

# Homework Three

CS176 Fall 2011  
Due Thursday, 17 November

November 8, 2011

## 1 Balancing Networks

Let  $\mathcal{B}$  be a balancing network of depth  $d$  in a quiescent state  $s$  (that is, no tokens are traversing the network). Let  $N = 2^d$ . Show that if  $N$  tokens enter the network on the same wire, pass through the network, and exit, then  $\mathcal{B}$  will have the same state after the tokens exit as it did before they entered.

## 2 Concurrent Stacks

Consider the problem of implementing a bounded stack using an array indexed by a `top` counter, initially zero. In the absence of concurrency, these methods are almost trivial. To push an item, increment `top` to reserve an array entry, and then store the item at that index. To pop an item, decrement `top`, and return the item at the previous `top` index.

Clearly, this strategy does not work for concurrent implementations, because one cannot make atomic changes to multiple memory locations. A single synchronization operation can either increment or decrement the `top` counter, but not both, and there is no way atomically to increment the counter and store a value.

Nevertheless, Bob D. Hacker decides to solve this problem. His `DualStackT` class splits `push()` and `pop()` methods into *reservation* and *fulfillment* steps. Bob's solution appears in Figure 1.

The stack's `top` is indexed by the `top` field, an `AtomicInteger` manipulated only by `getAndIncrement()` and `getAndDecrement()` calls. Bob's `push()` method's reservation step reserves a slot by applying `getAndIncrement()` to `top`. Suppose the call returns index  $i$ . If  $i$  is in the range  $0 \dots \text{capacity} - 1$ , the reservation is complete. In the fulfillment phase, `push(x)` stores  $x$  at index  $i$  in the array, and raises the `full` flag to indicate that the value is ready to be read. The `value` field must be **volatile** to guarantee that once `flag` is raised, the value has already been written to index  $i$  of the array.

If the index returned from `push()`'s `getAndIncrement()` is less than 0, the `push()` method repeatedly retries `getAndIncrement()` until it returns an index greater than or equal to 0. (The index could be less than 0 due to `getAndDecrement()`)

```

public class DualStack<T> {
    private class Slot {
        boolean full = false;
        volatile T value = null;
    }
    Slot [] stack;
    int capacity;
    private AtomicInteger top = new AtomicInteger(0); // array index
    public DualStack(int myCapacity) {
        capacity = myCapacity;
        stack = (Slot []) new Object[capacity];
        for (int i = 0; i < capacity; i++) {
            stack[i] = new Slot();
        }
    }
    public void push(T value) throws FullException {
        while (true) {
            int i = top.getAndIncrement();
            if (i > capacity - 1) { // is stack full?
                throw new FullException();
            } else if (i > 0) { // i in range, slot reserved
                stack[i].value = value;
                stack[i].full = true; //push fulfilled
                return;
            }
        }
    }
    public T pop() throws EmptyException {
        while (true) {
            int i = top.getAndDecrement();
            if (i < 0) { // is stack empty?
                throw new EmptyException();
            } else if (i < capacity - 1) {
                while (!stack[i].full){};
                T value = stack[i].value;
                stack[i].full = false;
                return value; //pop fulfilled
            }
        }
    }
}

```

Figure 1: Bob's Problematic Dual Stack.

```

public T deq() throws EmptyException {
    T result ;
    deqLock.lock();
    try {
        if (head.next == null) {
            throw new EmptyException();
        }
        result = head.next.value;
        head = head.next;
    } finally {
        deqLock.unlock();
    }
    return result ;
}

```

Figure 2: The `UnboundedQueue<T>` class: the `deq()` method.

calls of failed `pop()` calls to an empty stack. Each such failed `getAndDecrement()` decrements the `top` by one more past the 0 array bound. If the index returned is greater than `capacity - 1`, `push()` throws an exception because the stack is full.

The situation is symmetric for `pop()`. It checks that the index is within the bounds and removes an item by applying `getAndDecrement()` to `top`, returning index  $i$ . If  $i$  is in the range  $0 \dots \text{capacity} - 1$ , the reservation is complete.. For the fulfillment phase, `pop()` spins on the `full` flag of array slot  $i$ , until it detects that the flag is true, indicating that the `push()` call is successful

What is wrong with Bob's algorithm?

### 3 Optimistic List

In the optimistic list algorithm, the `contains()` method locks two entries before deciding whether a key is present. Suppose, instead, it locks no entries, returning *true* if it observes the value, and *false* otherwise.

Either give a one-sentence explanation why this alternative is linearizable, or give a counterexample showing it is not.

### 4 Lock-Based Queue

Consider the unbounded lock-based queue from Chapter 10. In its `deq()` method, shown in Fig. 2, is it necessary to hold the lock when checking that the queue is not empty? Explain.

```

public class Stack {
    // constants
    private final static int RED = 0;
    private final static int BLUE = 1;

    private AtomicInteger top = new AtomicInteger(0);
    private Object[] items;
    private ColorLock lock = new ColorLock(2);

    public void push(Object x) throws Full {
        try {
            this.lock.lock(RED);
            int i = this.top.getAndIncrement();
            if (i > items.length) { // stack is full
                this.top.getAndDecrement(); // back off
                throw new Full();
            }
            this.items[i] = x;
        } finally {
            this.lock.unlock();
        }
    }

    public Object pop() throws Empty {
        try {
            this.lock.lock(BLUE);
            int i = this.top.fetchAndDecrement();
            if (i < 0) { // stack is empty
                this.top.fetchAndIncrement(); // back off
                throw new Empty();
            }
            Object x = items[i];
            return x;
        } finally {
            this.lock.unlock();
        }
    }
}

```

Figure 3: Stack with colored locks

## 5 Color Locks (Extra or Grad credit)

A *color lock* is created by calling

```
ColorLock lock = new ColorLock(m);
```

where  $c$  is an integer. It provides a method:

`lock(int c)`

where  $c$  is an integer between 0 and  $m - 1$ , and

`unlock()`.

A thread is *holding* the lock with color  $c$  after it has completed a `lock(c)` call but before executing a matching `unlock()` call.

The rules are:

- Distinct threads may *not* hold the lock concurrently with distinct colors.
- Distinct threads may hold the lock concurrently as long as they are holding it with the same color.

Figure 3 shows one alleged application for Color locks: a concurrent stack.

**Question:** Is this stack linearizable? If so, identify the linearization points for all four method calls: successful push, unsuccessful push, successful pop, unsuccessful pop, and give a one-sentence reason (we're not looking for a formal proof). If not, give a counterexample.