RACE CONDITIONS & SYNCHRONIZATION

Lecture 23 – CS2110 – Spring 2016

Announcements


Recap

☐ A “race condition” arises if two threads try to read and write the same data
☐ Might see the data in the middle of an update in an inconsistent store
☐ A “race condition”: correctness depends on the update racing to completion without the reader managing to glimpse the in-progress update
☐ Synchronization (also known as mutual exclusion) solves this

Java Synchronization (Locking)

• You can lock on any object, including this

```java
public synchronized void doSomething() {
    ...
}
```

Above is syntactic sugar for the stuff below. They mean the same thing

```java
public void doSomething() {
    synchronized (this) {
        ...
    }
}
```

Deadlock involve cycles:

Simple solution to deadlock:
Acquire resources in a certain order:
1. think
2. eat (2 forks)
eat is then:
1. pick up smaller fork
2. pick up bigger fork
pick up food, shovel it
put down bigger fork
put down smaller fork
notify(), notifyAll(), wait(), wait(n)

There are times when a method in a synchronized block has to relinquish the lock and acquire it later.
Example: It needs to get an element from a set and the set is empty.
Example: It needs to add something to a set that can’t hold any more; must wait until something is deleted from the set.

For these reasons, these methods are in class Object:

notify(), notifyAll(), wait(), wait(n)

About wait(), wait(n), notify(), notifyAll()

A thread that holds a lock on object OB and is executing in its synchronized code can make (at least) these calls.

1. wait(); It is put into set 2. Another thread from set 1 gets the lock.
2. wait(n); It is put into set 2 and stays there for at least n milliseconds. Another thread from set 1 gets the lock.
3. notify(); Move one “possible” thread from set 2 to set 1.
4. notifyAll(); Move all “threads” from set 2 to set 1.

Two sets:
1. Runnable threads: Threads waiting to get the OB lock.
2. Waiting threads: Threads that called wait and are waiting to be notified.

Important example: bounded buffer

We illustrate these methods using an important example, which you should study and understand. Bounded Buffer

Example: A baker produces breads and puts them on the shelf, like a queue. Customers take them off the shelf.
- Threads A: produce loaves of bread and put them in the queue
- Threads B: consume loaves by taking them off the queue
- This is the produce/consumer model, using a bounded buffer, the shelf (which can contain at most 20 (say) loaves of bread).

Array implementation of a queue of max size 6

Array b[0..5]

<table>
<thead>
<tr>
<th>b</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

push values 5 3 6 2 4

For later purposes, we show how to implement a bounded queue—one with some maximum size—in an array.

A neat little implementation! We give you code for it on course website.

Values wrap around!!

Array implementation of a queue of max size 6

Array b[0..5]

<table>
<thead>
<tr>
<th>b</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

push values 5 3 6 2 4
pop, pop, pop
push value 1 3 5
**Bounded buffer**

```java
/** Implement a bounded-size queue in an array */
public class ArrayQueue<E> {
    int[] b; // The n elements of the queue are in
    int n; // b[h], b[(h+1) % b.length], ... b[(h+n-1) % b.length]
    int h; // 0 <= h < b.length

    public ArrayQueue(int n) {b = new int[n];}

    public void put(E e) {
        // Add e to the queue (error if full)
        b[(h+n) % b.length] = e;
        n = n+1;
    }

    public int get() {
        // remove head of queue and return it
        int e = b[h];
        // (error if empty)
        h = (h+1) % b.length; n = n-1;
        return e;
    }
}
```

Also other methods, like `isEmpty()`, `isFull()`.

**Bounded Buffer**

```java
/** An instance maintains a bounded buffer of limited size */
class BoundedBuffer {
    ArrayQueue aq; // bounded buffer is implemented in aq

    /** Put v into the bounded buffer. */
    public synchronized void produce(int v) {
        while (aq.isFull()) { try { wait(); } catch (InterruptedException e) {} }
        aq.put(v);
        this.notifyAll(); // Signal: not empty.
    }

    /** Remove first element from bounded buffer and return it. */
    public synchronized int consume() {
        while (aq.isEmpty()) { try { wait(); } catch (InterruptedException e) {} }
        int item = aq.get();
        this.notifyAll(); // Signal: not full
        return item;
    }
}
```

Things to notice

- Use a `while` loop because we can’t predict exactly which thread will wake up “next”
- `wait()` waits on the same object that is used for synchronizing (in our example, this, which is this instance of the bounded buffer)
- Method `notify()` wakes up one waiting thread, `notifyAll()` wakes all of them up

Problem with the above. The wait may throw an `InterruptedException`. The thread was interrupted. Need a throws clause OR … (next slide)
In an ideal world...

- Bounded buffer allows producer and consumer to run concurrently, with neither blocking
  - This happens if they run at the same average rate
  - ... and if the buffer is big enough to mask any brief rate surges by either of the two

- But if one does get ahead of the other, it waits
  - This avoids the risk of producing so many items that we run out of computer memory for them. Or of accidentally trying to consume a non-existent item.

Should one use notify() or notifyAll()?

- Lots of discussion on this on the web!
  - stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again
- notify() takes less time than notifyAll()
  - In consumer/producer problem, if there is only one kind of consumer (or producer), probably notify() is OK.
  - 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. Then, using notify() can lead to a situation in which no one can make progress. We illustrate with a project in Eclipse, which we will put on the course website.

Another example: simple counter

```java
/** An instance is a counter; can be incremented */
class Counter {
    private int counter = 0;
    /** increment counter and return it */
    public int increment() {
        return counter = counter + 1;
    }
}
```

Using synchronization

```java
class Counter {
    private int counter = 0;
    /** increment counter and return it */
    public synchronized int increment() {
        return counter = counter + 1;
    }
}
```

Using Concurrent Collections...

Java has a bunch of classes to make synchronization easier.

- It has an Atomic counter.
- It has synchronized versions of some of the Collections classes

```java
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

Conference on Jay Misra's retirement

- Gries was at a conference in Austin, Texas, in honor of the retirement of Jay Misra, a professor there.
- Two days of 15-minute talks by well-known computer scientists (including Tony Hoare), dealing mainly with Jay's main interests:
  - Concurrency
  - Correctness proofs
  - Programming methodology
- Both theory and practice
- Major in CS and you will hear more about the problems in these areas. Do you PhD, and you may work in these areas and help solve the problems.