Race Conditions & Synchronization

Lecture 25 – CS2110 – Fall 2016

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 a inconsistent stare”
  - 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)


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

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

`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

Java Synchronization (Locking)

- You can lock on any object, including **`this`**

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

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

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

Monitor design pattern

Threads often need to wait for something

- e.g. “the queue is full”, “the queue is empty”, “there’s space in the class”, “nobody’s going the opposite way on the bridge”, ...

A **Monitor** is a class that monitors the conditions under which threads can proceed or must block.

```
class Queue<E> {
  /** wait until elt. available,
   * remove and return it. */
  E poll();

  /** wait until space avail.
   * add elt. */
  void offer(E);
}
```

```
class Course {
  /** wait until class not full,
   * then enroll */
  void enroll();
}
```

```
class OneLaneBridge {
  /** wait until safe to go north on bridge;
   * then start crossing. */
  enterNorth();

  /** sim. */
  enterSouth();

  /** vacate the bridge */
  leaveNorth();
  leaveSouth();
}
```

Implementing a monitor

1. Write the method names, types, and specs
   - make them synchronized
2. Write down the things you want to wait for
   - they should be **predicates**: you should be able to tell if they are true or false by looking only at the fields
   - document them with your class invariants and fields
3. When you need something to be true, call **wait() while it is false**
4. When you make something true, be sure to **notifyAll()** of the waiting threads.

People always misuse wait() and notify() and it leads to broken programs! Don’t! Always follow this pattern.

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 ... e loaves by taki... off the queue
- This is th... /c... od... bounded ... s... ca... at most 20 (say) l... read).

producer

shelves

consumer

Array implementation of a queue of max size 6

Array b[0..5]

	0	1	2	3	4	5	b.length
b	5	3	6	2	4		

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.

Array implementation of a queue of max size 6

Array b[0..5] n = 6

	0	1	2	3	4	5	b.length
b	5	3	6	2	4		

push values 5 3 6 2 4

pop, pop, pop

Array implementation of a queue of max size 6

Array b[0..5]

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

Values wrap around!!

push values 5 3 6 2 4

pop, pop, pop

push value 1 3 5

Bounded Buffer


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

    /**  Constructor: empty bounded buffer of max size n*/
    public BoundedBuffer(int n) {
        aq= new ArrayQueue(n);
    }
}
```

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?

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

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

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 millisecs. 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

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 proje in Eclipse, which we will put on the course website.
- **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 an Atomic counter.

It has synchronized versions of some of the Collections classes

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