# Final exam 1: Solution

Exam rules:

1. Exam time: from 12.15 to 15.15.
2. The exam is closed book. No electronic devices are allowed.
3. You can use any notation for algorithms, but remember to write which variables represent shared objects (e.g., registers) and which are process-local.
4. Try to describe (briefly) the main idea behind every algorithm you give.
5. 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.
6. The exam grade will be computed in the following way: 1.0 (for handing in the exam) plus the total number of points obtained divided by 2.

Assumptions:

1. We assume an asynchronous, shared-memory system of $n$ processes, out of which $n-1$ might crash (i.e., wait-free).
2. Unless explicitly stated otherwise, we assume that every object is atomic (linearizable) and wait-free.

Good luck!

## Problem 1 (1 point)

1. Define the *consensus* object. **(0.5 points)**
2. What does it mean for an object $O$ to have consensus number $k$? **(0.5 points)**

**Solution.** A consensus object has one operation $propose(v)$, which returns a value. When the operation returns, we say that the process decides. The following properties are required:

1. No two processes decide differently.
2. Every decided value is a proposed value.
3. Each process decide after a finite number of steps.

An object $O$ has a consensus number $k$ consensus number, if $k$ is the maximum number of processes for which the object can solve a consensus problem.

## Problem 2 (2 points)

Implement a MWMR atomic, multivalued register from any number of SWMR regular, multivalued registers.

**Solution.** The required transformations are given in the slides on register transformations.

## Problem 3 (2 points)

A *stack* is a shared object that implements a LIFO (last in, first out) data structure and provides the following operations:

1. *push(v)* that puts element $v$ at the top of the stack,
2. $pop()$ that returns the element on the top of the stack and removes it from the stack. The $pop()$ operation will return the special value $\perp$ if called on an empty stack.

Your task is to:

1. Give an algorithm that implements wait-free consensus using (any number of) stacks and registers in a system of 2 processes. The stack can be initialized to an arbitrary state. **(1 point)**
2. Give an algorithm that implements wait-free consensus using (any number of) stacks and registers in a system of 2 processes. The stack is initially empty. **(1 point)**

**Solution.** The solution is (almost) identical to the one given in the exercises for queues.

## Problem 4 (2 points)

A *queue* is a shared object that implements a FIFO (first in, first out) data structure and provides the following operations:

1. *enqueue(v)* that puts element $v$ at the end of the queue,
2. *dequeue*() that returns the first element from the queue and removes it from the queue. This operation returns the special value $\perp$ if called on an empty queue.

Give an algorithm that implements a wait-free *shared queue* object using any number of compare-and-swap objects and registers in a system of $n$ processes.

**Solution.** Use the universal construction from lectures, implementing consensus objects using compare-and-swaps. There exist more complex solutions, however the universal construction is enough to get full mark.

## Problem 5 (1 point)

Consider a simple (distributed) *networking object* that has only one operation called *getsock*(). If a process $p_i$ invokes *getsock*() (with no parameters) $p_i$ is returned a unique *socket* object that it can later use (locally) to perform *networking operations* (i.e. send and receive packets over network). For simplicity we assume that:

1. Each socket is uniquely identified by an integer $1, 2, \ldots$
2. No process invokes *getsock* more than $M$ times in any execution, where $M$ is some known constant.

The *networking object* ensures the following (in every execution):

1. No two processes are returned the same socket by *getsock*().
2. The highest identifier of a socket returned by *getsock* (at any process) in a given execution is bounded by a function $f(k)$, where $k$ is the number of invocations of *getsock* in that execution, and $f$ is independent of the total number of processes $n$.

Write an algorithm that implements a wait-free *networking object* (i.e., its *getsock* operation) using only (MRMW atomic multi-valued wait-free) registers and test-and-set objects.

**Solution.** Implementing is networking object is essentially the same as giving a solution to the renaming problem presented in class. You were given a solution using only registers and test-and-set objects, which applies in this case as well. (There exist solutions using only registers, but they are more complex.)

## Problem 6 (1 point)

Prove that it is impossible to implement a wait-free consensus object using only queues and atomic registers in a system of 3 processes.

**Solution.** Refer to "Wait-Free Synchronization" paper (second reference on the course web page), Section 3.3.

## Problem 7 (1 point)

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
    $R_i \leftarrow$v;
upon $scan$ do
    return $R$;
```

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

```
upon $update_i(v)$ do
    $ts \leftarrow ts + 1$;
    $R_i \leftarrow (v, ts, scan())$;
upon $scan$ do
    $t_1 \leftarrow collect(), t_2 \leftarrow t_1$;
    while $true$ do
        $t_3 \leftarrow collect()$;
        if $t_3 = t_2$ then return $\langle t_3[1].val, \ldots, t_3[N].val \rangle$ ;
        ;
        for $k \leftarrow 1$ to $N$ do
            if $t_3[k].ts \geq t_1[k].ts + 1$ then return $t_3[k].snapshot$;
            ;
        $t_2 \leftarrow t_3$;
procedure $collect$
    for $j \leftarrow 1$ to $N$ do
        $x_j \leftarrow R_j$;
    return $x$;
```

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

**Solution.** Consider an execution given in the figure below. In the execution, the scan of $p_2$ records the snapshot that does not observe the concurrent update of $p_1$. Process $p_3$ performs a scan that starts after the update of $p_1$ is done, so it has to observe its effects. It performs one collect before $p_2$ writes the results of its scan into its position in the snapshot object and another one after. Because the timestamps of these two elements of the snapshot differ by one, it returns the scan of $p_2$. The scan does not include the update of $p_1$, so the atomicity is violated.

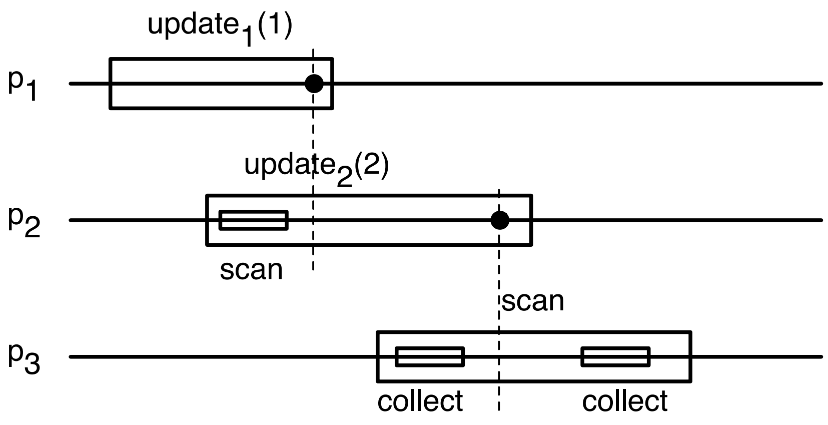

Figure 1: An execution violating atomicity