Dan Alistarh
Giorgi Nadiradze

May 22nd, 2017
Deadline: May 22nd, 2017

Advanced Data Structures

Spring Semester 2017

Exercise Set 14

Exercise 1:
Recall Peterson's two-thread mutual exclusion algorithm from class. We will now prove some more of its properties.
Question 1: Does the algorithm still guarantee mutual exclusion if we swap the order of the variable check in the while loop? More precisely, for Alice, we will have

$$\mathsf{while}(\mathsf{turn} == \mathsf{his}\&\mathsf{he\_wants} == \mathsf{true}),$$

and the symmetric for Bob.
Question 2: Prove that the algorithm satisfies *deadlock-freedom*: for any sufficiently long suffix, there is some lock or unlock operation which succeeds. (Crucially, you will need to use the fact that every thread is scheduled to take steps eventually.)
Question 3: Consider the following *livelock-freedom* property: for any sufficiently long suffix, every lock or unlock operation succeeds. Does the algorithm guarantee this property?

Exercise 2:
Peterson's algorithm is designed to work with only two threads.
Question: Can you build on this algorithm to solve mutex for an arbitrary (known) number of threads n? Write down the pseudocode in detail.

Exercise 3:
The implementation of Treiber's lock-free stack algorithm we presented in class today implicitly assumed that nodes are *immutable*, in the sense that once a node is popped, its memory should never be reused for some other node.
Question 1: Can you build an example where if a node's memory can be reused we *break* the correctness of the stack? More precisely, you need to construct an execution in which some thread would return an incorrect value. (Hint: your execution should build a scenario where a CAS that should fail doesn't.)
Question 2: Can you fix the stack implementation so that this issue doesn't occur?

Exercise 4:
A *shared queue* object implements *enqueue* and *dequeue* operations, with the same semantics as their sequential counterparts.
Question: Build a non-blocking shared queue using read, write and compare-and-swap operations.

Exercise 5:
A *binary consensus* shared object has a single operation *propose* that takes a value v equal to 0 or 1 as an argument and returns 0 or 1. When a thread p_i invokes *propose*(v), we say that p_i proposes value v. When p_i is returned value v' from *propose*(v), we say that p_i decides value v' (v' does not have to be equal to v). A binary consensus object satisfies the following properties:

Agreement No two threads decide different values.

Validity The value decided is one of the values proposed.

Termination Every thread that does not crash will eventually decide.

A *fetch-and-increment register* is a shared object with the following sequential specification:

```
Shared: register R, initially 0
upon fetch-and-inc( )
  x = R
  R = R + 1
  return x
```

Your tasks are:

1. Implement a binary consensus object using atomic read/write and fetch-and-increment operations in a system of 2 threads;
2. Implement a binary consensus object using read/write and one or more shared *queue* objects (with the usual FIFO semantics) in a system of 2 processes. The queues can be initialized with whatever you want;
3. ($\star$) Implement a binary consensus object for 2 processes using registers and one or more *queue* objects that are initially *empty*.
4. Implement a binary consensus object using atomic read/write and compare-and-swap operations in a system of n threads;
5. ($\star$, optional) Prove that one *cannot* implement binary consensus for 2 threads in a system where only read and write operations are available, and threads may stop taking steps (crash) at any point during the execution.