

Homework Two

CS176 Fall 2011
Due Thursday, 20 October

October 12, 2011

1 Mutual Exclusion and Registers

Does Peterson's two-thread mutual exclusion algorithm work if we replace shared atomic registers with regular registers?

2 Registers

Consider the following implementation of a **register** in a distributed, message-passing system. There are n processors P_0, \dots, P_{n-1} arranged in a ring, where P_i can send messages only to $P_{i+1 \bmod n}$. Messages are delivered in FIFO order along each link.

Each processor keeps a copy of the shared register.

- To read a register, the processor reads the copy in its local memory.
- To start a **write** call of value v to register x , each P_i sends the message " P_i : write v to x " to $P_{i+1 \bmod n}$.
- If P_i receives a message " P_j : write v to x ", for $i \neq j$, then it writes v to its local copy of x , and forwards the message to $P_{i+1 \bmod n}$.
- If P_i receives a message " P_i : write v to x ", then it writes v to its local copy of x , and discards the message. The **write** call is now complete.

Give a short justification or counterexample.

If only one processor ever calls **write**,

- Is this register implementation regular?
- Is it atomic?

If multiple processors call **write**,

- Is this register implementation atomic?

```

public class HWQueue<T> {
    AtomicReference<T>[] items;
    AtomicInteger tail ;
    static final int CAPACITY = 1024;

    public HWQueue() {
        items =(AtomicReference<T>[])Array.newInstance(AtomicReference.class,
            CAPACITY);
        for (int i = 0; i < items.length; i++) {
            items[i] = new AtomicReference<T>(null);
        }
        tail = new AtomicInteger(0);
    }
    public void enq(T x) {
        int i = tail .getAndIncrement();
        items[i] .set(x);
    }
    public T deq() {
        while (true) {
            int range = tail .get ();
            for (int i = 0; i < range; i++) {
                T value = items[i] .getAndSet(null);
                if (value != null) {
                    return value;
                }
            }
        }
    }
}

```

Figure 1: Herlihy/Wing queue.

3 More Linearizability

This exercise examines a queue implementation (Fig. 1) whose `enq()` method does not have a linearization point.

The queue stores its items in an `items` array, which for simplicity we will assume is unbounded. The `tail` field is an `AtomicInteger`, initially zero. The `enq()` method reserves a slot by incrementing `tail`, and then stores the item at that location. Note that these two steps are not atomic: there is an interval after `tail` has been incremented but before the item has been stored in the array.

The `deq()` method reads the value of `tail`, and then traverses the array in ascending order from slot zero to the `tail`. For each slot, it swaps `null` with the current contents, returning the first non-`null` item it finds. If all slots are `null`, the procedure is restarted.

Give an example execution showing that the linearization point for `enq()` cannot occur at Line 15.

Hint: give an execution where two `enq()` calls are not linearized in the order they execute Line 15.

Give another example execution showing that the linearization point for `enq()` cannot occur at Line 16.

Since these are the only two memory accesses in `enq()`, we must conclude that `enq()` has no single linearization point. Does this mean `enq()` is not linearizable?

Consensus

A *ternary* register holds values $\perp, 0, 1$, and provides `compareAndSet()` and `get()` methods with the usual meaning. Each such register is initially \perp . Give a protocol that uses one such register to solve n -thread consensus if the inputs of the threads are *binary*, that is, either 0 or 1.

Can you use multiple such registers (perhaps with atomic read/write registers) to solve n -thread consensus even if threads' inputs are in the range $0 \dots 2^K - 1$, for $K > 1$. (You may assume an input fits in an atomic register.) *Important:* remember that a consensus protocol must be wait-free.

We will accept a solution that uses n ternary registers. You get extra credit if you can do it with $O(K)$ ternary registers. Feel free to use all the atomic registers you want (they're cheap).

Graduate Credit (extra credit otherwise)

The `Stack<>` class provides two methods: `push(x)` pushes a value onto the top of the stack, and `pop()` removes and returns the most recently pushed value. Prove that the `Stack<>` class has consensus number *exactly* two.