Operating Systems

1DV512-HT19

Tutorial: “Java Programming with Threads”

Aris Alissandrakis (based on slides originally by Suejb Memeti)

Department of Computer Science & Media Technology

November 7, 2019

Introduction

The aim of this presentation is to introduce you to Java multi-threading

- Start, Interrupt and Sleep Threads
- Thread Synchronization
- The Volatile variables and Synchronized methods
- Locks, Multiple locks
- Thread Pools
- Wait and Notify commands
- Deadlocks
- Semaphores

Code examples

Questions

Starting Threads in Java

Extend the Thread class (see *demo1.ExtendThread.java*)

- Threads can be controlled using the Thread class
- Start the thread using the start() method in order to run it in a separate thread

```
class ClassName extends Thread {
      public void run() {
            //your code here
                  }
```

```
public static void main(String[] args) {
      ClassName t1 = new ClassName();
      t1.start();
}
```

Implement the Runnable interface (see *demo2.ImplementRunnable.java*)

- Implement runnable class and pass it to the constructor of Thread

```
class ClassName implements Runnable {
      public void run() {
            //your code here
                  }
```

```
public static void main(String[] args) {
      Thread t1 = new Thread(new ClassName());
      t1.start();
}
```


Starting Threads in Java - cont'd

Using Thread pools

- ExecutorService - starting multiple threads at once

```
ExecutorService exec = Executors.newFixedThreadPool(2);
for (int i = 0; i < 5; i++) {
                  exec.submit(new Runnable() {
                        public void run() {
                              //your code here
                        }
                  });
}
```


Putting the threads to sleep

- **Using the sleep() method**
  - The thread pauses/sleeps for a certain amount of time.
  - Accepts an integer which indicates the milliseconds you want the thread to sleep for

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


The volatile variables and Interrupting Threads in Java

- **Stop thread using shared data (see *demo3.Volatile.java*)**
  - It is possible that on some systems (or java implementation), when java optimizes the code, the thread (in our example “Processor”) decides to cache a variable (in our example the “running” public variable).
  - To prevent caching variables we can use ***volatile*** keyword

- **Thread Interruption (see *demo3.Interrupt(ThreadPool)?.java*)**
  - Using the interrupt() method, and handling the InterruptedException.
  - Interrupt thread pool using shutdownNow() method

The Synchronized methods (see *demo4.Synchronized.java*)

- **Problem: Thread interleaving**
  - Two threads reading/writing the same data

```
private int count = 0;
//T1
for(int i=0; i<1000; i++) {
        count ++;
}
//T2
for(int i=0; i<1000; i++) {
        count ++;
}
```

- **Solution: Synchronized command**
  - Makes sure that when one thread is performing an action, no other thread is performing the same action at the same time
  - First thread acquires an intrinsic lock to the method, and the second thread has to wait until the intrinsic lock is released.

```
public synchronized void increment() {
        count ++;
}
```


Multiple Locks using Synchronized Code Blocks

- **The synchronized code blocks (see *demo5.Synchronized(Methods|CodeBlocks).java*)**
  - Allow you to lock a part of your code and assign different lock object to each synchronized code block

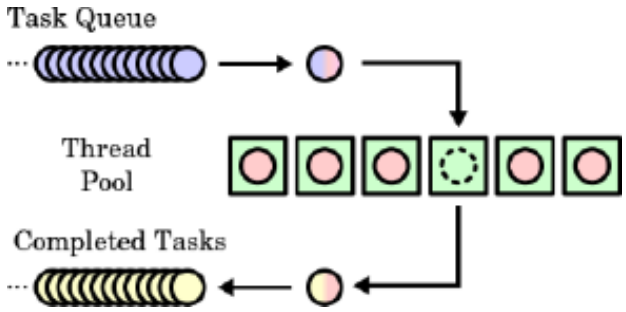
```
public synchronized void stageOne() {
      list1.add(random.nextInt(100));
}
public synchronized void stageTwo() {
      list2.add(random.nextInt(100));
}

public void process() {
      for (int i = 0; i < 1000; i++) {
            stageOne();
            stageTwo();
      }
}
```

```
private Object lock1 = new Object();
private Object lock2 = new Object();
public void stageOne() {
      synchronized (lock1) {
            list1.add(random.nextInt(100));
      }
}
public void stageTwo() {
      synchronized (lock2) {
            list2.add(random.nextInt(100));
      }
}
public void process() {
      for (int i = 0; i < 1000; i++) {
            stageOne();
            stageTwo();
      }
}
```


Thread Pools (see *demo6.ThreadPool.java*)

Way of managing lots of threads at the same time

- Thread pool is a group of threads waiting for tasks to execute
- The threads are always existing, which avoids the overhead of creating them every time
- Using ExecutorService tasks are added in a queue, and assigned one at a time to each thread
- You can think as having a number of workers in a factory, and having a larger number of tasks for these workers. When a worker completes a task, a new task will be assigned to him.



Wait and Notify (see *demo7.Processor.java*)

Wait()

- releases the lock of this object
- tells the calling thread to give up the monitor and go to sleep until the other thread enters the same monitor and calls notify()

Notify()

- wakes up the first thread that called wait() on the same object

NotifyAll()

- wakes up the all the threads that are waiting on the same object

Can be used inside synchronized method or code blocks.

Low vs High Level synchronization techniques

High level synchronization using Java Concurrent package (see *demo8.HighLevelSynchTechnique.java*)

- Contains set of classes that makes it easier to develop multithreaded applications in Java.
- Avoids the low level synchronization with the ***synchronized*** methods or code blocks
- Available in ***java.util.concurrent*** package

Low level synchronization (see *demo8.LowLevelSynchTechniques.java*)

- Manually handling the thread synchronization using ***synchronized***, ***wait***, ***notify*** …

Deadlocks

Deadlock is a situation where two or more threads are locked forever

- It can occur when locks are locked in different orders

Deadlock prevention (see *demo9.Runner.java*)

- Lock Ordering
  - Make sure the locks are always taken in the same order by any thread
- Lock Timeout
  - Put a timeout on lock attempts, If not successful in taking the necessary locks, backup, free all the acquired locks, wait for some time and retry.
- Deadlock Detection
  - The heavier deadlock prevention. Every time a thread takes a lock or requests a lock it is noted in a data structure (map, graph) of threads and locks.
  - The detection is done by traversing the lock graph.

Semaphores (see *demo10.Connection.java*)

Semaphores ensure that only a given number of processes can access a certain resource at a given time.

- Useful for limiting connections
- Limiting thread creation
- Limiting concurrent access to the disk

Always release what you acquire (try - finally blocks)

- acquire() will block until permits are available
- release() will always increment the number of permits

Literature

- Multithreaded Programming (Chapter 4), in book Operating System Concepts, pages 161-199
- The Java Tutorials (Oracle) https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- Steven Haines and Stephen Potts, “Java 2 Primer Plus”, Sams Publishing 2003
- Cave of Programming, http://www.caveofprogramming.com