CMSC 202H

Threads

Principle of Multitasking: Processes and Threads

- Core concept: user wants to have multiple, simultaneous flows-of-control in their computation: “do several things at once”
  - If they are independent applications, we call the units “processes”
  - If they are parallel execution streams within a shared application context, they are called “threads” (sometimes referred to as “lightweight processes”)
  - Originally, concurrency of these execution flows was simulated: “context-switched”
  - Modern systems actually support true parallelism in hardware (e.g.: “dual-core processors”)

Why Threads are Useful

There are several situations in which multi-threading is a useful model:

- User prefers to launch multiple tasks simultaneously at outset, instead of sequentially with waits

…or:

- Some tasks “block” (like I/O)—would like to continue other, independent computations in the meantime

…or:

- Parallel, coordinated tasks is a more intuitive model to implement

…or…

Threads: Simple vs. Complex

- In simplest form, threads are easy:
  - Start multiple, independent tasks, then wait for all to finish
- Trying to coordinate tasks quickly makes things very complex:
  - Need communication/coordination constructs
  - Need to control concurrent access to shared resources

Most of these issues exist *even if multitasking is only simulated!*

Example Complication: Race Conditions

Thread 1:

```
// x == 1 at start
y = x + 1;
x = y;
```

Thread 2:

```
y = x + 1;
x = y;

// What are results?
```

Example Complication: Race Conditions

Thread 1:

```
// x == 1 at start
y = x + 1;
x = y;
```

Thread 2:

```
y = x + 1;
x = y;
// x == 3 now
```

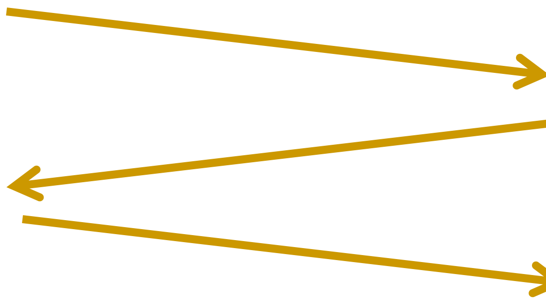
Example Complication: Race Conditions

Thread 1:

// x == 1 at start

y = x + 1;

x = y;



Thread 2:

y = x + 1;

x = y;

// But x == 2 now!

Example Thread Application: GUIs

- Need widgets to be independently responsive even though your application's flow-of-control continues
- You might want to be responsive to widget events even through a long computation
- By default, Swing event handling is single-stream, but you might want it to be parallel-processing

Java is Inherently Threaded

- Even for a simple application with one class and a simple main(), using threads:
  - main() is invoked from a foreground—or user—thread
  - Garbage collection is implemented as a background—or “daemon”—thread

    (Bonus question: do you know what the difference between a “daemon” and “demon” is?)

Creating Your Own Threads—Method A

- A simple way to create a thread is:

1. Create an instance of a **`Runnable`** type object
   - **`Runnable`** is an interface, with one method: **`public void run();`**
2. Instantiate a **`Thread`** class, passing your **`Runnable`** instance to the constructor
3. Invoke the new **`Thread`** instance's **`start()`** method, which will do some setup, then invoke your **`Runnable`**'s **`run()`** method

Creating Your Own Threads—Method A

```
public class MyRunnable implements Runnable {
  public void run() {
    for (int i = 0; i < 1000000; i++) {
      long j = i * i;
    }
    System.out.println("Done with thread!");
  }

  public static void main(String args[]) {
    Runnable task = new MyRunnable();
    Thread otherThread = new Thread(task);
    otherThread.start();
    // Following will likely be output before above
    System.out.println("Main thread here");
  }
}
```

Creating Your Own Threads—Method B

- The other way to create a thread is:

1. Extend the `Thread` class, overriding the `run()` method (let's call the new class `MyThread`)
2. Instantiate your new `Mythread` class, calling the no-arg constructor
3. Invoke the new `MyThread` instance's `start()` method, which will do some setup, then invoke its overriding `run()` method

Creating Your Own Threads—Method B

```
public class MyThread extends Thread {
  public void run() {
    for (int i = 0; i < 1000000; i++) {
      long j = i * i;
    }
    System.out.println("Done with thread!");
  }

  public static void main(String args[]) {
    Thread otherThread = new MyThread();
    otherThread.start();
    // Following will likely be output before above
    System.out.println("Main thread here");
  }
}
```

Threads: Advanced Topics

There are many additional facets to threaded programming, which we cannot cover here:

- Scheduling:
  - Pre-emptive scheduling, priorities
- Memory:
  - Race conditions, and thread-safe code
- Synchronization:
  - Locks, deadlocks