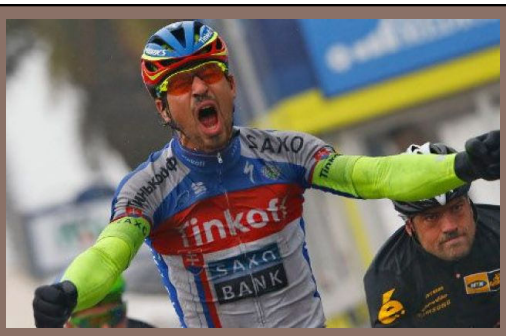

Race Conditions & Synchronization

Lecture 25 – Spring 2017

Gries office hours

A number of students have asked for time to talk over how to study for the prelim.

Gries will hold office hours at the times below for this purpose.

Also, read "basicLearningMethods.pdf" in the pinned Piazza note "Supplementary material".

Tues. (today) 1-4PM.

Wed. (tomorrow) 10-noon

How end of course works

A8 due last day of classes: Wednesday, 10 May

Firm deadline. Nothing after that.

We grade A8 as quickly as possible, then figure out tentative course letter grades. Make them available on the CMS.

You complete a CMS assignment: Do you accept grade or take final.

Taking the final can lower as well as raise your grade.

Final is Sunday, 21 May, at 2PM.

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.

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

Important example: bounded buffer

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 producer/consumer model, using a bounded buffer, the shelf (which can contain at most 20 (say) loaves of bread).

producer *shelf* *consumer*

Array implementation of a queue of max size 6

7

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

8

Array b[0..5]

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

pushed values 5 3 6 2 4

Now, pop, pop, pop

Array implementation of a queue of max size 6

9

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

Array implementation of a queue

h

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

Values wrap around!!

Fields and the class invariant

```
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
```

Insert 7 (assuming there's space)

```
b[(h+n)% b.length]= 5;
n= n+1;
```

Pop (assuming not empty)

```
value= b[h];
h= (h+1) % b.length
```

Java Synchronization (Locking)

11

```
public AQ<String> aq= new AQ<String>();

public void doSomethingTo-aq() {

    code to change aq

}
```

We need a way to prohibit other threads from using AQ aq while the code to change aq is being executed. For this purpose, we use Java keyword synchronize

Java Synchronization (Locking)

12

```
private AQ<String> aq= new AQ<String>();

public void doSomethingTo-aq() {
   synchronize {
      code to change aq
   }
}
```

`synchronized` block

While this method is executing the synchronized block, object aq is locked. No other thread can obtain the lock. Only one thread can own the lock at a time

Java Synchronization (Locking)

13

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

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

Why do this? Suppose method doSomething is within an object of class ArrayQueue, and we want to prohibit other methods within ArrayQueue from being executed.

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

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

Bounded Buffer

14

```
/** 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?

Bounded Buffer

15

```
/** 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(Integer v) {
        aq.put(v);
    }
}
```

We know more code will be put in method produce, and we want to be sure that no other method in this Bounded Buffer object can do anything with this object until method produce is finished. So stick in a synchronize keyword.

Bounded Buffer: producer

16

```
/** 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(Integer v) {
        aq.put(v);
    }
}
```

What happens of aq is full?

We have 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.
We use a while-loop to wait, and we also need to give p the lock so some other thread can buy (consume) a loaf of bread.

Bounded Buffer: producer

17

```
/** An instance maintains a bounded buffer of limited size */
class BoundedBuffer {
    ArrayQueue aq;  // bounded buffer implemented in aq
    /**  Put v into the bounded buffer.*/
    public synchronized void produce(Integer v) {
        while (aq.isFull())
            try { wait(); }
            catch (InterruptedExecution e) {}
        aq.put(v);
        … more to come …
    }
}
```

wait(): put on a list of waiting threads, give up lock so another thread can have it

Need a while-loop to wait. An if-statement will no work.

If the wait is interrupted for some reason, just continue

Bounded Buffer: producer

18

```
/** An instance maintains a bounded buffer of limited size */
class BoundedBuffer {
    ArrayQueue aq;  // bounded buffer implemented in aq
    /**  Put v into the bounded buffer.*/
    public synchronized void produce(Integer v) {
        while (aq.isFull())
            try { wait(); }
            catch (InterruptedExecution e) {}
        aq.put(v);
        notifyAll();
    }
}
```

Another thread may be waiting because the buffer is empty ---no more bread. Have to notify all waiting threads that it is not empty

The consumer —method consume— is similar.
Let's look at code.

Things to notice

19

- 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

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

20

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

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

21

A thread that 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.

3. notify(); Move one "possible" thread from set 2 to set 1.

4. notifyAll(); Move all threads from set 2 to set 1.

Notify() a lot less expensive than notifyAll(). Why not use it all the time? Because the wrong thread may be notified, giving deadlock

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()

22

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. We illustrate with a project in Eclipse, which is on the course website.

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

WHY use of notify() may hang.

23

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

Using Concurrent Collections...

24

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

25

… 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(...));

Using Concurrent Collections…

26

```
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

27

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