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.

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 two new Java methods: wait() and notifyAll() as an example, throughout, we use a bounded buffer.

Look at JavaHyperText, entry Thread !!!!!!!
ArrayQueue

Array b[0..5]

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

put values 5 3 6 2 4 into queue

ArrayQueue

Array b[0..5]

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

Values wrap around!!

put values 5 3 6 2 4 into queue

get, get, get

put values 1 3 5

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] { (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() {
        return aq.isEmpty() ? null : aq.get();
    }
}

Problems

1. Chef doesn't easily know whether bread was added.
2. Suppose race condition
   (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
/** 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); }  /* After finding aq not full, but before putting v, another chef might beat you to it and fill up buffer aq! race condition */
    }
}

Synchronized block

/** 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) {  /* Synchronized block */
            if (!aq.isFull()) { aq.put(v); }
        }
    }

    public synchronized void produce(E v) {  /* Synchronized Methods */
        synchronized(this) {  /* You can synchronize (lock) any object, including this. */
            if (!aq.isFull()) { aq.put(v); }
        }
    }
}

Synchronized Methods

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

public synchronized 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

/** 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); }
    }

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

Use of synchronized

Key is hanging the outhouse.
Anyone can grab the key, go inside, and lock the door. They have the key.

When they come out, they lock the door and hang the key by the front door. Anyone (only one) person can then grab the key, go inside, lock the door.
That’s what synchronized implements!

Bounded Buffer

When they come out, they lock the door and hang the key by the front door. Anyone (only one) person can then grab the key, go inside, lock the door.
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`
- `notifyAll()` moves all threads on the waitlist to the locklist
- `notify()` moves one thread from the waitlist to the locklist
  - Note: which thread is moved is arbitrary

**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.

**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 (rye or white bread)
Consumer 1: even ints (buy only rye bread)
Consumer 2: odd ints (buy only white bread)
Dropbox: 1-element bounded buffer (1 loaf of bread)

Locklist
Threads wanting the Dropbox
Waitlist
Threads who had Dropbox and waited
Dropbox: empty or 1 integer (loaf of bread)

Word of warning with synchronized

Key is hanging by front door.
Anyone can grab the key, go inside, and lock the door. They have the key.

But: You leave the back door open and tell your friends to go in whenever they want
Threads that don’t synchronize can get in. Dangerous but useful to increase efficiency.

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:

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

Race Conditions

Thread 1
Initially, i = 0
```java
tmp = load i;  Load 0 from memory
Load 0 from memory  tmp = load i;
tmp = tmp + 1;  Store tmp to i;
store tmp to i;
��time��
```
Finally, i = 1

Thread 2
```java
tmp = load i;  Load 0 from memory
Load 0 from memory  tmp = load i;
tmp = tmp + 1;  Store tmp to i;
store tmp to i;
��time��
```

Using Concurrent Collections...

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

class Counter {
    private static AtomicInteger counter;

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

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

Semaphore: a kind of synchronized counter (invented by Dijkstra in 1962-63, THE multiprogramming system)

The Windows and Linux and Apple O/S have kernel locking features, like file locking.

Python: acquire a **lock**. release the **lock**. Has semaphores.

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