Context switching from one process to another is also costly.

Process-based vs Thread-based

- Threads are light weight process.

  They share the same address space and cooperatively share the same heavy weight process.

- Inter thread communication is inexpensive.

- Context switching from one thread to the next is low cost.

Single-thread System

- Single-thread System use an approach called an event loop with pooling.
- A Single thread of control runs in an infinite loop, pooling a single event queue to decide what to do next.
- In a single-threaded environment, when a thread blocks because it is waiting for some resources the entire program stops running.

Multi Thread

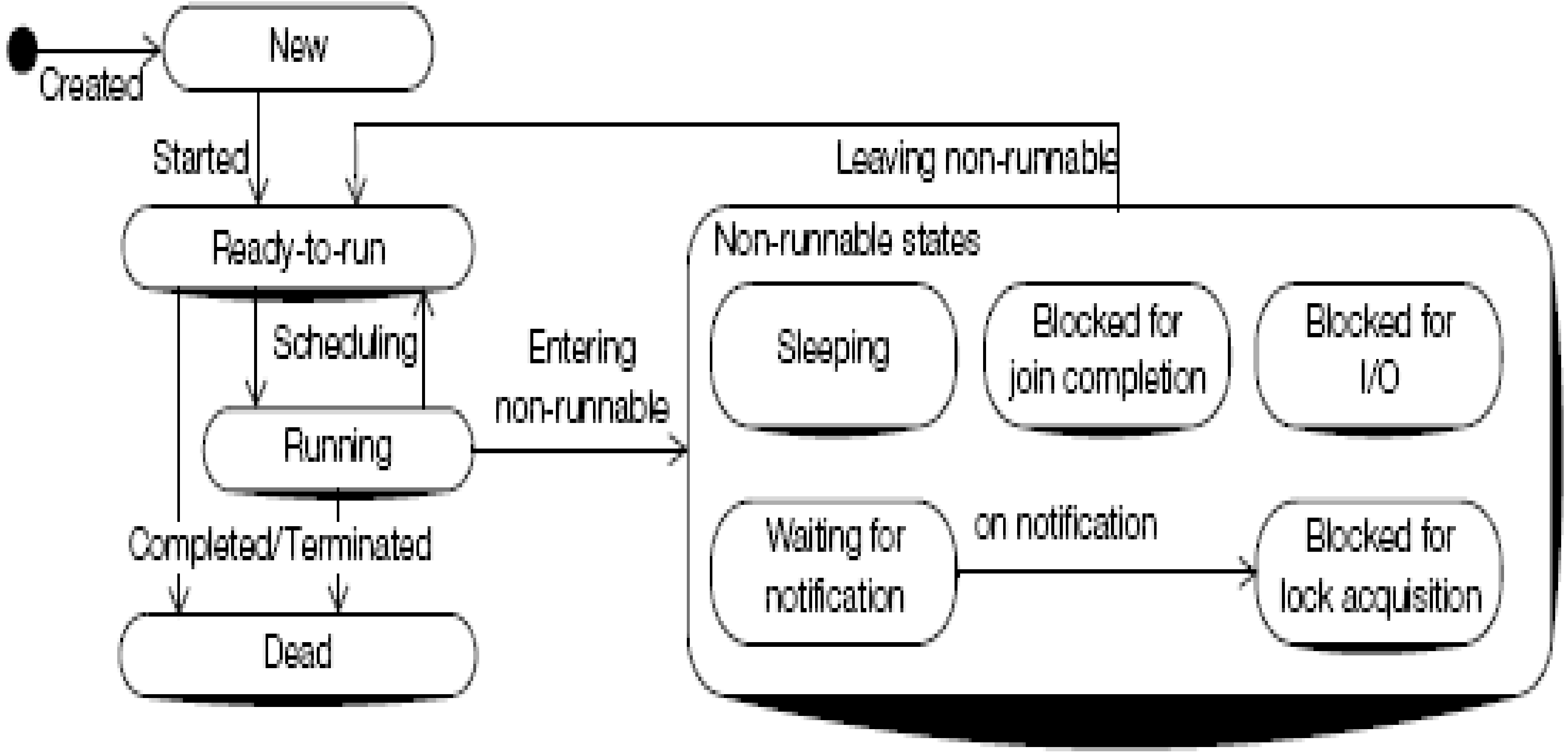
- Multithreading enables us to write very efficient programs that make maximum use of CPU. Because ideal time can be kept to a minimum.
- Java supports multi threading.
- The advantage of multi threading is that the main loop, pooling mechanism is eliminated.
- One thread can pause with out stopping other parts of the program.

Thread Life Cycle

- Thread can exist in several states:

1. New Born
2. Ready State
3. Running State
4. Blocked/Waiting State
5. Dead State

Thread States / Life Cycle of Thread



Thread States…

Constant in the `Thread.State` enum type	State in Figure	Description of the thread
`NEW`	*New*	Created but not yet started.
`RUNNABLE`	*Runnable*	Executing in the JVM.
`BLOCKED`	*Blocked for lock acquisition*	Blocked while waiting for a lock.
`WAITING`	*Waiting for notify,* *Blocked for join completion*	Waiting indefinitely for another thread to perform a particular action.
`TIMED_WAITING`	*Sleeping,* *Waiting for notify,* *Blocked for join completion*	Waiting for another thread to perform an action for up to a specified time.
`TERMINATED`	*Dead*	Completed execution.

Thread Life Cycle

- When a thread is dead, the same thread can’t able to restart it is possible to start some other new thread.
- Whenever thread is dead, it is handled by **ThreadDeath** class in Error super class.
- ThreadDeath class will be invoked whenever the stop() method is called.

Thread Creation

- Implementing threads is achieved in one of two ways:
  - implementing the j**ava.lang.Runnable** interface
  - extending the **java.lang.Thread** class
- The Thread class defines several methods that help to manage threads:

Method	Meaning
getName	Obtain a thread’s name.
getPriority	Obtain a thread’s priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Extending Thread 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.
- By using start() the control passes a request to processor for allocating resources.

Extending Thread Class

- After allocating the resources the last instruction is calling the run() method.
- start() implicitly calls the run() method.

Create a 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("Exiting child thread.");
}}
```

Create a thread by extending Thread…

```
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 exiting.");
}
}
```

Thread Priorities

- A thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

- Thread class supporting different priorities:

  1) MIN_PRIORITY

  2) NORM_PRIORITY

  3) MAX_PRIORITY

- Whenever a thread is created by default it is containing NORM_PRIORITY whose value is 5.

Thread Priorities

- **MIN_PRIORITY**

  The minimum priority that a thread can have

  public static final int **MIN_PRIORITY**

- **NORM_PRIORITY**

  The default priority that is assigned to a thread.

  public static final int **NORM_PRIORITY**

- **MAX_PRIORITY**

  The maximum priority that a thread can have

  public static final int **MAX_PRIORITY**

Thread Class Constructors

- **Thread():**

  Allocates a new Thread Object.

  E.g.,

  ```
  Thread t=new Thread();
  ```

Constructors

- **Thread(Runnable target):**

Allocates a new Thread object.

E.g.,

```
SecondThread st= new SecondThread();
Thread t =new Thread(st);
```

Constructors

- **Thread(Runnable target, String name):**

  Allocates a new Thread object.

E.g.,

```
SecondThread st=new SecondThread();
      Thread t= new Thread(st, ”Secondchild”);
```

Constructors

- **Thread(String name):**

  Allocates a new thread.

E.g.,

```
Thread t= new Thread( “FirstChild”);
```

Constructors

- **Thread(ThreadGroup group, String name)**:

  Allocates a new Thread object.

  Thread(group, null, name)

E.g.,

```
ThreadGroup tg=new ThreadGroup("Image Group");
        Thread t1= new Thread(tg,"frist child");
         Thread t2= new Thread(tg,"second child");
```

Constructors

- **Thread(ThreadGroup group, Runnable target)**:

  Allocates a new thread object.

  group - the thread group.

  target - the object whose run method is called.

Constructors

- E.g.,

```
ThreadGroup tg=new ThreadGroup(“Image Group”);

       SecondThread st= new SecondThread();
        Thread t1= new Thread(tg, st);
        Thread t2= new Thread(tg, st);
```

Constructors

- **Thread(ThreadGroup group, Runnable target,**

  String name):

  Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.

Constructors

- E.g.,

```
ThreadGroup tg= new ThreadGroup(“Image Group”);
SecondThread st= new SecondThread();
Thread t1= new Thread(tg, st, “First child”);
Thread t2= new Thread(tg, st, “Second child”);
```

Constructors

- **Thread(ThreadGroup group, Runnable target,**

  String name, long StackSize):

- Allocates a new Thread object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, and has the specified *stack size*.

Constructors

- E.g.,

```
ThreadGroup tg=new ThreadGroup(“Image Group”);

 SecondThread st= new SecondThread();

 Thread t1= new Thread(tg,st,”first child”,10);

  Thread t2= new Thread(tg,st,”second child”,10);
```

Thread Methods

- **currentThread()**:

  Returns a reference to the currently executing thread object.

```
public static Thread currentThread()
```

Thread Methods

- **setName**:

  Changes the name of this thread to be equal to the argument name.

public final void setName(String name)

Thread methods

- **getName():**

  Returns this thread’s name.

  public final String getName()

- **getPriority():**

  Returns this thread’s priority.

  public final int getPriority()

Thread Methods

- **getThreadGroup():**

  Returns the thread group to which this thread belongs. This method returns null if this thread has died.

  public final ThreadGroup getThreadGroup()

Thread Methods

- **setPriority():**

  Changes the priority of this thread.

  public final void setPriority(int newPriority)

❖ This method throws

1)IllegalAccessException: If the priority is not in the range MIN_PRIORITY to MAX_PRIORITY.

2)SecurityException**:** if the current thread cannot modify this thread.

Thread Methods

- E.g.,

```
ThreadTest t= new ThreadTest();
 t.setPriority(Thread.NORM_PRIORITY+3);
System.out.println(t.getPriority());
```

Thread Methods

- **sleep()**:
- Causes the currently executing thread to sleep for the specified number of milliseconds.

  public static void sleep(long millis) throws InterruptedException

  - InterruptedException: if another thread has interrupted the current thread.

Thread methods

```
public static void sleep(long millis, int nanos) throws  InterruptedException
```

- Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

  - millis - the length of time to sleep in milliseconds.
  - nanos - 0-999999 additional nanoseconds to sleep.

Thread Methods

- **interrupt():**

  Interrupts this thread.

  public void interrupt()

SecurityException: if the current thread cannot modify this thread

Thread methods

- **interrupted():**

  Tests whether the current thread has been interrupted.

public static boolean interrupted()

Thread Methods

- **isInterrupted()**:

  Tests whether this thread has been interrupted.

```
public boolean isInterrupted()
```

Thread Methods

- **isAlive():**

  Tests if this thread is alive. A thread is alive if it has been started and has not yet died.

Returns true if this thread is alive; false otherwise.

public final boolean **isAlive**()

Thread Methods

- **join():**

  Waits for this thread to die. This method waits until the thread on which it is called terminates.

  public final void **join**() throws InterruptedException

Example of isAlive() and join()

```
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.");
        } }
```

Example of isAlive() and join()...

```
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 exiting.");
} }
```

Thread Methods

- **yield():**

  Causes the currently executing thread object to temporarily pause and allow other threads to execute.

  public static void **yield**()

suspend(), resume() and stop()

- **suspend( ) and resume( ),** which are methods defined by Thread, to pause and restart the execution of a thread.
  - final void suspend( )
  - final void resume( )
- The Thread class also defines a method called stop( ) that stops a thread. Its signature is shown here:
  - final void stop( )

Example of suspend() and resume()

```
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 = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
}
} catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
}
        System.out.println(name + " exiting.");
} }
```

Example of suspend() and resume()...

```
class SuspendResume {
public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
try {
        Thread.sleep(1000);
        ob1.t.suspend();
        System.out.println("Suspending thread One");
        Thread.sleep(1000);
        ob1.t.resume();
        System.out.println("Resuming thread One");
        ob2.t.suspend();
        System.out.println("Suspending thread Two");
        Thread.sleep(1000);
        ob2.t.resume();
        System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");}
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
        ob1.t.join();
        ob2.t.join();
} catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
}
        System.out.println("Main thread exiting."); }}
```

Synchronization

- When two or more threads need access to a shared resource it is called as race condition.
- To avoid this race condition we need to use a mechanism called as *synchronization.*
- A monitor is an object that is used as a mutually exclusive lock, or mutex.
- Only one thread can own a monitor at a given time.
- When a thread enters in to the monitor all other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Synchronization…

- Synchronization can be achieved in two ways:

  1) Method level synchronization

  2) Block level synchronization

❖ In method level synchronization the total data of method is sensitive. Where as in block level synchronization part of the method is sensitive.

Example: Without synchronized

```
class Callme {
void call(String msg) {
        System.out.print("[" + msg);
try {
        Thread.sleep(1000);
} catch(InterruptedException e) {
        System.out.println("Interrupted");
}
System.out.println("]");
} }
class Caller implements Runnable {
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
}
public void run() {
        target.call(msg);
}}
```

Example: Without synchronized…

```
class Synch {
public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}  }}
```

Here is the output produced by this program:

```
[Hello[Synchronized[World]
]
]
```

Example: With synchronized

```
class Callme {
synchronized void call(String msg) {
        System.out.print("[" + msg);
try {
        Thread.sleep(1000);
} catch(InterruptedException e) {
        System.out.println("Interrupted");
}
System.out.println("]");
} }
class Caller implements Runnable {
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
}
public void run() {
        target.call(msg);
}}
```

Example: With synchronized…

```
class Synch {
public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}  } }
```

Here is the output produced by this program:

```
[Hello]
[Synchronized]
[World]
```

Interthread Communication

- **wait( ) tells the calling thread to give up the monitor and go to sleep until some** other thread enters the same monitor and calls **notify( ).**
- **notify( ) wakes up the first thread that called wait( ) on the same object.**
- **notifyAll( ) wakes up all the threads that called wait( ) on the same object.** The highest priority thread will run first.
- Example
  - Producer and Consumer Problem

Assignment

1. What is Thread?
2. How can you create thread in Java?
3. Explain the life cycle of Thread?
4. What is monitor?
5. What is the use of sychronized method and sychronized block?
6. What do mean by inter thread communication? Write the java program for producer and consumer problem?
7. How can you set thread priority?
8. Explain the isAlive(), join(), suspend(), and resume() with example.