



Midterm Review

ICS432 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

What to expect

- Closed notes, up to and including the Classic Concurrency Problems” module
- General short-answer questions about sequential code optimization, concurrency, and threads in C/Java
- Some code with concurrency bugs to find
 - Java: the whole gamut of Java concurrency stuff
 - locks/conds: pseudo-code
 - semaphores: pseudo-code
 - classic problems: pseudo-code
- “Write pseudo-code” questions
 - Standard problems, nothing too creative
 - And really, do it in pseudo-code!
- Some practice questions have been posted on Laulima



Concurrency Abstractions

- Locks + Condition Variables
- Semaphores
- Java Monitors

- All can be implemented in terms of the others

Lock Implementation

- Software-only implementations are very challenging
- Disabling interrupts is not typically feasible for safety concerns
- So one uses **atomic** hardware instructions
 - .e.g., compare-and-swap
- These instructions make it possible to implement **spinlocks**
- Another kind of locks is **blocking lock**
 - Involvement of the OS
- Both kinds serve the same purpose, but have different overhead/cost behaviors

Condition Variables

- Implemented by the OS
 - wait(): block until somebody calls notify*()
 - notify(): wake up a blocked thread
 - notify_all(): wake up all blocked threads
- A condition variable is always associated with a lock:
 - Calling wait() releases the lock

Semaphores

- P()
 - Atomic
 - Wait until the value is >0
 - Decrement it by 1 and return
- V()
 - Atomic
 - Increment the value by 1 (to infinitum!!)
- Can have any integer initial values
- Typical:
 - value either 0 or 1: *binary semaphore*
 - value ≥ 0 : *counting semaphore*

In Java?

- Each object has a hidden lock and condition variable
 - Locking/unlocking done via the synchronized keyword
 - Condition variables methods are called wait(), notify(), notifyAll()
- One can also use Lock, ConditionVariables, Semaphore classes provided by the Java concurrency package

Semaphores with Monitors

- In class we have implemented several basic concepts in Java
 - Semaphores, Barriers, a Blocking Lock by hand, etc.
 - Make sure you understand those (simple) implementations
- Let's now use our Semaphore implementation to implement locks...
 - Which is so straightforward it hurts??

Locks with Semaphores?

```
public class Lock {  
    Semaphore sem;  
  
    public Lock() {  
        sem = new Semaphore(1);  
    }  
    public lock() {  
        sem.P();  
    }  
    public unlock() {  
        sem.V();  
    }  
}
```

Locks with Semaphores?

```
public class Lock {  
    Semaphore sem;  
  
    public Lock() {  
        sem = new Semaphore(1);  
    }  
    public lock() {  
        sem.P();  
    }  
    public unlock() {  
        sem.V();  
    }  
}
```

What's wrong with this?

Locks with Semaphores?

```
public class Lock {  
    Semaphore sem;
```

```
    public Lock() {  
        sem = new Semaphore(1);  
    }
```

```
    public lock() {  
        sem.P();  
    }
```

```
    public unlock() {  
        sem.V();  
    }
```

What's wrong with this?

Two calls to unlock() will give the semaphore a value of 2, unless the semaphore is implemented as a binary semaphore!

Binary Semaphore

```
public class Semaphore {
    int semaphore;

    ...

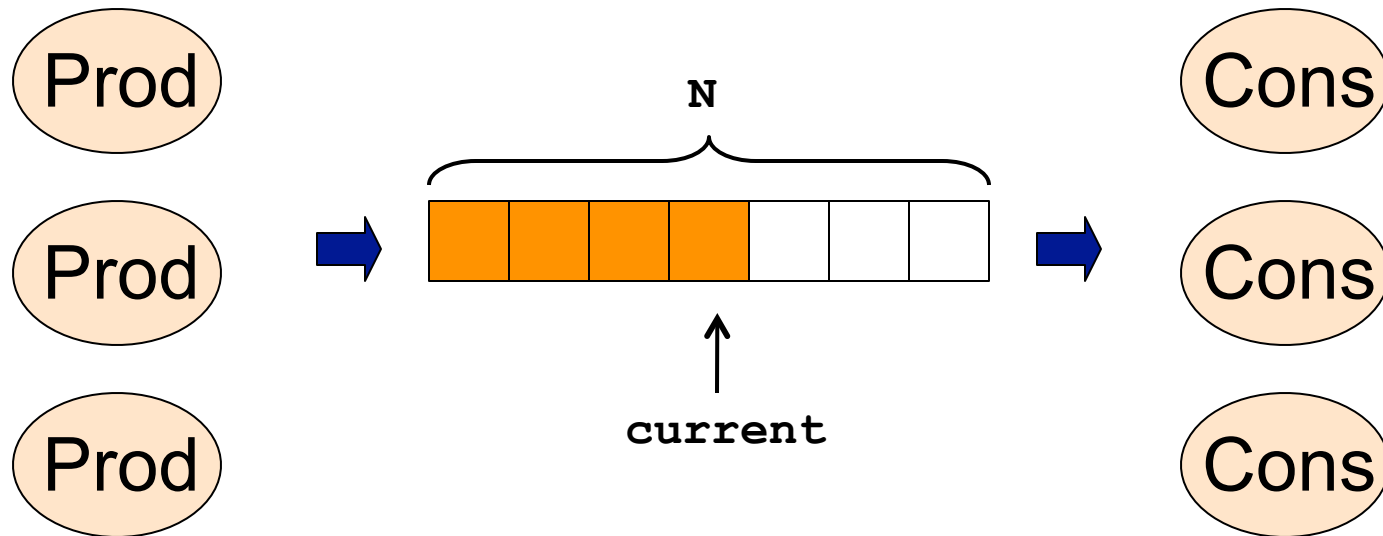
    public synchronized void V() {
        if (semaphore > 0)
            return; // unlocking an unlocked lock has no effect
        semaphore = 1;
        this.notify();
    }
}
```

So What?

- Probable a good idea to understand how to implement:
 - semaphores with locks and condition variables
 - locks and condition variables with semaphores
 - locks and condition variables with monitors
- And in Java using “monitors” (i.e., hidden locks and condition variables within objects)

Producer / Consumer

- The main abstraction we've seen is bounded Producer/Consumer



Solution with Semaphores

```
semaphore mutex=1, freeslots=N, takenslots=0;
```

```
int current=-1, buffer[N];
```

```
void producer() {  
    while(true) {  
        P(freeslots);  
        P(mutex);  
        current++;  
        buffer[current] = produce();  
        V(mutex);  
        V(takenslots);  
    }  
}
```

```
void consumer() {  
    while (true) {  
        P(takenslots);  
        P(mutex);  
        consume(buffer[current]);  
        current--;  
        V(mutex);  
        V(freeslots);  
    }  
}
```

Solution with Locks/Cond Vars

```
lock mutex;  
cond notfull, notempty;  
boolean empty=true, full=false;  
int current=-1, buffer[N];
```

```
void producer() {  
    while(true) {  
        lock(mutex);  
        if (full)  
            wait(notfull, mutex);  
        current++;  
        buffer[current] = produce();  
        empty = false;  
        if (current == N-1)  
            full = true;  
        unlock(mutex);  
    }  
}
```

```
void consumer() {  
    while (true) {  
        lock(mutex);  
        if (empty)  
            wait(notempty, mutex);  
        consume(buffer[current]);  
        current--;  
        full = false;  
        signal(notfull);  
        unlock(mutex);  
        if (current == -1)  
            empty = true;  
    }  
}
```

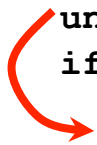
What's wrong with this code?

Solution with Locks/Cond Vars

```
lock mutex;  
cond notfull, notempty;  
boolean empty=true, full=false;  
int current=-1, buffer[N];
```

```
void producer() {  
    while(true) {  
        lock(mutex);  
        while (full)  
            wait(notfull, mutex);  
        current++;  
        buffer[current] = produce();  
        empty = false;  
        if (current == N-1)  
            full = true;  
        signal(notempty);  
        unlock(mutex);  
    }  
}
```

```
void consumer() {  
    while (true) {  
        lock(mutex);  
        while (empty)  
            wait(notempty, mutex);  
        consume(buffer[current]);  
        current--;  
        full = false;  
        signal(notfull);  
        unlock(mutex);  
        if (current == -1)  
            empty = true;  
    }  
}
```



One thing to note about signal()

- You should call signal() only once the boolean other threads are waiting for is true

```
while (!valid) {  
    wait(cond, mutex);  
}
```

```
valid = true;  
signal(cond);
```

```
signal(cond);  
valid = true;
```

The waiting thread may “miss” it and never be awakened again (rare bug because calling signal() in code after the things we’re signaling for has happened is very natural)

ProdCons with Monitors

```
monitor ProdCons {
    cond notempty, notfull;
    int buffer[N];
    int current=-1;
    void produce(int element) {
        while (current >= N-1) notfull.wait();
        current++;
        buffer[current] = element;
        notempty.notify();
    }
    int consume() {
        int tmp;
        while(current == -1) notempty.wait();
        tmp = buffer[current];
        current--;
        notfull.notify();
        return tmp;
    }
}
```

Straight Translation to Java

```
public class ProdCons {
    private int buffer[];
    private int current;
    private Object notfull, notempty;

    public ProdCons() { . . . . . }

    public void synchronized produce(int element) {
        while (current > N-1 ) { notfull.wait(); }
        current++;
        buffer[current] = element;
        notempty.notify();
    }

    public int synchronized consume() {
        while (current == -1) { notempty.wait(); }
        int tmp = buffer[current];
        current--;
        notfull.notify();
        return tmp;
    }
}
```



Is this OK?

Translation to Java: first try

```
public class ProdCons {
    private int buffer[];
    private int current;
    private Object notfull, notempty;

    public ProdCons() { . . . . . }

    public void synchronized produce(int element) {
        while (current > N-1 ) { notfull.wait(); }
        current++;
        buffer[current] = element;
        notempty.notify();
    }

    public int synchronized consume() {
        while (current == -1) { notempty.wait(); }
        int tmp = buffer[current];
        current--;
        notfull.notify();
        return tmp;
    }
}
```

should be in
synchronized(notfull) or in
synchronized(notempty)
blocks!!

We really have 3 locks
hidden here:
The one for “this”
The one for notfull
The one for notempty

We have to use them all

Brute-force Translation to Java

```
public class ProdCons {
    private int buffer[];
    private int current;
    private Object notfull, notempty;

    public ProdCons() { . . . . . }

    public void produce(int element) {
        synchronized(notfull) {
            while (current > N-1 ) {
                notfull.wait();
            }
        }
        synchronized(this) {
            current++;
            buffer[current] = element;
        }
        synchronized (notempty) {
            notempty.notify ();
        }
    }
    . . .
```

This is pretty messy

Using Semaphores could be cleaner

Typical Translation to Java

```
public class ProdCons {
    private int buffer[];
    private int current;

    public ProdCons() { . . . . . }

    public void synchronized produce(int element) {
        while (current > N-1 ) { this.wait(); }
        current++;
        buffer[current] = element;
        this.notifyAll();
    }

    public int synchronized consume() {
        while (current == -1) { this.wait(); }
        int tmp = buffer[current];
        current--;
        this.notifyAll();
        return tmp;
    }
}
```

One easy solution is just to wake up EVERYBODY and let whoever can get out of its while loop continue execution

It's a little bit wasteful



Locks, Conds, Sems, Monitors..

- Any question about all this?
- What about the homework assignments?

Reader/Writer

- Readers call `read()` on a shared object
- Writers call `write()` on a shared object
- We want to have either
 - 1 active writer, 0 active readers
 - N active readers, 0 active writers
- This is called “selective mutual exclusion”
- Question: how can we implement this?

- Should we review this?



Java and Concurrency

- Java has idiosyncrasies for Concurrency
- Thread interrupting, resuming, terminating
- The volatile keyword

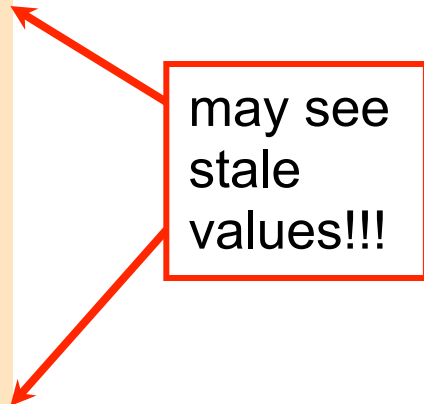
The volatile Keyword

- Volatile variables are synchronized across threads: **Each read of a volatile will see the last write to that volatile**

```
public class SomeClass {  
    private int var1;  
    private volatile int var2;  
  
    public int get1() {  
        return var1;  
    }  
  
    public int get2() {  
        return var2;  
    }  
}
```

```
SomeClass stuff;  
...  
// Thread 1  
System.out.println(stuff.get1());  
...  
System.out.println(stuff.get2());  
...  
  
...  
// Thread 2  
System.out.println(stuff.get1());  
...  
System.out.println(stuff.get2());  
...
```

may see
stale
values!!!



When Do I Use `volatile`?

- If multiple threads “update” a variable, you need synchronized methods/statements
 - In this case, there is no need for a volatile variable, because synchronized also ensures that the value written last is seen by all threads
- But if you have a variable written to by 1 thread and read by N threads, then you don't need to go synchronized and volatile will do the job
 - with less overhead to boot!

The interrupt() method

- The Thread class provides an interrupt() method
- Calling interrupt() causes an InterruptedException to be raised if/while the target thread is blocked
- As you see in compilation error messages, several blocking functions mandate a try block and a catch for the InterruptedException
- Example:

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException e) {  
    // Perhaps do something  
}
```



Killing/Pausing a Java thread

- Make sure you fully understand how to kill a Java thread, and how to pause a Java thread
- Questions about this?
- Should we look back at the lecture notes?

“Classic” problem

- On the exam you can expect one “classic” question for some real-life problem
- There are tons and tons of those, including very difficult ones
 - Which of course wouldn't be on the exam
- The deal here is to recognize that a problem is equivalent (or very close) to another problem we've looked at
 - producer-consumer, reader-writer, dining philosophers, barber shop, compute servers on the cloud, bank account
- So although the problem may be about animals drinking at a lake, patrons at a sushi bar, or cars at a toll booth, the idea is to think of what other problem it resembles



Any More Questions?