# Threads and Concurrency

Lecture 21 – CS211 – Summer 2008

## Final exam

- Two hour exam, 8:00am-10:00am
- Cumulative exam
  - Will cover all material presented in course
  - Special emphasis on post-prelim material
  - Topics presented this week will be on final, but we won't expect you to know them in as much detail



## What is a Thread?

- A separate process that can perform a task independently, concurrently with other threads
  - Most programs have only one thread
  - GUIs have a separate thread, the *event dispatching thread*
  - A program can have many threads
  - You can create new threads in Java



## What is a Thread?

- In reality, threads are an illusion
  - The processor shares its time among all the active threads
  - Implemented with support from underlying operating system or virtual machine
  - Gives the illusion of several threads running simultaneously



## Concurrency (aka Multitasking)

- Refers to situations in which several threads are running simultaneously
- Special problems arise
  - race conditions
  - deadlock



- The operating system provides support for multitasking
- In reality there is one processor doing all this
- But this is an illusion too – at the hardware level, lots of multitasking
  - memory subsystem
  - video controller
  - buses
  - instruction prefetching

## Threads in Java

- Threads are instances of the class **`Thread`**
  - can create as many as you like
- The Java Virtual Machine permits multiple concurrent threads
  - initially only one thread (executes **`main`**)
- Threads have a priority
  - higher priority threads are executed preferentially
  - a newly created **`Thread`** has initial priority equal to the thread that created it (but can change)



## Creating a new Thread (Method 1)

```
class PrimeThread extends Thread {
   long a, b;

   PrimeThread(long a, long b) {
      this.a = a; this.b = b;
   }

   public void run() {
      //compute primes between a and b
      ...
   }
}
```

overrides **`Thread.run()`**

can call **`run()`** directly – calling thread will run it

```
PrimeThread p = new PrimeThread(143, 195);
p.start();
```

or, can call **`start()`** – will run **`run()`** in new thread



## Creating a new Thread (Method 2)

```
class PrimeRun implements Runnable {
   long a, b;

   PrimeRun(long a, long b) {
      this.a = a; this.b = b;
   }

   public void run() {
      //compute primes between a and b
      ...
   }
}
```

```
PrimeRun p = new PrimeRun(143, 195);
new Thread(p).start();
```



## Example

```
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[Thread-0,5,main] 0
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
Thread[Thread-0,5,main] 1
Thread[Thread-0,5,main] 2
Thread[Thread-0,5,main] 3
Thread[Thread-0,5,main] 4
Thread[Thread-0,5,main] 5
Thread[Thread-0,5,main] 6
Thread[Thread-0,5,main] 7
Thread[Thread-0,5,main] 8
Thread[Thread-0,5,main] 9
```



## Example

```
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      currentThread().setPriority(4);
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
Thread[Thread-0,4,main] 0
Thread[Thread-0,4,main] 1
Thread[Thread-0,4,main] 2
Thread[Thread-0,4,main] 3
Thread[Thread-0,4,main] 4
Thread[Thread-0,4,main] 5
Thread[Thread-0,4,main] 6
Thread[Thread-0,4,main] 7
Thread[Thread-0,4,main] 8
Thread[Thread-0,4,main] 9
```



## Example

```
public class ThreadTest extends Thread {

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }

   public void run() {
      currentThread().setPriority(6);
      for (int i = 0; i < 10; i++) {
         System.out.format("%s %d\n",
            Thread.currentThread(), i);
      }
   }
}
```

```
Thread[main,5,main] 0
Thread[main,5,main] 1
Thread[main,5,main] 2
Thread[main,5,main] 3
Thread[main,5,main] 4
Thread[main,5,main] 5
Thread[Thread-0,6,main] 0
Thread[Thread-0,6,main] 1
Thread[Thread-0,6,main] 2
Thread[Thread-0,6,main] 3
Thread[Thread-0,6,main] 4
Thread[Thread-0,6,main] 5
Thread[Thread-0,6,main] 6
Thread[Thread-0,6,main] 7
Thread[Thread-0,6,main] 8
Thread[Thread-0,6,main] 9
Thread[main,5,main] 6
Thread[main,5,main] 7
Thread[main,5,main] 8
Thread[main,5,main] 9
```

## Example

```
public class ThreadTest extends Thread {
   static boolean ok = true;

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.println("waiting...");
      }
      ok = false;
   }

   public void run() {
      while (ok) {
         System.out.println("running...");
      }
      System.out.println("done");
   }
}
```

```
running...
running...
running...
running...
running...
running...
running...
running...
running...
running...
running...
running...
running...
running...
running...
running...
running...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
done
```

## Example

```
public class ThreadTest extends Thread {
   static boolean ok = true;

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.println("waiting...");
         yield();
      }
      ok = false;
   }

   public void run() {
      while (ok) {
         System.out.println("running...");
         yield();
      }
      System.out.println("done");
   }
}
```

allows other waiting threads to run

```
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
done
```



## Example

```
public class ThreadTest extends Thread {
   static boolean ok = true;

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.println("waiting...");
         yield();
      }
      ok = false;
   }

   public void run() {
      while (ok) {
         System.out.println("running...");
         yield();
      }
      System.out.println("done");
   }
}
```

allows other waiting threads to run

```
running...
waiting...
running...
running...
waiting...
running...
waiting...
waiting...
running...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
done
```



## A problem with threads

- Threads are abstraction that make some programs much easier to program
- But the programmer gives up some control
  - The O/S and JVM control when threads are scheduled, not the program
  - A multi-threaded program might run differently every time it is run!
  - Debugging can be very difficult.
  - Finding all bugs is virtually impossible. Buggy code might fail very infrequently.



## Stopping Threads

- Threads normally terminate by returning from their run method
- **`stop()`**, **`interrupt()`**, **`suspend()`**, **`destroy()`**, etc. all deprecated
  - can leave application in an inconsistent state
  - inherently unsafe
  - don't use them
  - instead, set a variable telling the thread to stop itself



## Daemon and Normal Threads

- A thread can be *daemon* or *normal*
  - the initial thread (the one that runs **`main`**) is normal
- Daemon threads are used for minor or background tasks (e.g. timers, sounds)
- A thread is initially a daemon iff its creating thread is
  - but this can be changed
- The application halts when either
  - **`System.exit(int)`** is called, or
  - all normal (non-daemon) threads have terminated

## Race Conditions

- A *race condition* can arise when two or more threads try to access data simultaneously
- Thread B may try to read some data while thread A is updating it
  - updating may not be an atomic operation
  - thread B may sneak in at the wrong time and read the data in an inconsistent state
- Results can be unpredictable!



## Example – A Lucky Scenario

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   if (stack.isEmpty()) return;
   String s = stack.pop();
   //do something with s...
}
```

Suppose threads A and B want to call **`doSomething()`**, and there is one element on the stack

1. thread A tests **`stack.isEmpty()`** ⇒ false
2. thread A pops ⇒ stack is now empty
3. thread B tests **`stack.isEmpty()`** ⇒ true
4. thread B just returns – nothing to do



## Example – An Unlucky Scenario

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   if (stack.isEmpty()) return;
   String s = stack.pop();
   //do something with s...
}
```

Suppose threads A and B want to call **`doSomething()`**, and there is one element on the stack

1. thread A tests **`stack.isEmpty()`** ⇒ false
2. thread B tests **`stack.isEmpty()`** ⇒ false
3. thread A pops ⇒ stack is now empty
4. thread B pops ⇒ Exception!



## Solution – Locking

```
private Stack<String> stack = new Stack<String>();

public void doSomething() {
   synchronized (stack) {
      if (stack.isEmpty()) return;
      String s = stack.pop();
   }
   //do something with s...
}
```

**`synchronized`** block

- Put critical operations in a **`synchronized`** block
- The **`stack`** object acts as a lock
- Only one thread can own the lock at a time



## Solution – Locking

- You can lock on any object, including **`this`**

```
public synchronized void doSomething() {
   ...
}
```

is equivalent to

```
public void doSomething() {
   synchronized (this) {
      ...
   }
}
```



## File Locking

- In file systems, if two or more processes could access a file simultaneously, this could result in data corruption
- A process must *open* a file to use it – gives exclusive access until it is *closed*
- This is called *file locking* – enforced by the operating system
- Same concept as **`synchronized(obj)`** in Java

## Deadlock

- The downside of locking – *deadlock*
- A *deadlock* occurs when two or more competing threads are waiting for the other to relinquish a lock, so neither ever does
- Example:
  - thread A tries to open file X, then file Y
  - thread B tries to open file Y, then file X
  - A gets X, B gets Y
  - Each is waiting for the other forever



## Thread synchronization

- What if thread A needs to wait for B to finish a task?

```
public class ThreadTest extends Thread {
    static char turn = 'A';

    public static void main(String[] args) {
        new ThreadTest().start();
        while(true) {
            while(turn != 'A') { }
            System.out.println("waiting...");
            turn = 'B';
        }
    }

    public void run() {
        while (true) {
            while(turn != 'B') { }
            System.out.println("running...");
            turn = 'A';
        }
    }
}
```

- One solution: spin locks
- Use a shared variable to store whose turn it is
- Threads do nothing (spin) until it's their turn
- Downside: spinning wastes processor time



## `wait/notify`

- Another option: wait/notify statements

```
public class ThreadTest extends Thread {

    public static void main(String[] args) {
        Thread t = new ThreadTest(); t.start();
        while(true) {
            synchronized(t) {
                System.out.println("waiting...");
                t.notify();
                t.wait();
            }
        }
    }

    public synchronized void run() {
        try {
            while (true) {
                System.out.println("running...");
                notify();
                wait();
            }
        } catch(Exception e) {}
    }
}
```

- wait() and notify() provide a rudimentary mechanism for passing messages
- wait() pauses execution until someone calls notify() on the same object
- More efficient: doesn't waste processor cycles



## Conclusion

- Threads are an important abstraction
  - Multi-threaded programs are very common
- But programming with threads is difficult
  - Program execution becomes nondeterministic
  - Subtle bugs involving race conditions, etc.
  - Use synchronization primitives to avoid these problems
- You'll learn much more about threads in later CS courses