Concurrent Algorithms
Mock Midterm Exam

December 20th, 2021

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	2	
3	3	
4	3	
5	2	
Total	12	

Problem 1 (2 points)

Tasks.

1. Write a wait-free algorithm that implements a safe MRSW binary register using (any number of) SRSW safe binary registers.

2. Write a wait-free algorithm that implements a regular MRSW binary register using (any number of) safe MRSW binary registers.

Problem 2 (2 points)

A *snapshot* object maintains an array of registers R of size n, has operations $scan()$ and $update_i()$ and the following sequential specification:

```
upon update_i(v) do
    R_i ← v
upon scan do
    return R
```

Figure 1: Sequential specification of the snapshot object.

The following algorithm (incorrectly) implements an atomic *snapshot* object using an array of shared registers R:

```
upon update_i(v) do
    ts ← ts + 1
    R_i ← (v, ts, scan())
upon scan do
    t_1 ← collect(), t_2 ← t_1
    while true do
        t_3 ← collect()
        if t_3 = t_2 then return ⟨ t_3[1].val, ..., t_3[N].val ⟩

        for k ← 1 to N do
            if t_3[k].ts ≥ t_1[k].ts + 1 then return t_3[k].snapshot

        t_2 ← t_3
upon collect do
    for j ← 1 to N do
        x_j ← R_j;
    return x
```

Figure 2: Incorrect implementation of the snapshot object.

Task. Give an execution of the algorithm which violates atomicity of the *snapshot* object.

Problem 3 (3 points)

Consider the linearizable and wait-free log object. The log object supports two operations: append and getLog. The sequential specification of the log object is shown below:

```
Given:
Sequential linked list L that is initially empty.

procedure append(obj)
    L.append(obj)

procedure getLog()
    result[] ← ⊥
    k ← length(L)
    i ← 1
    while i ≤ k do
        i ← i + 1
        result[i] ← element(L, i) // the element(L, i) function call returns the i-th element of list L
    return result
```

Figure 3: Sequential specification of the log object.

Furthermore, consider a linearizable and wait-free fetch-and-increment object where its sequential specification is shown below:

```
Given:
Register R that is initially 0.

procedure fetchAndIncrement()
    old ← R
    R ← old + 1
    return old
```

Figure 4: Sequential specification of the fetch-and-increment object.

Is it possible to implement the linearizable and wait-free log object by using any number of read-write registers and fetch-and-increment objects? Explain your answer.

Problem 4 (3 points)

An atomic shared counter maintains an integer x, initially 0, and has two operations `inc()` and `read()`. The sequential specification is as follows:

```
x integer, initially 0
upon read(x) do
    return x
upon inc(x) do
    x ← x + 1
```

Consider the following, *incorrect*, implementation of an obstruction-free consensus object from shared counters:

```
uses: C_0, C_1 – atomic shared counters initialized to 0
upon propose(v) do
    while true do
        (x_0, x_1) ← readCounters()
        if x_0 > x_1 then
            v ← 0
        else if x_1 > x_0 then
            v ← 1
        if |x_0 − x_1| ≥ 1 then
            return v
        C_v.inc()
upon readCounters() do
    while true do
        x_0 ← C_0.read()
        x_1 ← C_1.read()
        x'_0 ← C_0.read()
        if x_0 = x'_0 then
            return (x_0, x_1)
```

Give an execution of the above algorithm that shows that the algorithm is not a correct implementation of an obstruction-free consensus object, i.e. an execution in which some property (obstruction-freedom, validity, or agreement) of obstruction-free consensus is violated.

Problem 5 (2 points)

An atomic 0-**set-once** object is a shared object that has three states $\perp$, 0, and 1. $\perp$ is the initial state. It provides only one operation *set(v)* where $v \in \{0, 1\}$, such that:

- If the object is in state $\perp$, then *set(v)* changes the state of the object to v and returns v.
- If the object is in state s where $s \in \{0, 1\}$, then *set(v)* changes the state of the object to $s \wedge \neg v$ and returns the new state of the object (i.e., $s \wedge \neg v$).

Tasks.

1. Explain what it means for a shared object to have infinite consensus number.
2. Prove that the atomic 0-**set-once** object has infinite consensus number.