COMP241
Software Engineering Development
Lecture 20: Networking and Threads 2

Mark Hall

Readings: Horstmann Ch 9

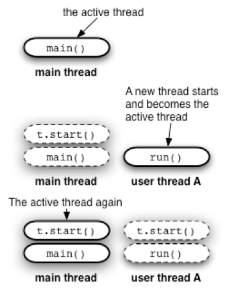
- Threads
- Multiple call stacks
- `Runnable`
- States of a thread
- Thread scheduler
- `Thread.sleep()`
- Concurrency issues
  - Lost update problem
- `synchronize`

DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

Last time…

- We introduced Java's `Thread` class
  - Saw how to create a thread but…
  - The thread needs a job to execute
- With more than one thread of execution, we have more than one call stack

More than one call stack



1. The JVM calls the main() method

```
public static void main(String [] args) {
  . . .
}
```

2. main() starts a new thread. The main thread is temporarily frozen while the new thread starts running.

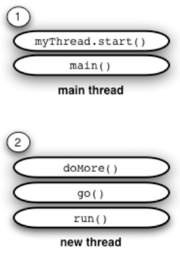
```
Runnable r = new MyThreadJob();
Thread t = new Thread(r);
t.start();
Dog d = new Dog();
```

3. The JVM switches between the new thread (user thread A) and the original main thread, until both threads complete.

Launching a new thread

1. Make a Runnable object (the thread's job)
   `Runnable threadJob` `= new MyRunnable();`
2. Make a Thread object (the worker) and give it the Runnable (the job)
   `Thread myThread = new Thread(`**`threadJob`**`);`
3. Start the Thread
   `myThread`**`.start()`**`;`
   - The thread calls the **`run()`** method defined in the **`Runnable`** interface

```
public class MyRunnable implements Runnable {

  public void run() {
    go();
  }

  public void go() {
    doMore();
  }
}

class ThreadTester {

  public static void main(String [] args) {

    Runnable threadJob = new MyRunnable();
    Thread myThread = new Thread(threadJob);

    myThread.start();
    System.out.println("back in main");
  }
}
```



States of a new thread

- New
  - After construction, waiting be started
  - Thread object, but no *thread of execution*
- Runnable
  - After starting (`start()` called) it moves into the runnable state
  - Waiting to be chosen for execution
  - Has a new call stack at this point
- Running
  - The currently running thread
  - Only the JVM **thread scheduler** decides which thread to executed
    - You can sometimes *influence* the decision, but not force a thread to move from runnable to running

States of a thread

- Once a thread becomes runnable, it can move back and forth between runnable, running and an additional state: **blocked** (temporarily not runnable)
  - The thread scheduler can move a thread to a blocked state for various reasons
    - E.g. Thread might be reading from a Socket input stream, but there isn't any data to read
    - E.g. executing code may have told the thread to put itself to sleep (sleep())
    - E.g. Thread might be trying to call a method on an object, and that object was 'locked'

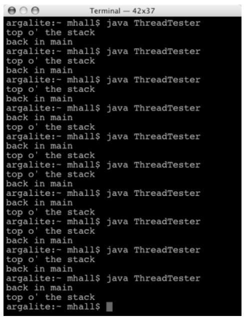
Thread Scheduler

- Controls the moving of threads from runnable to running
- There are no guarantees about scheduling

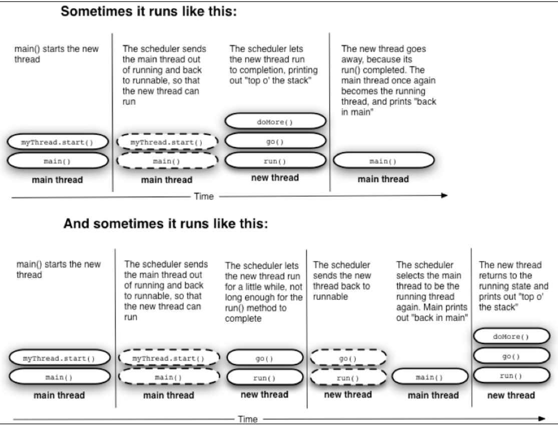
  Do not base your program's correctness on the scheduler working in a particular way!
  - Implementations of the scheduler are different for different JVMs
    - Even running the same program on the same machine can give different results
  - Use `sleep()` to give another thread a chance to run
    - A sleeping thread will **not** become the currently-running thread before the length of the sleep time has expired

Unpredictability example

```
public class MyRunnable implements Runnable {

  public void run() {
    go();
  }

  public void go() {
    doMore();
  }

  public void doMore() {
    System.out.println("top o' the stack");
  }
}

class ThreadTester {

  public static void main(String [] args) {

    Runnable threadJob = new MyRunnable();
    Thread myThread = new Thread(threadJob);

    myThread.start();
    System.out.println("back in main");
  }
}
```



```
argalite:~ mhall$ java ThreadTester
top o' the stack
back in main
argalite:~ mhall$ java ThreadTester
top o' the stack
back in main
argalite:~ mhall$ java ThreadTester
top o' the stack
back in main
argalite:~ mhall$ java ThreadTester
back in main
top o' the stack
argalite:~ mhall$ java ThreadTester
back in main
top o' the stack
argalite:~ mhall$ java ThreadTester
top o' the stack
back in main
argalite:~ mhall$ java ThreadTester
back in main
top o' the stack
argalite:~ mhall$ java ThreadTester
top o' the stack
back in main
argalite:~ mhall$ java ThreadTester
top o' the stack
back in main
argalite:~ mhall$ java ThreadTester
back in main
top o' the stack
argalite:~ mhall$
```


Sometimes it runs like this:
main() starts the new thread
The scheduler sends the main thread out of running and back to runnable, so that the new thread can run
The scheduler lets the new thread run to completion, printing out "top o' the stack"
The new thread goes away, because its run() completed. The main thread once again becomes the running thread, and prints "back in main"
myThread.start()
main()
main thread
myThread.start()
main()
main thread
doMore()
go()
run()
new thread
main()
main thread
Time
And sometimes it runs like this:
main() starts the new thread
The scheduler sends the main thread out of running and back to runnable, so that the new thread can run
The scheduler lets the new thread run for a little while, not long enough for the run() method to complete
The scheduler sends the new thread back to runnable
The scheduler selects the main thread to be the running thread again. Main prints out "back in main"
The new thread returns to the running state and prints out "top o' the stack"
myThread.start()
main()
main thread
myThread.start()
main()
main thread
go()
run()
new thread
go()
run()
new thread
main()
main thread
doMore()
go()
run()
new thread
Time

Putting a thread to sleep

- One of the best ways to help your threads take turns is to put them to sleep periodically
  - For example:

    ```
    Thread.sleep(2000);
    ```
  - Will knock a thread out of the running state, and keep it out of the *runnable* state for two seconds
  - **`sleep()`** throws an **`InterruptedException`**

```
try {
  Thread.sleep(2000);
} catch (InterruptedException ex) {
  ex.printStackTrace();
}
```

Using sleep to make our program more predictable

```
public class MyRunnable implements Runnable {

  public void run() {
    go();
  }

  public void go() {
    try {
      Thread.sleep(2000);
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    doMore();
  }

  public void doMore() {
    System.out.println("top o' the stack");
  }
}

class ThreadTester {

  public static void main(String [] args) {

    Runnable threadJob = new MyRunnable();
    Thread myThread = new Thread(threadJob);

    myThread.start();
    System.out.println("back in main");
  }
}
```

Concurrency Issues

- Consider the following:
  - We have *two* threads, Ryan and Monica, who share a singe object, the **`BankAccount`**
  - Two classes: **`BankAccount`** and **`RyanAndMonicaJob`**
  - **`RyanAndMonicaJob`** implements **`Runnable`**
    - Behaviours of checking the balance and making withdrawals

1 **Make one instance of RyanAndMonicaJob**
The RyanAndMonicaJob class is the Runnable, and since both Monica and Ryan do the same thing, we need only one instance.

```
RyanAndMonicaJob theJob = new RyanAndMonicaJob();
```

2 **Make two threads with the same runnable**

```
Thread one = new Thread(theJob);
Thread two = new Thread(theJob);
```

3 **Name and start the threads**

```
one.setName("Ryan");
two.setName("Monica");
one.start(); two.start();
```

4 **Watch both threads execute the run() method**
One thread represents Ryan, the other represents Monica. Both threads continually check the balance and then make a withdrawal, but only if its safe!

```
if (account.getBalance() >= amount) {
  try {
    Thread.sleep(500);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
```

In the run() method check the balance and, if there's enough money, make the withdrawal.

This should protect against overdrawing the account.

Except... Ryan and Monica always fall asleep **after** they check the balance but **before** they finish the withdrawal.

Ryan and Monica Example

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

  public int getBalance() {
    return balance;
  }

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

public class RyanAndMonicaJob implements Runnable {
  private BankAccount account = new BankAccount();

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

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

Ryan and Monica Example

```
private void makeWithdrawal(int amount) {
  if (account.getBalance() >= amount) {
    // use the static currentThread method to access the currently running
    // thread
    System.out.println(Thread.currentThread().getName()
                       + " is about to withdraw");

    try {
      System.out.println(Thread.currentThread().getName()
                         + " is going to sleep");
      Thread.sleep(500);
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    }

    System.out.println(Thread.currentThread().getName()
                       + " woke up");
    account.withdraw(amount);
    System.out.println(Thread.currentThread().getName()
                       + " completes the withdrawal");
  } else {
    System.out.println("Sorry, not enough for "
                       + Thread.currentThread().getName());
  }
}
}
```

```
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up
Monica completes the withdrawal
Monica is about to withdraw
Monica is going to sleep
Ryan woke up
Ryan completes the withdrawal
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up
Monica completes the withdrawal
Monica is about to withdraw
Monica is going to sleep
Ryan woke up
Ryan completes the withdrawal
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up
Monica completes the withdrawal
Monica is about to withdraw
Monica is going to sleep
Ryan woke up
Ryan completes the withdrawal
Ryan is about to withdraw
Ryan is going to sleep
Monica woke up
Monica completes the withdrawal
Sorry, not enough for Monica
Sorry, not enough for Monica
Sorry, not enough for Monica
Sorry, not enough for Monica
Sorry, not enough for Monica
Ryan woke up
Ryan completes the withdrawal
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
Sorry, not enough for Ryan
Overdrawn!
```

The makeWithdrawal() method always checks the balance before making a withdrawal, but still we overdraw the account.

Here is one scenario:

Ryan checks the balance, sees that there is enough money, and then falls asleep.

Meanwhile, Monica comes in and checks the balance. She, too, sees that there is enough money. She has no idea that Ryan is going to wake up and complete a withdrawal.

Monica falls asleep.

Ryan wakes up and completes his withdrawal.

Monica wakes up and completes her withdrawal. Big problem! In between the time when she checked the balance and made the withdrawal, Ryan woke up and pulled money from the account.

Monica's check of the account was not valid, because Ryan had already checked and was in the middle of making a withdrawal.

Monica must be stopped from getting into the account until Ryan wakes up and finishes his transaction. And vice-versa.

Need a **lock** for account access

- We need to make the `makeWithdrawal()` method to run as one **atomic** thing
  - Make sure that once a thread enters `makeWithdrawal()`, *it must be allowed to finish the method before any other thread can enter*
  - The **`synchronized`** keyword means that a thread needs a **key** in order to access the synchronized code
    - To protect your **data**, synchronize the methods that act on that data

`Public `**`synchronized`**` void makeWithdrawal(int amount) {`
…

Using an object's lock

- Every object has a *lock*
  - Most of the time, the lock is unlocked, and you can imagine a virtual key sitting with it
- Object locks come into play when there are one or more synchronized methods

  A thread can enter a synchronized method only if the thread can get the key to the object's lock

Using an object's lock

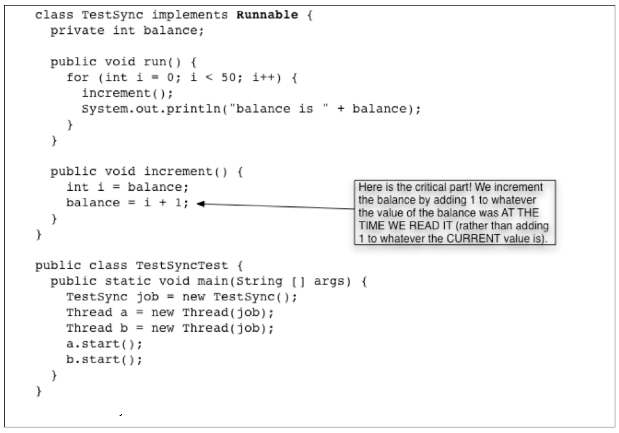
- The locks are not per *method*, they are per *object*
- If an object has two synchronized methods…
  - Not simply the case that you can't have two threads entering the **same** method
  - It means you can't have two threads entering **any** of the synchronized methods
- If you have multiple methods that can potentially act on instance variables, all those methods need to be protected with synchronized
  - *The goal of synchronization is to protect critical data*

The dreaded "Lost Update" problem

- Classic concurrency problem from the database world
  - Closely related to the Ryan and Monica example
- Revolves around one process:
  - Step 1: Get the balance of the account
    `int i = balance;`
  - Step 2: Add 1 to that balance
    `balance = i + 1;`
  - Force the computer to take two steps to complete the change
    - Of course, we'd normally do this via `balance++;`

The dreaded "Lost Update" problem

- By forcing *two* steps, the process becomes non-atomic
  - We could imagine that the two steps involved are more complex and couldn't be done in one statement
- In the "Lost Update" problem we have two threads, both trying to increment the balance



```
class TestSync implements Runnable {
  private int balance;

  public void run() {
    for (int i = 0; i < 50; i++) {
      increment();
      System.out.println("balance is " + balance);
    }
  }

  public void increment() {
    int i = balance;
    balance = i + 1;
  }
}

public class TestSyncTest {
  public static void main(String [] args) {
    TestSync job = new TestSync();
    Thread a = new Thread(job);
    Thread b = new Thread(job);
    a.start();
    b.start();
  }
}
```

Here is the critical part! We increment the balance by adding 1 to whatever the value of the balance was AT THE TIME WE READ IT (rather than adding 1 to whatever the CURRENT value is).

Lost Update

- The problem occurs when one of the threads is sent back to runnable *after* reading the value of balance but *before* writing it
  - Any updates from another thread are lost when this one returns to complete its update
- Solution: make the **`increment()`** method atomic by synchronizing it

Synchronize the bare minimum

- A good rule of thumb is to synchronize only the bare minimum that should be synchronized

```
public void go() {
  doStuff();

  synchronized(this) {
    criticalStuff();
    moreCriticalStuff();
  }
}
```

doStuff() doesn't need to be synchronized, so we don't synchronize the whole method.

Now only these two method calls are grouped into one *atomic* unit. When you use the synchronized keyword WITHIN a method, rather than in a method declaration, you have to provide an argument that is the object whose key the thread needs to get.

The deadly side of synchronization

- All it takes for a deadlock are two objects and two threads
- A simple deadlock scenario:
  1. Thread A enters a synchronized method of object *foo*, and gets the key
     - Thread A goes to sleep holding the *foo* key
  2. Thread B enters a synchronized method of object *bar*, and gets the key
     - Thread B tries to enter a synchronized method of object *foo,* but can't get **that** key
     - B **waits** until the *foo* key is available
     - B **keeps** the *bar* key

The deadly side of synchronization

- A simple deadlock scenario cont.
  3. Thread A wakes up and tries to enter a synchronized method on object *bar,* but can't get **that** key because B has it
     - A **waits** until the *bar* key is available (never will be)

  Thread A can't run until it gets the bar key and thread B can't run until it gets the foo key