THREAD

Tanjina Helaly

Concurrent Programming

- Different parts of the same program can be executing at the same time, or behave as if they are executing at the same time.
- Java uses threads to achieve concurrency.
- Writing concurrent programs presents a number of issues that do not occur in writing sequential code.

Issues Involve in Concurrent Programming

- Safety
  - Two different threads could write to the same memory location at the same time, leaving the memory location in an improper state.
- Liveness
  - Threads can become deadlocked, each thread waiting forever for the other to perform a task. Threads can become livelocked, waiting forever to get their turn to execute.
- Nondeterminism
  - Thread activities can become intertwined. Different executions of a program with the same input can produce different results. This can make program hard to debug.
- Communication
  - Different threads in the same program execute autonomously from each other. Communication between threads is an issue.

What is Thread?

- Thread is an independent path of execution through program code.
- A multi threaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
- When multiple threads execute, one thread's path through the same code usually differs from the others.

Why we use Threads?

- To perform asynchronous or background processing
- Increases the responsiveness of GUI applications while performing other tasks, such as repaginating or printing a document.
- Take advantage of multiprocessor systems
- Simplify program logic when there are multiple independent entities

What happens when a thread is invoked?

- When a thread is invoked, there will be **two paths** of execution.
  - One path will execute the thread and
  - the other path will follow the statement after the thread invocation.
  - There will be a separate stack and memory space for each thread.

How to Create and Run Thread

How to Create and Run Thread

- **1) Define and Create a Thread object either by**
  - Implementing the Runnable interface. And pass the object as job of the Thread object.
  - Extending the java.lang.Thread class.
- 2) *For both cases we need to override "public void run()" method which should contain the code that needs to be run as Thread.*
- 3) Start the thread of execution by calling the start() method.
- **Do not call the run() method directly**.
  - Calling the run() directly executes the method in the normal sequential manner.
- **Do not call the start() method more than once.**
  - After **starting** a **thread**, it **can** never be started again. If you **does** so, an IllegalThreadStateException is thrown. In such case, **thread will run** once but for second time, it **will** throw exception.

Implementing the Runnable Interface

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.
  - First, have your class implement the Runnable interface, which has one method, run().
    - This run() plays the same role as the run() in the Thread subclass in the first method.
  - Second, create an instance of the Thread class, passing an instance of your class to the constructor.
  - Finally, send the thread object the start() method.

Implementing the Runnable Interface

```
public class ThreadTest {
    public static void main(String[] args) {
        Thread t1 = new Thread(new JobForThread(), "1st thread");
        Thread t2 = new Thread(new JobForThread(), "2nd thread");
        t1.start();
        t2.start();
        System.out.println("Main completed");
    }
}

public class JobForThread implements Runnable {
    public void run() {
        for (int i = 0; i<50; i++) {
            System.out.println(Thread.currentThread().getName()+":"+i);
            try {
              Thread.sleep(200);
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
        }
    }
}
```

Extending the Thread Class

- In the Thread subclass, implement the run() method. The signature of run() must same as Thread class as shown in the example.
  - run() is the entry point or starting point (or main) of your thread.
- To start a thread, create an object from your Thread class. Call the "start()“ method to the thread object.
  - This will create the new thread, start it as an active entity in your program, and call the run() method in the thread object.

Extending the Thread Class

```
public class ThreadTest extends Thread{
    public static void main(String[] args) {
        Thread t1 = new ThreadTest();
        Thread t2 = new ThreadTest();
        t1.start();
        t2.start();
        System.out.println("Main completed");
    }

    public void run() {
        for (int i = 0; i<50; i++) {
            System.out.println(Thread.currentThread().getName()+":"+i);
            try {
              Thread.sleep(200);
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
        }
    }
}
```

Extends Thread class vs Implements Runnable Interface?

- Extending the Thread class will make your class unable to extend other classes, because of the single inheritance feature in JAVA. However, this will give you a simpler code structure. If you implement Runnable, you can gain better object-oriented design and consistency and also avoid the single inheritance problems.
- If you just want to achieve basic functionality of a thread you can simply implement Runnable interface and override run() method. But if you want to do something serious with thread object as it has other methods like suspend(), resume(), ..etc which are not available in Runnable interface then you may prefer to extend the Thread class.

Thread Scheduling & Life Cycle

Thread scheduling

- In an idealized world, all program threads would have their own processors on which to run.
- In real world, threads often must share one or more processors
- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- You can't control the scheduler. There is no API for calling methods on the scheduler..)
- Only one thread at a time can run in a single process.

Thread scheduling - Steps

- All thread that are at runnable state will wait in a pool.
- Scheduler will pick one of them and run.
- Typically, a JVM's thread scheduler chooses the highest-priority thread and allows that thread to run until it either terminates or blocks.
- At that time, the thread scheduler chooses a thread of the next highest priority. That thread (usually) runs until it terminates or blocks.
- Java is priority-preemptive
  - If a high-priority thread wakes up/unblocks, and a low-priority thread is running
  - Then the high-priority thread gets to run immediately
- But this is not guaranteed. In some circumstances, a lower priority Thread will be chosen.

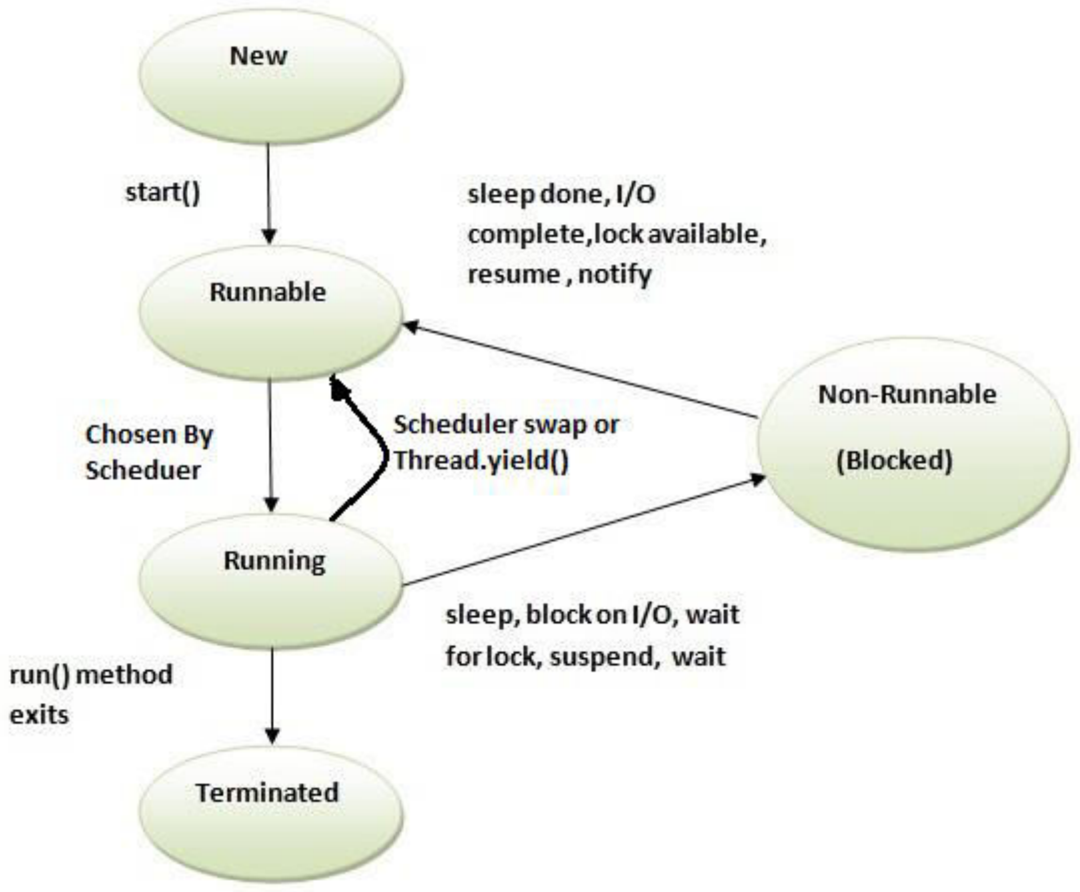
Thread Priority

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- As an absolute value, a priority is meaningless;
  - a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.
  - Instead, a thread's priority is used to decide when to switch from one running thread to the next.
- We can set the priority using setPriority(int) method.
  - The value will be between 1 and 10.
  - Or can use one of the constant defined in Thread class
    - Thread.MIN_PRIORITY - minimum thread priority (set to 1)
    - Thread.MAX_PRIORITY - maximum thread priority (set to 10)
    - Thread.NORM_PRIORITY – normal thread priority(set to 5)
  - Default priority is 5.
- **Remember** that the **priorities should be set before the threads start** method is invoked.

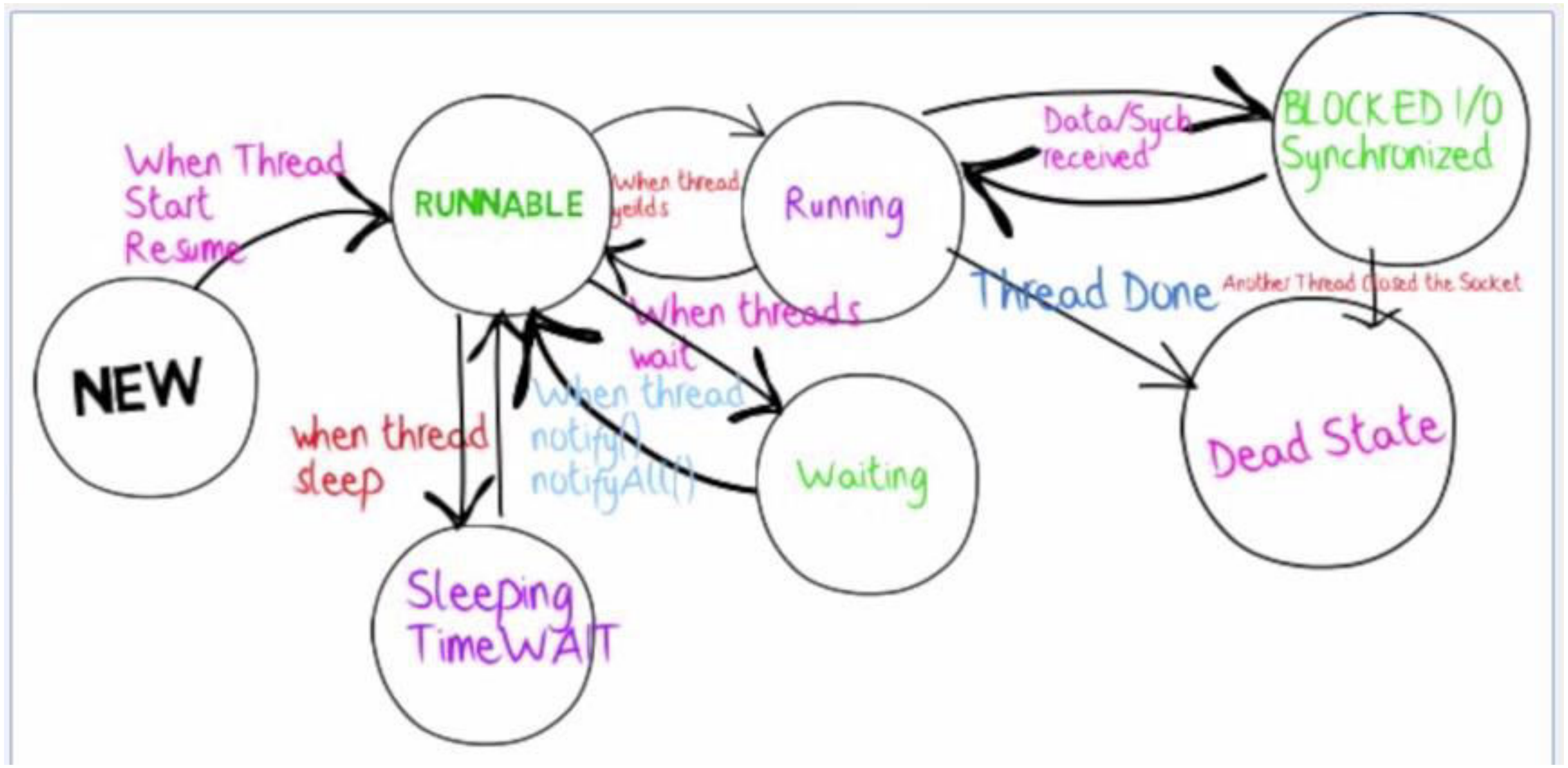
Thread States

- **Initial state:** A program has created a thread's thread object, but the thread does not yet exist because the thread object's start() method has not yet been called.
- **Runnable state:** This is a thread's default state. After the call to start() completes, a thread becomes runnable. Although many threads might be runnable, only one currently runs.
- **Blocked state:** Thread is neither running nor in a position to run. (You can probably think of other times when a thread would wait for something to happen.)
    - When a thread executes the sleep(), wait(), or join() methods,
    - when a thread attempts to read data not yet available from a network or stream, and
    - when a thread waits to acquire a lock, that thread is in the blocked state:
    - When a blocked thread unblocks, that thread moves to the runnable state.
- **Terminating state:** Once execution leaves a thread's run() method, that thread is in the terminating state. In other words, the thread ceases to exist.

Thread Life Cycle


New
start()
Runnable
sleep done, I/O complete,lock available, resume , notify
Chosen By Scheduer
Scheduler swap or Thread.yield()
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

Thread Life Cycle – Another version



Some methods – influence scheduling

- Some of the methods from the java.lang.Thread Class that can help us **influence thread scheduling** are as follows:
  - public static void sleep(long millis) throws InterruptedException

  - public static void yield()
    - Theoretically, **to 'yield' means to let go, to give up, to surrender**.
    - A yielding thread tells the virtual machine that it's willing to let other threads be scheduled in its place.
    - This indicates that it's not doing something too critical. Note that *it's only a hint*, though, and not guaranteed to have any effect at all.
    - Moves from running state to runnable state

  - public final void join() throws InterruptedException
    - Allows one thread to wait for the completion of another.
    - If t is a Thread object whose thread is currently executing,

      t.join();

      causes the current thread to pause execution until t's thread terminates.
    - Overloads of join allow the programmer to specify a waiting period.

  - public final void setPriority(int newPriority)

Some methods

- Few other methods
  - public static native Thread currentThread();
  - public final String getName()
  - public final synchronized void setName(String name)
  - public final native boolean isAlive();

Synchronization

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 resource is locked by one thread so that no one can get access to the resource.
- Method level

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

- Object level

```
synchronized(objRef) {
// statements to be synchronized
}
```

  - Mainly when the class is not designed by you and you do not have access to source code.
  - Class is not designed to handle multi threading(no synchronized keyword),
  - Don’t want to synchronized at method level.

Synchronization

- For both cases the lock is acquired at object level not method.
- Every Java object has a lock. And a lock has only one 'key. .
- Most of the time, the lock is at unlocked and nobody cares.
- If an object has synchronized methods, a thread can enter one of the synchronized methods ONLY if the "key for the object's lock is available.
    - In other words, only if another thread hasn't already grabbed the one "key.

Synchronization – Any disadvantages?

- Synchronization doesn't come *for free.*
- *First, a synchronized method has a certain* amount of overhead.
  - When code hits a synchronized method, there's going to be a performance hit while the matter of “Is the key available“ is resolved.
- Second, a synchronized method can slow your program down because synchronization restricts concurrency.
  - a synchronized method forces other threads to get in line and wait their turn.
- Third, and most frightening, synchronized methods can lead to deadlock.
  - Thread deadlock happens when you have two threads, both of which are holding a key the other thread wants. There's no way out of this scenario, so the two threads will simply sit and wait. And wait. And wait.

Synchronization – How to overcome?

- A good rule of thumb Is to synchronize only the bare minimum that should be synchronized.
- And In fact, you can synchronize at a granularity that's even smaller than a method.
  - you can use the synchronized keyword to synchronize at the more fine-grained level of one or more statements, rather than at the whole-method level.

Inter-thread Communication

- To avoid polling, Java includes an elegant inter-process 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( )**
  - when wait() method is invoked on **an object**,
  - the thread executing that code gives up its lock on the object immediately
  - and moves the thread to the wait/blocked state.
  - throws InterruptedException
- **notify()**:
  - This wakes up threads that called wait() on the same object and
  - moves the thread to ready/runnable state.
- **notifyAll()**:
  - This wakes up all the threads that called wait() on the same object.
  - One of the threads will be granted access.

Producer-Consumer Problem

- In computing, the **producer–consumer problem** (also known as the bounded-buffer **problem**) is a classic example of a multi-process synchronization **problem**. The **problem** describes two processes, the **producer** and the **consumer**, who share a common, fixed-size buffer used as a queue.
- Producer
  - puts items into the buffer
  - must wait until the buffer has space before it can put something in
- Consumers
  - takes items out of the buffer
  - must wait until something is in the buffer before it can take something out.
- A condition variable represents a queue of threads waiting for some condition to be signaled.

Producer-Consumer Problem

- Common Resource - MyStack

```
import java.util.ArrayList;
public class MyStack {
    private ArrayList<Integer> data = new ArrayList<Integer>();

    public synchronized void push(int num){
        // add the num to the ArrayList (data)
        data.add(num);
        System.out.println("Pushed:" + num + ":"+ data);
        notify();
    }

    public synchronized int pop() {
        if (data.size() == 0){
            try {
            wait();
            } catch (InterruptedException e) {
            System.out.println(e.getMessage());
            }
        }

        // return the last item and remove that item from the list
        int item = data.remove(data.size() - 1);
        System.out.println("Poped:" + item);
        return item;
        }
    }
}
```

Producer-Consumer Problem

- Producer

```
import java.util.Random;

public class Producer implements Runnable{
	MyStack myStack;

	public Producer(MyStack s) {
			myStack = s;
	}

	public void run() {
		int num;
		Random rand = new Random();
		for (int i=0; i<50; i++){
			// generate a random number between 0 to 10 and call the push method using myStack.
			System.out.println(Thread.currentThread().getName());
			num = rand.nextInt(10);
			myStack.push(num);

			// Call the sleep method and pass ~ 100 ms.
			try {
			Thread.sleep(100);}
			catch (InterruptedException e) {
			e.printStackTrace();
			}
		}
	}
}
```

Producer-Consumer Problem

- Consumer

```
public class Consumer implements Runnable{
    MyStack myStack;

    public Consumer(MyStack s) {
            myStack = s;
    }

    public void run() {
        int num;
        for (int i=0; i<50; i++){
            System.out.println(Thread.currentThread().getName());
            // Call the pop method and print the number. Also sleep for ~ 100 ms.
            try {
            num = myStack.pop();
            } catch (Exception e1) {
            System.out.println(e1.getMessage());;
            }

            try {
            Thread.sleep(100);
            } catch (InterruptedException e) {
            e.printStackTrace();
            }
        }
    }
}
```

Producer-Consumer Problem

- Application Class

```
public class MainClass {

    public static void main(String[] args) {
        MyStack st = new MyStack();
        // Start 2 Threads; 1 for Producer and 1 for Consumer.
        // For both Producer and Consumer pass the same st object as parameter

        new Thread(new Producer(st), "Producer").start();
        new Thread(new Consumer(st), "Consumer ").start();
        }
}
```

Reference

- Java: Complete Reference - Chapter 11