

# Homework One

CS176 Fall 2011  
Due Thursday, 6 October

September 27, 2011

## 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, characterize the applications that would benefit from the multiprocessor in terms of how much of it is parallelizable.

## 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 starvation

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)

```
class Bouncer {
    public static final int DOWN = 0;
    public static final int RIGHT = 1;
    public static final int STOP = 2;
    private boolean goRight = false;
    private ThreadLocal<Integer> myIndex;
    private volatile int last = -1;
    int visit () {
        int i = myIndex.get();
        last = i;
        if (goRight)
            return RIGHT;
        goRight = true;
        if (last == i)
            return STOP;
        else
            return DOWN;
    }
}
```

Figure 1: The Bouncer class implementation.

Suppose  $n$  threads call the `visit()` method of the Bouncer class shown in Fig. 1. Prove that—

- At most one thread gets the value STOP.
- At most  $n - 1$  threads get the value DOWN.

- At most  $n - 1$  threads get the value RIGHT.

Note that the last two proofs are *not* symmetric.