The final is optional

As soon as A8 is graded and all grades are on the CMS, we will determine a tentative letter grade for you.

(1) You can accept it, and that will be your course grade.
(2) You can decide to take the final with the hope of raising your grade. Taking the final can decrease as well as raise your course grade.

You will tell us your decision on the CMS.

Please don’t email in the coming weeks, asking where you stand and whether you should take the final! We can’t say at this point! Look at your prelim averages. That gives you a rough idea what grade you may be getting.

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.

Last time: Learned about
1. Race conditions
2. Deadlock
3. How to create a thread in Java.

An Example: bounded buffer

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 buffer

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?
ArrayQueue

Array b[0..5]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

put values 5 3 6 2 4 into queue

ArrayQueue

Array b[0..5]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

put values 5 3 6 2 4 into queue

get, get, get

ArrayQueue

Array b[0..5]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Values wrap around!!

put values 5 3 6 2 4 into queue

get, get, get

put values 1 3 5

ArrayQueue

Array b[0..5]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Values wrap around!!

int[] b; // 0 <= h < b.length. The queue contains the
int h; // n elements b[h], b[h+1], b[h+2], ...
int n; // b[h+n-1] (all indices mod b.length)

/** 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() {
        if(aq.isEmpty()) ? return null : return aq.get();
    }
}

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); }
    }

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

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

Problems

1. Chef doesn’t easily know whether bread was added.
2. Suppose
   (a) First chef finds it not full.
   (b) another chef butts in and adds a bread
   (c) First chef tries to add and can’t because
      it’s full. Need a way to prevent this
**Synchronized block**

a.k.a. locks or mutual exclusion

```java
synchronized (object) { ... }
```

**Execution of the synchronized block:**
1. “Acquire” the object, so that no other thread can acquire it and use it.
2. Execute the block.
3. “Release” the object, so that other threads can acquire it.

**1.** Might have to wait if other thread has acquired object.

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

---

**Bounded Buffer**

```java
/** 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) {
        synchronized (aq) {
            if (!aq.isFull()) { aq.put(v); }
        }
    }
    // After finding aq not full, but before putting v, another chef might beat you to it and fill up buffer aq!
}
```

**Synchronized Methods**

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

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

**Or you can synchronize methods**

This is the same as wrapping the entire method implementation in a synchronized(this) block.

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

---

**Bounded buffer**

```java
/** 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); }
    }
    // What happens of aq is full?
    public synchronized void produce(E v) {
        if (!aq.isFull()) { aq.put(v); }
    }
    // 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.
    // After finding aq not full, but before putting v, another chef might beat you to it and fill up buffer aq!
}
```
Two lists for a synchronized object

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

   Method `wait()` is defined in `Object`

**notify()** and **notifyAll()**

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

**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(): is put in set 2.
2. Consumer `R` gets lock, wants Rye bread, finds buffer empty, wait(): 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(): is put in set 2.
6. Producer gets lock, finds buffer full, wait(): is put in set 2.
   All 3 threads are waiting in set 2. **Nothing more happens.**

**Wait()**

```java
class BoundedBuffer<E> {
    ArrayQueue<E> aq;
    /** 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()
        }
    } notifyAll()
    ...
    locklist waitlist
```

**notify()** and **notifyAll()**

```java
/** 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()
    }
    ...
```

**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()`
Eclipse Example

Producer: produce random ints
Consumer 1: even ints
Consumer 2: odd ints
Dropbox: 1-element bounded buffer

Locklist
Threads wanting the Dropbox
Waitlist
Threads who had Dropbox and waited

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.

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

Initially, i = 0
Load 0 from memory
Load 0 from memory
Store 1 to memory
Store 1 to memory
Finally, i = 1

Fancier forms of locking

Java. synchronized is the core mechanism.
But, Java has a class Semaphore. It can be used to allow a limited number of threads (or kinds of threads) to work at the same time. Acquire the semaphore, release the semaphore.

The Windows and Linux and Apple O/S have kernel locking features, like file locking.
Python: acquire a lock, release the lock. Has semaphores.

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