

Introduction to Multi-Threaded Programming in Java

CS 351

Design of Large Programs

Instructor: **Joel Castellanos**

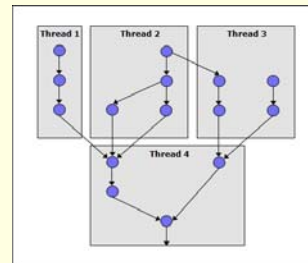
e-mail: joel@unm.edu

Web: <http://cs.unm.edu/~joel/>

Office: Farris Engineering
Center (FEC) room 321

Lab Instructor: **David Godinez**

e-mail: dgodinez@cs.unm.edu



1/28/2009

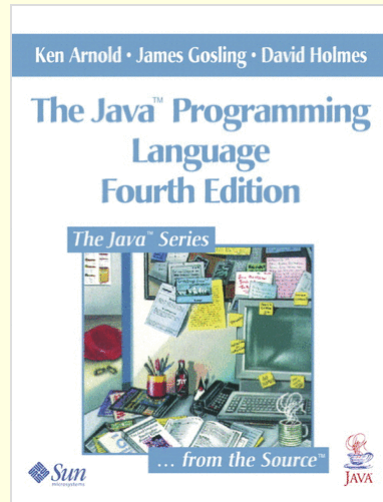
Quiz 1-1: Synchronous Updates

In the context of updating an iterative, time-step simulation, what does *synchronous* mean?

- a) That the algorithm is thread save.
- b) That the algorithm is *not* thread save.
- c) That the algorithm uses an update mechanism to *acquire* and *release* a data lock.
- d) That updates running in different threads are synchronized to use the same time-step.
- e) That updates made on the same iteration have no effect on each other.

2

Chapter 14: Threads



- What you need for the snowflake project is only a small part of what is covered in this chapter.
- We will need more of what is in Chapter 14 for project 2.

3

Threads versus Processes

- Threads and Processes are both methods of parallelizing.
- Processes are independent execution units that contain their own state information, use their own address spaces, and only interact via interprocess communication:
 - Sockets,
 - TCP/IP,
 - Pipes,
 - Files (disk files, RAM files, hardware shared memory)
 - High-level, Remote Procedure Call (RPC) systems.
- Threads within a single process share the same address space. Hence, they can access the same global variables.
 - Threads have shared heaps, but separate stacks.

4

Processes/Threads - Heavy/Light

- Spawning a *processes* is heavy:
 - Start up has significant overhead.
 - Interprocess communication has significant overhead.
 - A spawned process can continue to run after the original process is killed.
- Spawning a *thread* is relatively light:
 - It is often efficient to spawn a thread for a short-term task such as performing a complex mathematical computation using parallelism or initializing a large matrix.
 - In Java, calling `System.exit(0)` in any thread, causes all threads running in that process to exit.

5

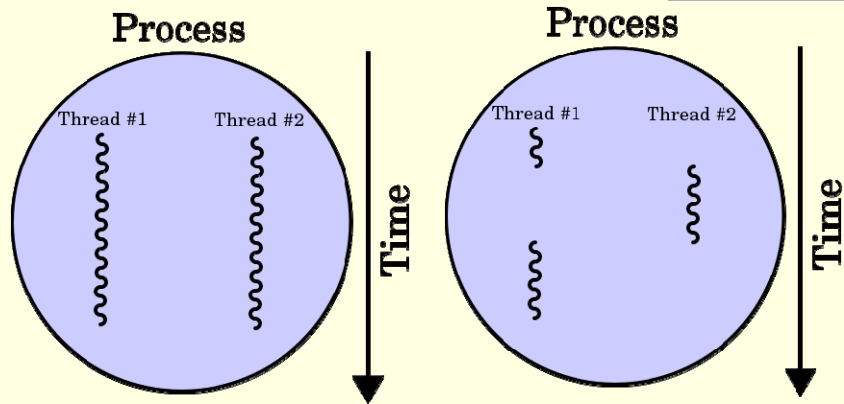
Native Threads Versus Green Threads

Two main ways of implementing threads:

- **Native threads** are implemented by the kernel.
 - Heavier because context switching at the kernel level is comparatively expensive.
 - Can take advantage of multiple processors.
 - Modern Java, C, C++, Matlab.
- **Green threads** are implemented at the interpreter or virtual machine level.
 - Lighter weight.
 - Can't take advantage of multiple CPUs.
 - MzScheme, Haskell, Smalltalk, Python.

6

Threads: Concurrent versus Swapped



With native threads, when there are more processors than threads, each runs *concurrently*.

When there are more threads than processors, the threads are *swapped* in and out of control. Swapping requires overhead.

7

Quiz 1-2: Java Threads

In the current versions of Java, threads are implemented:

- a) With context switching at the kernel level.
- b) Within the Java Virtual Machine (JVM).
- c) As *green* threads.
- d) As *black* threads.
- e) As *master* threads.

8

Categories of Thread Applications

1. A single, computationally intensive problem that can be divided into independent parts where each part requires significant CPU time.
 - Only makes sense on multiprocessor machine.
2. A computationally intensive problem that has a GUI or some other component that needs to be able to interrupt the main computation.
 - Used with single and multiprocessors.
3. A task that spends most of its time waiting for some resource.
 - Used with single and multiprocessors.

9

Static Methods of java.lang.Thread

These static methods can be use from any Java program without creating a Thread object.

- `public static void sleep(long millis)`
`throws InterruptedException`

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds (`millis`). The thread does not lose ownership of any monitors.

- `public static void yield()`

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

10

Work Thread: Keeping GUI Response

```
public static void main(String[] args)
{ ...
  WorkerThread worker = new WorkerThread();
  worker.run();
}

public class WorkerThread extends Thread
{ ...
  public void run()
  { for (;;)
    { if (myGuiPanel.isPaused())
      { try {Thread.sleep(500);}
        catch (InterruptedException e){}
      }
      else
      { ...
    }
  }
}
```

Simple Polling

Significant work,
but less than 500
milliseconds.

11

Volatile Keyword

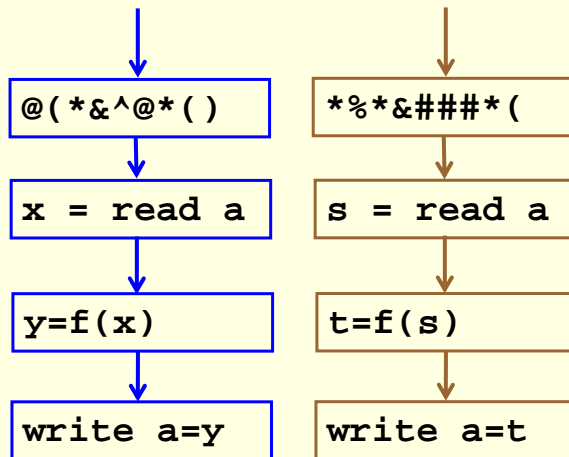
In Java, declaring a volatile variable means:

- The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory"
- Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

```
public class GUI_Panel extends JPanel
{ ...
  public volatile boolean
                        paused = true;
  ...
  public boolean isPaused() return paused;
}
```

12

Race Condition



Lock:

"I am using this object. You need to wait."

Polling Solution:

"Are you done yet?"

Notify Solution:

"I will just sleep until you wake me."

13

Quiz 1-3: Race Condition

In this loop, which calculations can be preformed in *different* threads without a race condition?

```
for (int i=0; i<1000000; i++)
1 { c = f(a) + f(c);
2   d = f(d+c);
3   e = f(e) + f(a) + f(e+a);
4   b = g(d);
}
```

- a) 1 and 2
- c) 1 and 4

- b) 1 and 3
- d) 2 and 4

14



"None of the Above"

```
for (int i=0; i<1000000; i++)  
1 { c = f(a) + f(c);  
2   d = f(d+c);  
3   e = f(e) + f(a) + f(e+a);  
4   b = g(d);  
}
```

Running line 3 in parallel with 1, 2 and 4 *requires* the *independence* of calls to methods **f** and **g**.

It *requires* that method **f** does not access global variables or other resources written to by **f** or **g**.

15

Not all Java JIT Compilers are Equal

- All must meet Java Language specifications.
- Some have *much* better optimization.
- IBM's Java JIT compiler performs particularly high.

```
for (int i=0; i<1000000; i++)  
1 { c = f(a) + f(c);  
2   d = f(d+c);  
3   e = f(e) + f(a) + f(e+a);  
4   b = g(d);  
}
```

Assuming independence can be verified, a good optimizer would parallelize line 3.

16

Quiz 1-4: Race Condition #2

In this loop, which calculations can be preformed in *different* threads without a race condition?

```
for (int i=0; i<1000000; i++)  
1 { x = g(x) + f(w);  
2   y = k(y+w);  
3   z = f(y) + h(v) + f(y+v);  
4   v = g(d);  
}
```

a) 1 and 2
c) 3 and 4

b) 2 and 3
d) 2 and 4