WOLDIA UNIVERSITY
INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTING

DEPARTMENT OF SOFTWARE ENGINEERING
ADVANCED PROGRAMMING

Chapter Four
Multithreading

by Demeke G.

AY- 2017

Outline

- **Introduction**
- **Thread vs. Process**
- **Life Cycle of a Thread**
- **Creating and Executing Thread**
- **Thread Synchronization**

Introduction

- A thread is a single sequence of executable code within a larger program
- Multithreading is a process of executing multiple threads simultaneously to maximize CPU utilization.
- Multithreading allows a program to perform multiple tasks concurrently, making it more efficient and responsive
- Operating systems on single-processor computers create the **illusion** of concurrent execution by rapidly switching between activities,
- but on such computers only a single instruction can execute at once
- Single-threaded program can handle one task at any time.

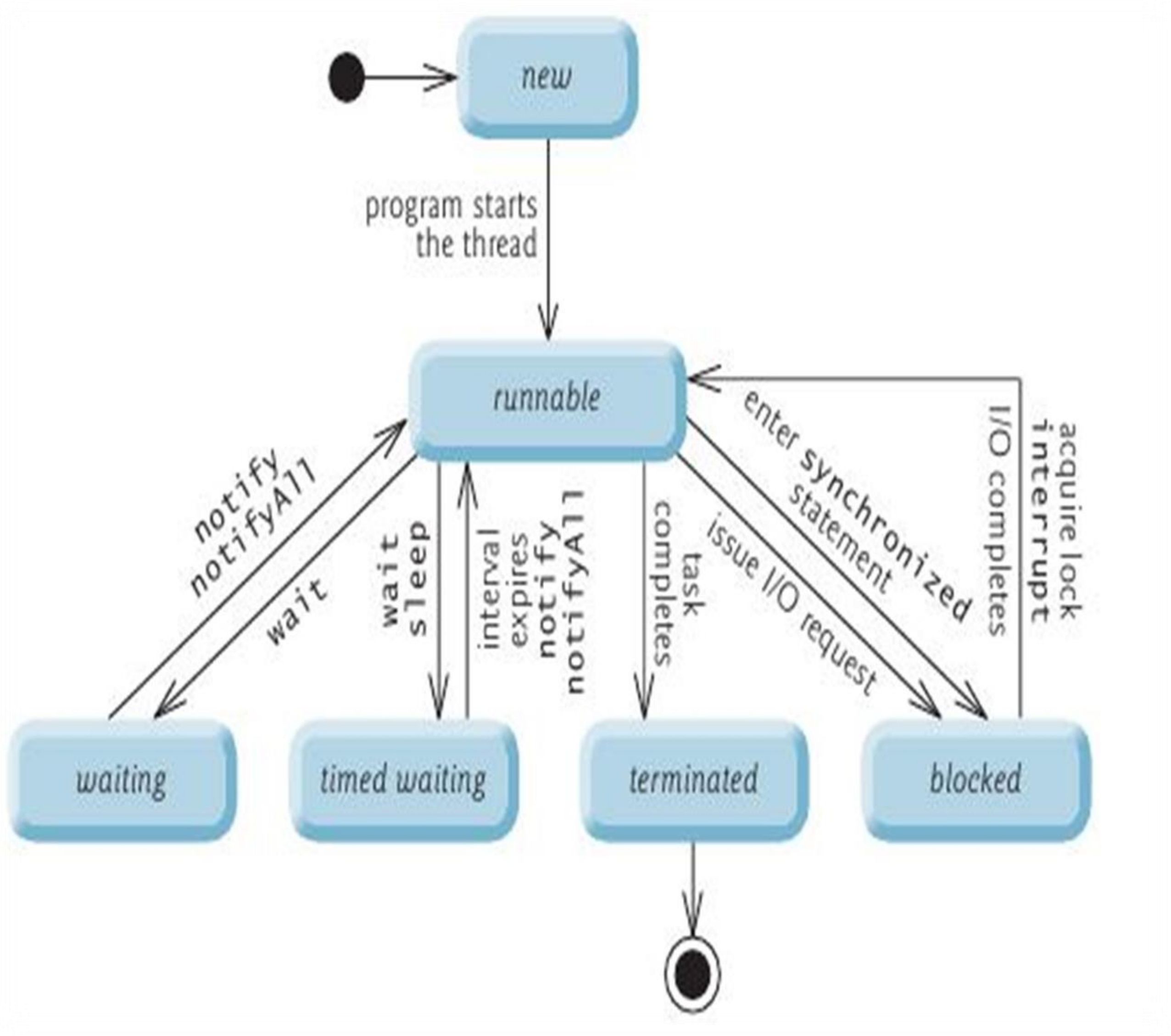
Threads vs. Processes

- Both ***threads*** and ***processes*** are methods of parallelizing an application.
- **Processes** are **independent** execution units that
  - ***contain their own state information,***
  - ***use their own address spaces, and***
  - ***only interact with each other via inter-process communication mechanisms (managed by the OS).***
- A single process might contains multiple threads.
- All threads within a process:
  - ***share the same state*** ,
  - ***same memory space***, and
  - can ***communicate with each other directly***, because they share the same variables.

Advantages of Multithreading

- Better resource utilization :
  - i.e. Utilize the idle time of the CPU
- Prioritize your work depending on priority
- Server can handle multiple clients simultaneously
- Allows performing I/O and CPU tasks concurrently.

Thread States: Life Cycle of a Thread



Threads life cycle

- ***New and Runnable States:*** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread
    - A thread in the ***runnable*** state is considered to be executing its task.
- ***Waiting State:*** Sometimes a runnable thread transitions to the waiting state while it waits for another thread to perform a task.
    - A waiting thread transitions back to the runnable state only when another thread notifies it to continue executing.

Cont..

- ***Timed Waiting State:*** A runnable thread can enter the timed waiting state for a specified interval of time.
  - It transitions back to the runnable state when that time interval expires .
  - Another way to place a thread in the timed waiting state is to put a runnable thread to ***sleep***.
- ***Blocked State: A*** runnable thread transitions to the blocked state when it attempts to perform a task that **cannot be completed immediately.**
  - it must temporarily wait until that task completes.
  - A blocked thread cannot use a processor.

Example

- when a thread issues an input/output request, the operating system **blocks** the thread until that I/O request completes.
- After I/O request completed the blocked thread transitions to the runnable state, so it can resume execution.

Terminated State

A runnable thread enters the terminated state when it successfully completes its task or due to an error

Creating a Thread

- There are two ways to create a thread.

1. Extend the **java.lang.Thread** class
2. Implement the **java.lang.Runnable** interface.

Thread Methods:

- start(): Starts the thread and invokes run().
- run(): Defines the task to be executed.
- sleep(milliseconds): Pauses the thread for a specified time.
- join(): Waits for a thread to complete before proceeding.
- isAlive(): Checks if a thread is still running.
- getName() / setName(): Gets or sets the thread name.
- getPriority() / setPriority(): Gets or sets the thread priority.

1. Extending the Thread class

- The easiest way to create a thread is to write a class that extends the *Thread* class.

```
class MyThread extends Thread{
 public void run() {
  System.out.println("concurrent thread started running..");
 }
}
Class MyThreadDemo{
 public static void main( String args[] ) {
  MyThread mt = new  MyThread();
 Thread t= new Thread(mt)
  t.start();
 }
}
```

- **Thread cannot be started twice.**

2.Implementing the Runnable Interface

- You can create a class that implements the ***Runnable*** interface rather than ***extends*** the Thread class.
- The ***Runnable*** interface marks an object that can be run as a thread.
  - It has only one method, ***run*****,** that contains the code that's executed in the thread.
- The Runnable instance can be reused by different threads, making it more flexible
- Implementing Runnable lets the class inherit from other classes. This promotes better design by separating the task logic from the thread management.

❖ ***To use the Runnable interface and create and start a thread, you have to do the following:***

1. Create a class that implements Runnable.
2. Provide a run method in the Runnable class.
3. Create an instance of the Thread class and pass your Runnable object to its constructor as a parameter. *A Thread object is created that can run your Runnable class.*
4. Call the Thread object’s start method.
5. The run method of your Runnable object is called, which executes in a separate thread.

    i.e. assuming that your Runnable class is named

Cont..

```
class MyThread implements Runnable{
 public void run() {
  System.out.println("concurrent thread started running..");
 }
 }
class MyThreadDemo{
 public static void main( String args[] ) {
  MyThread mt = new MyThread();
  Thread  t = new Thread(mt);
  t.start();
 }}
```

Thread Priorities and Thread Scheduling

- Every Java thread has a ***thread priority*** that helps to determine the order in which threads are scheduled.
- Each thread is assigned a default priority of ***Thread.NORM_PRIORITY*** **(constant of 5).**
- You can reset the priority using ***setPriority(int priority).***
- Some constants for priorities include
  - ***Thread.MIN_PRIORITY ,***
  - ***Thread.MAX_PRIORITY and***
  - ***Thread.NORM_PRIORITY .***

Cont..

- By default, a thread has the priority level of the thread that created it .
- An operating system's thread scheduler determines which thread runs next.
- Most operating systems use ***timeslicing*** for threads of equal priority.
  - **Preemptive scheduling:** when a thread of higher priority enters the running state, it preempts the current thread.
  - ***Starvation*****:** Higher-priority threads can postpone the execution of lower-priority threads.

Thread Synchronization

- When multiple threads share an object and it’s modified by one or more of them, ***indeterminate*** results may occur.
- The problem can be solved by giving only one thread at a time ***exclusive access*** to code
- During that time, other threads desiring to manipulate the object are kept waiting.
- When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting is allowed to proceed.
- This process, called ***thread synchronization***
- A common way to perform synchronization is to use Java’s built-in monitors.

Cont..

- Every object has a ***monitor and a monitor lock***.
- The monitor ensures that its object's monitor lock is held by a maximum of **only one thread at any time.**
- To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a ***synchronized*** statement.
- The monitor allows only one thread at a time to execute statements within ***synchronized*** statements that lock on the same object.
- The ***synchronized*** statements are declared using the ***synchronized*** keyword

```
synchronized ( object )
{    statements
}
```

wait(), notify(), and notifyAll()

- These methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Other wise, an ***IllegalMonitorStateException*** would occur.
- The **wait()** method lets the thread wait until some condition occurs.
- Use the **notify()** or **notifyAll()** methods to notify the waiting threads to resume normal execution.
- The ***notifyAll()*** method wakes up all waiting threads, while ***notify()*** picks up only one thread from a waiting queue.

Example 1: without synchronization(1/3)

```
public class First {
    public void display(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("]");
    }
}
```

Example 1: without synchronization(2/3)

```
class Second extends Thread {
    String msg;
    First fobj;
    Second(First fp, String str) {
        fobj = fp;
        msg = str;
}
    public void run() {
      fobj.display(msg);
           }
}
```

Example 1: without synchronization(3/3)

```
public class MainTest {
public static void main (String[] args)
 {
 First fnew = new First();
 Second ss = new Second(fnew, "welcome");
 Second ss1= new Second (fnew,"new");
 Second ss2 = new Second(fnew, "programmer");
       Thread t1 = new Thread(ss);
       Thread t2 = new Thread(ss1);
       Thread t3 = new Thread(ss2);
       t1.start();
       t2.start();
       t3.start();
 }
}
```

```
[new[welcome[programmer]
]
]
```

Example 1: with synchronization(1/3)

```
public class First {
    public void display(String msg) {
        System.out.print("[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
        System.out.println("]");
    }
}
```

Example 1: with synchronization(2/3)

```
class Second extends Thread {
    String msg;
    First fobj;
    Second(First fp, String str) {
        fobj = fp;
        msg = str;
        start();
    }
    public void run() {
        synchronized (fobj) //Synchronized block
        {
            fobj.display(msg);
        }
    }
}
```

Example 1: without synchronization(3/3)

```
public class MainTest {
public static void main (String[] args)
 {
 First fnew = new First();
 Second ss = new Second(fnew, "welcome");
 Second ss1= new Second (fnew,“new");
 Second ss2 = new Second(fnew, "programmer");
      Thread t1 = new Thread(ss);
       Thread t2 = new Thread(ss1);
       Thread t3 = new Thread(ss2);
       t1.start();
       t2.start();
       t3.start();
 }
}
```

```
[welcome]
[programmer]
[new]
```

End of Chapter