Threads & Concurrency

Lecture 24– CS2110 – Spring 2017

Due date of A7 About A5-A6

We have changed the due date of A7 Friday, 28 April.

But the last date to submit A7 remains the same:

29 April.

We make the last date be 29 April so that people who are working on A8 can use our solution to A7 beginning on 30 April

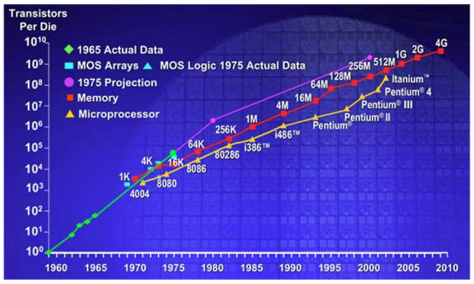
We will get caught up on grading A6 and regrading A5 and A6.

Today: New topic: concurrency

- Modern computers have "multiple cores"
  - Instead of a single CPU (central processing unit) on the chip 5-10 common. Intel has prototypes with 80!
- We often run many programs at the same time
- Even with a single core, your program may have more than one thing "to do" at a time
  - Argues for having a way to do many things at once

Why multicore?

Moore's Law: Computer speeds and memory densities nearly double each year



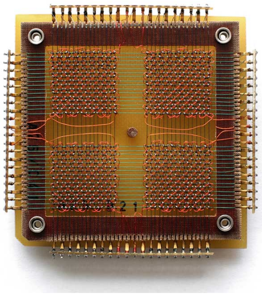
Magnetic-core memory. A penny a bit

What memory was like, 1955..1975.

At 1 penny per bit, a gigabyte would cost

$80,000, 000

From Wikipedia upload.wikimedia.org/ wikipedia/commons/d/da/ KL_CoreMemory.jpg



Magnetic-core memory. A penny a bit

Each ring is either magnetized or unmagnetized (1 or 0).

To set (Y4, X3) send impulse down wires Y4 and X3. Enough voltage to change that one but not others. Sense wire goes through all cores and detects whether something changed.

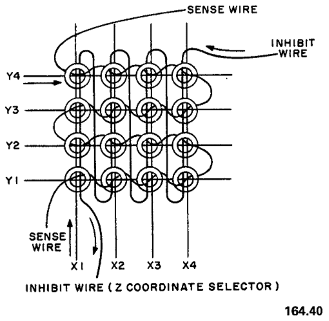
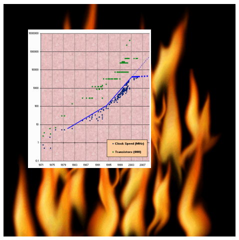


Figure 6-9.—Inhibit wire of a core array.

But a fast computer runs hot

- □ Power dissipation rises as square of the clock rate
- □ Chips were heading toward melting down!
- □ Multicore: with four CPUs (cores) on one chip, even if we run each at half speed we can perform more overall computations!



Programming a Cluster...

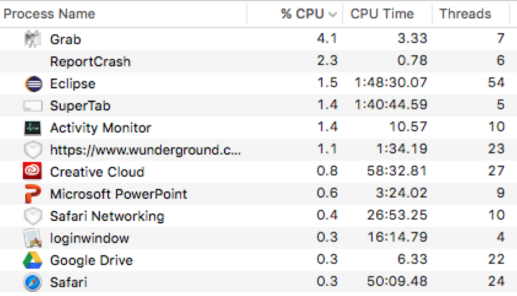
- Sometimes you want to write a program that is executed on many machines!
- Atlas Cluster (at Cornell):
- 768 cores
- 1536 GB RAM
- 24 TB Storage
- 96 NICs (Network Interface Controller)

Many processes are executed simultaneously on your computer

- Operating system provides support for multiple "processes"
- Usually fewer processors than processes
- Processes are an abstraction: at hardware level, lots of multitasking
  - –memory subsystem
  - –video controller
  - –buses
  - –instruction prefetching

Part of Activity Monitor in Gries's laptop

>100 processes are competing for time. Here's some of them:



Process Name	% CPU	CPU Time	Threads
Grab	4.1	3.33	7
ReportCrash	2.3	0.78	6
Eclipse	1.5	1:48:30.07	54
SuperTab	1.4	1:40:44.59	5
Activity Monitor	1.4	10.57	10
https://www.wunderground.c...	1.1	1:34.19	23
Creative Cloud	0.8	58:32.81	27
Microsoft PowerPoint	0.6	3:24.02	9
Safari Networking	0.4	26:53.25	10
loginwindow	0.3	16:14.79	4
Google Drive	0.3	6.33	22
Safari	0.3	50:09.48	24

Concurrency

- □ *Concurrency* refers to a single program in which several processes, called threads, are running simultaneously
  - □ Special problems arise
  - □ They see the same data and hence can interfere with each other, e.g. one process modifies a complex structure like a heap while another is trying to read it
- □ CS2110: we focus on two main issues:
  - □ Race conditions
  - □ Deadlock

Race conditions

- □ A "race condition" arises if two or more processes access the same variables or objects concurrently and at least one does updates
- □ Example: Processes t1 and t2 x= x + 1; for some static global x.

```
Process t1          Process t2
…                   …
x= x + 1;           x= x + 1;
```

But x= x+1; is not an "atomic action": it takes several steps

Race conditions

- Suppose x is initially 5

Thread t1	Thread t2
LOAD x	...
	LOAD x
ADD 1	ADD 1
	STORE x
STORE x	

- ... after finishing, x = 6! We “lost” an update

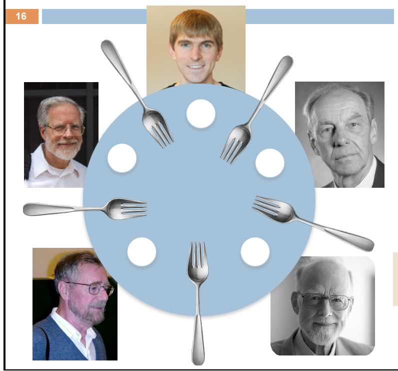
Race conditions

- Typical race condition: two processes wanting to change a stack at the same time. Or make conflicting changes to a database at the same time.
- Race conditions are bad news
  - Race conditions can cause many kinds of bugs, not just the example we see here!
  - Common cause for “blue screens”: null pointer exceptions, damaged data structures
  - Concurrency makes proving programs correct much harder!

Deadlock

- To prevent race conditions, one often requires a process to “acquire” resources before accessing them, and only one process can “acquire” a given resource at a time.
- Examples of resources are:
  - A file to be read
  - An object that maintains a stack, a linked list, a hash table, etc.
- But if processes have to acquire two or more resources at the same time in order to do their work, **deadlock** can occur. This is the subject of the next slides.

Dining philosopher problem



Five philosophers sitting at a table.

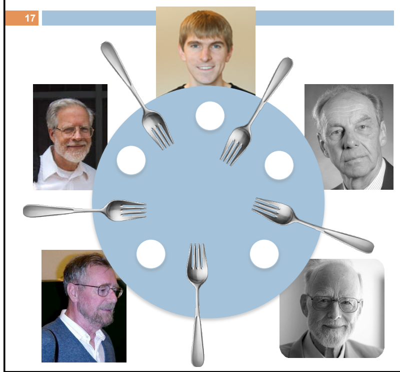
Each repeatedly does this:

1. think
2. eat

What do they eat? spaghetti.

Need TWO forks to eat spaghetti!

Dining philosopher problem



Each does repeatedly :

1. think
2. eat (2 forks)

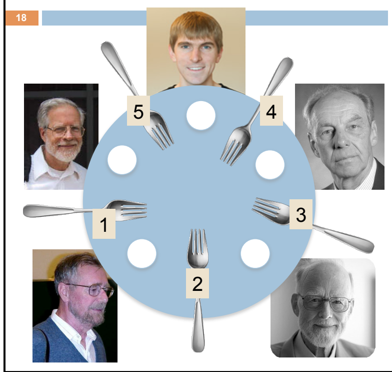
eat is then:

pick up left fork
pick up right fork
pick up food, eat
put down left fork
put down rght fork

At one point, they all pick up their left forks

DEADLOCK!

Dining philosopher problem



Simple solution to deadlock:

Number the forks. Pick up smaller one first

1. think
2. eat (2 forks)

eat is then:

pick up smaller fork
pick up bigger fork
pick up food, eat
put down bigger fork
put down smallerfork

Java: What is a Thread?

- *A separate "execution" that runs within a single program and can perform a computational task independently and concurrently with other threads*
- Many applications do their work in just a single thread: the one that called main() at startup
  - But there may still be extra threads...
  - ... Garbage collection runs in a "background" thread
  - GUIs have a separate thread that listens for events and "dispatches" calls to methods to process them
- Today: learn to create new threads of our own in Java

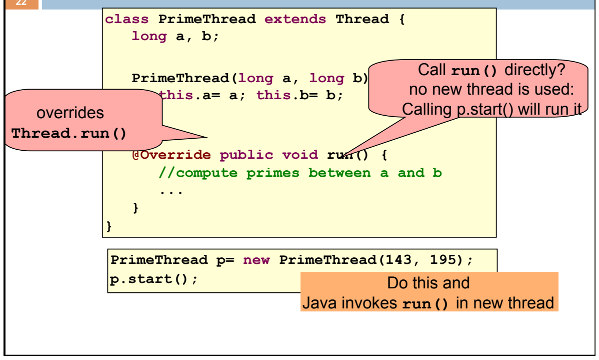
Thread

- A thread is an object that "independently computes"
  - Needs to be created, like any object
  - Then "started" --causes some method to be called. It runs side by side with other threads in the same program; they see the same global data
- The actual executions could occur on different CPU cores, but but don't have to
  - We can also simulate threads by *multiplexing* a smaller number of cores over a larger number of threads

Java class Thread

- threads are instances of class Thread
  - Can create many, but they do consume space & time
- The Java Virtual Machine creates the thread that executes your main method.
- Threads have a priority
  - Higher priority threads are executed preferentially
  - By default, newly created threads have initial priority equal to the thread that created it (but priority can be changed)

Creating a new Thread (Method 1)



```
class PrimeThread extends Thread {
   long a, b;

   PrimeThread(long a, long b) {
      this.a= a; this.b= b;
   }

   @Override public void run() {
      //compute primes between a and b
      ...
   }
}
```

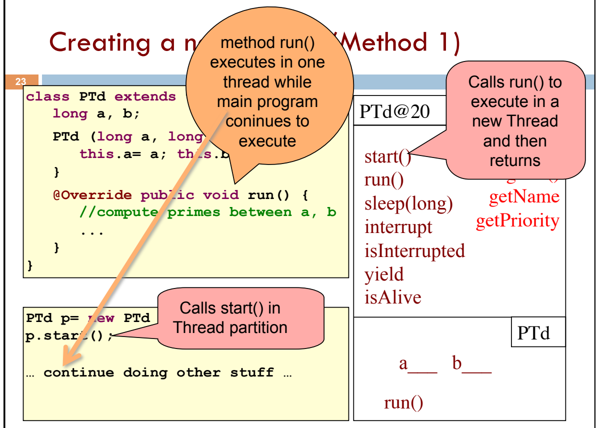
```
PrimeThread p= new PrimeThread(143, 195);
p.start();
```

overrides **Thread.run()**

Call **run()** directly? no new thread is used: Calling p.start() will run it

Do this and Java invokes **run()** in new thread

Creating a n Method 1)



```
class PTd extends
   long a, b;
   PTd (long a, long
      this.a= a; this.b
   }
   @Override public void run() {
      //compute primes between a, b
      ...
   }
}
```

```
PTd p= new PTd
p.start();

… continue doing other stuff …
```

method run() executes in one thread while main program coninues to execute

Calls run() to execute in a new Thread and then returns

Calls start() in Thread partition

Creating a new Thread (Method 2)

```
class PrimeRun implements Runnable {
   long a, b;

   PrimeRun(long a, long b) {
      this.a= a; this.b= b;
   }

   public void run() {
      //compute primes between a and b
      ...
   }
}
```

```
PrimeRun p= new PrimeRun(143, 195);
new Thread(p).start();
```

Example

```
public class ThreadTest extends Thread {
  int M= 1000;      int R= 600;
  public static void main(String[] args) {
     new ThreadTest().start();
     for (int h= 0; true; h= h+1) {
          sleep(M);
          System.out.format("%s %d\n", Thread.currentThread(), h);
     }
}

@Override public void run() {
   for (int k= 0; true; k= k+1) {
          sleep(R);
          System.out.format("%s %d\n", Thread.currentThread(), k);
     }
}
```

We'll demo this with different values of M and R. Code will be on course website

sleep(…) requires a throws clause —or else catch it

Example

Thread name, priority, thread group

```
public class ThreadTest extends Thread {
  int M= 1000;      int R= 600;
  public static void main(String[] args) {
     new ThreadTest().start();
     for (int h= 0; true; h= h+1) {
          sleep(M);
          …format("%s %d\n", Thread.curren
     }
}

@Override public void run() {
   for (int k= 0; true; k= k+1) {
          sleep(R);
          …format("%s %d\n", Thread.currentThread(), k);
     }
}
```

```
Thread[Thread-0,5,main] 0
Thread[main,5,main] 0
Thread[Thread-0,5,main] 1
Thread[Thread-0,5,main] 2
Thread[main,5,main] 1
Thread[Thread-0,5,main] 3
Thread[main,5,main] 2
Thread[Thread-0,5,main] 4
Thread[Thread-0,5,main] 5
Thread[main,5,main] 3
…
```

Example

```
public class ThreadTest extends Thread {
   static boolean ok = true;

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.println("waiting...");
         yield();
      }
      ok = false;
   }

   public void run() {
      while (ok) {
         System.out.println("running...");
         yield();
      }
      System.out.println("done");
   }
}
```

```
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...

waiting...
running...
waiting...
running...
done
```

If threads happen to be sharing a CPU, yield allows other waiting threads to run.

Terminating Threads is tricky

- Easily done... but only in certain ways
  - *Safe way to terminate a thread: return from method run*
  - *Thread throws uncaught exception? whole program will be halted (but it can take a second or two ...)*
- Some old APIs have issues: stop(), interrupt(), suspend(), destroy(), etc.
  - Issue: Can easily leave application in a "broken" internal state.
  - Many applications have some kind of variable telling the thread to stop itself.

Background (daemon) Threads

- In many applications we have a notion of "foreground" and "background" (daemon) threads
  - Foreground threads are doing visible work, like interacting with the user or updating the display
  - Background threads do things like maintaining data structures (rebalancing trees, garbage collection, etc.) A daemon can continue even when the thread that created it stops.
- On your computer, the same notion of background workers explains why so many things are always running in the task manager.

Background (daemon) Threads

- demon: an evil spirit
- daemon. Fernando Corbato, 1963, first to use term. Inspired by Maxwell's daemon, an imaginary agent in physics and thermodynamics that helped to sort molecules.
- from the Greek δαιμων. Unix System Administration Handbook, page 403: … "Daemons have no particular bias toward good or evil but rather serve to help define a person's character or personality. The ancient Greeks' concept of a "personal daemon" was similar to the modern concept of a "guardian angel"—eudaemonia is the state of being helped or protected by a kindly spirit. As a rule, UNIX systems seem to be infested with both daemons and demons.

Beginning to think about avoiding race conditions

You know that race conditions can create problems:

Basic idea of race condition: Two different threads access the same variable in a way that destroys correctness.

- Process t1 … x= x + 1; Process t2 ... x= x + 1;

But x= x+1; is not an "atomic action": it takes several step

Two threads may want to use the same stack, or Hash table, or linked list, or … at the same time.

Synchronization

- Java has one primary tool for preventing race conditions. you must use it by carefully and explicitly – it isn't automatic.
  - Called a synchronization barrier
  - Think of it as a kind of lock
    - Even if several threads try to acquire the lock at once, only one can succeed at a time, while others wait
    - When it releases the lock, another thread can acquire it
    - Can't predict the order in which contending threads get the lock but it should be "fair" if priorities are the same

Solution: use with synchronization

```
private Stack<String> stack= new Stack<String>();

public void doSomething() {
   synchronized (stack) {
      if (stack.isEmpty()) return;
      String s= stack.pop();
   }
   //do something with s...
}
```

synchronized block

- Put critical operations in a synchronized block
- Can't be interrupted by other **synchronized blocks on the same object**
- Can run concurrently with non-synchronized code
- Or code synchronized on a different object!

Synchronization

- Java has one primary tool for preventing race conditions. you must use it by carefully and explicitly – it isn't automatic.
  - Called a synchronization barrier
  - Think of it as a kind of lock
    - Even if several threads try to acquire the lock at once, only one can succeed at a time, while others wait
    - When it releases the lock, another thread can acquire it
    - Can't predict the order in which contending threads get the lock but it should be "fair" if priorities are the same

Example: a lucky scenario

```
private Stack<String> stack= new Stack<String>();

public void doSomething() {
   if (stack.isEmpty()) return;
   String s= stack.pop();
   //do something with s...
}
```

Suppose threads A and B want to call **doSomething()**, and there is one element on the stack

1. thread A tests **stack.isEmpty()** false
2. thread A pops ⇒ stack is now empty
3. thread B tests **stack.isEmpty()** ⇒ true
4. thread B just returns – nothing to do