Concurrency

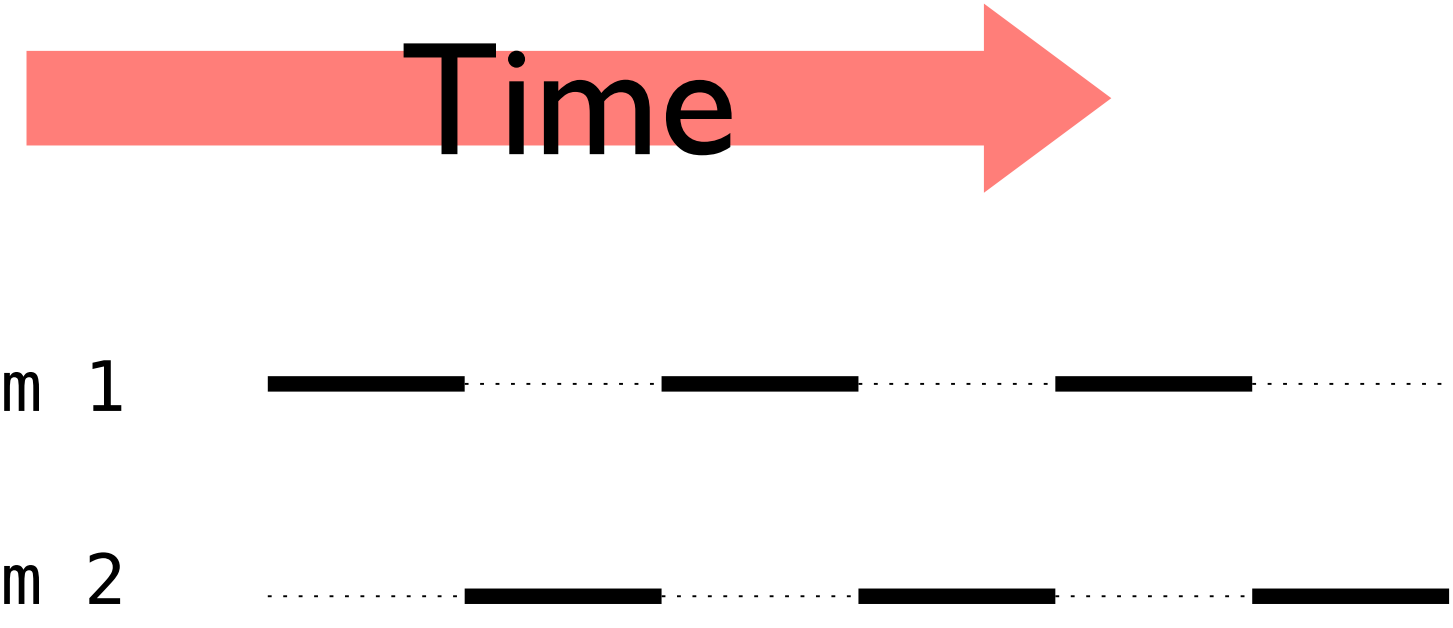
March 7–8, 2012
CS 60: Principles of Computer Science

Assignment 6 due March 7: Spampede!
Midterm due back 5pm Friday, March 9

Multitasking

Multitasking: running several programs "at once" (on the same CPU).

Modern operating systems use "preemptive time slicing"



The Multicore Future

Parallel code is inevitable.

- ✓ Moore's Law gives us more *transistors*
- ✓ 2-core, 4-core, 8-core, . . . 128-core, . . . CPUs
- ✓ Multi-cpu motherboards (e.g., church.cs.hmc.edu has 48 cores)
- ✓ Intel was prototyping 80-core CPUs five years ago
- ✓ 512-core GPUs readily available

Using Multiprocessors

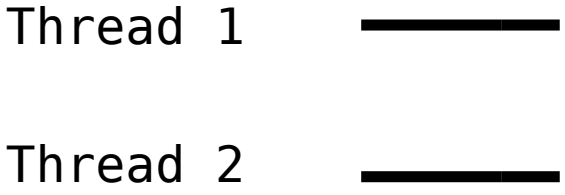
We could run different programs at the *same* time:


```
Program 1 ——————————
Program 2 ——————————
```

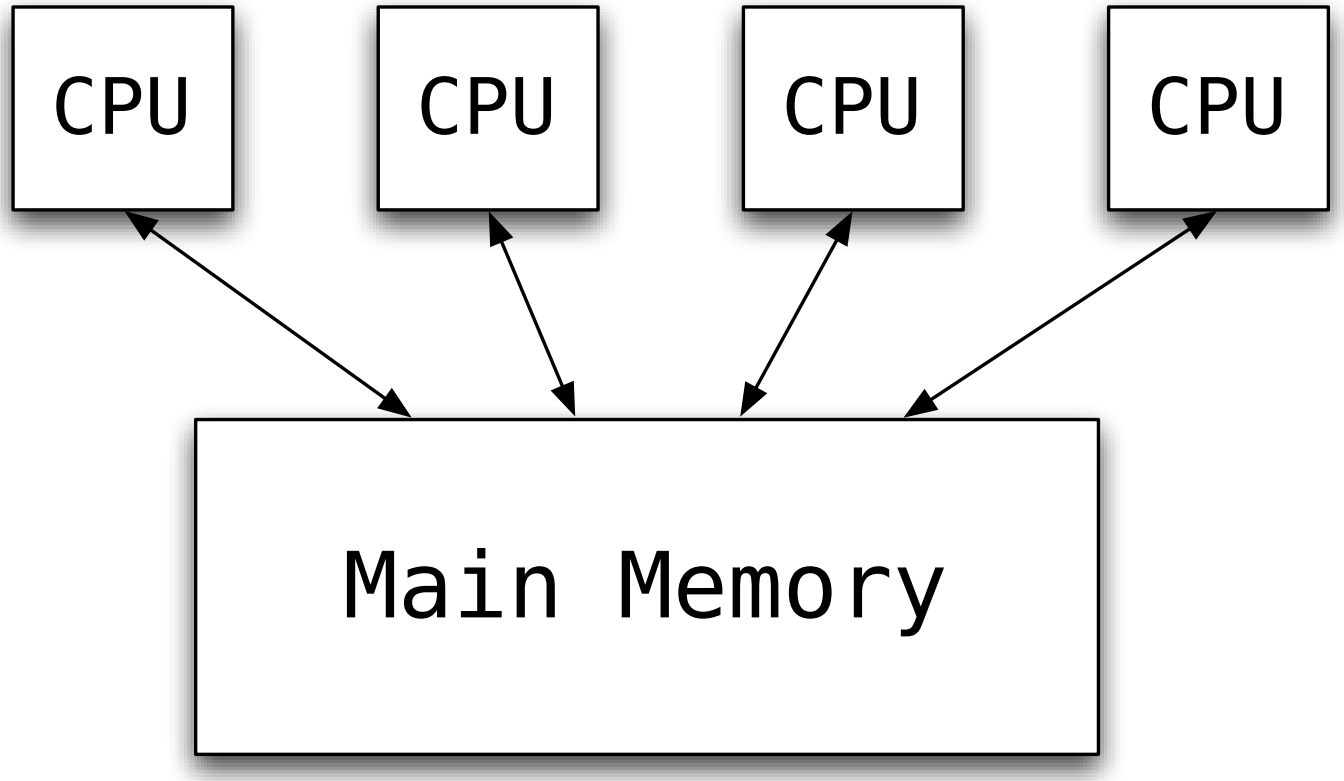
Using Multiprocessors

We could run *different programs* at the *same* time:

Or, break *one program* into multiple "threads of control," and finish faster:

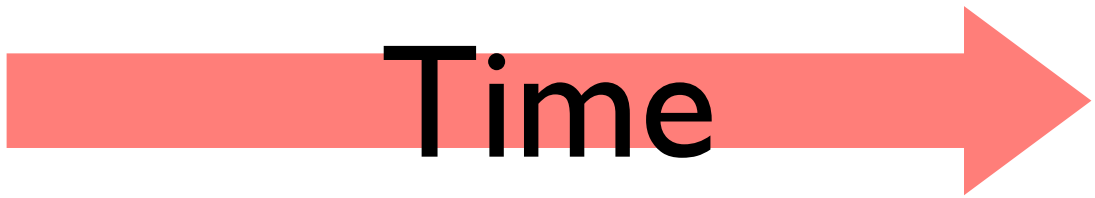


Shared Memory Concurrency (Idealized)



Thinking About Shared Memory

There's only one memory, we can view the reads and writes of threads as being "interleaved" (as in multitasking).




```
Thread 1  ----------------------------------
Thread 2  ----------------------------------
```

A Simple Example

Two threads, both trying to increment the variable `x`.

```
int x = 0;
```

```
r1 = x;
x = r1 + 1;
```

```
r2 = x;
x = r2 + 1;
```

A Simple Example

Two threads, both trying to increment the variable `x`.

```
int x = 0;
```

```
r1 = x;
x = r1 + 1;
```

```
r2 = x;
x = r2 + 1;
```

```
x == 2
```

A Simple Example

Two threads, both trying to increment the variable `x`.

```
int x = 0;
```

```
r2 = x;
x = r2 + 1;
```

```
r1 = x;
x = r1 + 1;
```

```
x == 2
```

A Simple Example

Two threads, both trying to increment the variable `x`.

```
                    int x = 0;

r1 = x;
                                    r2 = x;
x = r1 + 1;
                                    x = r2 + 1;

                      x == 1
```

Puzzle

What are the possible final values of **x**, if each thread loops 10 times?

```
int x = 0;
```

```
for (int i=0; i<10; ++i) {
   r1 = x;
   x = r1 + 1;
}
```

```
for (int j=0; j<10; ++j) {
   r2 = x;
   x = r2 + 1;
}
```

Puzzle

What are the possible final values of **x**, if each thread loops 10 times?

```
int x = 0;
```

```
for (int i=0; i<10; ++i) {
   r1 = x;
   x = r1 + 1;
}
```

```
for (int j=0; j<10; ++j) {
   r2 = x;
   x = r2 + 1;
}
```

x could be as high as 20, and as low as 2

How x Could Be 2

```
read  0
                read 0
write 1
read 1
write 2
read 2
...
write 9
                write 1
read 1
                read 1
                write 2
                ...
                read 9
                write 10
write 2
```

Conventional Solution: Locks

Recall: locks ensure exclusive access, with one owner at a time. So,

1. Associate *locks* with shared data
2. Don't access shared data unless we hold the locks

Conventional Solution: Locks

Recall: locks ensure exclusive access, with one owner at a time. So,

1. Associate *locks* with shared data
2. Don't access shared data unless we hold the locks

```
int x = 0;
```

```
lock(x);
r1 = x;
x = r1 + 1;
unlock(x);
```

```
lock(x);
r2 = x;
x = r2 + 1;
unlock(x);
```

Conventional Solution: Locks

Recall: locks ensure exclusive access, with one owner at a time. So,

1. Associate *locks* with shared data
2. Don't access shared data unless we hold the locks

```
int x = 0;
```

```
lock(x);
r1 = x;
x = r1 + 1;
unlock(x);
```

```
lock(x);
r2 = x;
x = r2 + 1;
unlock(x);
```

Conventional Solution: Locks

Recall: locks ensure exclusive access, with one owner at a time. So,

1. Associate *locks* with shared data
2. Don't access shared data unless we hold the locks

```
int x = 0;
```

```
lock(x);
r2 = x;
x = r2 + 1;
unlock(x);
```

```
lock(x);
r1 = x;
x = r1 + 1;
unlock(x);
```

Locks Are No Fun

Easy to acquire too few (or too many!) locks

Easy to deadlock

Locks Are No Fun

Easy to acquire too few (or too many!) locks

Easy to deadlock

```
lock(a);
lock(b);
r1 = a;
b = r1 + 1;
unlock(b);
unlock(a);
```

```
lock(b);
lock(a);
r2 = b;
a = r2 * 2;
unlock(a);
unlock(b);
```

Locks Are No Fun

Easy to acquire too few (or too many!) locks

Easy to deadlock

```
lock(a);

lock(b);
r1 = a;
b = r1 + 1;
unlock(b);
unlock(a);
```

```
lock(b);
lock(a);
r2 = b;
a = r2 * 2;
unlock(a);
unlock(b);
```

Threads in Java

1. Create an object supporting the `Runnable` interface:

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

2. Use it to construct a `Thread` object

3. Invoke the thread's `.start()` method.

Example

```
class Counter implements Runnable {
  public void run() {
    for (int i = 0; i < 1000; ++i)
       System.out.println(i);
  }
}

...

  Thread t1 = new Thread(new Counter());
  Thread t2 = new Thread(new Counter());

  t1.start();
  t2.start();
```

Example

```
class Counter implements Runnable {
  public void run() {
    for (int i = 0; i < 1000; ++i)
       System.out.println(i);
  }
}

...

  Thread t1 = new Thread(new Counter());
  Thread t2 = new Thread(new Counter());

  t1.start();
  t2.start();
```

```
0
1
2
...
150
151
0
1
2
3
4
...
296
297
152
153
...
```

Locks in Java

Every object comes with an associated lock!

Locks in Java

Every object comes with an associated lock!

These locks can be acquired/released using `synchronized` blocks:

```
synchronized (objectToLock) {          // lock
  ...do stuff
}                                      // unlock
```

Example: Stacks

What could go wrong if two different threads try to **push** onto the same stack at once?

```
class StringStack {
  ...

  public void push(String data)
  {
     StackCell newCell = new StackCell(data, this.top);
     this.top = newCell;
  }
  ...
}
```

How can we avoid this?

Solution 1

```
class StringStack {
  ...

  public void push(String data)
  {
     synchronized (this) {
        StackCell newCell = new StackCell(data, this.top);
        this.top = newCell;
     }
  }
  ...
}
```

Solution 2

```
class StringStack {
  ...

  synchronized public void push(String data)
  {
     StackCell newCell = new StackCell(data, this.top);
     this.top = newCell;
  }
  ...
}
```

Concurrency in Spampede

There are two threads in Spampede:

1. The "implicit" applet thread is processing events as they arrive in the event queue. [view and controller]
   - E.g., calling `keyPressed` and `actionPerformed` and redisplaying the buffer

2. A second thread is calling `cycle` every so often [model]
   - E.g., calling `advancePede` and `updateSpam` and `drawEnvironment`

Communication:

- ✓ Fields in the (one) Spampede object can be seen by both threads (e.g., the current direction `dir`)
- ✓ When the second thread calls `repaint()`, this puts a "copy the buffer to the screen" event in the event queue, to be processed by the first thread!

Preparing for a new thread...

```
public class Spampede extends JApplet
  implements ActionListener, KeyListener, Runnable
{
  ...

  /*
   * This is the method that calls the "cycle()"
   * method every so often (every sleepTime milliseconds).
   */
  public void run()
  {
   ...
  }
  ...
}
```

Creating the new thread...

```
public class Spampede extends JApplet
  implements ActionListener, KeyListener, Runnable
{
  Thread thread = null;
  ...

  /* This is the method attached to the "Start" button */
  public synchronized void go()
  {
    if (thread == null)  {
      thread = new Thread(this);
      thread.start();
      ...
    } ...
  }
  ...
}
```