17 - Threads

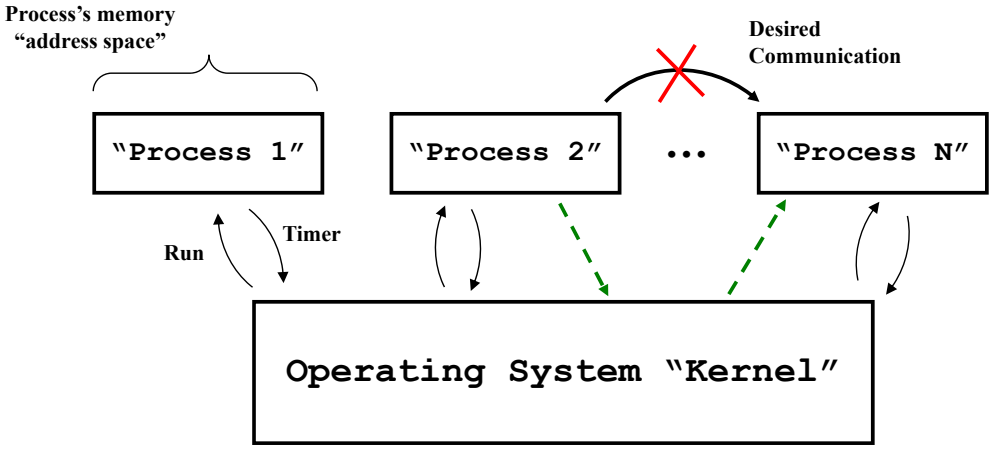
Computer Science Department
California State University, Sacramento

Overview

- **Threads vs. Processes**
- **Java/CN1 Threads**
  - **Thread Class**
  - **Runnable Interface**
- **Thread Synchronization**
- **Application Uses**

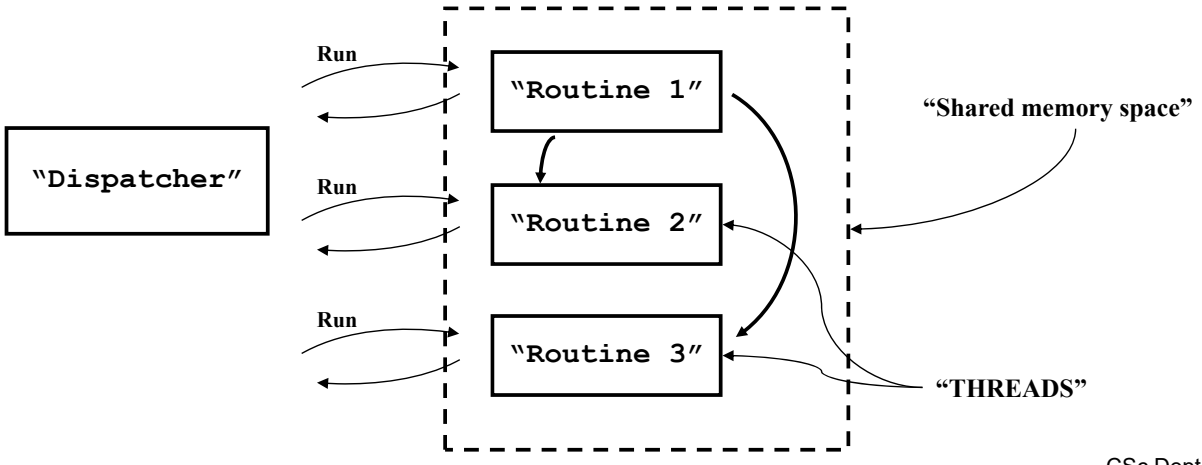
Threads vs. Processes

- OS shares CPU between "processes"
  - Processes cannot access outside their own "address space"
  - Processes can only communicate via kernel-controlled mechanisms



Threads vs. Processes

- Sometimes we *want* to allow separate pieces of code to communicate
  - Put them in the same memory space
  - Provide a mechanism to run each one independently
  - Allow the *programmer* to worry about "conflicts"



Java/CN1 Threads

- Two creation mechanisms:
  - Extend class **`Thread`**
  - Construct a thread from an object that implements Interface **`Runnable`**

```
public interface Runnable
{
   void run();
}
```

```
public class MyClass implements Runnable {
   public void run() {
      ...
   }
   ...
}
...
Runnable r1 = new MyClass();
Thread t1 = new Thread(r1);
t1.start();
```


Example 1: Counter Thread

```
/** This class constructs a separate thread running a
 *  "Counter" object, starts that thread running, and
 *  waits until the thread completes before terminating.
 */

public class CounterTest {

   public CounterTest() {

      // Create a "Runnable" object
      Runnable r1 = new Counter(25);

      // Construct an instance of Thread, passing the
      // Runnable as the code to be run by the Thread.
      Thread t1 = new Thread(r1);

      // Start the thread running
      t1.start();
   }
}
```

Example 1: Counter Thread (cont.)

```
public class Counter implements Runnable {
  private int loopLimit;
  private Random rand;

  public Counter(int loopLimit) {
    this.loopLimit = loopLimit;
    rand = new Random();
  }

  // Specify the runnable (thread) behavior.
  public void run() {
    for (int i=1; i<=loopLimit; i++) {
      System.out.println(i);       // display current loop count
      pause(rand.nextFloat());     // sleep for up to 1 second
    }
  }

  private void pause(double seconds) {
    try {
      Thread.sleep(Math.round(1000.0*seconds));
    }
    catch(InterruptedException ie) {
      System.err.println ("Sleep interrupted");
    }
  }
}
```


Example 2: Concurrent Output

```
public class ConcurrentOutput {

  public ConcurrentOutput() {

    Runnable r1 = new Counter(25);
    Thread t1 = new Thread(r1);
    t1.start();

    // cause some output from main program
    for (int i=0; i<20; i++) {
      try {
        Thread.sleep(500);
      }
      catch (Exception e) {
        System.err.println ("Sleep interrupted");
      }
      System.out.println ("*****");
    }
    System.out.println ("Main: done.");
  }
```

Example 2: Concurrent Output (cont.)

```
public class Counter implements Runnable {
  ... (initialization here -- same as before)
  public void run() {
    for (int i=1; i<=loopLimit; i++) {
      System.out.println(i);
      pause(rand.nextFloat());
    }
    System.out.println ("Counter: done.");
  }
  private void pause(double seconds) {
    ... as before
  }
}
```


Example 3: Multiple User Threads

```
public class MultipleCounters {
  public MultipleCounters() {
    /* Create multiple "Runnable" objects */
    Runnable r1 = new Counter2(8);
    Runnable r2 = new Counter2(8);
    Runnable r3 = new Counter2(8);
    /* Create threads for each runnable */
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);
    Thread t3 = new Thread(r3);
    /* Start the threads running  */
    t1.start();
    t2.start();
    t3.start();
  }
}
```

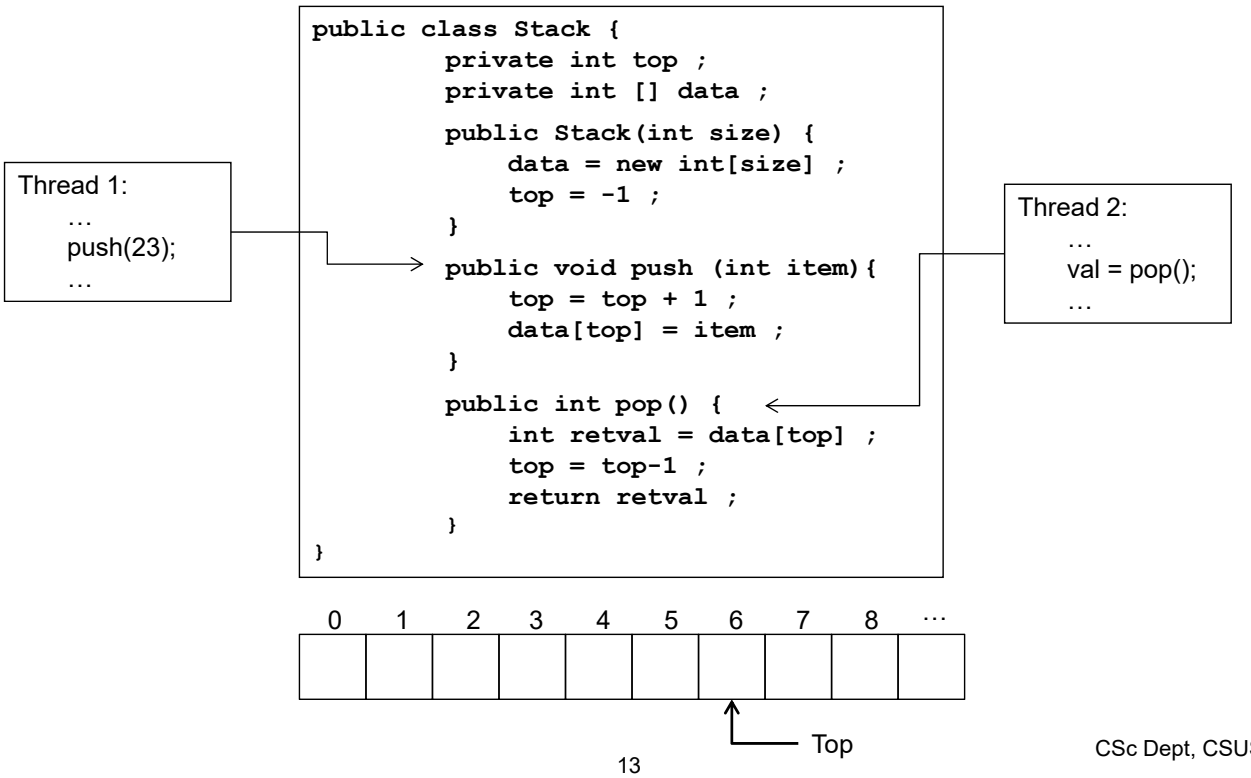
Multiple User Threads (cont.)

```
public class Counter2 implements Runnable {
  private static int totalCounters = 0;    //counts instances
  private int myNum, loopLimit;
  private Random rand;
  public Counter2(int loopLimit) {
    this.loopLimit = loopLimit;
    myNum = totalCounters++;   //assign this instance a unique number
    rand = new Random();
  }
  public void run() {
    for(int i=1; i<=loopLimit; i++) {
      System.out.println("Counter " + myNum + ": " + i);
      pause(rand.nextFloat());
    }
  }
  private void pause(double seconds) { ... }        // as before
}
```


Thread Synchronization

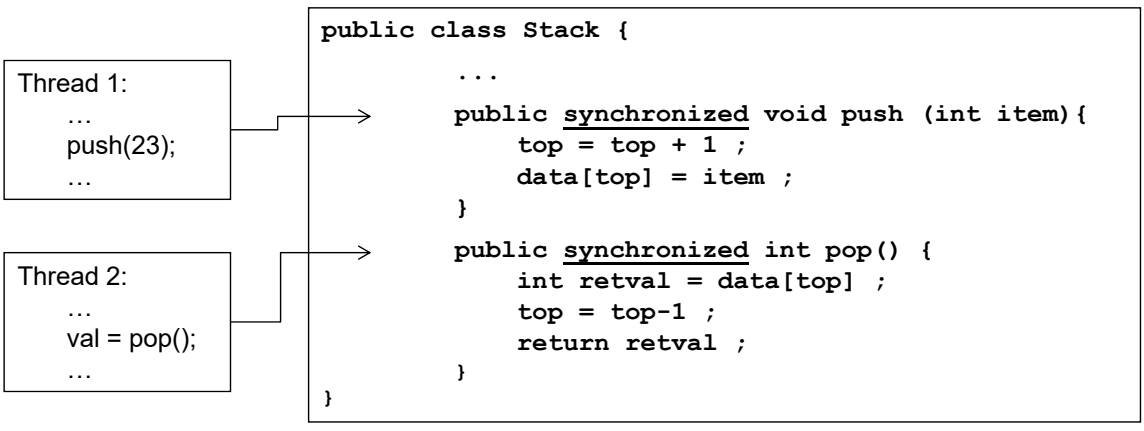
- Parallel execution can lead to problems
  - Corruption of shared data
  - Race conditions
  - Deadlock
  - Starvation

Example: Data Corruption



Java/CN1 Thread Synchronization

Example: Synchronized Methods




```
//Another example: synchronized code blocks
public class SomeClass {
   public void someMethod() {
      ...
       synchronized (this) { // a block of code synchronized on this object, note
                             // that instead of "this" you can provide an
                             // arbitrary object
         ... // arbitrary code here
       }
   }
}
```

Other Important Thread Methods

- **`sleep()`**
  - forces current thread to stop for a specified amount of time
- **`yield()`**
  - forces current thread to give up control to threads of equal priority
- **`currentThread()`**
  - returns the currently executing thread
- **`join()`**
  - E.g. myOtherThread.join() blocks THIS thread until myOtherThread dies (finishes). Hence, it forces a "sync point" where the threads 'join together'.

Common Thread Uses

- Update vs. Display of Game Worlds
- Event Handling
- Image Loading
- Audio File Playing