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 1 (2 points)

Your tasks:

1. Write an algorithm that implements an MRSW regular $M$-valued register using (any number of) MRSW regular binary registers.

2. In your algorithm, if you replace the base registers (MRSW regular binary registers) by MRSW atomic binary registers, does your algorithm yield an MRSW atomic $M$-valued register? Justify your answer (i.e., give a counterexample if your algorithm does not work, or prove correctness if it does).

Solution

1. Please refer to page 15 of lecture slides “Registers”.

2. If your algorithm, denoted by $A$, is the same as shown in page 15 of lecture slides “Registers”, then the answer here is no. A counterexample is described below. Assume that $M = 2$ and two processes $p_1$ and $p_2$. First $p_1$ starts Read(); $p_1$ then pauses after $p_1$ has read $Reg[0]$ and $Reg[1]$; $p_1$ is going to read $Reg[2]$ but pauses. Then $p_2$ starts Write(1), finishes Write(1) and starts Write(2); $p_2$ pauses after $p_1$ has written $Reg[2]$; $p_1$ is going to write $Reg[1]$ but pauses. Next $p_1$ resumes and read $Reg[2]$; thus $p_1$ finds $Reg[2] = 1$ and returns 2; $p_1$ continues with a new Read(), and finishes Read(). However, this time $p_1$ returns 1 as $p_1$ finds $Reg[1] = 1$. Now $p_2$ resumes and finishes Write(2). This execution breaks atomicity in that at $p_1$, there is a new-old inversion and it is impossible to linearize $p_1$ and $p_2$’s operations.
Problem 2  (2 points)

This problem is about computing with anonymous processes.

Recall that a weak counter shared object has a single operation \( wInc \) that takes no argument, and returns a value \( ts \). A weak counter satisfies the following property. Let \( w_1 \) and \( w_2 \) be two invocations of \( wInc \). If \( w_1 \) returns before \( w_2 \) starts, then \( w_2 \) returns a larger value than \( w_1 \).

Consider the following implementation of snapshot with anonymous processes (as given in the lecture). Recall that a snapshot shared object has two operations update and scan, and maintains an array of size \( m \). No component of this array is devoted to a process. The number \( n \) of anonymous processes can be arbitrary but known to the implementation.

Using: an array of atomic multi-valued MRMW shared registers \( \text{Reg}[1, 2, \ldots, m] \), initialized to 0;
Using: a weak counter \( C \), initialized to 0;

\[
\begin{align*}
\text{update}(i, x) & \{ \\
& ts := C.wInc(); \\
& v := \text{scan}(); \\
& \text{write} (x, v, ts) \text{ in } \text{Reg}[i]; \\
\} \\
\text{scan()} & \{ \\
& ts := C.wInc(); \\
& \text{while (true) do} \\
& \quad \text{read } \text{Reg}[1], \text{Reg}[2], \ldots, \text{Reg}[m]; \\
& \quad \text{if some register contained } (*, v, t) \text{ with } t \geq ts \\
& \quad \quad \text{then return } v; \\
& \quad \text{else if } n + 1 \text{ sets of reads gave the same results} \\
& \quad \quad \text{then return the first field of each value in such a set;} \\
& \} \\
\end{align*}
\]

The second return condition (underlined) checks if \( n + 1 \) sets of reads gave the same results. Your task is to explain what happens if the condition is replaced by the following:

- if \( n + 2 \) sets of reads gave the same results.

More specifically, please answer whether the implementation is still correct after the replacement. If not, please provide a concrete counter-example (for which it is sufficient for you to specify an \( n \) of your choice). If the implementation is still correct, please justify your answer for any arbitrary \( n \) and explain whether the behavior of executions of the implementation changes, and if so, please articulate the change(s).

Solution

The implementation is correct. It is easy to see that the implementation does not break linearizability. To see that the implementation eventually terminates, we emphasize here that according to the first return condition, one scan collects at most \( n - 1 \) different sets of reads (as the \( n \)th different set of reads must include an update that starts later than \( wInc \) in the current scan and thus have a higher timestamp.)

The behavior changes in that \( \text{scan}() \) now takes more time to terminate.
Problem 3  (4 points)

In this problem, we consider a system of \( n \) processes.

An \((m, n)\)-assignment object, where \( n \geq m > 1 \), has \( n \) fields (for instance, an \( n \)-element array) and two operations: assign() and read(). The assign() operation takes as arguments \( m \) values \( v_1, \ldots, v_m \) and \( m \) indices \( i_1, \ldots, i_m \) and atomically assigns value \( v_j \) to array element \( i_j \), for \( j = 1, \ldots, m \). Note: the entire sequence of \( m \) assignments is atomic. The read() operation takes an index argument \( i \) and returns the \( i \)th array element.

Your task is to prove that atomic \((n, \frac{n(n+1)}{2})\)-assignment objects, where \( n > 1 \), have consensus number at least \( n \).

**Hint:** Give a consensus protocol for \( n \) processes using atomic \((n, \frac{n(n+1)}{2})\)-assignment objects and atomic registers.

**Solution**

Please refer to Section 3.6 of the paper “Wait-free Synchronization” by Maurice Herlihy:

https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf