COSC441 Concurrent Programming

Memory is weird

Richard A. O'Keefe

July 18, 2017

Outline

- The Mutual Exclusion Problem
- Dekker's Algorithm
- What it assumes
- How the compiler wrecks it
- How modern hardware wrecks it
- Lazy initialisation
- Double-checked locking in Java
- The Memory Hierarchy

The Mutual Exclusion Problem

- There is some resource (variable, file, *etc.*)
- It can only be used by one thread at a time.
- Thread must do something to avoid interference.
- But what?

Java Example

```
public class ConcurrentCounter {
    private long count = 0;
    public void increment() {
        synchronized (this) { count++; }
    }
    public long value() {
        synchronized (this) { return count; }
    }
}
```

But how does that work?

- **synchronized** is how Java does it.
- That has to be implemented at a lower level.
- How do you do that?
- Without special hardware support?

Dekker's Algorithm

- First correct mutual exclusion algorithm.
- Didn't need special hardware support.
- Only supports two threads.

Dekker data

```
bool wants_to_enter[2] = {false, false};
int turn = 0; // or 1
```

wants_to_enter[*i*] is true if thread *i* wants to enter its critical section.
turn alternates to indicate which thread gets priority if both want to enter their critical region.

Dekker code

```
process p(int me, int other) {
    wants_to_enter[me] = true;
    while (wants_to_enter[other]) {
        if (turn $\neq$ me) {
            wants_to_enter[me] = false;
            while (turn $\neq$ me) /* busy wait */;
            wants_to_enter[me] = true;
        }
    }
    /* critical section here */
    turn = other;
    wants_to_enter[me] = false;
}
```

Questions

- How does it work? — Look it up.
- *Does* it work? — It did, but not now.
- What must we assume for it to work?

Assumptions

- Instructions may be arbitrarily interleaved, but instructions reading or writing the same location act *as if* serialised.
- Assignment is atomic.
  Fails two ways: assignment of small values may involve an instruction sequence, and assignment of large values may involve multiple stores, which leads to **tearing**.
- The change made by an assignment is immediately visible to all other threads.
- Variable references always go to memory.

How the compiler wrecks it

```
    move wants_to_enter[other] to R1;
    go to L3 if R1;
L1: move turn to R1;
    go to L3 if R1 $\neq$ me;
    move wants_to_enter[me] to R1;
L2: go to L2;
L3: /* critical section */
    move other to turn;
    move 0 to wants_to_enter[me];
```

Compiler optimisations

- Dead code is eliminated.
- Unchanged tests aren't retested.
- It takes less time to use data in registers than data in memory.
- Compilers go to a lot of trouble to move data into registers and keep it there as long as it is useful.
- Compilers are explicitly allowed to optimise code as if there was only one thread.
- This means that an assignment may update a local *copy* of a variable, not the shared memory.
- This has been true since the 1960s. . .

Defeating the compiler

- DECLARE X BINARY FIXED **ABNORMAL**;
- "The ABNORMAL attribute specifies that the value of the variable can change between statements or within a statement. An abnormal variable is fetched from or stored in storage each time it is needed or each time it is changed. All optimisation is inhibited for an abnormal variable." (From 1965.)
- C picked this up in 1989 and called it **volatile**.
- C++ got it from C and Java got it from C++.
- **volatile bool** wants_to_enter[2] = . . . ;
- **volatile int** turn = . . . ;

How hardware wrecks it

- Can we save Dekker's algorithm by declaring the key variables **volatile**?
- No: hardware is also allowed to "optimise" *as if* there is only one thread.
- A common feature is a "store buffer"; an assignment is queued up and written to memory when convenient. Reads look in the queue.
- An assignment may have been queued, the processor that did it will see the change, others may not.

Defeating the hardware

- Needs special hardware support!
- Called a *memory fence*.
- Waits for all writes to complete.
- Library implementations of locking include appropriate memory fences.
- Assume that even volatile assignments are *not* visible to other threads until a fence or locking operation is done.

Lazy initialisation

- An OO idiom.
- An instance of X has a reference to Y.
- It is expensive to make a Y.
- The Y might not be needed.
- Don't create it until you know you need it.

Java example

```
public class X {
    private Y myY = null;
    public void foo() {
        if (myY == null) myY = new myY();
        /* use myY */
    }
}
```

Locked version

The code above is not thread-safe.

```
synchronized (this) {
    if (myY == null) myY = new myY();
}
```

Locking is expensive so. . .

```
if (myY == null) {
    synchronized (this) {
        if (myY == null) myY = new myY();
    }
}
```

What could go wrong?

It doesn't work

- Adding **volatile helps**.
- But not enough. You need a memory fence too.
- The only way to get it is to use locking.

Memory Hierarchy

Read Ulrich Drepper's 2007 report "What Every Programmer Should Know About Memory" at https://people.freebsd.org/∼lstewart/articles/cpumemory.pdf