CSCD 330
Network Programming

Lecture 12
More Client-Server Programming
Winter 2016

Reading: References at end of Lecture

Introduction

- So far,
- Looked at client-server programs with Java
  - Sockets – TCP and UDP based
  - Reading and writing to a socket
  - Sockets – with threads
  - Multiple clients at a time
- Today
  - Look at more complexity with threads
  - Synchronizing threads
  - Come in handy with future program

Threads

- Recall … A thread is
  - Separate thread of execution
  - Called from an existing program
  - A java class, Thread represents a thread in Java
  - Has a separate stack
  - Allows you to run separate processes from main thread
  - Can do concurrency

Threads States

Thread t = new Thread (r);

t.start();

Running!

Thread created but not started

Thread ready to run, runnable state
Waiting to be selected for execution

Thread selected to run and is the currently running thread

Once thread is runnable, can go back and forth between running, runnable and blocked

Threads

- The JVM thread scheduler is responsible for deciding who gets to run next
- You, have little control over the decision
- Don’t base your program’s correctness on the scheduler working in a particular way

Example:

Two threads executing

Example of Two Threads

```
public class RunThreads implements Runnable {
    RunThreads runner = new RunThreads ();
    Thread alpha = new Thread (runner);
    Thread beta = new Thread (runner);
        alpha.setName (“Alpha thread”);
        beta.setName (“Beta thread”);
        alpha.start ();
        beta.start();
}
```

Example of Two Threads

```
public void run() {
   for (int i = 0; i < 25; i++) {
      String threadName = Thread.currentThread().getName ();
      System.out.println (threadName + " is running");
   }
}
}
```

Lets execute this and see what happens ….

Will run two examples of this.

How can we Control Scheduler

- What can we do to influence the thread execution?

Example of Two Threads

- We can do something about the Scheduler
- Can't control it, but we can force it to execute things in order
- There is

  Thread.sleep(arg)

  Will put one thread to sleep

  Call the sleep method and pass it the sleep duration in milliseconds

  Will knock the thread out of the running state and keep it out for the duration

  Example: Thread.sleep (2000); thread sleeps 2 secs

Sleep method throws in InterruptedException, so all calls must be wrapped in a try/catch

---

```
public void run() {

    String threadName = Thread.currentThread().getName ();
    if (threadName.equals("Alpha thread")) {
      try {
        Thread.sleep(500);
      } catch (InterruptedException ex) {
        ex.printStackTrace ();
      }
    }

    for (int i = 0; i < 25; i++) {
        System.out.println (threadName + "is running");
  }
}
```

Put one thread to sleep

Other thread can execute first

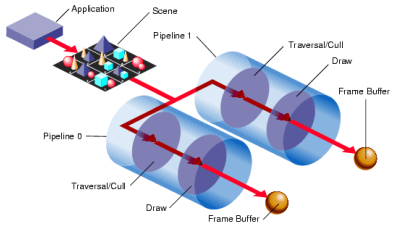
Execute again, RunThreads4 or RunThreads3

Thread States

- After putting thread to sleep ...
  - When it wakes up, its runnable
  - Can be selected to run

- No guarantee that it will be run!!

Concurrency and Threads



- So, with threads there can be concurrency issues
  - When you need to coordinate access to same object between two or more threads
  - Left hand side doesn't know what right hand side is doing
  - Operating from two different stacks
  - When thread is not running, it is completely unaware that anything else is happening
  - Like being unconscious
  - Lets look at an example ...

Ryan and Monica Two Thread Story

- Two threads one object
  - Ryan and Monica, a couple
  - One bank account, limited amount of money!!
  - Trouble for sure
  - RyanandMonica are a Job
    - Its runnable
  - Two threads, both operate on the same job
  - Artificially create a race condition in the code
  - They each need to withdraw an amount without overdrawing the account

Ryan and Monica Two Thread Story

- Run method,
  - Ryan and Monica each running in a separate thread
    - Check the balance,
    - if enough money, withdraw amount
    - Should protect against overdrawing the account … but
    - They fall asleep after checking the balance

      but before they finish withdrawal

```
class BankAccount {
   private int balance = 100;

   public int getBalance() {
      return balance;
   }

   public void withdraw(int amount) {
      balance = balance - amount;
   }
}
```

withdraw amount

```
public class RyanAndMonica
    implements Runnable {

  private BankAccount account =
        new BankAccount();

  public static void main (String [] args)
   {
    RyanAndMonica  theJob =
        new RyanAndMonica();
    Thread one = new Thread (theJob);
    Thread two = new Thread (theJob);
    one.setName ("Ryan");
    two.setName ("Monica");
    one.start();
    two.start();
```

Job

Start two threads

Ryan and Monica
Two Thread Story

```
public void run() {
   for (int x = 0; x < 10; x++) {
      makeWithdrawal(10);
      if (account.getBalance() < 0) {
        System.out.println("Overdrawn!");
      }
    }
 }
```

Try to withdraw 10 times

Flag overdrawn

Within makeWithdrawal, put checks to prevent overdrawn condition

```
private void makeWithdrawal(int amount) {
    if (account.getBalance() >= amount) {
    System.out.println(Thread.currentThread().getName() + " is about to
    withdraw");

     try {
       System.out.println(Thread.currentThread().getName() + " is going to
       sleep");
       Thread.sleep(1000);
     } catch(InterruptedException ex) {ex.printStackTrace(); }

     System.out.println(Thread.currentThread().getName() + " woke up.");
     account.withdraw(amount);

     System.out.println(Thread.currentThread().getName() + " completes the
    withdrawl");
    }
   else {
     System.out.println("Sorry, not enough money for " +
    Thread.currentThread().getName());
  }
 }
}
```

check account

force sleep

make withdrawal

Ryan and Monica …. run it and see ….

RyanAndMonica class

Ryan and Monica Continued

- What happened?
  - Ryan was in the middle of his transaction when Monica thread checked balance and withdrew amount
  - Invalid transaction
  - Solution?
  - Make the transaction one single action
  - **Atomic transaction**
  - Will see this when you take Database
  - Must check and withdraw money all in one action

Monica and Ryan

- Need a lock associated with the bank account transaction
- One key and it stays with the lock
- One person at a time gets the lock and key until transaction finished
- Key is not released until transaction over
- Use the synchronized key word in Java
- Prevents multiple threads from using a method at a time

Ryan and Monica

```
private synchronized void makeWithdrawal(int amount) {
    if (account.getBalance() >= amount) {
    System.out.println(Thread.currentThread().getName() +
   " is about to withdraw");
     try {
       System.out.println(Thread.currentThread().getName()
   + " is going to sleep");
       Thread.sleep(1000);
     } catch(InterruptedException ex) {ex.printStackTrace();
   }
```

……

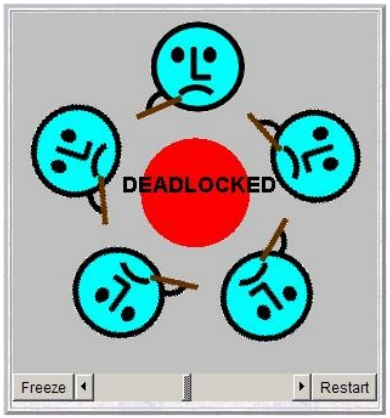
Re-run example …. RyanAndMonica3

Threads and Concurrency

- Goal of synchronization
  - Protect critical data
  - Don’t lock the data
  - Synchronize methods that access the data

- Problem with Synchronization?

Problems with Concurrency



- Problem is …
  - Deadlock
  - Java has no real way to deal with deadlock
  - You must design your applications to handle this case
  - Pay attention to the order in which you start your threads

Deadlock Scenario

- Simple Deadlock scenario
  - Thread A enters synchronized method – object foo
    - Get key
    - Thread A goes to sleep holding foo key
  - Thread B enters synchronized method – object bar
    - Get key
    - Thread B tries to enter synchronized method of object foo but can’t get that key
    - Goes to waiting room, keeps bar key
  - Thread A wakes up, tries to enter synchronized method of object bar, can’t, key not available
    - Goes to waiting room
  - Problem?

Other Thread Methods

- Another way besides sleep() to influence threads to move between runnable and ready-to-run is
- Thread.yield();
  - All it does is move one thread to ready-to-run for a while
  - May not result in threads taking turns
- Demo with RyanAndMonica4.java

Other Thread Methods

- Turns out that once a thread gets a lock, does not want to give it up
- Thus, unless we can break Monica's hold on the bank book
  - Resulting from the Synchronized method
- Ryan is out of luck for getting any money!
- Wait (); - Turns out, you can make Monica wait for a while and allow Ryan to get in and get some money
  - Wait - makes the lock holding thread pause
  - Notify - used to tell waiting thread lock is free

Final Solution to Ryan and Monica

- So, the wait() method will help regulate Monica's behavior
- Set up the synchronized method to create a critical section
- Then, make her release the lock through the wait method so Ryan has a chance !!!

- Demo RyanAndMonica5

Summary

- Covered Threads and Synchronization
- Demonstrated that the JVM scheduler can’t be trusted to execute threads in a given order
- Concurrency issues can lead to deadlock
- Ways around this ... but need to be aware of issues

References

More Intro Type of Information

http://www.javabeginner.com/learn-java/java-threads-tutorial

Goes into Semaphores and Locking

http://www.ibm.com/developerworks/library/j-thread.html

Goes into Synchronization of Java Threads

http://java.sun.com/docs/books/tutorial/essential/concurrency/sync.html

Insider view of Java Virtual Machine more detailed ...

http://www.artima.com/insidejvm/ed2/threadsynch.html

Next:

Network Layer – Routers and Routing