

Homework One

CS176 Fall 2008
Due Tuesday, 30 September

September 23, 2008

Amdahl's Law

You have a choice between buying one uniprocessor that executes five zillion instructions per second, or a ten-processor multiprocessor where each processor executes one zillion instructions per second. Using Amdahl's law, explain how you would decide which to buy for a particular application.

Peterson's Algorithm

Another way to generalize the two-thread Peterson lock is to arrange a number of two-thread Peterson locks in a binary tree. Suppose n is a power of two. Each thread is assigned a leaf lock which it shares with one other thread. In the tree-lock's acquire method, the thread acquires every two-thread Peterson lock from that thread's leaf to the root. The tree-lock's release method for the tree-lock unlocks each of the two-thread Peterson locks that thread has acquired, from the root back to its leaf.

At any time, a thread can be delayed for a finite duration. (In other words, threads can take naps, or even vacations, but they do not drop dead.) For each property, either sketch a proof that it holds, or describe a (possibly infinite) execution where it is violated.

1. mutual exclusion
2. freedom from deadlock
3. freedom from livelock

Is there an upper bound on the number of times the tree-lock can be acquired and released between the time a thread starts acquiring the tree-lock and when it succeeds?

Compare-and-Swap

The `AtomicInteger` class (in the `java.util.concurrent.atomic` package) is a container for an integer value. It provides many interesting methods, but the one that concerns us here is

boolean `compareAndSet(int expect, int update)`

This method compares the object's current value to `expect`. If the values are equal, then it atomically replaces the object's value with `update` and returns *true*. Otherwise, it leaves the object's value unchanged, and returns *false*. This class also provides

int `get()`

which returns the object's actual value.

Consider the following FIFO queue implementation, called `IQueue`. It stores objects in an unbounded array field `data`. It has two `AtomicInteger` fields: `tail` is the index of the next slot from which to remove an object, and `head` is the index of the next slot in which to place an object.

```
import java.util.concurrent.atomic.AtomicInteger;
class IQueue {
    AtomicInteger head = new AtomicInteger(0); // initialized to zero
    AtomicInteger tail = new AtomicInteger(0); // initialized to zero
    // effectively unbounded array
    Object data = new Object[Integer.MAX_VALUE]; // Kids, don't try this at home!

    public void enq(Object x) {
        int slot;
        do {
            slot = tail.get();
        } while (! tail.compareAndSet(slot, slot+1));
        data[slot] = x;
    }

    public Object deq() throws Empty {
        Object value;
        int slot;
        do {
            slot = head.get();
            value = data[slot];
            if (value == null)
                throw new Empty;
        } while (! head.compareAndSet(slot, slot+1));
        return value;
    }
}
```

Give an example showing that this implementation is *not* linearizable.

Bakery Algorithm

A *flickering* register has the property that if a `read()` call overlaps a `write()` call, then the value returned can be either the new value or the old. A `read()` that does not overlap a `write()` returns the last value written. (Note that a flickering register is not linearizable.)

An *unsafe* register has the property that if a `read()` call overlaps a `write()` call, then the value returned can be arbitrary. A `read()` that does not overlap a `write()` returns the last value written.

An *wraparound* register has the property that there is a value v such that adding 1 to v yields 0, not $v + 1$.

If we replace the Bakery algorithm's shared variables with either flickering, unsafe, or wrap-around registers, then does it still satisfy (1) mutual exclusion, (2) first-come-first-served ordering?

You should provide six answers (some may imply others). Justify each claim.

Graduate Credit (extra credit otherwise)

You are one of P recently-arrested prisoners. The warden, a deranged Computer Scientist, makes the following announcement:

You may meet together today and plan a strategy, but after today you will be in isolated cells and have no communication with one another.

I have set up a "switch room" which contains a light switch, which is either *on* or *off*. The switch is not connected to anything.

Every now and then, I will select one prisoner at random to enter the "switch room". This prisoner may throw the switch (from *on* to *off*, or vice-versa), or may leave the switch unchanged. Only one prisoner at a time will enter the room, and only prisoners will be allowed to enter the room.

Each prisoner will visit the switch room arbitrarily often. More precisely, for any N , eventually each of you will visit the switch room at least N times.

At any time, any of you may declare: "we have all visited the switch room at least once." If the claim is correct, I will set you free. If the claim is incorrect, I will feed all of you to the crocodiles. Choose wisely!

- Devise a winning strategy when you know that the initial state of the switch is *off*.
- Devise a winning strategy when you do not know whether the initial state of the switch is *on* or *off*.

Hint: not all prisoners need to do the same thing.