

# Introduction

## ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

## What to expect

- General question about concurrency
- Questions about “interesting” features of Java for multi-threading
- Some code with concurrency bugs to find
  - Java
  - locks/cond. var
  - semaphores
- “what does this code print” questions
- “write pseudo-code” questions
  - Simple standard problems, nothing too creative
- Questions related to the homework assignments

## Lock Implementation

- Software-only implementations are very challenging
- Disabling interrupts is not typically feasible for safety concerns
- So one uses **atomic** hardware instructions
- These instructions make it possible to implement **spin locks**
  - // T&S: load from memory, if 0 then store 1
  - DADDI R1, R0, #0
  - Lock: T&S R1, <lock>
  - BNZ R1, Lock
  - RET

## Monitors as Locks

- How can I implement a lock with monitors?

```
public class Lock {
    private boolean locked = false;
    public synchronized void lock() {
        while (locked) {
            this.wait();
        }
        locked = true;
    }
    public synchronized void unlock() {
        locked = false;
        this.notify();
    }
}
```

## Locks as Monitors

- Note that our Lock implementation uses condition variables
  - So it's not a spin lock at all
- In fact, it's very different from the lock that's embedded inside the monitor
  - the one used for the synchronized methods and blocks
- How about condition variables?

## Monitors as Cond. Vars

```
public class CondVar {
    public synchronized void wait() {
        this.wait();
    }
    public synchronized void signal() {
        this.notify();
    }
}
```

- In this case there is a pretty simple mapping between monitors and condition variables

## Locks and Cond Variables

- Remember that when doing a wait() on a cond. variable and holding a lock, one wants to release the lock
- So could modify our implementation:

```
public class CondVar {
    public synchronized void wait() { this.wait(); }

    public synchronized void wait(Lock lock) {
        lock.unlock();
        this.wait();
        lock.lock();
    }

    public synchronized void signal() { this.notify(); }
}
```

## Semaphores

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

## Solution with Semaphores

```
semaphore mutex=1, freeslots=N, takenslots=0;
int current=-1, buffer[N];
```

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

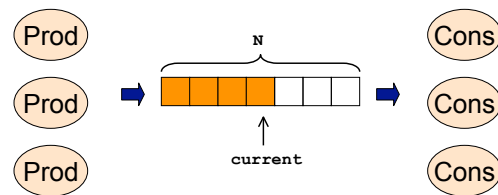
void consumer() {
    while (true) {
        P(takenslots);
        consume(buffer[current]);
        P(mutex);
        current--;
        V(freeslots);
        V(mutex);
    }
}
```

What's wrong with this code?

## Locks and Cond Vars

- So, we now have a Java implementation that looks just like Pthreads
- We “simulate” a lock and a cond var with Monitors
- Under the cover we know we have 2 condition variables and 2 locks!
- So we have a pretty high-overhead implementation
  - but our lock is not a spin lock
- So although you can do this, it's probably simpler to buy into Java monitors

## Bounded Prod/Cons



## Solution with Semaphores

```
semaphore mutex=1, freeslots=N, takenslots=0;
int current=-1, buffer[N];
```

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

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

What's wrong with this code?

## 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;
        unlock(mutex);
        if (current == N-1)
            full = true;
    }
}

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

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;
        signal(notempty);
        if (current == N-1)
            full = true;
        unlock(mutex);
    }
}

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

## Solution with Monitors

```
monitor ProdCons {
    cond notempty, notfull;
    int buffer[N];
    int current=-1;
    void produce(int element) {
        while (current > N-2) 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;
    }
}
```

## 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-2) { 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;
    }
}
```

What's wrong with this code?

## 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-2) { 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're back to the nested  
synchronized problem!

## Translation to Java

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

    public ProdCons() { . . . . . }

    public void synchronized produce(int element) {
        while (current > N-2) { 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

(We could have used this approach for our previous non-Java specific solutions as well)

## 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-2) {
                notfull.wait();
            }
        }
        synchronized(this) {
            current++;
            buffer[current] = element;
        }
        synchronized(notempty) {
            notempty.notify();
        }
    }
}
```

Create synchronized blocks is probably best, if messy

Using Semaphores could be cleaner actually

## 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
- Let's first look at it with semaphores

## Naive solution

semaphore mutex=1;

```
void reader() {
    P(mutex);
    read();
    V(mutex);
}

void writer() {
    P(mutex);
    write();
    V(mutex);
}
```

- Easy but not “selective”

## Reader-Preferred Solution

- One common solution is to have readers function as follows
  - Check if I am the first reader to get access to the shared data
  - If I am then I acquire the semaphore that allows me to access the shared data without conflicting with writers (i.e., enter the critical section)
  - If I am not the first, then somebody else has acquire the semaphore and is a reader, so I can move right along
  - If I am the last reader to leave the critical section, then I release the semaphore

## Reader-Preferred Solution

```
semaphore mutex = 1, rw = 1;  int nr = 0;

void read() {
    P(mutex);
    if (nr == 0) P(rw);
    nr++;
    V(mutex);
    // read shared data
    P(mutex);
    if (nr == 1) V(rw);
    nr --
    V(mutex);
}

void write() {
    P(rw);
    // write shared data
    V(rw);
}
```

## With Locks only

```
lock mutex1, mutex2;  int nr = 0;

void read() {
    lock(mutex1);
    if (nr == 0)
        lock(mutex2);
    nr++;
    unlock(mutex1);
    // read shared data
    lock(mutex1);
    if (nr == 1)
        unlock(mutex2);
    nr --
    unlock(mutex1);
}

void write() {
    lock(mutex2);
    // write shared data
    unlock(mutex2);
}
```

## Problem with Reader-Preferred

- The problem is that if there is a constant stream of readers, then the writer(s) could get constantly denied
- It's difficult to "fix" this
- The idea is to bound the number of readers to N and to use a token analogy
  - There are N tokens on a table
  - A writer needs them all to write
  - A reader needs only one to read
- This causes a problem if there are two writers
  - e.g., Each of them could get 1/2 the tokens and be stuck
- Therefore, writers must grab tokens in mutual exclusion of each others!

## Decent Reader/Writer Solution

```
semaphore_t sem = N;
semaphore_t wmutex = 1;
```

```
void reader(){
    while(true){
        P(sem);
        <read from the DB>
        V(sem);
    }
}
```

```
void writer() {
    while(true) {
        P(wmutex);
        for (i=0; i<N; i++)
            P(sem);
        V(wmutex);
        <write to the DB>
        for (i=0; i<N; i++)
            V(sem);
    }
}
```

## Java and Concurrency

- Thread interrupting, resuming
- The volatile keyword

## Thread Stopping

```
public class MyThread extends Thread {
    private volatile boolean stopped = false;

    public void stopThread() {
        stopped = true;
    }

    public void run() {
        while(true) {
            <do some work>
            if (stopped) {
                break;
            }
            <clean up>
            return;
        }
    }
}
```

```
...
MyThread thread
thread = new MyThread();
...
thread.start();
...
// stop the thread
thread.stopThread();
...
```

## 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 different values!!!

## volatile and synchronized

```
public class SomeValue {
    private float value = 0.0;

    synchronized void set(float v) {
        value = v;
    }

    synchronized float get() {
        return value;
    }
    ...
    SomeValue value = new SomeValue();
    ...
}
```

```
...
volatile float value = 0.0;
...
```

- A volatile variable removes the need for a small class that would be used just so that all threads see the value written last!
- You can replace the whole code on the left with the code on the right
- But not if the class on the left has an "update" method

## So When Do I Use volatile?

- Clearly, 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!
- We will see this in action on our next topic, stopping threads

## The interrupt() method

- The Thread class provides an interrupt() method
  - This is a very confusing name, so beware
- 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 a Thread

```
public class MyThread extends Thread {
    private volatile boolean stopped = false;
    public void stopThread() {
        stopped = true;
    }

    public void run() {
        ...
        if (stopped) { return; }
        ...
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            if (stopped)
                return;
        }
        ...
    }
}
```

```
...
MyThread thread
thread = new MyThread();
...
thread.start();
...
// stop the thread
thread.stopThread();
thread.interrupt();
...
```

- To cover all bases one typically both sets the "stopped" variable to true AND call interrupt();

## Any More Questions?