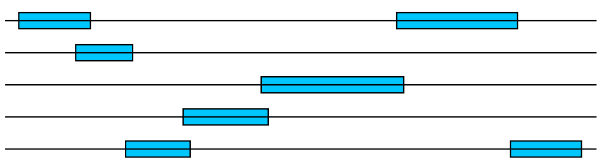
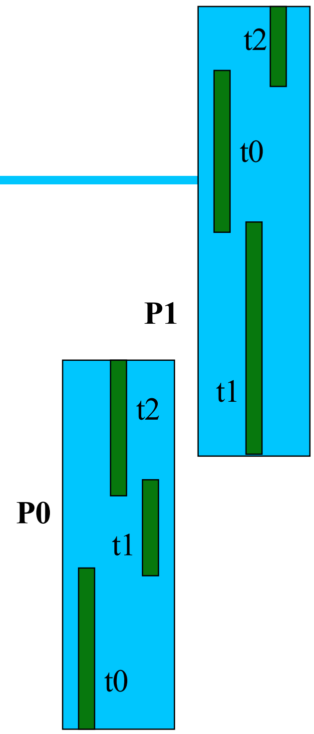
Multi-Threaded Programming in JAVA





In the Beginning…

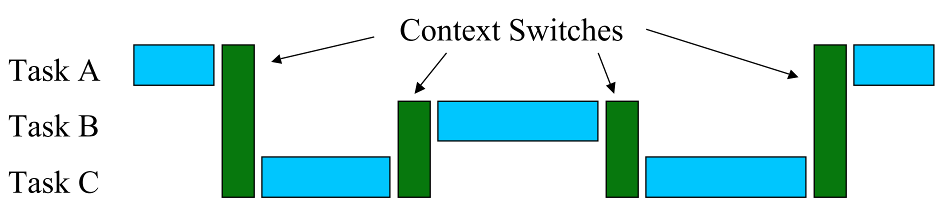
- Computers ran a single task at a time…
  - Punch cards were placed into a feeder.
  - The cards were then read, compiled and run.
- Batch processing extended this…
  - Groups of punch cards could be run one after the next.
  - This increases return of investment in the hardware.

Then Came the Operating System

- The OS sits between programs and the hardware.
- Contributions include:
  - Uniform interaction between hardware
    - I/O abstractions (e.g., filesystems)
    - Standardized interaction libraries (e.g., libc)
  - Multi-user Capabilities
    - Memory management and protection (virtual address space)
    - Scheduling and time sharing

Multiple Users ~ Multiple Tasks

- Hardware is shared between many users
- Each user can run multiple tasks
  - Better return on investment… just like batch systems
  - Processor is idle less often
  - Some time is “wasted” switching tasks



Copyright 1999-2002 Simon Lok Reproduction and/or redistribution in whole or part without written authorization is expressively prohibited

Multi-Tasking

- K users share the hardware running N tasks
- Tasks are time-sliced quickly to give the illusion that all tasks are running in parallel
- Each task thinks it's the only one on the machine
- Cooperative Multi-Tasking: (Win3.1, MacOS)
  - Each process yield the processor when it feels fit
- Pre-Emptive Multi-Tasking: (UNIX)
  - The OS scheduler decides who should run when

Multi-Threading

- If each user can run many tasks…
  - Why can't each task have many "sub-tasks"?
  - This is usually called multi-threading.
- Threads are like "lightweight" tasks…
  - Scheduling for execution is pretty much the same
- Differences include:
  - They share the same memory space.
  - They may not have as much OS overhead.

Copyright 1999-2002 Simon Lok Reproduction and/or redistribution in whole or part without written authorization is expressively prohibited

Why do we want threads?

- Why multi-user / multi-tasking?
  - Processor is idle less, people can share a computer.
  - Better return on hardware investment
- We use threads for somewhat similar reasons:
  - Make sure processors are fully utilized
    - Don't block on I/O
    - True parallel execution on multiprocessor hardware
  - Other cool things
    - Games: intelligent user agents, animation
    - Automatic garbage collection, hidden from the user

How does the JVM affect this?

- Each JVM is a separate task on the native OS
- Most JVMs run a single JAVA program
- Each JVM (JAVA program) has many threads
  - In the simplest case, the GC and your main thread
  - JVM threads to OS interaction depends on JVM
  - The newest JVMs (e.g., Sun HotSpot) will take advantage of physical multiprocessor hardware

Copyright 1999-2002 Simon Lok Reproduction and/or redistribution in whole or part without written authorization is expressively prohibited

Programming Threads in JAVA

- Two ways it can be done
  - Create a class, extend Thread
    - Override the run() method
    - Instantiate the class
    - Call start()
  - Create a class, implement Runnable
    - Implement the run() method
    - Instantiate your class
    - Instantiate a Thread class, pass your class in constructor
    - Call start()

A Simple Thread Example

```
public class TwoThreadsExample {
   public TwoThreadsExample() {
      (new SimpleThread("First Thread")).start();
      (new SimpleThread("Second Thread")).start();
   }
   private class SimpleThread extends Thread {
      public SimpleThread(String str) { super(str); }
      public void run() {
         for (int i = 0; i < 10; i++) {
           System.out.println(getName() + " says " + i);
           try{ sleep((long)(Math.random() * 1000)); }
           catch (InterruptedException e) {}
         }
         System.out.println(getName() + " is done.");
      }
   }
   public static void main (String[] args) {
      new TwoThreadsExample();
   }
}
```

TwoThreadsTest Output

simon@sltpt20: /misc/ud/classes/java/codelib/threads

```
[simon@sltpt20 threads]$ java TwoThreadsExample
First Thread says 0
Second Thread says 0
First Thread says 1
First Thread says 2
Second Thread says 1
Second Thread says 2
First Thread says 3
First Thread says 4
Second Thread says 3
Second Thread says 4
First Thread says 5
First Thread says 6
First Thread says 7
First Thread says 8
Second Thread says 5
First Thread says 9
First Thread is done.
Second Thread says 6
Second Thread says 7
Second Thread says 8
Second Thread says 9
Second Thread is done.
[simon@sltpt20 threads]$
```


I/O Blocking Example

- We want to serve many clients using sockets
  - Each client that connects is serviced by a thread
  - This provides “parallel” service to may clients
- Two components, **`Listener`** and **`Handler`**
  - The **`Handler`** implements **`Runnable`**
  - The client servicing is done in the **`Handler`**
  - The **`Listener`** spawns **`Handler`**s using the new keyword and wrapping **`Handler`**s inside **`Thread`**s
  - Try/catch blocks are missing from this code

The Listener

```
public class Listener {
public static void main(String[] args) {
  ServerSocket srvSock = new ServerSocket(4567);
  while (keepRunning) {
    // when we get a connection, spawn off a
    // thread to handle it… this means we can
    // keep listening for other connections
    // while the first client is serviced
    Socket conn = srvSock.accept();
    (new Thread(new sockHandler(conn))).start();
  }
}
}
```


The Handler

```
public class sockHandler implements Runnable {
   private Socket conn = null;
   public sockHandler(Socket conn) {
      this.conn = conn;
   }
   public void run() {
     InputStreamReader ISR = new
        InputStreamReader(conn.getInputStream());
     BufferedReader fromClient = new
        BufferedReader(ISR);
     OutputStreamReader OSR = new
        OutputStreamReader(conn.getOutputStream());
     PrintWriter toClient = new
        PrintWriter(OSR);
     // DO CLIENT SERVICING HERE
   }
}
```

Data Parallel Programming

- We spawn off many threads to estimate PI
- As each thread completes, we update our estimate
- If we were running on MP hardware with a Hotspot JVM, these threads would run on separate processors and harness true parallelism
- Notice that the threads share a single memory space… that's why we can communicate between the sub-tasks and controller without RMI

PI Estimation Task Thread

```
public class PiEstimatorTask extends Thread {
  private EstimatePi Parent = null;
  private static final int iterations = 100000;
  public PiEstimatorTask(EstimatePi Parent) {
    this.Parent = Parent;
  }
  public void run() {
    int in = 0, out = 0;
    for (int i = 0; i < iterations; i++) {
       double x=2.0*Math.random()-1.0, y=2.0*Math.random()-1.0;
        if ((Math.sqrt(x*x+y*y) < 1.0)) { in++; } else { out++; }
     }
     double estimate = 4.0 * (double)in / (double)iterations;
     Parent.updateEstimate(estimate);
  }
}
```

PI Estimation Control Program

```
public class EstimatePi {
  private double pi = 0.0;
  private final int numTasks = 12; // one for each processor
  private int allFinished = 0;
  private long starttime = 0;

  public synchronized void updateEstimate(double est) {
    long rt = System.currentTimeMillis() - starttime;
    System.out.println("Terminated at " + rt + " ms, est " + est);
    pi = (allFinished == 0) ? est : (pi + est) / 2;
    allFinished++;
  }

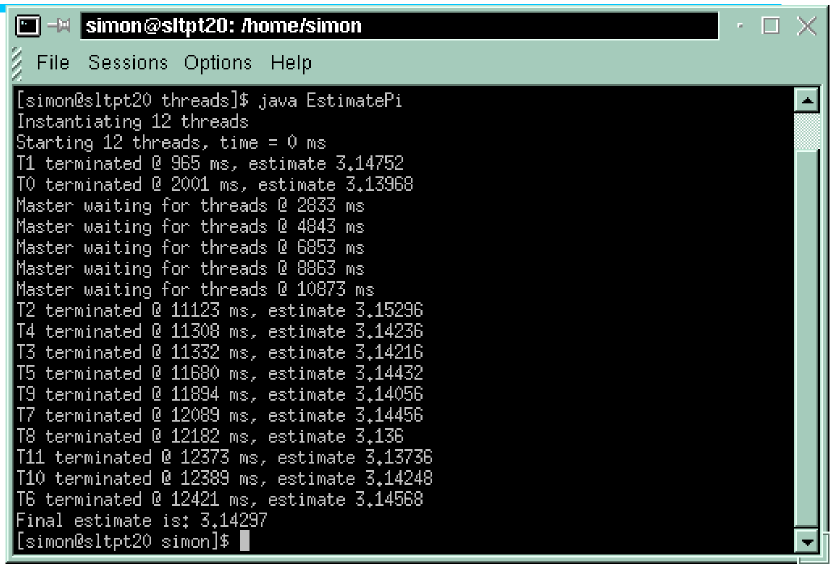
  public double getPi() { return pi; }


  // . . . continued on next slide . . .
```


PI Estimation Control Program

```
  public void go() {
    PiEstimatorTask[] PiTasks = new PiEstimatorTask[numTasks];
    System.out.println("Instantiating " + numTasks + " threads");
    for (int i = 0; i < numTasks; i++) {
      PiTasks[i] = new PiEstimatorTask(this);
    }
    starttime = System.currentTimeMillis();
    System.out.println("Starting threads, time = 0 ms");
    for (int i = 0; i < numTasks; i++)
      (PiTasks[i]).start();
    } // FIXME: try/catch InterruptedExeception below
    while(allFinished < numTasks) { Thread.sleep(1000); }
  }
  public static void main(String[] args) {
    EstimatePi MCP = new EstimatePi();
    MCP.go();
    System.out.println("Final estimate is: " + MCP.getPi());
  }
}
```

PI Estimation Output (Uniproessor Hardware)



```
simon@sltpt20: /home/simon
File  Sessions  Options  Help
[simon@sltpt20 threads]$ java EstimatePi
Instantiating 12 threads
Starting 12 threads, time = 0 ms
T1 terminated @ 965 ms, estimate 3.14752
T0 terminated @ 2001 ms, estimate 3.13968
Master waiting for threads @ 2833 ms
Master waiting for threads @ 4843 ms
Master waiting for threads @ 6853 ms
Master waiting for threads @ 8863 ms
Master waiting for threads @ 10873 ms
T2 terminated @ 11123 ms, estimate 3.15296
T4 terminated @ 11308 ms, estimate 3.14236
T3 terminated @ 11332 ms, estimate 3.14216
T5 terminated @ 11680 ms, estimate 3.14432
T9 terminated @ 11894 ms, estimate 3.14056
T7 terminated @ 12089 ms, estimate 3.14456
T8 terminated @ 12182 ms, estimate 3.136
T11 terminated @ 12373 ms, estimate 3.13736
T10 terminated @ 12389 ms, estimate 3.14248
T6 terminated @ 12421 ms, estimate 3.14568
Final estimate is: 3.14297
[simon@sltpt20 simon]$
```


Implementing Runnable

- Change the PiEstimationTask Thread from extends Thread to implements Runnable
- Change the instantiation code in MCP to be:

```
Thread PiTasks[] = new Thread[numTasks];
for (int i = 0; i < numTasks; i++) {
  PiTasks[i] = new Thread(new (PiEstimatorTask(this));
}
```

- Using this approach is superior because the task child can extend a class that does useful abstraction rather than java.lang.Thread

Did you notice?

- The method `updateEstimate` in the control program is declared `synchronized`?
- This is JAVA's way of providing MutEx
  - MutEx is short for Mutual Exclusion
  - Only one person can go at a time
- When you have parallel processing, you will almost always run into MuTex problems

Synchronized Methods

- ```
  public void synchronized doSomething() {
     // guaranteed that only one Thread
     // will be running this method
     // at any time
  }
  ```
- If one thread is running this method, any other thread calling it will wait until the first thread has returned from the method.
- Note that this can slow things down tremendously! Use with caution.

Copyright 1999-2002 Simon Lok Reproduction and/or redistribution in whole or part without written authorization is expressively prohibited
  ```

Synchronized Blocks

```
public void someMethod() {
  synchronized(someObject) {
    // do some stuff
  }
}
```

- Only one thread can be running the block at any time
- The someObject is the data that is critical
- Fine grain atomicity
  - Generally results in better performance
  - Harder to create code
  - Harder to read code later on

Dead Locks

- If you have multiple piece of critical data:

```
synchronized(someObject) {
  synchronized(otherObject) {
    // critical section
  }
}
```

- Always gather the locks in the same order!
- If someplace else you get otherObject before someObject, you might end up with deadlock.

In Summary

- Multi-Threading enables use make better use of contemporary computers:
  - Prevents idle/busy waiting CPU
  - Non-blocking I/O
  - Parallel execution (on MP hardware)
  - Automatic garbage collection
  - Fun uses for games (animation / bad guys with AI)