Multithreaded Programming

Siddharth Shah

Outline - 1

- Concept
- Multiprocessing vs. Multithreading
- Why Multithreading?
- Life-Cycle of a Thread
- Thread States
- Thread Class and Runnable Interface
- The Main Thread
- Creating a Thread
- Creating Multiple Threads
- Using ***isAlive()*** and ***join()***

Outline - 2

- Thread Priorities
- Synchronization
  - Monitor
  - Synchronized Method
  - Synchronized Statement
- Interthread Communication
  - Deadlock
- Suspending, Resuming and Stopping Threads
- Using Multithreading

Concept

- 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.
- Thus, multithreading is a specialized form of multitasking.

Multiprocessing vs Multithreading - 1

- Two distinct types of multitasking: **process-based** and **thread-based**.
- Process-based multitasking is the feature that allows computer to run two or more programs concurrently.
- For example, processbased multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.
- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Multiprocessing vs Multithreading - 2

- In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code.
- This means that a single program can perform two or more tasks simultaneously.
- For instance, 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.
- Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multiprocessing vs Multithreading - 3

Multiprocessing	Multithreading
Processes are heavyweight tasks that require their own separate address spaces	Threads, on the other hand, are lighter weight, They share the same address space and cooperatively share the same heavyweight process
Interprocess communication is expensive and limited	Interprocess communication is inexpensive
Context switching from one process to another is also costly	Context switching from one thread to the next is lower in cost

- While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java's control. However, multithreaded multitasking is.

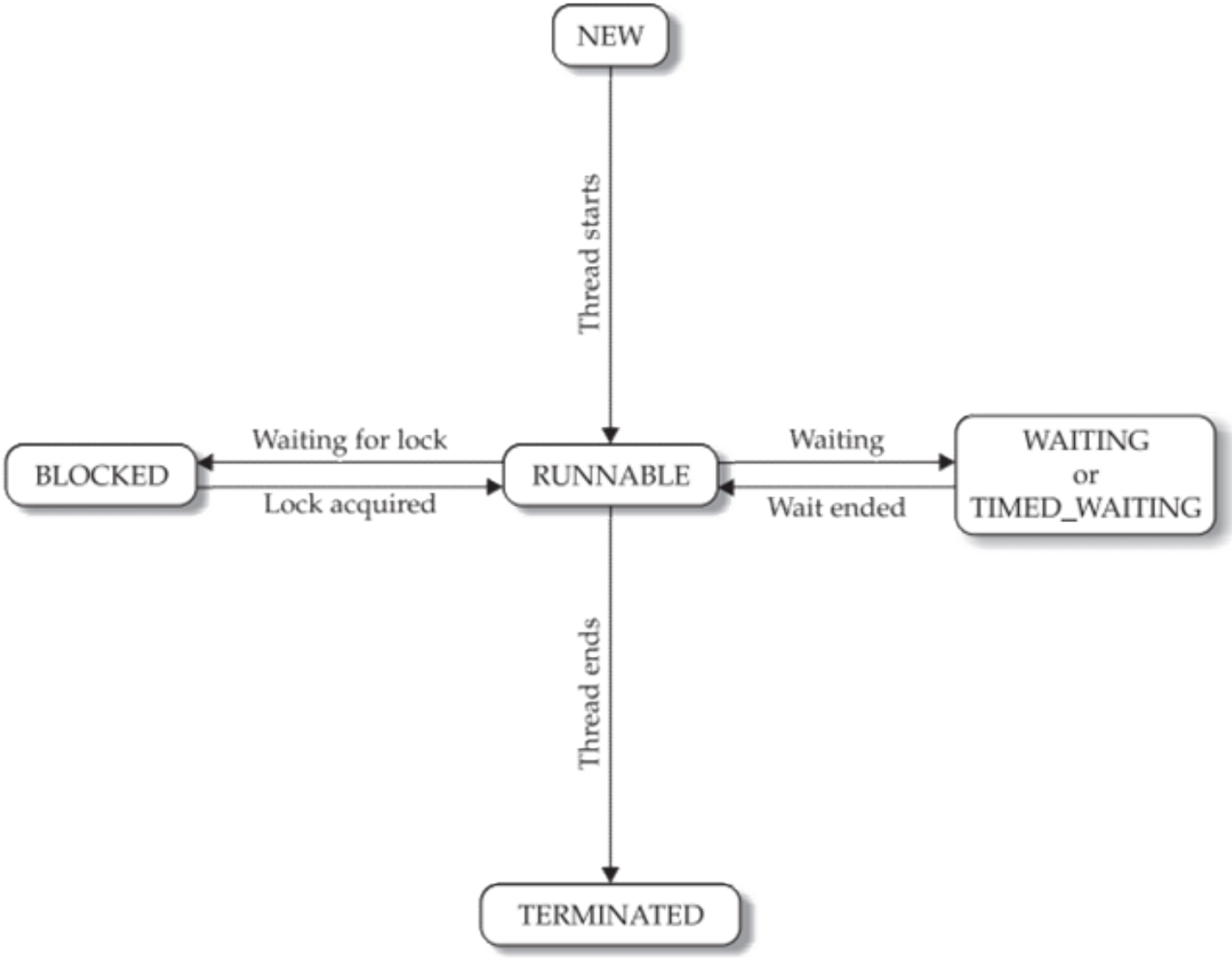
Why Multithreading ? - 1

- Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system.
- One important way multithreading achieves this is by keeping idle time to a minimum.
- This is especially important for the interactive, networked environment in which Java operates because idle time is common.

Why Multithreading ? - 2

- Examples of some tasks
  - Example 1: the transmission rate of data over a network is much slower than the rate at which the computer can process it.
  - Example 2: local file system resources are read and written at a much slower pace than they can be processed by the CPU.
  - Example 3: user input is much slower than the computer.
- In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though most of the time the program is idle, waiting for input.
- Multithreading helps you reduce this idle time because another thread can run when one is waiting.

Life-Cycle of a Thread



Thread States

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.

Thread Class & Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- **Thread** encapsulates a thread of execution.
- Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it.
- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

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:
  1. It is the thread from which other “child” threads will be spawned.
  2. 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.

Creating a Thread

- In the most general sense, you create a thread by instantiating an object of type Thread.
- Java defines two ways in which this can be accomplished:
  - Implement the Runnable interface.
  - Extend the Thread class, itself.

Implementing Runnable - 1

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- A thread can be constructed on any object that implements Runnable.
- To implement Runnable, a class needs to implement a single method called run( ), which is declared like this:
  - **public void run( )**

Implementing Runnable - 2

- Inside run( ), 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.

Implementing Runnable - 3

- After creating a class that implements Runnable, instantiate an object of type **Thread** from within that class.
- Though Thread class defines several constructors. The one that we will use is shown here:
  - **Thread(Runnable threadOb, String threadName)**
- In this constructor, **threadOb** is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin.
- The name of the new thread is specified by threadName.

Implementing Runnable - 4

- After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread.
- In essence, start( ) executes a call to run( ).
- The start( ) method is shown here:
  - **void start( )**

Implementing Runnable : Summary

- Create a class which implements Runnable interface.
- Implement run() method by writing a code that you want to execute in your thread.
- Instantiate an object of Thread inside your class by passing the reference to your class.
- Call start() method on the newly created Thread object which in turn will call run() method that you have implemented.

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.

Choosing an Approach

- Though there are 2 different ways to create a thread in Java, most programmers feel that creating a thread by implementing Runnable is better. The reasons for that are as follows:
  - Whether extending Thread class or implementing Runnable, in both cases we implements only run() method. Class should be extended if its behavior needs to be modified.
  - By implementing Runnable, your thread class does not need to inherit Thread, making it free to inherit a different class.

Using ***isAlive()*** and ***join()*** - 1

- As mentioned, often main thread is to finish last.
- In the preceding examples, 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.
- However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended?
- Fortunately, Thread provides a means by which you can answer this question.

Using ***isAlive()*** and ***join()*** - 2

- Two ways exist to determine whether a thread has finished.
- First, you can call isAlive( ) on the thread. This method is defined by Thread, and its general form is shown here:
  - **final boolean isAlive( )**
- The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

Using ***isAlive()*** and ***join()*** - 3

- While isAlive( ) is occasionally useful, the method that is more commonly used to wait for a thread to finish is called join( ), shown here:
  - **final void join( ) throws InterruptedException**
- This method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread joins it.
- Additional forms of **join( )** allow you to specify a maximum amount of time that program wants to wait for the specified thread to terminate.

Thread Priorities - 1

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

Thread Priorities - 2

- In theory, threads of equal priority should get equal access to the CPU.
- But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others.
- For safety, threads that share the same priority should yield control once in a while.
- This ensures that all threads have a chance to run under a nonpreemptive operating system.

Thread Priorities - 3

- In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O.
- When this happens, the blocked thread is suspended and other threads can run.
- But, if you want smooth multithreaded execution, you are better off not relying on this.
- Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run.

Thread Priorities - 4

- To set a thread's priority, use the setPriority( ) method, which is a member of Thread. The general form is:
  - **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**.
- Java provides unique, language-level support for it.
- Key to synchronization is the concept of the monitor.

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.
- You can synchronize your code in either of two ways:
  - Synchronized Method
  - Synchronized Statement

Synchronized Methods

- To achieve synchronization, thread has to enter into object's monitor.
- To enter an object's monitor, just 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

Synchronized Statements - 1

- While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- To understand why, consider the following.
- Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods.
- Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add synchronized to the appropriate methods within the class.
- How can access to an object of this class be synchronized?

Synchronized Statements - 2

- Fortunately, the solution to this problem is quite easy: You 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.

Interthread Communication - 1

- As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units.
- Threads also provide a secondary benefit: they do away with polling.
- Polling is usually implemented by a loop that is used to check some condition repeatedly.
- Once the condition is true, appropriate action is taken.
- This wastes CPU time.

Interthread Communication - 2

- For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it.
- To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

Interthread Communication - 3

- 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.

Interthread Communication - 4

- Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:
- **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( ).
  - Additional forms of wait( ) exist that allow you to specify a period of time to wait.
- **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.

Deadlock - 1

- A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.
- If the thread in X tries to call any synchronized method on Y, it will block as expected.
- However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Deadlock - 2

- Deadlock is a difficult error to debug for two reasons:
  - In general, it occurs only rarely, when the two threads time-slice in just the right way.
  - It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

Suspending, Resuming and Stopping Threads - 1

- Prior to Java 2, a program used **suspend( )**, **resume( )**, and **stop( )**, which are methods defined by Thread, to pause, restart, and stop the execution of a thread.
- Although these methods seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they are deprecated and should not be used.
- **suspend()** can sometimes cause serious system failures like deadlock.
- **resume(),** no resume without suspend()
- **stop()** can sometimes cause serious system failures like leave a data in corrupted state.

Suspending, Resuming and Stopping Threads - 2

- A thread must be designed so that the run( ) method periodically checks to determine whether that thread should suspend, resume, or stop its own execution.
- Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread.
- As long as this flag is set to “running,” the run( ) method must continue to let the thread execute.
- If this variable is set to “suspend,” the thread must pause.
- If it is set to “stop,” the thread must terminate.
- Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

Using Multithreading

- The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially. With the careful use of multithreading, you can create very efficient programs.
- However if you create too many threads, you can actually degrade the performance of your program rather than enhance it. As some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program.
- To create compute-intensive applications that can automatically scale to make use of the available processors in a multi-core system, consider using the new Fork/Join Framework.