MULTITHREADED PROGRAMMING

Ebey S.Raj

Multithreaded Programming

- Java provides built-in support for multithreaded programming.
- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution (multiple flow of control).
- A program that contains multiple flow of controls is called Multithreaded programming.
- Multithreading is a specialized form of Multitasking.

Multitasking

- Two distinct types of Multitasking
  - Process based Multitasking
    - Process-based multitasking allows your computer to run two or more programs concurrently.
    - A program is the smallest unit of dispatchable code.
    - Eg: run the Java compiler at the same time that you are using a text editor or visiting a web site.
  - Thread based Multitasking
    - A single program can perform two or more tasks simultaneously.
    - Thread is the smallest unit of dispatchable code.
    - Eg: a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Process based Multitasking Vs Thread based Multitasking

- Multitasking processes requires more overhead.
- Processes are heavyweight tasks that require their own separate address spaces.
- Interprocess communication is expensive and limited.
- Context switching from one process to another is also costly.

- Multitasking threads require less overhead.
- Threads, are lighter weight and they share the same address space and cooperatively share the same heavyweight process.
- Interthread communication is inexpensive
- Context switching from one thread to the next is lower in cost

Multithreading

- Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system.
- Keeps idle time to a minimum.

Java Thread Model

- When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.
- In a single-core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Two or more threads do not actually run at the same time, but idle CPU time is utilized.
- In multi-core systems, it is possible for two or more threads to actually execute simultaneously. This improves program efficiency and increase the speed of certain operations.

Thread Priorities

- Thread priorities are integers that specify the relative priority of one thread to another.
- Thread's priority is used to decide when to switch from one running thread to the next. This is called a **context switch**.
- Rules that determine when a context switch takes place :
  - **A thread can voluntarily relinquish control.**
    - Done by explicitly sleeping or blocking on pending I/O.
    - In this case, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
  - **A thread can be preempted by a higher-priority thread.**
    - A lower-priority thread that is running is simply preempted by a higher-priority thread.
    - This is called preemptive multitasking.

Thread Priorities

- If two threads are of equal priority
  - For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion.
  - For other types of operating systems, threads of equal priority must voluntarily yield control to their peers.

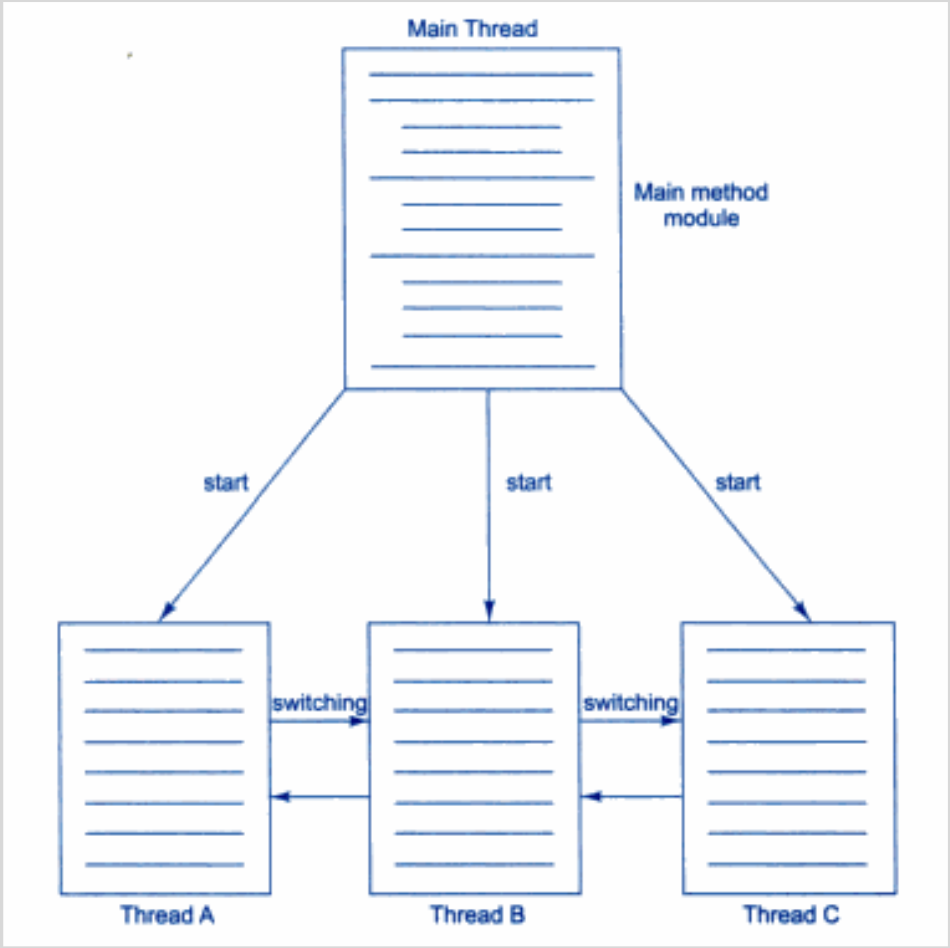
Thread class and Runnable Interface

- To create a new thread, our program will either extend Thread or implement the Runnable interface.
- The Thread class defines several methods that help to manage threads. Several of those are shown here:

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 the program.
- Main thread is actually the main method module.
- It 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.



The Main Thread

- We can obtain a reference to a thread by calling the method **currentThread( ),** which is a public static member of Thread.
- General form is shown here:

static Thread currentThread( )

This method returns a reference to the thread in which it is called.

Example: CurrentThreadDemo.java

A thread group is a data structure that controls the state of a collection of threads as a whole.

Used functions in the example

- The sleep( ) method causes the thread to suspend execution for the specified period of milliseconds.

  General form: **static void sleep(long milliseconds) throws InterruptedException**

  Second form:

  static void sleep(long ms, int nanoseconds) throws InterruptedException

- We can set the name of a thread by using setName( ).

  General form: **final void setName(String threadName)**

- We can obtain the name of a thread by calling getName( )

  General form: **final String getName( )**

Creating a Thread

- Java defines two ways for creating a thread:
  - By implementing the **Runnable interface.**
  - By extending the **Thread class.**

Creating a Thread

- In both cases. threads are implemented by creating a class that contain a method called **run()**.
  - run() makes up the entire body of a thread and is the only method in which the thread's behavior is implemented.

```
public void run( ){
        ……..
        ……..          //statements for implementing thread
        ……..
        ……..
}
```

- Initially create a thread object and initiate it with the help of another thread method called **start().**

Creating a Thread: By implementing Runnable Interface

- The Runnable interface declares the run() method that is required for implementing threads in our programs.
- To implement Runnable, a class need only implement a single method called run( ).
- Steps
  - Declare the class as implementing the Runnable Interface.
  - Implement the run() method that is responsible for executing the sequence of code that the thread will execute.
  - Create a thread by defining an object that is instantiated from this “runnable” class as the target of the thread.
  - Call the thread’s start method to run the thread.

Creating a Thread: By implementing Runnable Interface

- Declare the class as implementing the Runnable Interface.

```
class MyThread implements Runnable{
        ……….
        ……….
}
```

Creating a Thread: By implementing Runnable Interface

- Implement the run() method that is responsible for executing the sequence of code that the thread will execute.

```
public void run(){
        ……..                    //Thread code here
        ……..
}
```

Creating a Thread: By implementing Runnable Interface

- Create a thread by defining an object that is instantiated from this “runnable” class as the target of the thread.
  - Thread defines several constructors. One of them is shown here:

    Thread(Runnable threadOb, String threadName)

    threadOb is an instance of a class that implements the Runnable interface.

    threadName is the name of the new thread.

```
MyThread mythread=new MyThread();
Thread newthread=new Thread(mythread,"Child Thread");
```

Creating a Thread: By implementing Runnable Interface

- The new thread starts running only when you call its start( ) method. In essence, start( ) executes a call to run( ).
- General form of start( ) method is shown here: **void start( )**

```
newthread.start();
```

Example: ThreadDemo.java

Example: ThreadDemo1.java

Creating a Thread: By extending Thread class

- Steps
  - Declare the class as extending the Thread class
  - Implement the **run()** method.
  - Create a thread object and call the **start()** method to initiate the thread execution.

Creating a Thread: By extending Thread class

- Declare the class as extending the Thread class

```
class MyThread extends Thread{
        ……..
        ……..
}
```

Creating a Thread: By extending Thread class

- Implement the **run()** method.

```
public void run(){
        ……..                    //Thread code here
        ……..
}
```

Creating a Thread: By extending Thread class

- Create a thread object and call the **start()** method to initiate the thread execution.

```
MyThread mythread=new MyThread();
mythread.start();
```

Notice the call to **super( )** inside **MyThread**.
This invokes the following form of the **Thread** constructor:

public Thread(String *threadName*)

Here, *threadName* specifies the name of the thread.

Example: ExtendThread.java

Example: ExtendThread1.java

Blocking a thread

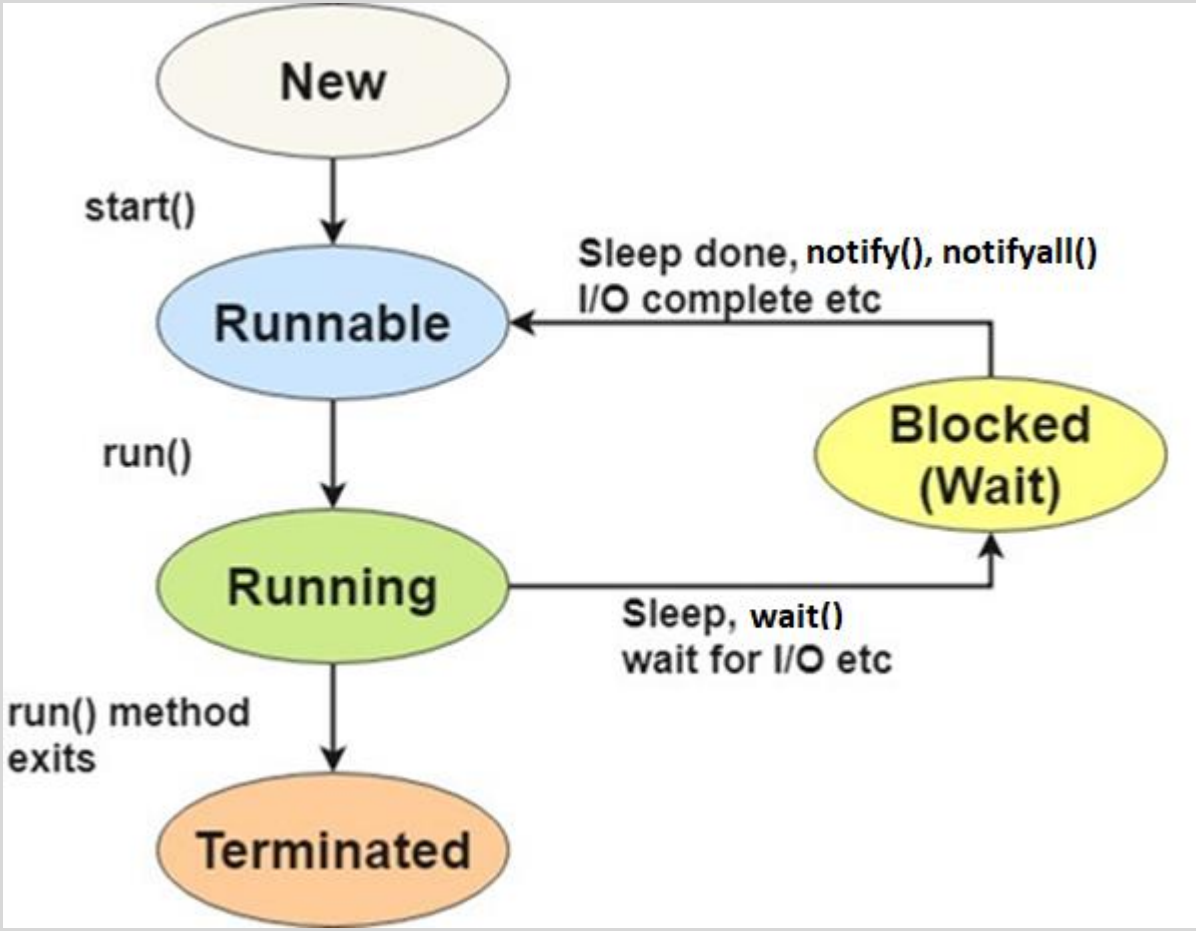
- Blocking a thread
  - A thread can be temporarily suspended or blocked from entering into a runnable and subsequently running state by using either of the following thread methods

    sleep(): Suspended for a specified time; Return to runnable state when the specified time is elapsed.

    wait(): Blocked until certain condition occurs; Return to runnable state when the **notify()** is invoked.

Life Cycle of a thread

- During the life time of a thread, there are many states they can enter.
  - New
  - Runnable
  - Running
  - Blocked
  - Terminated



States of a Thread

- New State
  - When we create a thread object, the thread is born and is said to be in New state.
  - We can schedule it for running using start() method.
- Runnable State
  - The state that the thread is ready for execution and is waiting for the availability of the processor.
    - The thread has joined in the queue of threads that are waiting for execution.
  - If all the threads have equal priority, they are given time slots in round robin fashion.

States of a thread

- Running state
  - Processor has given its time to the thread for its execution.
  - Thread runs until it relinquishes control on its own (sleep() or wait()) or it is preempted by a higher priority thread.
- Blocked state
  - When the thread is prevented from entering into the runnable state and subsequently the running state (**sleep() or wait()**).
- Dead state(Terminated State)
  - When the thread has completed executing its **run()** method.

Using isAlive() and join()

- Two ways exist to determine whether a thread has finished.
  - isAlive( ) method
    - This method is defined by Thread, and its general form is shown here:

      final boolean isAlive( )

      isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.
  - join() method
    - join() method waits until the thread on which it is called terminates.
    - General form: **final void join( ) throws InterruptedException**

      Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Creating Multiple threads

- Multiple Threads with join() and isAlive()

Example: MultipleThreads.java

Example: JoinDemo.java

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- A higher-priority thread can also preempt a lower-priority one.
- We can obtain the current priority value by calling the **getPriority( )** method of Thread.
- General form: **final int getPriority( )**

Thread Priorities

- To set a thread's priority, use the **setPriority( )** method of Thread.
- General form: **final void setPriority(int level)**

  level specifies the new priority value; Value of level must be within the range **MIN_PRIORITY(=1)** and **MAX_PRIORITY(=10).**
- To return a thread to default priority, specify **NORM_PRIORITY(=5)**
- These priorities are defined as static final variables within Thread.

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.**

Synchronization: Monitor in Java

- 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.
- 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.

Synchronization

- We can synchronize our code in either of two ways.
  - Using synchronized methods.
  - Using synchronized statement.

Synchronization: Using synchronized method

- In Java, all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- 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.

Synchronization: Using synchronized method

- Example of Non-synchronized Methods: NSynch.java

In this example, all three threads are calling the same method, on the same object, at the same time. This is known as a ***race condition***, because the three threads are racing each other to complete the method.

Synchronization: Using synchronized method

- To serialize access to **call( ),** we need to modify the **definition of call()** by using the **synchronized** keyword.
- Example of Synchronized Methods (Runnable Interface): Synch0.java
- Example of Synchronized Methods (Thread class): Synch1.java

Once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance.

Synchronization: Using synchronized statement

- Imagine that the class does not use synchronized methods and you want to synchronize access to its objects. [And the class was not created by you].
- How can access to an object of this class be synchronized?

Simply put calls to the methods defined by this class inside a **synchronized block.**

Synchronization: Using synchronized statement

- General form:

synchronized(*object*) {

…… // statements to be synchronized

}

object is a reference to the object being synchronized.

Example(Runnable Interface): Synch2.java

Example(Thread class): Synch3.java

Interthread Communication

- Multithreading replaces Event Loop programming with Polling, which wastes CPU time.
  - Polling is usually implemented by a loop that is used to check some condition repeatedly.
  - Producer Consumer Problem.
- To avoid polling, Java includes an elegant 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.

```
final void wait( ) throws InterruptedException
final void notify( )
final void notify All( )
```

Interthread Communication

- Rules for using these methods:
  - **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 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.

Example: PC.java

Example1: PC1.java

Reading Assignment

- **Deadlock**
- **Suspending, Resuming, and Stopping Threads**

References

- Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.