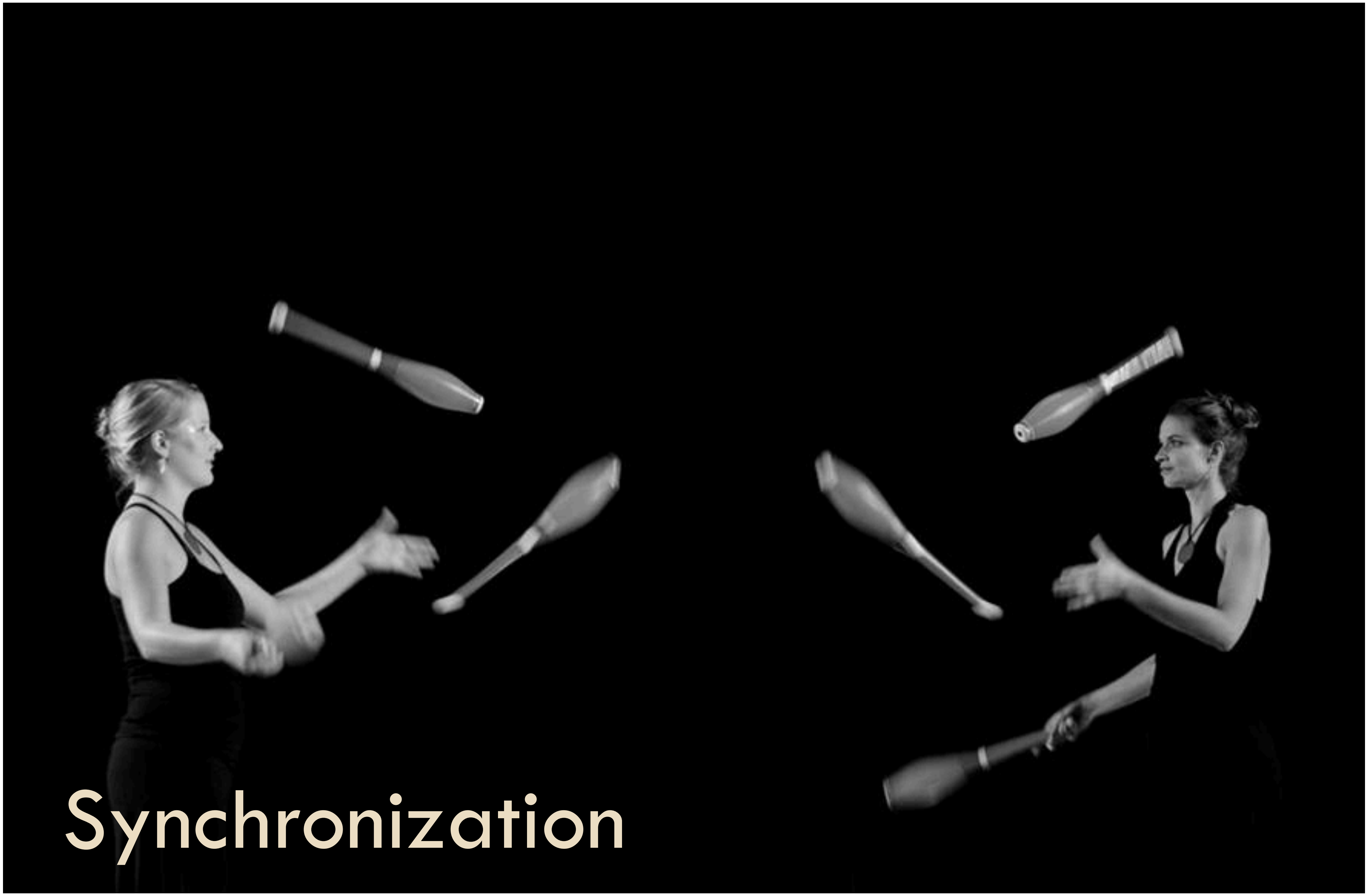


Synchronization

Lecture 23 – Fall 2017

Announcements

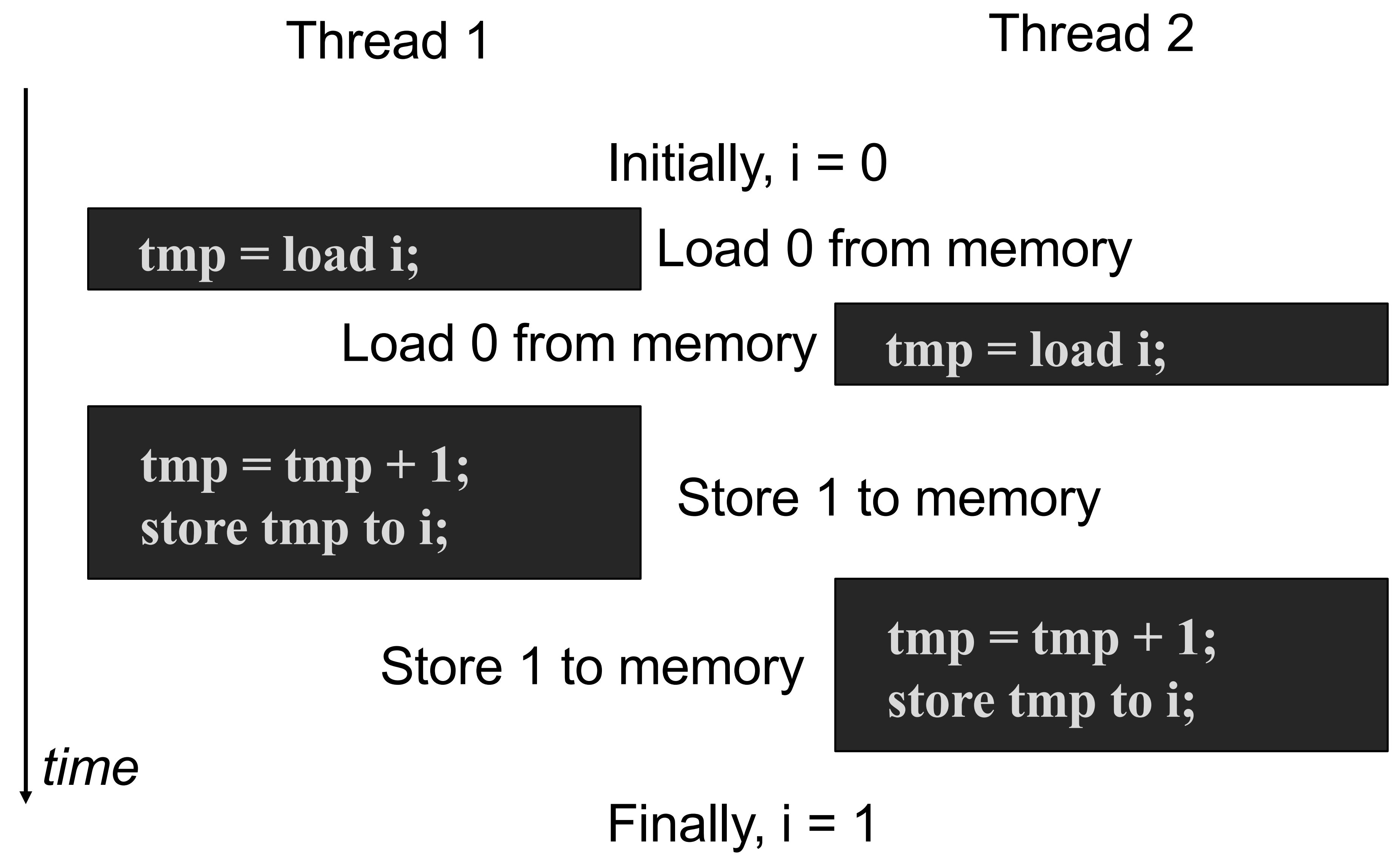
- A8 released today, Due: 11/21
  - Late deadline is after Thanksgiving
  - You can use your A6/A7 solutions or ours
  - A7 correctness scores have been posted
  - Next week's recitation will focus on A8
- Prelim 2 is in one week
  - Deadline for conflicts is today
  - Review session on Sunday 11/14

Concurrent Programs

A *thread* or *thread of execution* is a sequential stream of computational work.

Concurrency is about controlling access by multiple *threads* to shared resources.

Race Conditions



```
tmp = load i;
```

```
tmp = load i;
```

```
tmp = tmp + 1;
store tmp to i;
```

```
tmp = tmp + 1;
store tmp to i;
```

Race Conditions

A race condition is a situation in which the result of executing two or more processes in parallel can depend on the relative timing of the execution of the processes.

- A race condition can arises if two threads try to read and write the same data.
- Often occurs if a thread might see the data in the middle of an update (in a "inconsistent stare")
- Can lead to subtle and hard-to-fix bugs
- Solved by synchronization

Purpose of this lecture

Show you Java constructs for eliminating race conditions, allowing threads to access a data structure in a safe way but allowing as much concurrency as possible. This requires

- (1) The locking of an object so that others cannot access it, called synchronization.
- (2) Use of other Java methods: Wait() and NotifyAll()

As an example, throughout, we use a bounded buffer.

An Example: Bounded Buffers

finite capacity (e.g. 20 loaves)
implemented as a queue

Threads A: produce loaves of bread and put them in the queue

Threads B: consume loaves by taking them off the queue

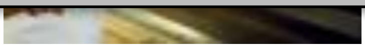
An Example: Bounded Buffers

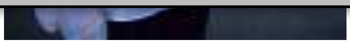
finite capacity (e.g. 20 loaves)

implemented as a queue

Separation of concerns:

1. How do you implement a queue in an array?
2. How do you implement a bounded buffer, which allows producers to add to it and consumers to take things from it, all in parallel?



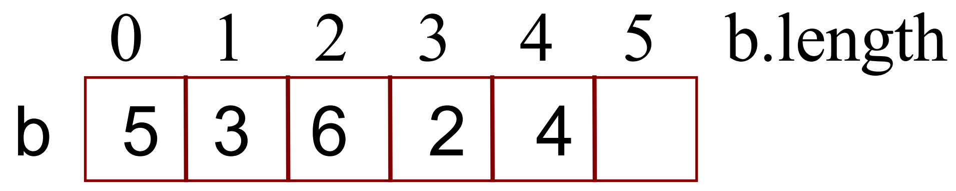


Threads A: produce loaves of bread and put them in the queue

Threads B: consume loaves by taking them off the queue

ArrayQueue

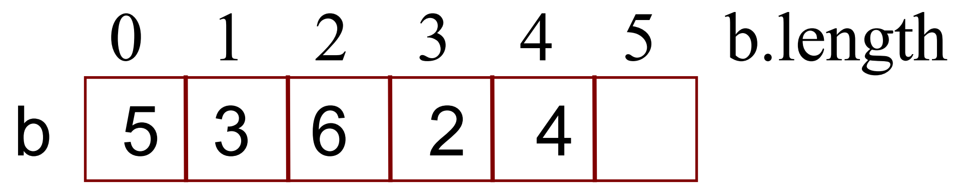
Array b[0..5]



put values 5 3 6 2 4 into queue

ArrayQueue

Array b[0..5]



put values 5 3 6 2 4 into queue

get, get, get

ArrayQueue

Array b[0..5]

	0	1	2	3	4	5	b.length
b	3	5		2	4	1	

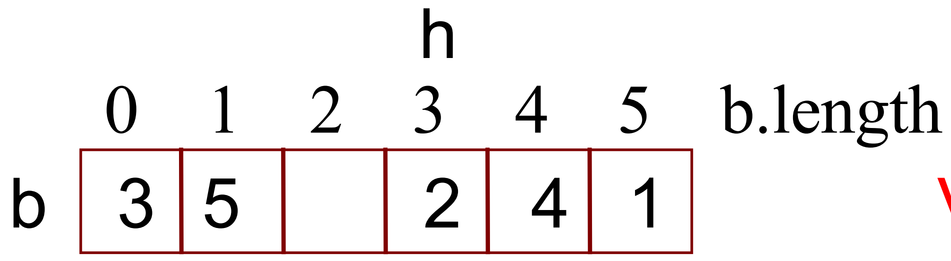
Values wrap around!!

put values 5 3 6 2 4 into queue

get, get, get

put values 1 3 5

ArrayQueue



Values wrap around!!

```
int[] b; // The array elements of the queue are in
int h;   // location of head, 0 <= h < b.length
int n;   // number of elements currently in queue
```

```
/** Pre: there is space */
public void put(int v){
    b[(h+n) % b.length]= v;
    n= n+1;
}
```

```
/** Pre: not empty */
public int get(){
    int v= b[h];
    h= (h+1) % b.length
    n= n-1;
    return v;
}
```

Bounded Buffer

```
/** An instance maintains a bounded buffer of fixed size */
class BoundedBuffer<E> {

    ArrayQueue<E> aq;

    /**  Put v into the bounded buffer.*/
    public void produce(E v) {
          if(!aq.isFull()){ aq.put(v) };
    }


    /**  Consume v from the bounded buffer.*/
    public E consume() {
          aq.isEmpty() ? return  null : return aq.get();
    }
}
```

Synchronized Blocks

a.k.a. *locks* or *mutual exclusion*

```
synchronized (q) {
  if (!q.isEmpty()) {
    q.remove();
  }
}
```

At most one consumer thread can be trying to remove something from the queue at a time.

While this method is executing the synchronized block, object aq is locked. No other thread can obtain the lock.

Synchronized Blocks

```
public void produce(E v) {
    synchronized(this){
        if(!aq.isFull()){ aq.put(v); }
    }
}
```

You can synchronize (lock) any object, including this.

Synchronized Methods

```
public void produce(E v) {
    synchronized(this){
        if(!aq.isFull()){ aq.put(v); }
    }
}
```

You can synchronize (lock) any object, including this.

```
public synchronized void produce(E v) {
    if(!aq.isFull()){ aq.put(v); }
}
```

Or you can synchronize methods
This is the same as wraping the entire method implementation in a synchronized(this) block

Bounded Buffer

```
/** An instance maintains a bounded buffer of fixed size */
class BoundedBuffer<E> {

    ArrayQueue<E> aq;

    /**  Put v into the bounded buffer.*/
    public synchronized void produce(E v) {
         if(!aq.isFull()){ aq.put(v); }
    }

}
```

What happens of aq is full?

We want to wait until it becomes non-full —until there is a place to put v.

Somebody has to buy a loaf of bread before we can put more bread on the shelf.

Wait()

For every synchronized object *sobj*, Java maintains:

1. **locklist:** a list of threads that are waiting to obtain the lock on *sobj*
2. **waitlist:** a list of threads that had the lock but executed wait()
   - e.g., because they couldn't proceed

wait() is a method defined in Object

Wait()

```
/** An instance maintains a bounded buffer of fixed size */
class BoundedBuffer<E> {

    ArrayQueue<E> aq;

    /**  Put v into the bounded buffer.*/
    public synchronized void produce(E v) {
        while(aq.isFull()){
            try { wait(); }
            catch(InterruptedException e){}
        }
        aq.put(v);
        notifyAll()
    }

    ...
}
```

need while loop (not if statement) to prevent race conditions

puts thread on the wait list

threads can be interrupted if this happens just continue

notify() and notifyAll()

- notify() and notifyAll() are methods defined in Object
- notify() moves one thread from the waitlist to the locklist
  - Note: which thread is moved is arbitrary
- notifyAll() moves all the threads on the waitlist to the locklist

notify() and notifyAll()

```
/** An instance maintains a bounded buffer of fixed size */
class BoundedBuffer<E> {

     ArrayQueue<E> aq;

     /**  Put v into the bounded buffer.*/
     public synchronized void produce(E v) {
          while(aq.isFull()){
               try { wait(); }
               catch(InterruptedException e){}
          }
          aq.put(v);
          notifyAll()
     }
     ...
}
```

WHY use of notify() may hang.

Work with a bounded buffer of length 1.
1. Consumer W gets lock, wants White bread, finds buffer empty, and wait()s: is put in set 2.
2. Consumer R gets lock, wants Rye bread, finds buffer empty, wait()s: is put in set 2.
3. Producer gets lock, puts Rye in the buffer, does notify(), gives up lock.
4. The notify() causes one waiting thread to be moved from set 2 to set 1. Choose W.
5. No one has lock, so one Runnable thread, W, is given lock. W wants white, not rye, so wait()s: is put in set 2.
6. Producer gets lock, finds buffer full, wait()s: is put in set 2.
All 3 threads are waiting in set 2. **Nothing more happens.**

Two sets:

1. Runnable: threads waiting to get lock.

2. Waiting: threads waiting to be notified

Should one use notify() or notifyAll()

But suppose there are two kinds of bread on the shelf —and one still picks the head of the queue, if it's the right kind of bread.

Using notify() can lead to a situation in which no one can make progress.

notifyAll() always works; you need to write documentation if you optimize by using notify()

Using Concurrent Collections...

Java has a bunch of classes to make synchronization easier.

It has synchronized versions of some of the Collections classes

It has an Atomic counter.

From spec for HashSet

… this implementation is not synchronized. If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using method Collections.synchronizedSet This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

Race Conditions

Thread 1 | Thread 2

Initially, i = 0

```
tmp = load i;
```

Load 0 from memory

Load 0 from memory

```
tmp = load i;
```

```
tmp = tmp + 1;
store tmp to i;
```

Store 1 to memory

Store 1 to memory

```
tmp = tmp + 1;
store tmp to i;
```

time

Finally, i = 1

Using Concurrent Collections...

```
import java.util.concurrent.atomic.*;

public class Counter {
  private static AtomicInteger counter;

  public Counter() {
    counter= new AtomicInteger(0);
  }

  public static int getCount() {
    return counter.getAndIncrement();
  }
}
```

Summary

Use of multiple processes and multiple threads within each process can exploit concurrency

- may be real (multicore) or virtual (an illusion)

Be careful when using threads:

- synchronize shared memory to avoid race conditions
- avoid deadlock

Even with proper locking concurrent programs can have other problems such as “livelock”

Serious treatment of concurrency is a complex topic (covered in more detail in cs3410 and cs4410)

Nice tutorial at http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html