Concurrent Algorithms 2018
Midterm Exam

November 26th, 2018

Time: 1h45

Instructions:

- This midterm is "closed book": no notes, electronics, or cheat sheets allowed.
- When solving a problem, do not assume any known result from the lectures, unless we explicitly state that you might use some known result.
- Keep in mind that only one operation on one shared object (e.g., a read or a write of a register) can be executed by a process in a single step. To avoid confusion (and common mistakes) write only a single atomic step in each line of an algorithm.
- Remember to write which variables represent shared objects (e.g., registers).
- Unless otherwise stated, we assume atomic multi-valued MRMW shared registers.
- Unless otherwise stated, we ask for *wait-free* algorithms.
- Unless otherwise stated, we assume a system of n asynchronous processes which might crash.
- For every algorithm you write, provide a short explanation of why the algorithm is correct.

- Make sure that your name and SCIPER number appear on **every** sheet of paper you hand in.
- You are **only** allowed to use additional pages handed to you upon request by the TAs.

Good luck!

Problem	Max Points	Score
1	2	
2	3	
3	3	
4	2	
Total	10	

Problem 1 (2 points)

Task. Write an algorithm that implements a MRSW atomic multi-valued register using (any number of) SRSW regular multi-valued registers.

Solution

First implement SRSW atomic multi-valued registers using SRSW regular multi-valued registers. Then implement MRSW atomic multi-valued registers using SRSW atomic multi-valued registers. The implementations can be found in the lecture slides (slides 17-21).

Problem 2 (3 points)

The register-swap operation is an atomic operation with the following sequential specification.

```
Variables:
Shared MWMR atomic registers $A$ and $B$.
procedure register-swap($A$, $B$)
    $tempA \leftarrow A$.read()
    $tempB \leftarrow B$.read()
    $A$.write($tempB$)
    $B$.write($tempA$)
```

Figure 1: Sequential specification of the register-swap atomic operation.

Consider an asynchronous shared-memory system with n processes, $n - 1$ of which may fail by crashing. Assume atomic MRMW registers, on which you can perform atomic read, write, and register-swap operations.

Task. Write an algorithm that implements wait-free consensus for n processes in this setting.

Solution

Processes share two arrays of registers $R[1, \ldots, n]$, and $Winner[1, \ldots, n]$, whose initial values are all $\perp$. They also share a register *Decider*, with initial value won. The algorithm proceeds as follows.

Each process i first writes its proposal value *val* to $R[i]$. It then uses a register-swap operation to swap the contents of the $Winner[i]$ register with the contents of the *Decider* register. The key idea is that the process that manages to get the won value into its $Winner[i]$ register will impose its decision value on the consensus object.

More precisely, after the register-swap($Winner[i], Decider$) operation, the process collects all the values in the *Winner* array. Let j be the unique index such that $Winner[j] =$ won in the collected array. Each process decides $R[j]$.

```
procedure propose$_i$($v$)
$R[i] \leftarrow v$
register-swap($Winner[i], Decider$)
$j \leftarrow$ unique index in $Winner$ with $Winner[j] =$ won
return $R[j]$
```

Figure 2: Consensus out of register-swap.

Problem 3 (3 points)

The sequential specification of a test-and-set object is the following.

```
Variables:
Binary Register $V$, initially 0.
procedure test-and-set()
    $temp \leftarrow V$.read()
    if $temp = 0$ then
        $V$.write(1)
    return $temp$
```

Figure 3: Sequential specification of test-and-set.

Consider an asynchronous shared-memory system with three processes, two of which may fail by crashing. Assume that atomic multi-reader multi-writer registers and test-and-set objects are available.

Task. Prove that one cannot solve wait-free consensus in this system.

Solution

The solution is a variant of the FLP argument seen in class. You can find the proof in the paper *Wait-free synchronization* by Maurice Herlihy, available on the course website. Alternatively, you can prove that test-and-set can be implemented with a queue, and then follow the proof of Theorem 6 in that paper to prove that a queue cannot implement consensus, therefore test-and-set cannot implement consensus.

Problem 4 (2 points)

The augmented double-ended queue object is the same as the queue object except that it supports `enqueue` and `dequeue` operations on both of its ends (HEAD or TAIL). In addition, processes can invoke a total of 3 `peek` operations. Each of the first 3 `peek` calls returns the value at the selected end of the queue. Every subsequent `peek` call returns $\perp$.

```
Variables:
elements := List(), the list of elements in the queue
peeks_invoked := 0, the number of invoked peeks
procedure enqueue(end,val)
    if end = HEAD
        elements.add_first(val)
    else
        elements.add_last(val)
procedure dequeue(end)
    if end = HEAD
        val ← elements.first()
        elements.remove_first()
    else
        val ← elements.last()
        elements.remove_last()
    return val
procedure peek(end)
    if peeks_invoked = 3
        return ⊥
    peeks_invoked ← peeks_invoked + 1
    if end = HEAD
        return elements.first()
    else
        return elements.last()
```

Figure 4: Sequential specification of the augmented double-ended queue.

The sequential specification of the object using a doubly linked list is given in Figure 4.

Task. Write an algorithm that implements a wait-free consensus object in a system of at most 4 processes using augmented double-ended queue objects that support at most 3 `peek` operations and registers.

Solution

The fact that the queue is double-ended is a red-herring: the same solution can apply for a regular queue. All the processes first enqueue their proposed values at either end of the queue, provided they use the same end. The winning value will be at the front of the queue and the processes can call peek to read it. 1 out of the 4 processes will read $\perp$, indicating that everyone else has read the winning value. That process can then safely call dequeue to read the winning value.

```
Variables:
deque := Deque(), a double-ended queue object
procedure propose(val)
    deque.enqueue(HEAD, val)
    winner ← deque.peek(TAIL)
    if winner != ⊥
        return winner
    else
        return deque.dequeue(TAIL)
```

Figure 5: Sample implementation of wait-free consensus for 4 processes.