Concurrent Algorithms 2019
Final Exam Solutions

January 27th, 2020

Time: 12h15 - 15h15 (3 hours)

Instructions:

- This exam is "closed book": no notes, electronics, or cheat sheets are 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 variable represents which shared object (e.g., registers).
- Unless otherwise stated, we assume atomic multi-valued MRMW shared registers.
- Unless otherwise stated, we ask for *linearizable* and *wait-free* algorithms.
- Unless otherwise stated, we assume a system of n asynchronous processes which might crash.

- 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 (available upon request).

Good luck!

Problem	Max Points	Score
1	6	
2	2	
3	2	
4	6	
5	2	
6	2	
Total	20	

Problem 1 (6 points)

1. Any execution in which a read concurrent with a write returns a value other than the value being written or the previous value, or any execution that exhibits new-old inversion.

2. The transformations can be found in the "Registers" lecture (slides 19–24).

Problem 2 (2 points)

One possible solution is to read the registers in reverse order, i.e. from 3 to 1. This way the reader can read an inconsistent state from at most one register. If the reader finds two consecutive registers having the same value, this value must be a consistent value and can be returned. Otherwise, it follows that only one of these two registers could hold an inconsistent value, so the value in the other register is returned.

```
uses: $c[3 \times M]$ local array of single-bit registers
upon $read()$ do
    for $i = 1$ to $M$ do
        $c[2 \times M + i] \leftarrow b[2 \times M + i]$
    for $i = 1$ to $M$ do
        $c[M + i] \leftarrow b[M + i]$
    for $i = 1$ to $M$ do
        $c[i] \leftarrow b[i]$
    if $c[M + 1 \dots 2M] = c[2M + 1 \dots 3M]$ then
        return Base10Conversion $(c[2M + 1 \dots 3M])$
    else
        return Base10Conversion $(c[1 \dots M])$
```

Problem 3 (2 points)

1. The obstruction-free consensus object satisfies 3 properties:

   Obstruction-free consensus satisfies the following three properties:

   - **obstruction-free termination**: If a correct process proposes and eventually executes alone, then the process eventually decides
   - **agreement**: No two processes decide differently
   - **validity**: Any value decided must have been proposed

   Lock-free consensus and wait-free consensus differ on the termination property. The properties are seen below:

   lock-free termination: If a correct process proposes, then some process eventually decides

   wait-free termination: Every correct process eventually decides

   However, since the consensus object is a single-shot object, the lock-free consensus object is equivalent to the wait-free consensus object and they both satisfy the wait-free termination property.

2. Figure 1 presents an execution that violates agreement. The problem arises due to the $ts := ts + 1$ line of code, since this way different processes could get the same ts value. In the original algorithm presented in class, ts is updated by n in every iteration of the *while* loop, where n is the number of processes. Therefore, in the original algorithm it is impossible for two different processes to get the same ts value and hence the execution presented in Figure 1 cannot occur.

	propose$_1$(v)	propose$_2$(v’)	
	ts := 1		
		ts := 2	
	T[1] := 1		
		T[2] := 2	
	val = ⊥? ✓		
		val = ⊥? ✓	
	V[1] := (v, 1)		
	ts = 2(maxts)? ✗		
	ts := 2		
	T[1] := 2		
	v(val) = ⊥? ✗		
	V[1] := (v, 2)		
		V[2] := (v’, 2)	
	ts = 2(maxts)? ✓		
		ts = 2(maxts)? ✓	
	return (v)	return (v’)	

Figure 1: Execution that violates agreement.

Problem 4 (6 points)

1. See slide 12 from the “The Power of Registers” lecture.
2. See slide 21 from the “The Power of Registers” lecture.
3. No, the algorithm does *not* correctly implement an atomic lock-free snapshot object if we replace the base registers by regular ones. An execution where atomicity is violated when using regular registers is presented in Figure 2. Note that in Figure 2 atomicity is violated since the first scan returns $(0, v)$ while the second one returns $(0, v')$ (new-old inversion). Such an execution can occur when using regular registers since when the second scan reads register $Reg[2]$ that is concurrently being written, it could as well read the previous written value v.

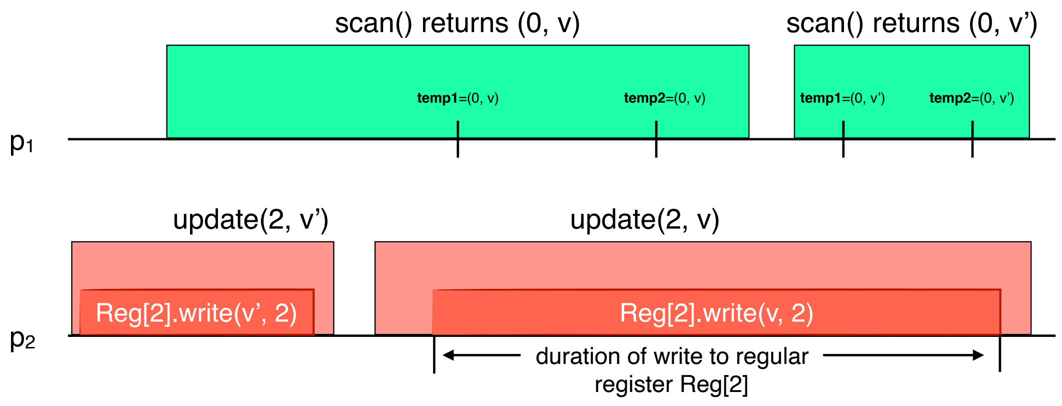

Figure 2: Execution that violates atomicity.

Problem 5 (2 points)

Yes, the implementation is correct.

Validity. All non-$\perp$ values in S_1 are proposed values. Therefore, if a process p writes $(true, v)$ or $(false, v)$ in S_2, then v must have been proposed by some process q (possibly by p itself). Since the output value of a process is taken either from S_1 or from S_2, validity is satisfied.

Lemma 1. *If S_2 contains two entries $(true, v_1)$ and $(true, v_2)$, then $v_1 = v_2$.*

Proof. Assume not. Since every process writes in S_1 and S_2 at most once, it must be that some process p_1 wrote $(true, v_1)$ and some other process p_2 wrote $(true, v_2)$. Thus, it must be that p_1 wrote v_1 in S_1, took a snapshot of S_1 and only saw v_1 in that snapshot. Similarly, it must be that p_2 wrote v_2 in S_1, took a snapshot of S_1 and only saw v_2 in that snapshot. This is impossible: since the snapshot object is atomic and the processes update S_1 before scanning, it must be that either p_1 saw p_2's value, or vice-versa. We have reached a contradiction. $\square$

Agreement. In order for a process p to commit v, p must write v to S_1, scan S_1 and see only entries equal to v; p must then write $(true, v)$ to S_2, scan S_2 and see only entries equal to $(true, v)$ and finally return $(commit, v)$.

Assume by contradiction that process p commits v and some process q commits or adopts $v' \neq v$. q's scan of S_2 cannot include the $(true, v)$ entry written by p, otherwise q would adopt v (remember that by Lemma 1, q cannot see any entry $(true, v')$ with $v' \neq v$ in S_2 if p has already written $(true, v)$ to S_2). Therefore, q's scan of S_2 must happen before p's write to S_2. Furthermore, q's scan of S_2 must include some entry $e = (\cdot, v')$ with $v' \neq v$ (written either by q or some other process). But then p's scan of S_2 (which is after p's write to S_2 and therefore after q's scan of S_2) will also include e, and thus p cannot commit v. We have reached a contradiction.

Commitment. Assume all proposed values are equal. Then no process can write $(false, \cdot)$ in S_2; S_2 contains only entries of the form $(true, \cdot)$. By Lemma 1, all such entries have equal values, so all processes that return must commit.

Termination. The code does not contain any waiting, loops, or *goto* statements, and the snapshot objects are wait-free, so every correct process will return in a finite number of steps.

Problem 6 (2 points)

Solution

We will prove the claim by mathematical induction on the number of processes n.

Base case: Let $n = 1$. When there is only one process it trivially decides its own value.

Inductive step: Assume there is an implementation C_n of consensus using 0-*set-once* objects and registers in a system of n processes. We need to prove that there is an implementation C_{n+1} of consensus using 0-*set-once* objects and registers in a system of $n+1$ processes. The implementation of C_{n+1} uses a 0-*set-once* object and a C_n consensus object for the first n processes and just a 0-*set-once* object for process p_{n+1}:

Operation $propose(v)$ for processes $p_1, \ldots, p_n$:

uses: C_n – shared consensus object for n processes.
uses: $R[0], R[1]$ – shared *registers*
uses: S – shared 0-*set-once* object initialized to $\perp$.

```
upon $propose(v)$ do
    $val \leftarrow C_n.propose(v)$
    $R[0] \leftarrow val$
    $t \leftarrow S.set(0)$
    if $t = 0$ then
        return $R[0]$
    else
        return $R[1]$
```

Operation $propose(v)$ for process p_{n+1}:

```
upon $propose(v)$ do
    $R[1] \leftarrow v$
    $t \leftarrow S.set(1)$
    if $t = 1$ then
        return $R[1]$
    else
        return $R[0]$
```