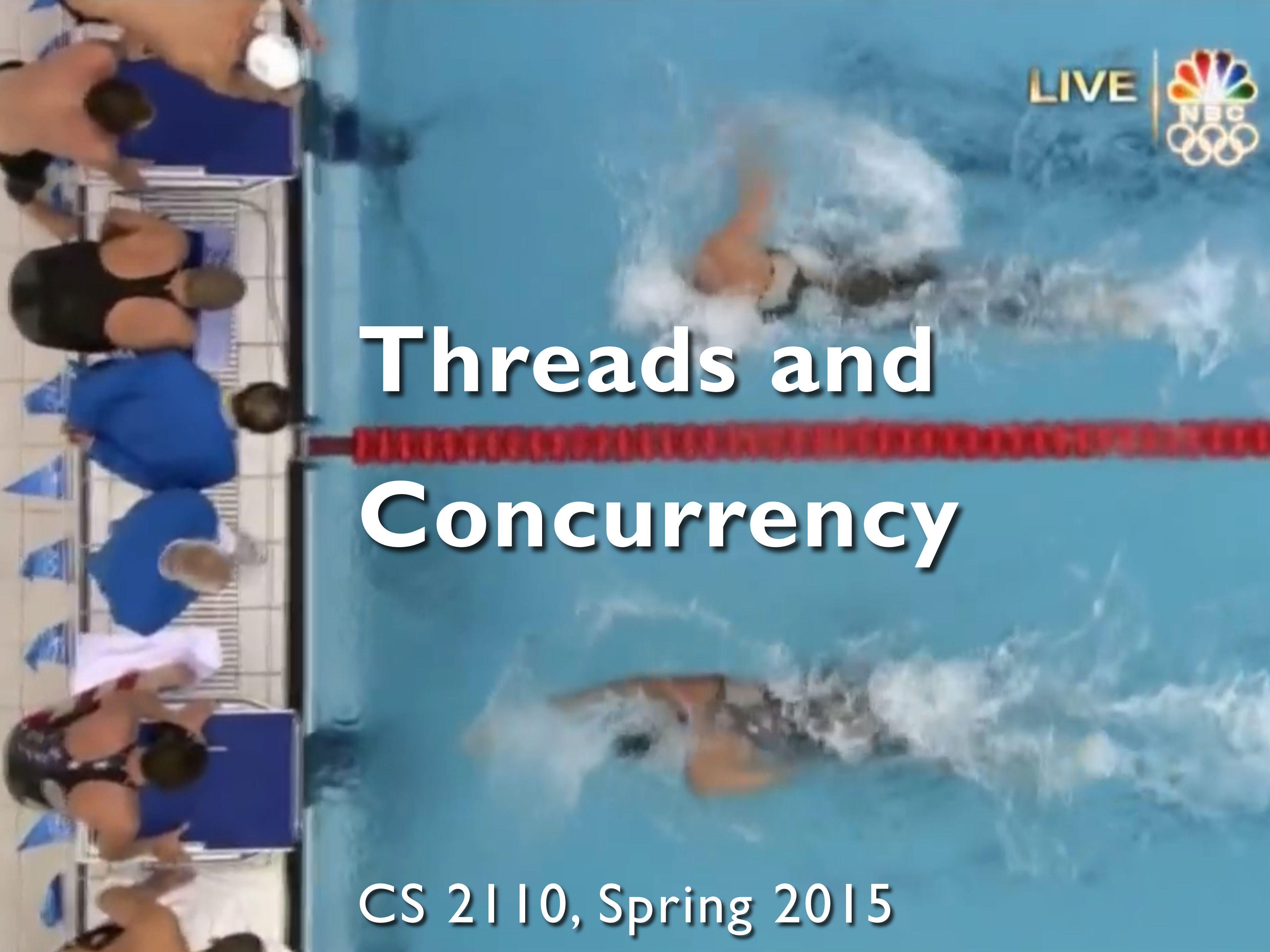
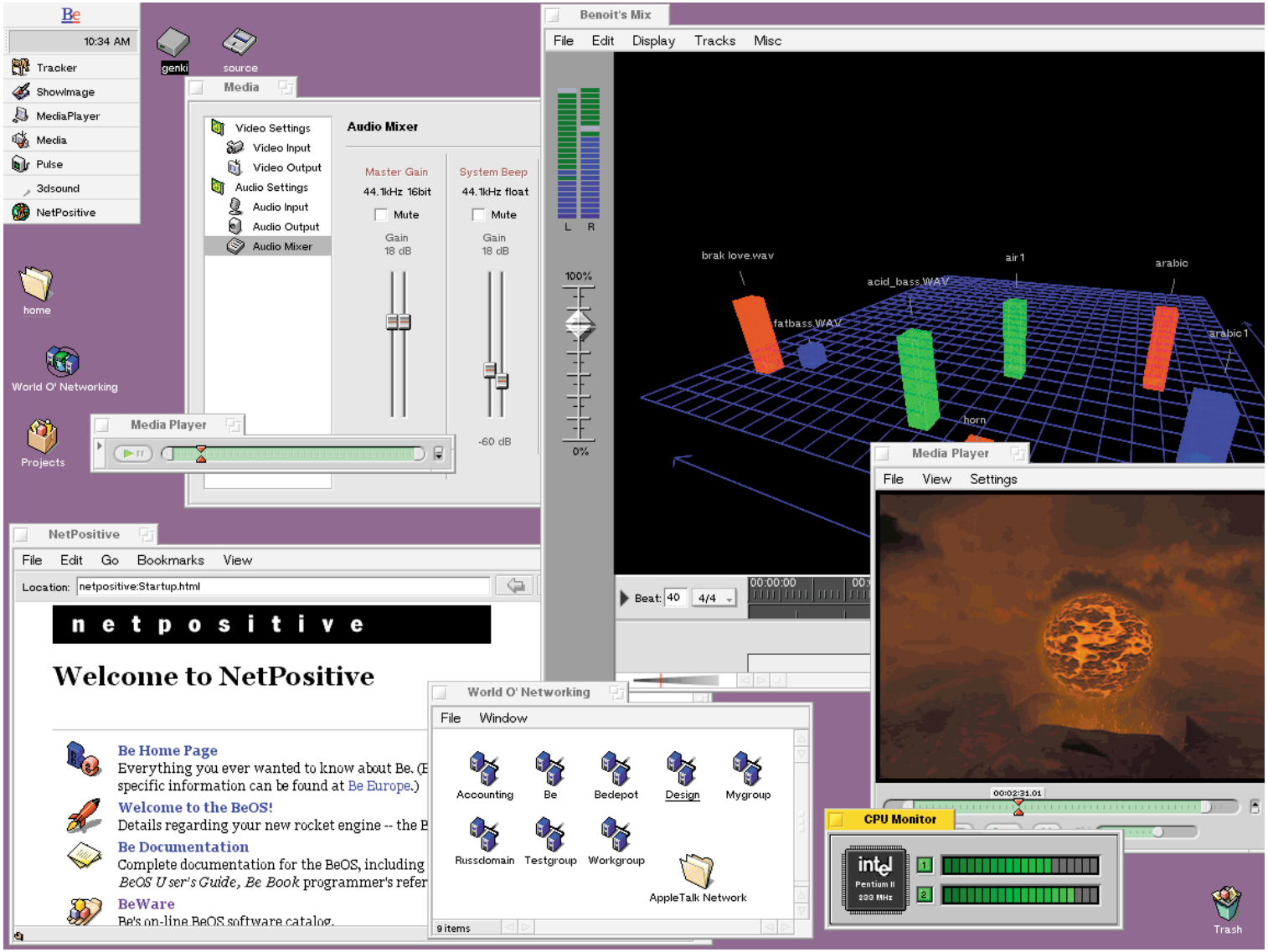
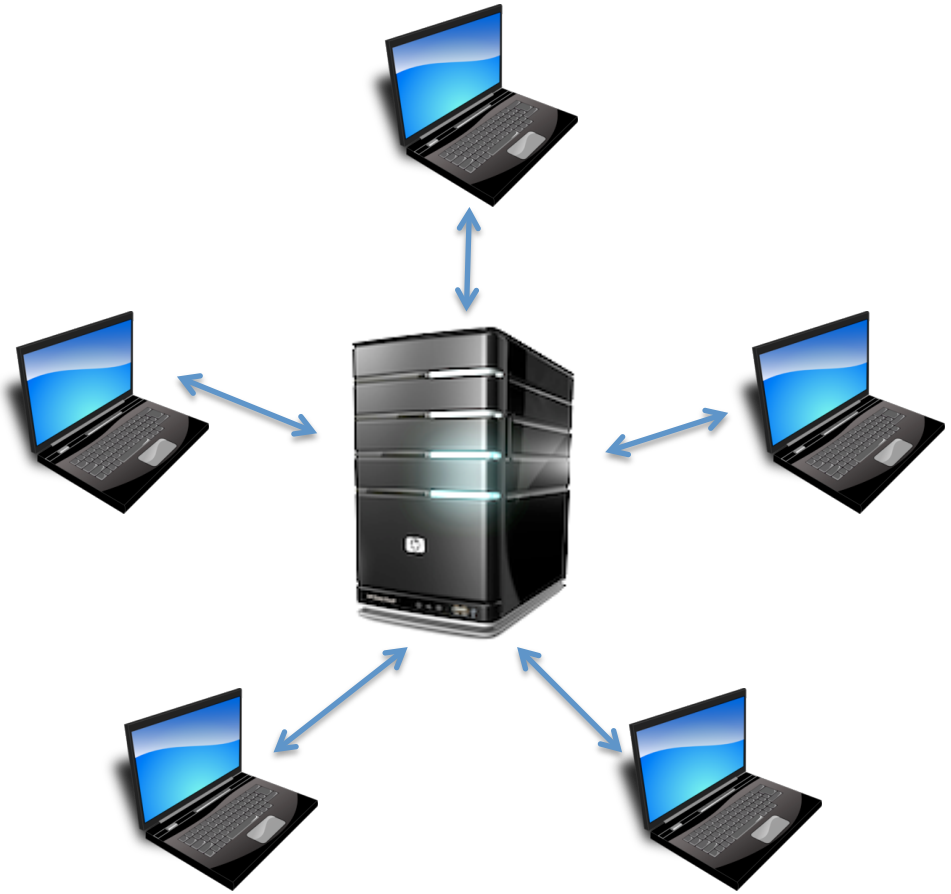


Threads and Concurrency

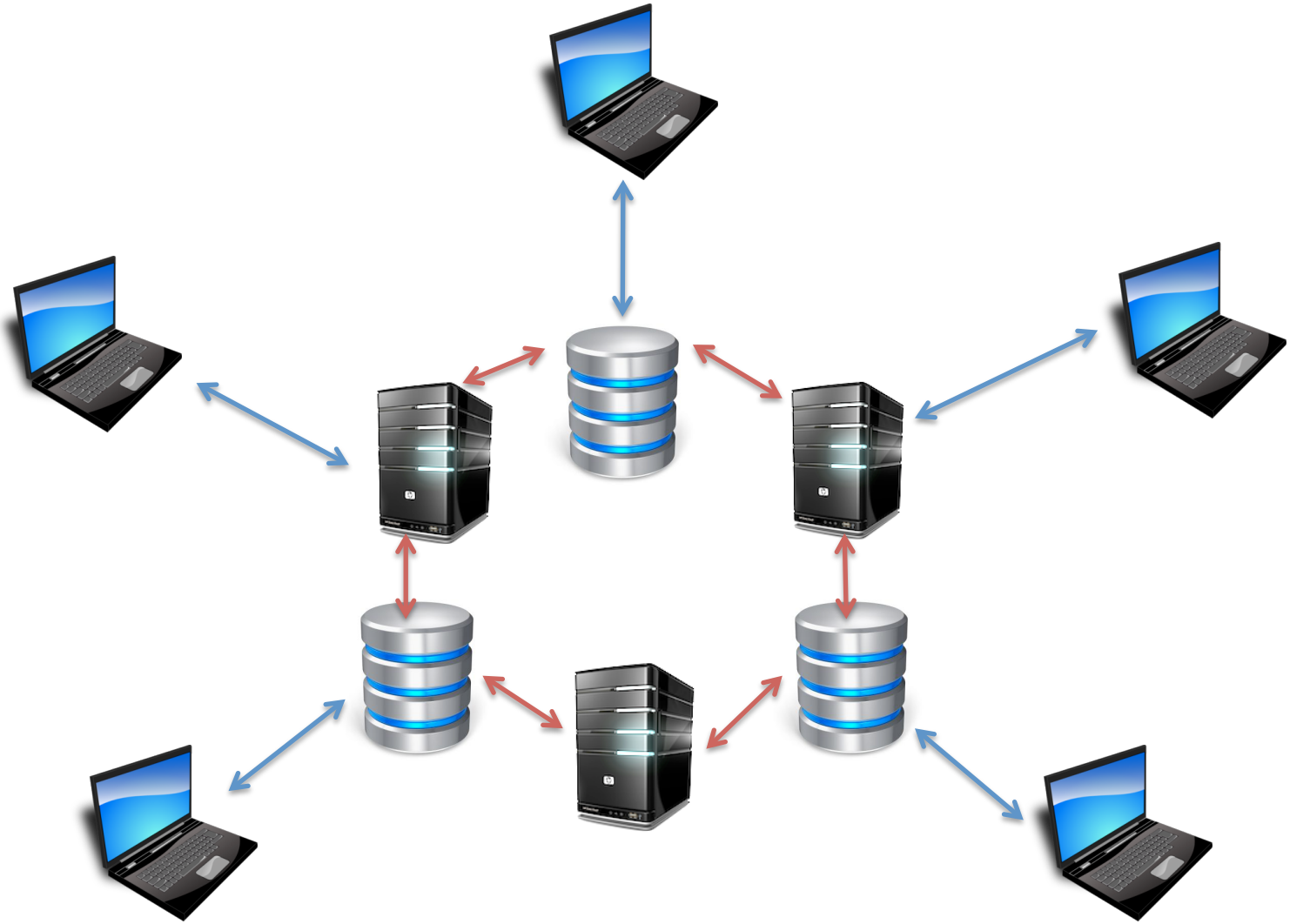
CS 2110, Spring 2015



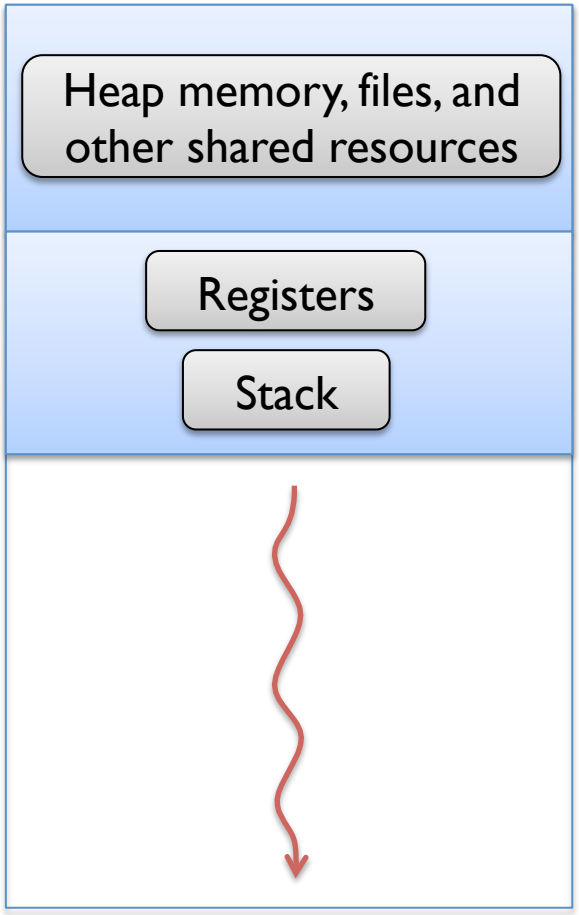
A single computer can simultaneously run several programs



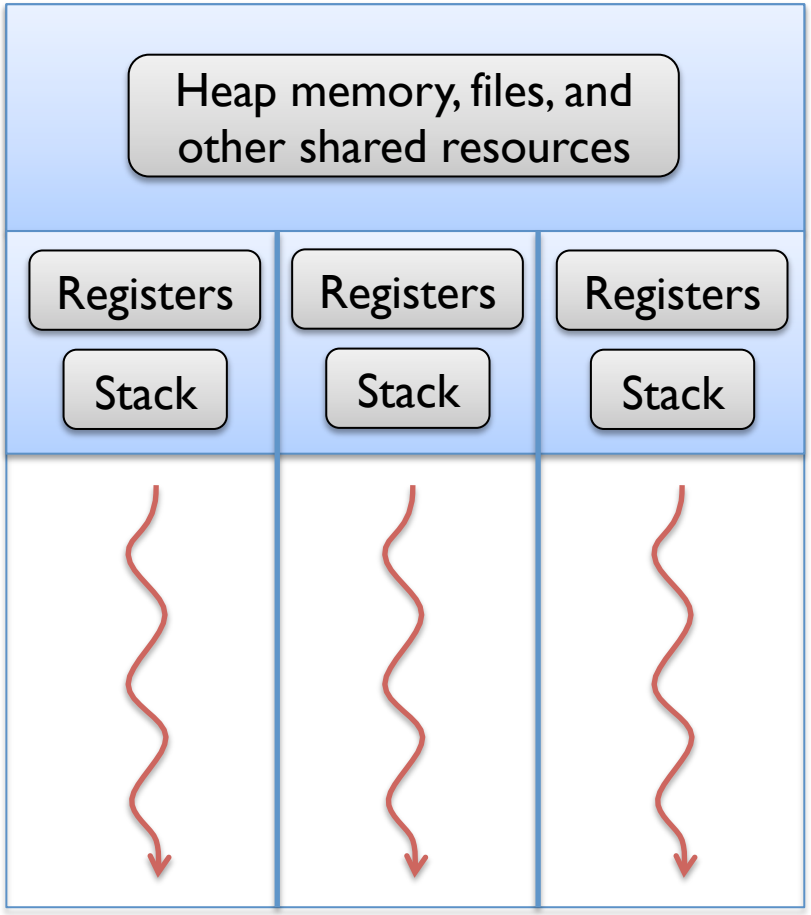
Several computers can simultaneously access a shared server



The server can itself be a distributed system with different data and programs on different nodes

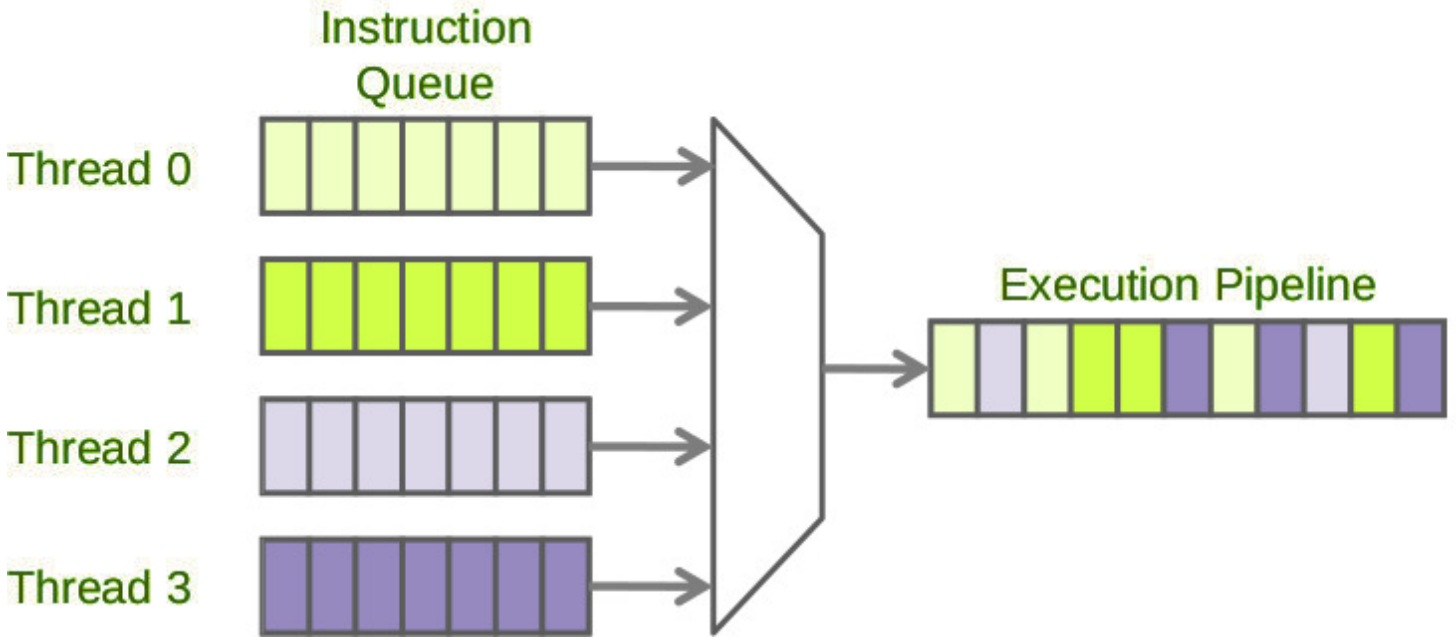


Single-threaded

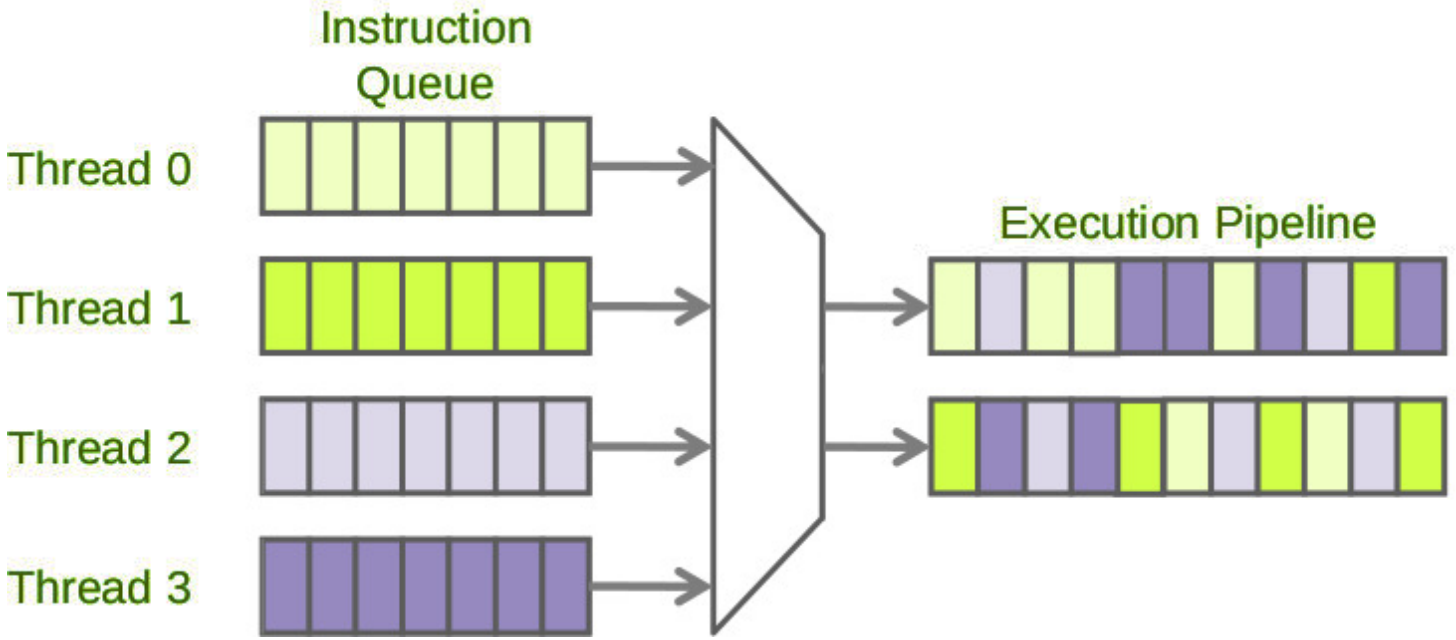


Multi-threaded

Each program can have several threads of execution



Several threads can be run on a single processor pipeline. Each thread is run for a short time and then suspended, giving the effect that they are running simultaneously



If the processor has multiple cores (modern CPUs often have 2, 4, 8 or more cores), the threads may be assigned to different cores (or more generally, to different "hardware threads") and actually run simultaneously

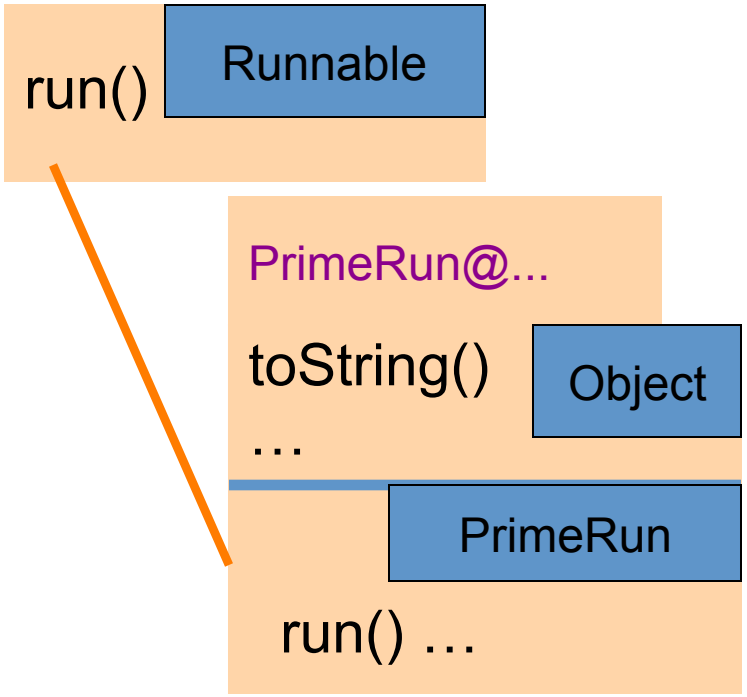
Threads in Java

- Threads are instances of class Thread
  - Can create many, but they consume space & time
- The Java Virtual Machine created the initial Thread that executes your method main
- Threads have a priority
  - Higher priority Threads are executed preferentially
  - A newly created Thread has initial priority equal to the Thread that created it (but can change)

A Java Thread runs a Runnable object

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

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

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

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

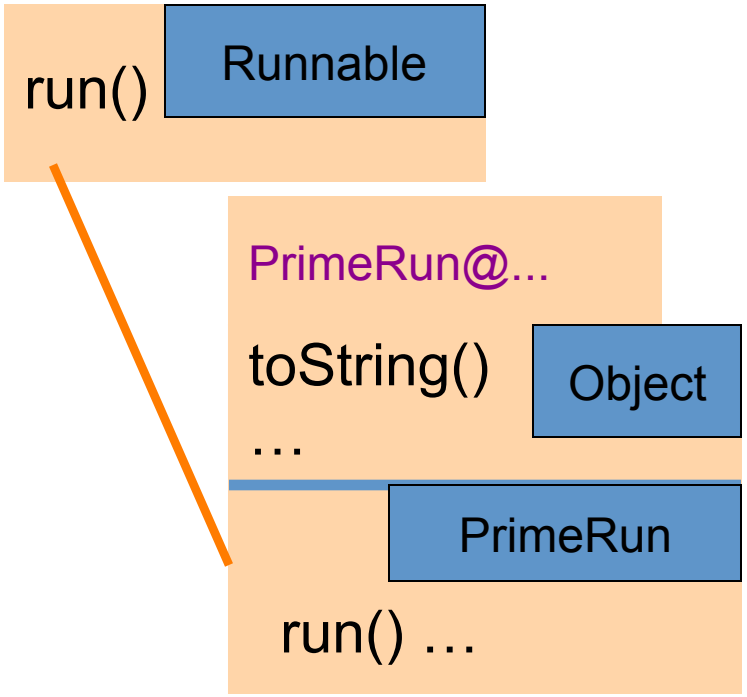


Method start() will call p's method run() in the new thread of execution

A Java Thread runs a Runnable object

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

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

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

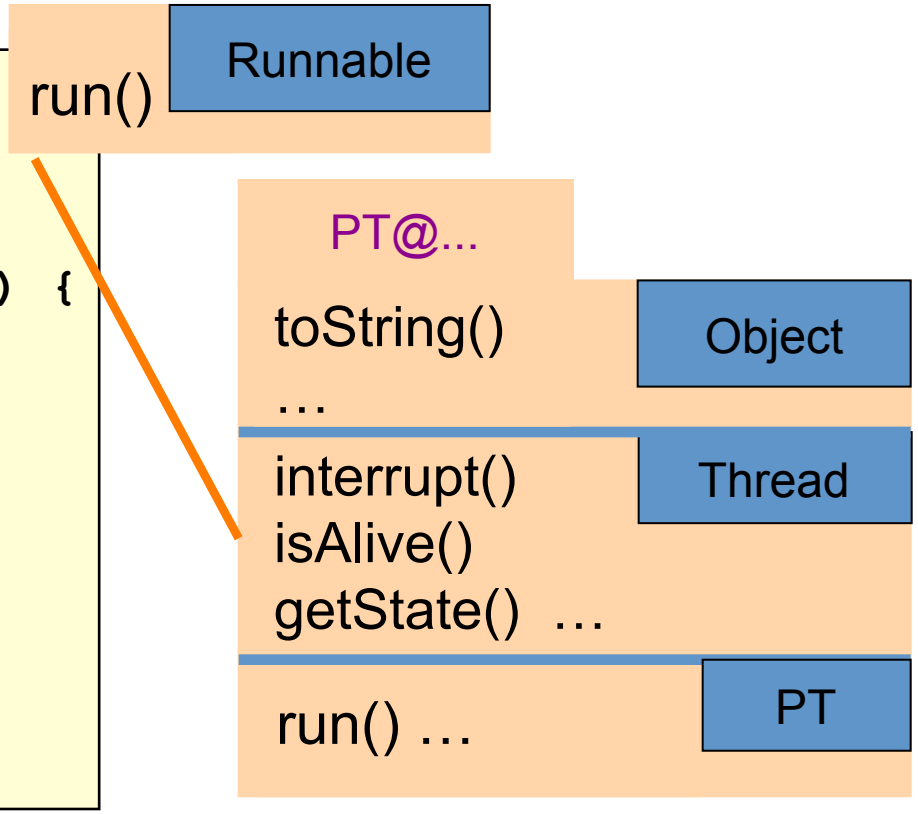
```
PrimeRun p=
        new PrimeRun(143, 195);
p.run();
```



No new thread!!!
run() runs in same thread as its caller.

Another way of creating a Thread

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

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

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

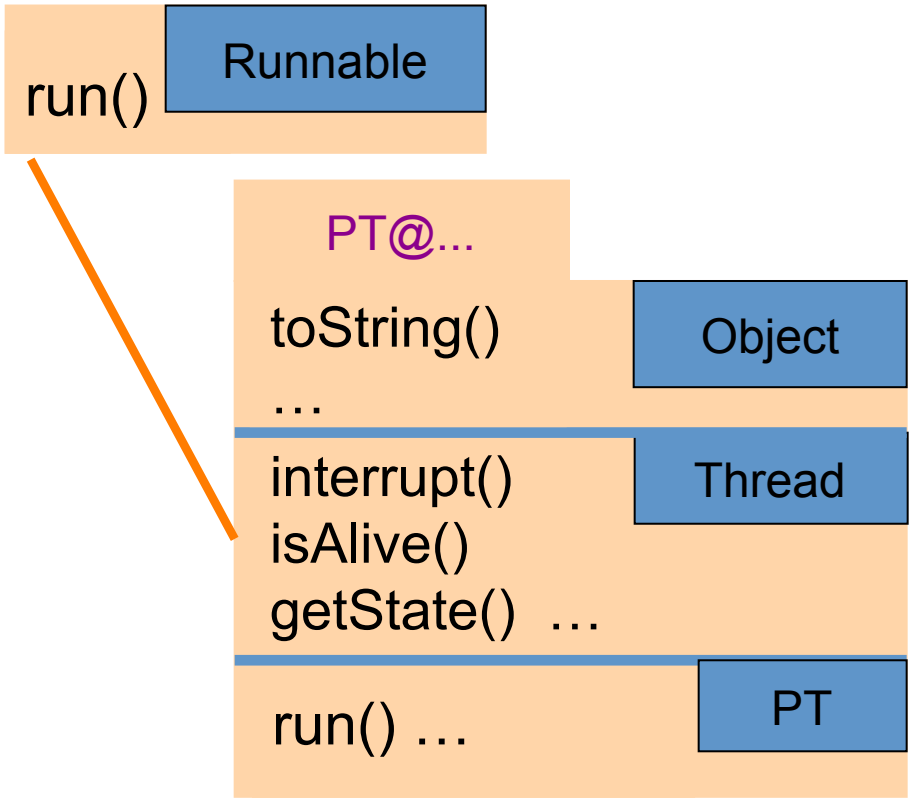



```
PrimeThread p=
      new PrimeThread(143, 195);
p.start();
```

Class Thread has methods to allow more control over threads

Class Thread has methods to handle threads

You can interrupt a thread, maintain a group of threads, set/change its priority, sleep it for a while, etc.



PrimeThread extends Thread, which implements Runnable


NBC
1660
KENYA
2367


NBC
LAP 65.37
9600m
Final Lap
26:29.0

Race Condition

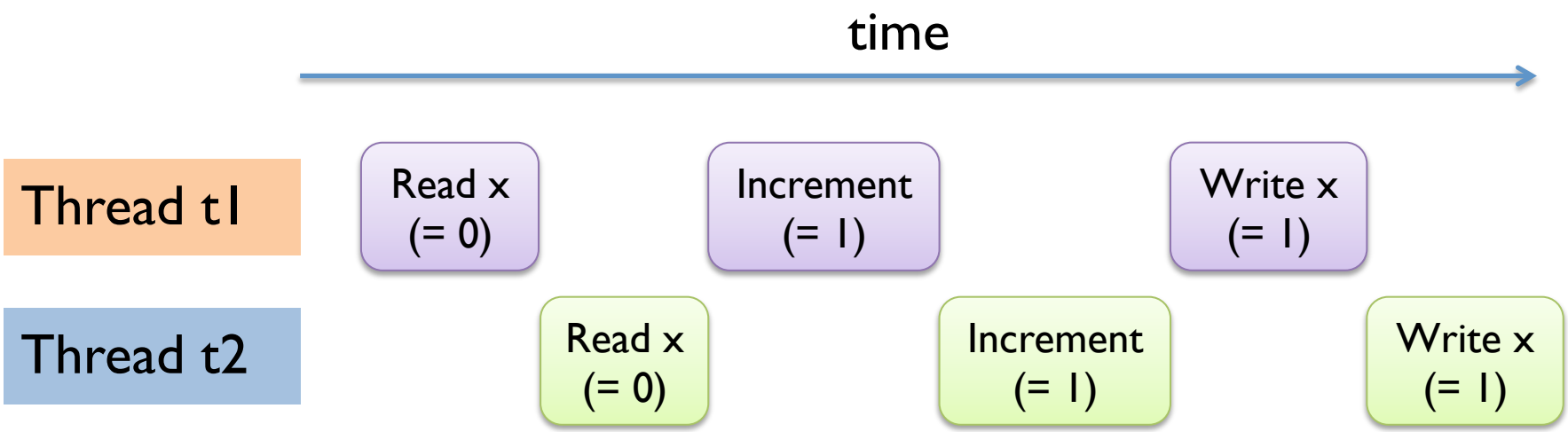
- Two or more simultaneous threads of execution (concurrency)
- Outcome depends on the exact order in which they are executed
- … which cannot be predicted in advance
  - Betting on races does not guarantee winnings
  - Two chefs can cook great dishes one after the other, but not if they're trying to simultaneously use the same stove

Race conditions yield unexpected results

Suppose x is initially 0

Thread t1	Thread t2
x= x + 1;	x= x + 1;

... after finishing, x = 1, not 2! Why?



Synchronization

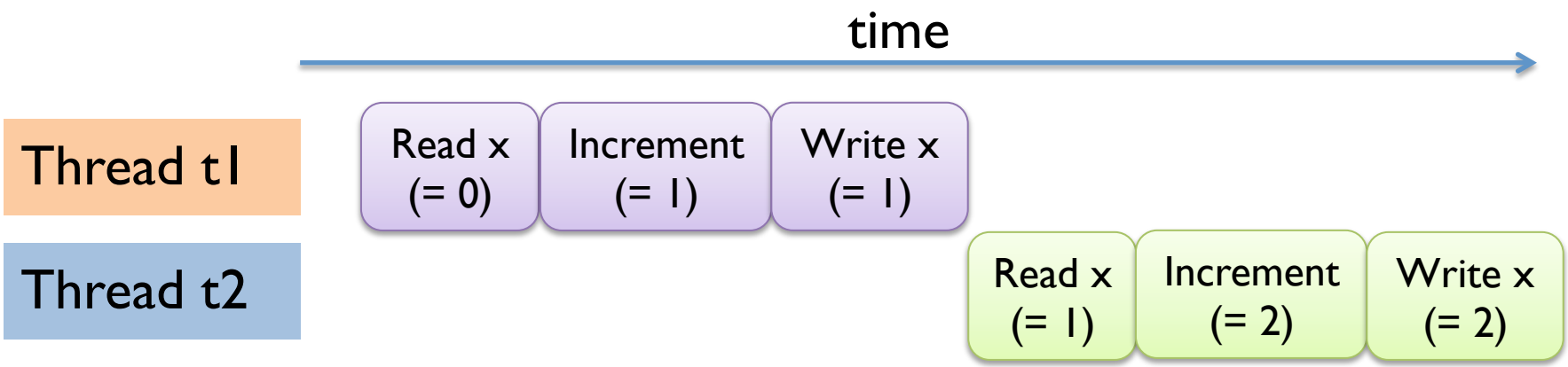
- Writing correct concurrent programs is very hard
  - Ideally, two threads would never access the same data
  - This is frequently unrealistic
- We need some form of **synchronization**
  - E.g. ensure a thread completes its read-modify-write sequence on a piece of data before another thread is allowed to touch it
  - E.g. ensure a thread accesses a resource only after another thread has finished accessing it
- There are many methods. We will only look at Java's synchronized keyword.

Fixing the x= x + 1 bug

```
class Counter {
  int value = 0;

  public synchronized void inc() {
    value= value + 1;
   }
}
```

Only one thread can execute this method on a given counter at a time

```
class CounterThread
      extends Thread {

  static Counter x=
         new Counter();

  public void run() {
    x.inc();
  }
}
```



The synchronized block

```
Stack<String> s= new Stack<String>();
```

```
{
   This is a block of code
}
```

```
synchronized(s) {
    This is a synchronized
    block of code
}
```

Only one thread can be executing a block B synchronized on s at any given time. All other threads trying to execute a block synchronized on s (need not be the same as B) must wait until the first thread finishes executing B.

The synchronized block is a primary tool for eliminating shared data problems. (There are others)

Accessing a stack in a threadsafe way

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

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

Synchronized block

- Put critical operations in a **`synchronized`** block
- The **`Stack`** object acts as a lock
- Only one thread can own the lock at a time
- Make synchronized blocks as small as possible

Locking on this, and synchronized methods

You can lock on any object, including `this`.

```
public void doSomething(){
    synchronized (this) {
        // body
    }
}
```

Note: the whole body is synchronized on **this**. There's a shorthand for this in Java

is equivalent to

```
public synchronized void doSomething(){
    // body
}
```

A threadsafe `Stack<T>` class will have

public synchronized T pop() { … }

public synchronized void push(…) { … } etc

Note: the lock is **this** stack. Two threads can access two *different* stacks simultaneously.

Synchronized collections

- Study class Collections and the following methods before working on A8:

  synchronizedCollection

  synchronizedSet

  synchronizedSortedSet

  synchronizedList

  synchronizedMap

  synchronizedSortedMap