Lecture 22 — Debugging III

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 29, 2014

Debugging Parallel Programming

Bugs are our enemies.

Last time we talked about Heisenbugs and race conditions.

Race condition: the order of the computation steps matters and it's possible that if the steps execute in a certain order, the result is incorrect.

Debugging Parallel Programming

Let's imagine we have an instance of an object `Location` that has two co-ordinates, `x` and `y`.

```
location.setX(5);    location.setX(10);
location.setY(7);    location.setY(0);
```

Location Example

Even if each method is atomic, we cannot guarantee an order.

Consider this order:

1. Set x to 5
2. Set x to 10
3. Set y to 0
4. Set y to 7

Result: $(10, 7)$ - Inconsistent!

Location Example

This only occurs some of the time - Heisenbug!

Most of the time, the answer is right.

Debugging or printing statements might hide the error.

Location Example

Is our example realistic?

Imagine two users editing contact information for a vendor.

Multithreaded bug - are single threaded programs immune?

Parallel Bugs in Single Threaded Systems

Let's use an example from the labs: step counter.

Even a simple operation like `steps++;` is broken down into smaller operations.

You've just detected a step. Let's assume the current value of the variable is 4.

Parallel Bugs in Single Threaded Systems

Normally, `steps++;` works like this:

1. Read the current value of `steps` (read 4)
2. Add 1 to the value (now it's 5)
3. Write the changed value back to memory (write 5)

Parallel Bugs in Single Threaded Systems

Imagine an interrupt comes at the worst possible time.

The user presses the reset button. Should set `steps` = 0.

1. Read the current value of `steps` (read 4)
2. Add 1 to the value (now it's 5)
3. INTERRUPT (control goes to the interrupt handler)
4. Write 0 to the total number of steps (write 0)
5. END INTERRUPT (control returns to where it was)
6. Write the changed value back to memory (write 5)

Parallel Bugs in Single Threaded Systems

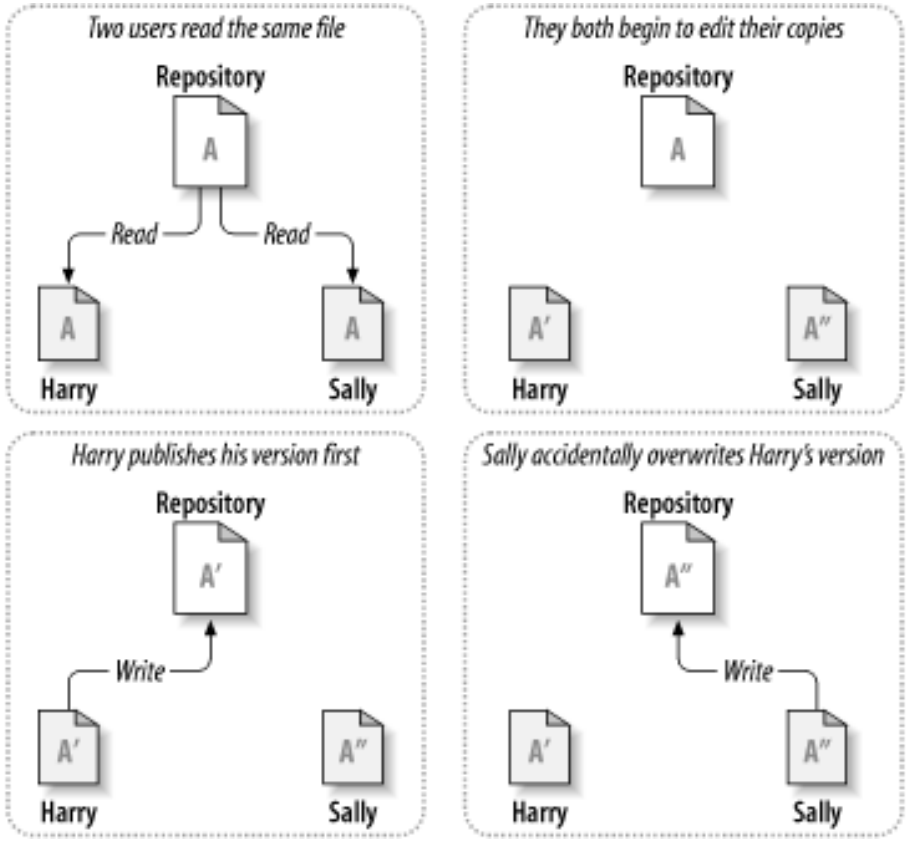
If the interrupt came after “write 5”, `steps` would be 0. OK.

If the interrupt came before “read 4”, `steps` would be 1. OK.

At the end of the last slide’s sequence, `steps` is 5. Wrong!

Wait a minute – changes overwritten? This looks familiar…

Parallel Bugs in Single Threaded Systems



http://svnbook.red-bean.com/en/1.6/svn.basic.version-control-basics.html

Solution: Lock-Modify-Unlock

Back to our earlier example:

```
lock(location);
location.setX(5);
location.setY(7);
unlock(location);
```

```
lock(location);
location.setX(10);
location.setY(0);
unlock(location);
```

Solution: Lock-Modify-Unlock

How the `lock` and `unlock` functions work is beyond the scope of this course; we are just interested in the concept.

Thread 1 locks the `location` object. Thread 2's attempt to lock the same object cannot succeed until Thread 1 unlocks it.

Thread 2 must wait.

Solution: Lock-Modify-Unlock

At the end of execution, the data could be $(5, 7)$ or $(10, 0)$.

It cannot be some mixup of the two like $(10, 7)$.

Locking & Unlocking in Java

"Knock knock." "Race Condition." "Who's there?"

The Java Keyword

In Java, there is a keyword that can be used to implement this lock-modify-unlock behaviour: `synchronized`.

A synchronized block in Java is synchronized "on" some object.

All synchronized blocks synchronized on the same object can only have one thread executing inside the synchronized area at a time.

All other threads attempting to enter the synchronized block must wait their turn.

Applying the Keyword

We can apply the keyword to the following blocks:

1. Instance methods
2. Static methods
3. Code blocks methods

Applying Synchronized

To make a method synchronized, just add the keyword `synchronized` to the function signature.

```
public synchronized int foo().
```

A synchronized instance method in Java is synchronized on the instance of the class.

Applying Synchronized

What If the method is also static?

```
public static synchronized int bar()
```

Synchronized on the class.

Synchronizing on a Block

It's not necessary to make the entire method synchronized.

The synchronized section should be as small as it can be.

If you use the statement `synchronized(s)` the following block will be synchronized on that object `s`:

```
synchronized(s) {
    s.count++;
}
```

Synchronizing on a Block

At `synchronized(s)`, the this thread will try to get the lock.

If someone has the lock, this thread will wait until the owner releases it and it is this thread's turn.

If nobody else has the lock, or it becomes this thread's turn, it succeeds in getting the lock.

Only once it has successfully acquired the lock can the code inside the braces execute.

When execution reaches the closing brace of the synchronized block, the lock is released.

Applying this to Location

Let's take that technique and apply it to the location example:

```
synchronized(location) {
    location.setX(5);
    location.setY(7);
}
```

```
synchronized(location) {
    location.setX(10);
    location.setY(0);
}
```

Problem occurred if a switch happened after the first thread set the X value and before it set the Y value.

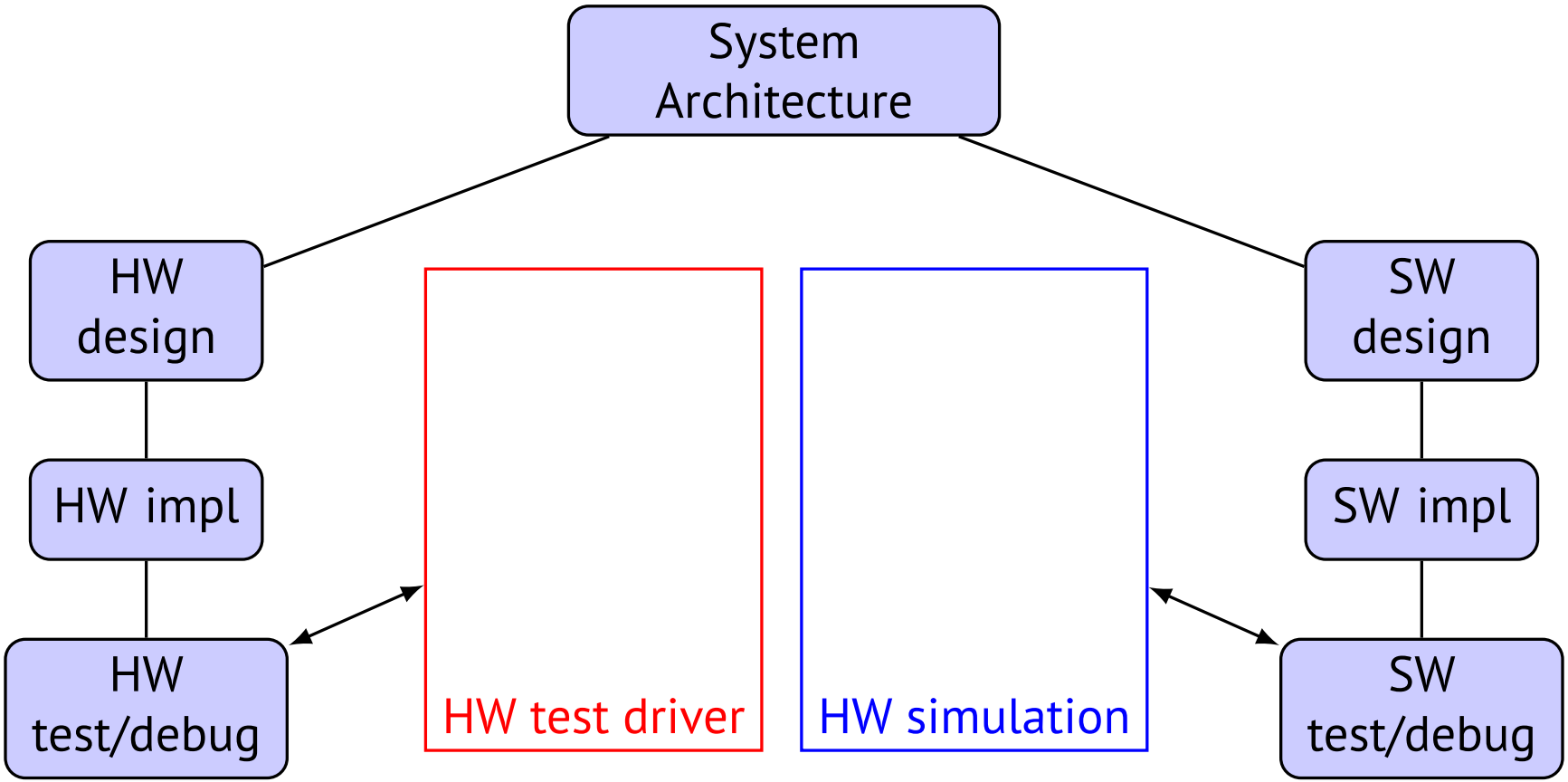
What happens in that scenario now that we have locks?

Debugging Embedded Systems

Embedded systems present a number of special debugging challenges.

There may be no console to print debug messages (or maybe no screen).

Embedded Systems diagram



Debugging Embedded Systems: Simulation

Simulate!

Reproduce the problem in the emulator (if possible).

Simulators often have consoles, and we can use a debugger.

Not always available.

Debugging Embedded Systems: Actuators

Use your actuators.

Make LEDs flash, or make noise.

Show an error code on a numeric display. (Microwave?)

Famous example: BIOS Power On Self Test (POST) in PCs.

Debugging Embedded Systems: Debugging Functions

Add debugging functions.

Allows you to put the system in a specific state.

Circumvent normal rules of the system.

You can use a physical button or software input to do it.

Real life example: cheat codes (`iddqd`).

Debugging Embedded Systems: Standard Interface

Use a standard interface.

JTAG is a standard interface for testing and I/O in many embedded systems.

It's possible (but not necessarily a good idea), to use the JTAG interface to flash your XBOX 360.

In ECE 327, you'll learn more about how JTAG works. In this course, you don't need to know any details about JTAG.