Java Programming

Lecture-13

Multithreaded programming

- **Multithreading in Java** is a process of executing multiple threads simultaneously.

- A thread is a lightweight sub-process, the smallest unit of processing.

- Multiprocessing and multithreading, both are used to achieve multitasking.

- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

- Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

- Multitasking is a process of executing multiple tasks simultaneously.
- We use multitasking to utilize the CPU.
- Multitasking can be achieved in two ways:
  - Process-based Multitasking (Multiprocessing)
  - Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

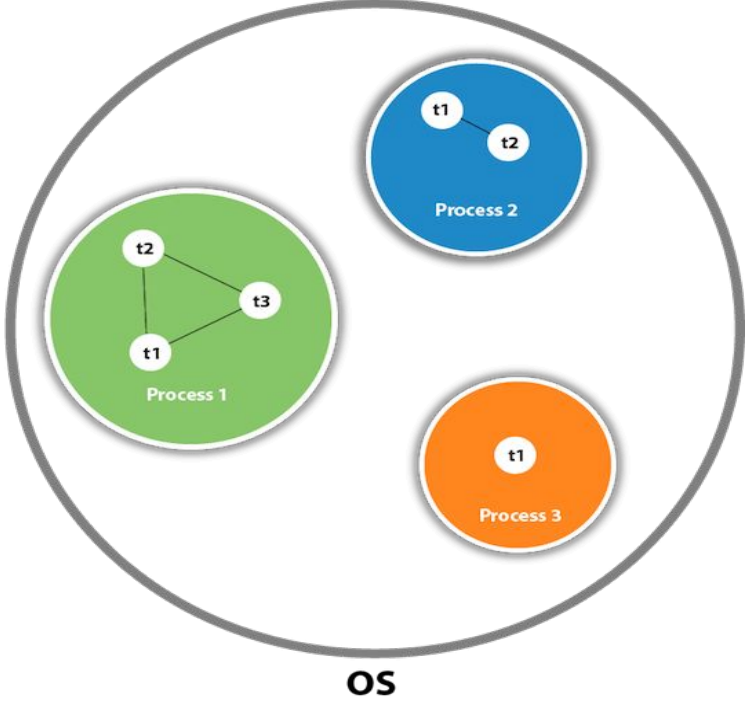
- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

What is 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.



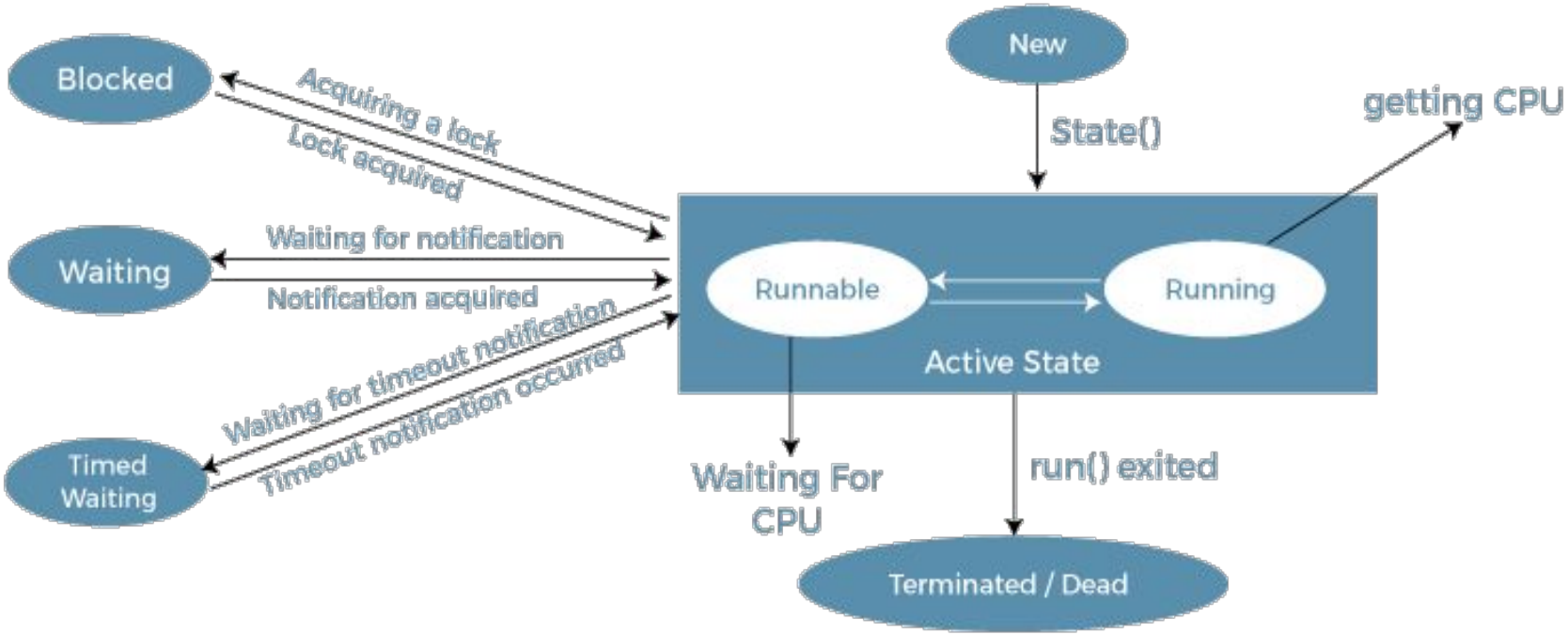
- As shown in the above figure, a thread is executed inside the process.
- There is context-switching between the threads.
- There can be multiple processes inside the OS, and one process can have multiple threads.

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active/Runnable
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

Thread States



Life Cycle of a Thread

Explanation of Different Thread States

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

Runnable: A thread, that is ready to run is moved to the runnable state.

- In the runnable state, the thread may be running or may be ready to run at any given instant of time.

- It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.

- A program implementing multithreading acquires a fixed slice of time to each individual thread.

- Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time.

- Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

Running: When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

- When the main thread invokes the **join()** method then, the main thread is in the waiting state.
- The main thread then waits for the child threads to complete their tasks.
- When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.
- If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

Timed Waiting: Sometimes, *waiting for* leads to starvation.

- For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section.
- In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation.
- To avoid such scenario, a timed waiting state is given to thread B.
- Thus, thread lies in the waiting state for a specific span of time, and not forever.
- A real example of timed waiting is when we invoke the sleep() method on a specific thread.
- The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The Thread Class and the Runnable Interface

- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

- The **Thread** class defines several methods that help 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.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.

The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object.

To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.

```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

Creating a Thread

We can create a thread by instantiating an object of type **Thread**.

Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.

- You 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 like this:

public void run( )

- Inside **run( )**, you will define the code that constitutes the new thread.

- It is important to understand that **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 your program.

- This thread will end when **run( )** returns.

Here is an example that creates a new thread and starts it running:

```
// 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("Exiting child thread.");
  }
```

```
}

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 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 you 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 in single- core systems, until their loops finish.

Output:

```
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
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread

- The second 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.

```
// 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("Exiting child 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.");
  }
}
```

Creating Multiple 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.");
  }
}
```

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

Output:

```
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.
```

Using isAlive( ) and join( )

- We want the main thread to finish last, 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.

- Another way to ensure this is to use **isAlive()** and **join()** methods.

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

- **join()** method waits until the thread on which it is called terminates.

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

- Over a given period of time, higher-priority threads get more CPU time than lower-priority threads.

- A higher-priority thread can also preempt a lower-priority one.

- threads of equal priority should get equal access to the CPU.

- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

General form:

final void setPriority(int level)

- Here, *level* specifies the new priority setting for the calling thread.
- The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**.
- Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5.
- These priorities are defined as **static final** variables within **Thread**.
- You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

final int getPriority( )

Synchronization

- When two or more threads need access to a shared resource, they need some way 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***.

- Key to synchronization is the concept of the monitor.

- A ***monitor*** is an object that is used as a mutually exclusive lock.

- Only one thread can ***own*** a monitor at a given time.

Synchronization

- When a thread acquires a **lock**, it is said to have *entered* the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread ***exits*** the monitor.
- These other threads are said to be ***waiting*** for the monitor.
- A thread that owns a monitor can **reenter** the same monitor if it so desires.

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.

- To enter an object's monitor, we call a method that has been modified with the **synchronized** keyword.

- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.

- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

The synchronized Statement

- To achieve synchronization, simply put calls to the methods defined by this class inside a **synchronized** block.

- This is the general form of the ***synchronized statement***:

  synchronized(objRef) ***{***
  // statements to be synchronized

  }

- Here, *objRef* is a reference to the object being synchronized.

- A synchronized block ensures that a call to a synchronized method that is a member of *objRef*'s class occurs only after the current thread has successfully entered *objRef*'s monitor.

Another method: **Using Synchronized Methods**

```
class Callme {
  synchronized void call(String msg) {
  ...
```

Interthread Communication

To avoid polling, Java includes an interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods.

These methods are implemented as **final** methods in **Object**, so all classes have them.

All three methods can be called only from within a **synchronized** context.

- **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( ) or notifyAll( ).

- **notify( )** wakes up a thread that called wait( ) on the same object.

- **notifyAll( )** wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

Thank you