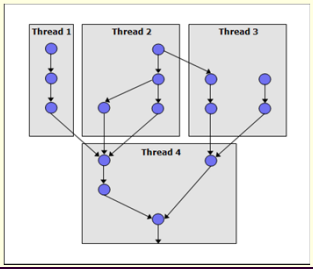
CS 351
Design of Large Programs
Introduction to Multi-Threaded Programming in Java

Instructor: **Joel Castellanos**
e-mail: joel@unm.edu
Web: http://cs.unm.edu/~joel/
Office: Farris Engineering Center (FEC) room 319



2/2/2017

Quiz: Creating Threads In Java

- In Java, there are two ways to have a class be able to start a new thread. What are they?
- In what situation would the one be used and in what situation would the other?

`extends Thread`

Software Engineering Principle: If you want a thread, default to extending **`Thread`** unless your class already extends something. This makes it clear to someone reading the code that the class is an instance of **`Thread`**.

`implements Runnable`

Personal Project Principle: If you want a thread, default to implementing runnable. That way, if later you want the class to extend something, there are less changes.

2

There is No Space in README

- For your personal files never to be moved from your own computer, it is fine to use spaces in file names.
- For any files you produce for school, work or sharing across computers, DO NOT USE SPACES IN THE FILE NAMES.
- Most command line programs (on Windows, Linux and MacOS) use spaces to delimit arguments.
- Many programs that move data from one system to another do not correctly handle file names with spaces.

3

Static

- Static fields DO NOT get "instantiated".
- Static is not "better" than an instance field, nor is an instance field "better" than a static field - any more than a tire is "better" than a clutch.
- Both static and instance fields can be public or private.
- A static initializer does NOT run when ever it is called from anywhere in the program.
- Static initializers are NOT called when the program is run.
- static ≠ final. Static variables can be changed (unless they are also final).

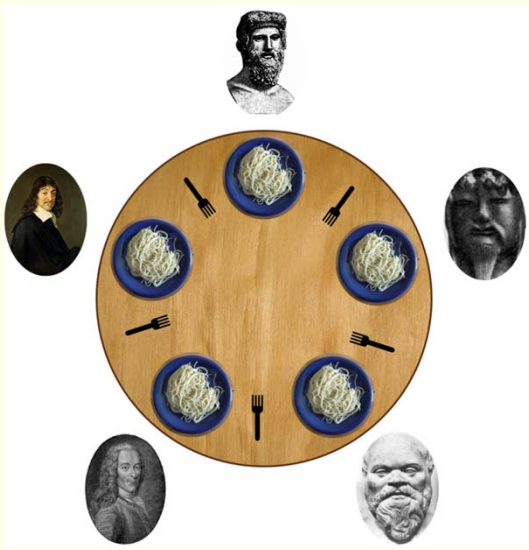
4

Dining Philosophers Problem

Each philosopher must alternately think and eat.
A philosopher requires both a left and right fork to eat.

What can go wrong with:

1) Think until the left fork is available; Then, pick it up.
2) Think until the right fork is available; Then, pick it up.
3) When both forks are held, eat for a fixed amount of time.
4) Then, put the right fork down;
5) Then, put the left fork down;
6) Think for a while then goto 1.



5

Superscalar Processor

A ***superscalar processor*** is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor.

A superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor.

Each execution unit is not a separate processor (or a core if the processor is a multi-core processor), but an execution resource within a single CPU such as an *Arithmetic Logic Unit*, an *Instruction Fetch Unit*, a *Memory Access Unit*, and a *Register Write Back Unit*.

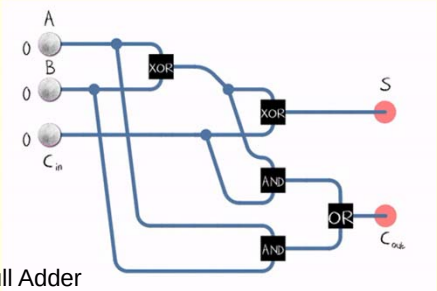
6

Superscalar and Vector Processor

A ***vector processor*** allows an array of numbers to be processed in different steps of a single processing unit.

Compared to a scalar processor with the same clock rate and the same number of circuit steps in a given operation, any particular instruction takes the same amount of time to complete (has the same latency).

However, like an assembly line, superscalar and vector CPUs can process an entire batch of operations much faster.



Full Adder

7

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.

8

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.

9

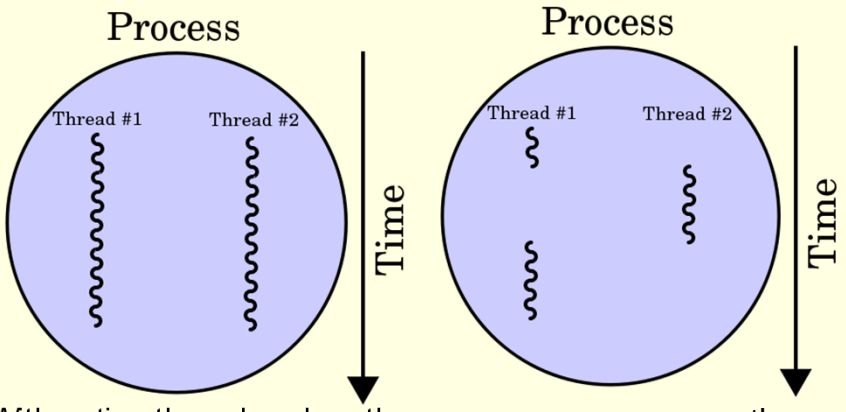
Native Threads Versus Green Threads

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 processers.
  - 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.

10

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 an out of control. Swapping requires overhead.

11

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.

12

java.lang.Thread: Sleep and Yeild

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.

13

Work Thread: Keeping GUI Response

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

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

Starts a new thread and calls the thread's **`.run()`**.

Do ***NOT*** call **`.run()`** directly.

Simple Polling

Must ***NOT*** override start()

Significant work, but less than 500 milliseconds.

14

`Volatile` Keyword

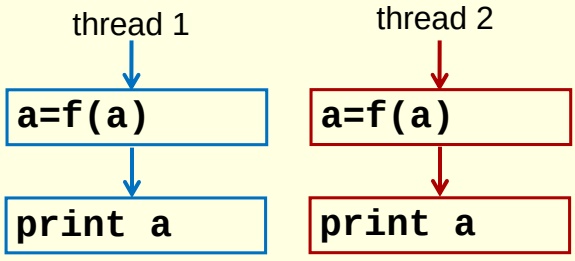
In Java, declaring a volatile variable means:

- The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory"
- A volatile variable ***never holds a lock***. Volatile is not suitable for cases where read-update-write must be atomic.

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

15

Race Condition:



Solutions:

Polling: "Hay, are you done yet?"

Lock: "I am blocked until the lock is released."

Notify: "I will sleep. Wake me when you are done".

16

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

17

"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**.

18

Java: Synchronized Method

```
public synchronized double foo()
{
  a=f(a);
  return a;
}
```

When thread 2's program counter enters **`foo()`**, thread 2 obtains a ***lock*** on **`foo()`**.

If thread 1 calls **`foo()`** while thread 2's program counter is in **`foo()`**, then thread 1 will be ***blocked*** until thread 2's program counter leaves foo.

19

Synchronized Access

```
1)  public synchronized void
2)    copyP(Point destination)
3)  {
4)    destination.x = p.x;
5)    destination.y = p.y;
6)  }
7)
8)  public synchronized void addP(int n)
9)  {
10)   p.x += n;
11)   p.y += n;
12) }
```

a

c

b

Both a ***read*** and a ***write.*** ***+= is NOT atomic.***

The two methods both use the instance to store the lock.

After thread 1 has read p.x on line 4, and before it reads p.y on line 5, ***without synchronized***, thread 2 could write to p.y.

20

Java: Synchronized Block

```
1) public void copyP(Point destination)
2) { synchronized(p)
3)   { destination.x = p.x;
4)     destination.y = p.y;
5)   }
6) }
7)
8) public void addP()(int n)
9) { synchronized(p)
10)  {
11)    p.x += n;
12)    p.y += n;
13)  }
14)}
```

When thread 1 executes line 3, it ***obtains a lock on instance variable p.*** If thread 2 reaches line 10, it will block until thread 1 releases the lock at line 6.

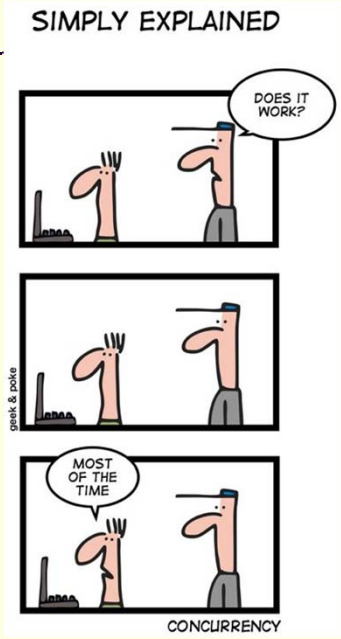
21

Synchronized Methods

Placing the synchronized keyword in the method header behaves identically to surrounding the code inside the method with a synchronization block on ‘this’.

A lock is ***stored*** in an object instance.

A lock is ***held*** by a thread.



22

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 it can be verified that f and g have no side effects, a good optimizer would parallelize line 3.

23

Quiz 1-4: Race Condition #2

Assuming it can be verified that function calls have no side effects, In this loop, which calculations can be preformed in *different* threads without a race condition?

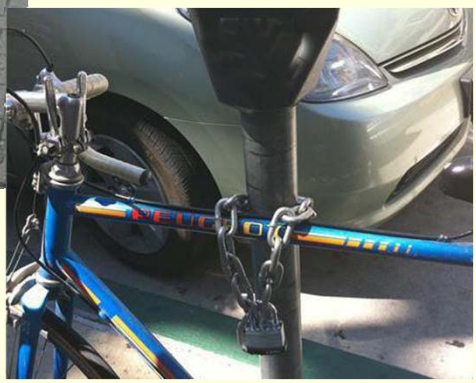
```
  for (int i=0; i<1000000; i++)
1 { x = g(x) + f(w);
2   y = k(y+w);
3   z = f(y) + h(v) + f(y+v);
4   v = g(d);
  }
```

a) 1 and 2
b) 2 and 3
c) 3 and 4
d) 2 and 4

24

The Key to Using Locks

Just because you use a lock does not mean your code is thread safe!



25

Lab 2: Threads of a Fibonacci Walk

- Simple, 15 Point lab
- Due: Midnight, Monday, January 30.
- Some of you will finish this in lab class. At most, it should take a few extra hours.
- Attach executable JAR in Blackboard Learn

26

Threads of a Fibonacci Walk (specs 1 of 5)

- Main thread of program must start up two worker threads.
- Each worker thread must independently walk the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...).
- Each thread must keep track of the following:

```
private final String NAME;  //Name of thread
private long step = 0;      //Steps since start.
private long z;             //   fib(step)
private long y = 1;         //   fib(step-1)
private long x = 1;         //   fib(step-2)
```

-

```
if (z == 7540113804746346429L)
{ x = 1;
  y = 1;
}
```

27

Threads of a Fibonacci Walk (specs 2 of 5)

- Implement in single file (public class and inner class).
- Every 2 seconds the main thread must print:
  - Each worker thread's name,
  - Each worker thread's step number
  - Each worker thread's three active sequence values (x, y and z).
- The info about x, y and z ***must*** be printed form the ***main thread***! The point of this is to verify that the main thread can access a ***consistent set of the data*** from each of the worker threads.
- In this context, what identifiable properties must a ***consistent set of data*** have?

28

Threads of a Fibonacci Walk (specs 3 of 5)

Do not confuse ***thread*** with ***class***:

- Code in different classes, can execute in the same thread.
- Similarly, code in the same class can execute in different threads.
- For example, a class that extends java.lang.Thread might have a public accessor method:
  - If that accessor is called from the child thread, then it runs in the child thread.
  - However, if that same accessor is called from the parent thread then it runs in the parent thread.
  - Indeed, the same accessor can be called by many threads at the same time.

29

Threads of a Fibonacci Walk (specs 4 of 5)

- Every 2 seconds the main thread must print each worker thread's name, step number and x, y z values.
- After 10 loops of the main thread, (20 seconds), the main thread must tell each of the workers (via a method you create) that they should exit. It will take some (short) time for each of the threads to notice they should exit, to print a short message such as "This is thread X signing out", and to actually exit. This message MUST print from the worker thread's run() method (NOT from whatever method main calls to tell the worker it should exit).
- Meanwhile, the main thread should be polling each thread's .isAlive() method. When .isAlive() returns false for both threads, then main must print "Program Exit" and call System.exit().

30

Threads of a Fibonacci Walk (specs 5 of 5)

- Note that each Fibonacci walk, even if started at almost the same time, will, in general, not be on the same step: they are running on different processors that are also running other stuff such as your e-mail, whatever webpages you have open, the operating system, various services such as Adobe checking for updates, ... You should expect that sometimes one thread is ahead of the other and sometimes the other thread is a head.
- While the Fibonacci numbers get reset to 1, 1, after reaching the end of the sequence, ***the step number must continue*** counting +1 for each step.
- On the first cycle through the Fibonacci numbers, step 1 is when z=2. Step 2 is when z=3. Step 3 is when z=5, ...

31

Grading Rubric (out of 15)

-5: Code does not follow the CS-351 standard.

+10: Program correctly uses thread synchronization so that no thread is waiting when there is no need for it to be waiting AND printing an consistent set of values is guaranteed.

+5: The each thread of the program exits after printing a goodbye message and the main thread calls System.exit() as specified.

32

Java's `final`

- The specs for the Fibonacci Walk, say:
- Each thread must keep track of the following:

```
private final String NAME;  //Name of thread.
```

- How is it possible for:
  a) This one class to be used for different threads,
  b) Each thread to be given a different name,
  c) AND for that different name to be `final`?

33

Wrong!

- I did not specify an output format. That means you are free to choose the output format.
- However, the below output is wrong. Why?

```
Step 0:
   Thread 1: x = 1, y = 1, z = 2
   Thread 2: x = 1, y = 1, z = 2

(2 second pause)

Step 1:
   Thread 1: x = 1, y = 2, z = 3
   Thread 2: x = 1, y = 2, z = 3
```

34

Example Correct Output

```
A (88618483) 308061521170129, 498454011879264, 806515533049393
B (88539684) 17711, 28657, 46368

A (242799173) 20365011074, 32951280099, 53316291173
B (242609894) 267914296, 433494437, 701408733

A (336637159) 5527939700884757, 8944394323791464, 14472334024676221
B (336406692) 55, 89, 144

A (489495460) 39088169, 63245986, 102334155
B (489226550) 4807526976, 7778742049, 12586269025
```

Δ=154,180,690
Δ=154,070,210

Δ= 93,837,986
Δ= 93,796,798

Δ=152,858,301
Δ=152,819,858

```
Main thread waiting for workers to die....
B Dies gracefully on step=1413066869
A Dies gracefully on step=1413648142
All workers are dead. Goodbye.
```

By random chance, B printed before A.

Above, A and B were printed in main. Therefore, the print order was ***deterministic***.

Properties of the Output:

I did not specify an output format - which means you are free to deliver any ***reasonable*** format.

1) The step numbers must be ***strictly increasing***.
2) The probability that the two worker threads, when polled, will be on the same step is near 0.
3) The step number for the two worker threads should be on same ***order of magnitude***.
4) Each ***difference*** between step numbers printed on consecutive two second intervals should be on same order of magnitude.
5) Each worker thread must report being done followed by main reporting done.

36

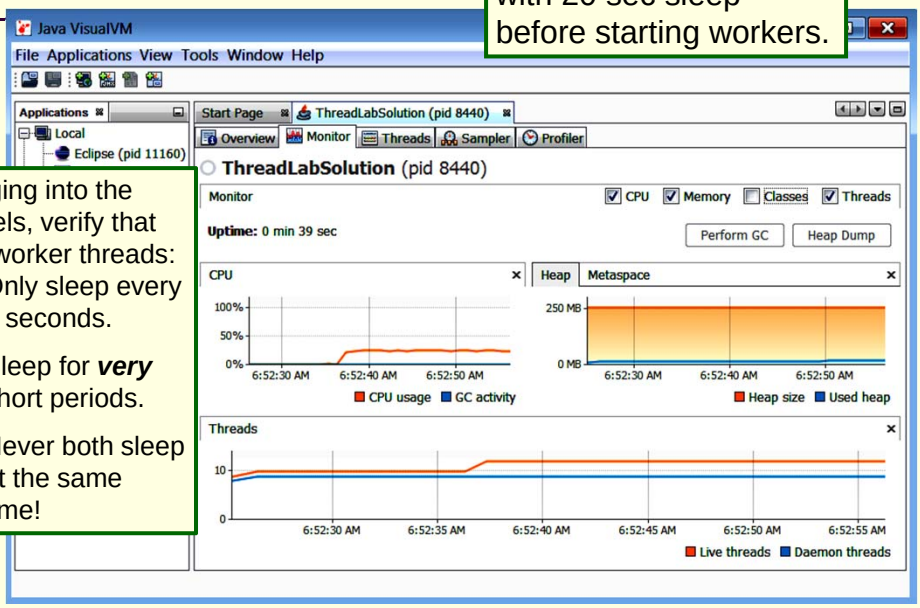
`jdk1.8.x\bin\jvisualvm.exe`

- VisualVM provides detailed information about Java applications while they are running on the Java Virutal Machine (JVM). VisualVM's graphical user interface enables you to quickly and easily see information about multiple Java applications.
- Watch video on **`https://visualvm.java.net/gettingstarted.html?Java_VisualVM`**
- VisualVM (or some similar tool of your choice) will be ***strongly*** needed for the Game of Live Lab.

37

Java VisualVM

Run of Fibonacci Lab with 20 sec sleep before starting workers.



Digging into the panels, verify that the worker threads:

1) Only sleep every 2 seconds.
2) Sleep for ***very*** short periods.
3) Never both sleep at the same time!

38

Example: *Parallelism in Traffic Sim*

- Segment by times of day: No Synchronization
- Segment by location (grid) **OR** system (trains, roads, ...):
  - Transitions within section, no synchronization. Transitions between sections do require synchronization.
  - How could this be optimized?
- Path-finding: Each timestep, when an agent comes to an intersection, it must use some intelligence to decide which way to go. In general, this requires global information.
  - Synchronization of ***simulation state***: read only.
  - Synchronization of ***search state***: each thread has personal copy.

39