Multithreaded Programming Part 1

Prof. Siddharth Shah
Department of Computer Engineering
Dharmsinh Desai University

Outline

- Concept
- Multiprocessing vs. Multithreading
- Why Multithreading?
- Life-Cycle of a Thread
- Thread States
- Thread Class and Runnable Interface
- The Main Thread

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.

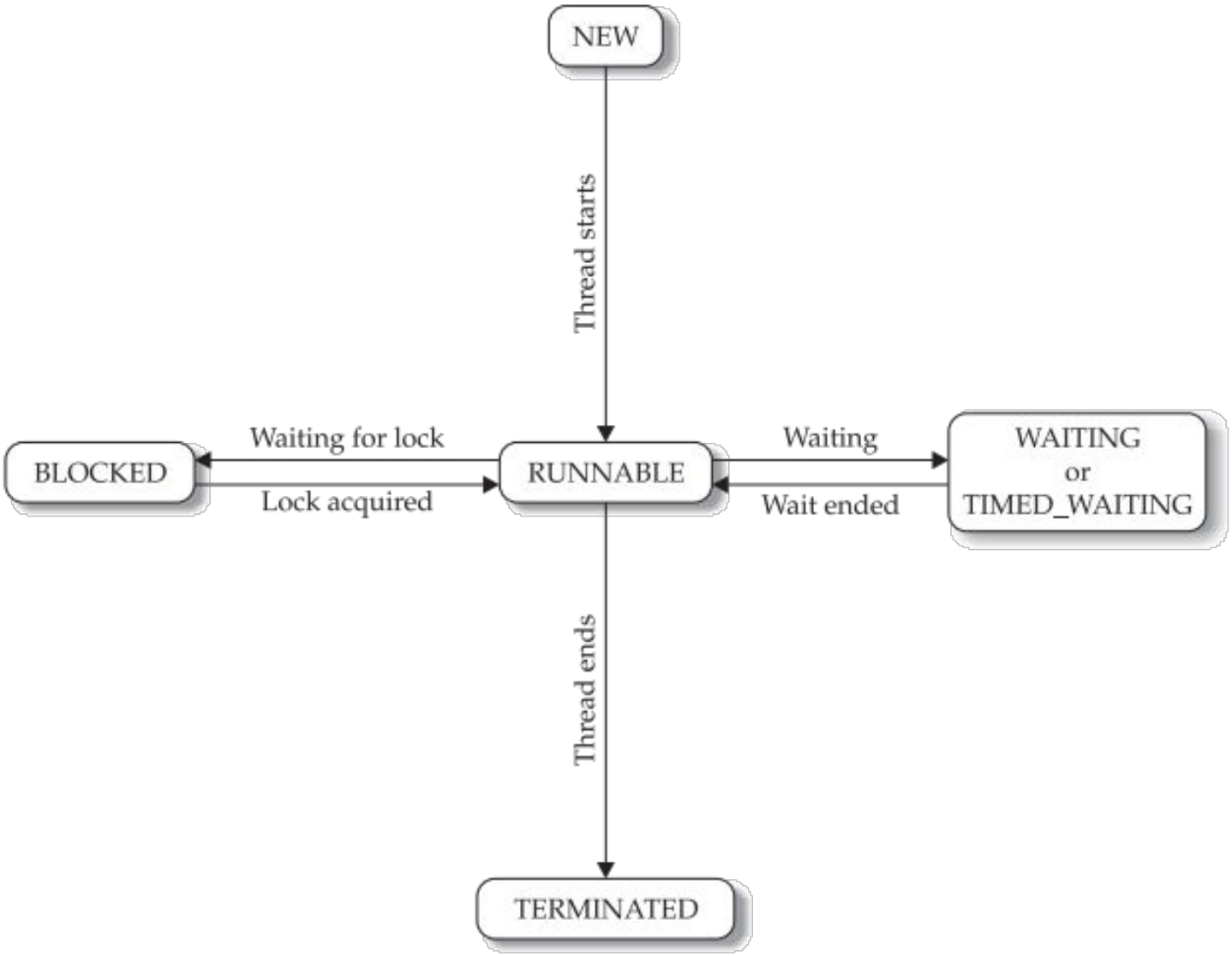
Why Multithreading ? - 1

- 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.