Concurrent Algorithms 2015
Midterm Exam
Solutions

December 1st, 2015

Time: 1h45

Instructions:

- This midterm is "closed book": no notes, electronics, or cheat sheets allowed.
- When solving a problem, do not assume any known results 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

A *binary consensus* shared object has a single operation *propose* that takes a value v equal to 0 or 1 as an argument, and returns 0 or 1. When a process p_i invokes $propose(v)$, we say that p_i proposes value v. When p_i has returned value v' from $propose(v)$, we say that p_i *decides* value v' (notice that v' does not have to be equal to v). A binary consensus object satisfies the following properties (for *binary* values):

Agreement No two processes decide different values.

Validity The value decided is one of the values proposed.

Your task is to show that with sufficiently many *binary consensus* shared objects and *m-valued* atomic registers one can implement an *m-valued consensus* shared object. An *m-valued consensus* shared object is similar to the *binary consensus* object, except that it allows processes to *propose* (and consequently *decide*) among m values. Explain why your algorithm satisfies the **Agreement** and **Validity** properties.

Solution

We agree on the value one bit at a time. The processes share an n-element array of atomic registers, and an array of K consensus objects, where K is the number of bits necessary to encode the largest possible proposed value. Each process i announces its input in the i-th index of a shared array of *m-valued* atomic registers. At any time each process has a *preference*, the value it is trying to convince the others to decide.

Initially, each process's preference is its input. At round i, each process uses the i-th bit of its preference as input to the i-th binary consensus object. If it wins, it continues to the $i+1$-th binary consensus object with the same preference. If it loses, it scans the announcement array for an input that agrees with all the binary values decided in prior consensus rounds, and uses that value for its preference in the next round.

Problem 2

Consider the following **incorrect** algorithm invoked by process i:

uses: $label[1,\ldots,N]$ — shared array of registers, $flag[1,\ldots,N]$ — shared array of boolean registers
initially: $label[1,\ldots,N] \leftarrow 0$, $flag[1,\ldots,N] \leftarrow$ **false**

```
1 upon lock_i() do
2 |  label[i] ← max(label[1],...,label[N]) + 1
3 |  flag[i] ← true
4 |  while (∃k ≠ i)(flag[k] and ((label[k],k) ≪ (label[i],i))) do
5 |  |  ;
6 upon unlock() do
7 |  flag[i] ← false
```

Where $(label[k],k) \ll (label[i],i))$ is true in the following two cases:

- $label[k] < label[i]$, **or**
- $label[k] = label[i]$ **and** $k < i$

This algorithm *incorrectly* implements the bakery *lock* algorithm. Its purpose is to ensure the *First-Come-First-Served* (*FCFS*) and *mutual exclusion* properties. This means that when a process gets a label, it waits until no process with an earlier label is trying to enter the critical section protected by the lock before entering (*FCFS*), and that only one process can be in the critical section protected by the lock at any point in time (*mutual exclusion*). Your tasks are the following:

a) Show that the algorithm presented is incorrect and correct the error(s).

b) We saw *safe*, *regular* and *atomic* registers in the course lectures. What type of register is the minimum necessary for the *correct* bakery algorithm to satisfy both the *FCFS* and *mutual exclusion* properties? Explain why.

c) We define a new *wraparound* register which is atomic but also has the following property: there is a value v such that adding 1 to v yields 0 and not $v+1$. If we use *wraparound* registers for the *correct* Bakery algorithm's variables, does the algorithm still satisfy the two properties (*FCFS* and *mutual exclusion*)? Explain why.

Solution

Step	Process 1	Process 2
1	$label[1].Read() \rightarrow 0$	
2	$label[2].Read() \rightarrow 0$	
3	$label[1] \leftarrow 1$	
4		$label[1].Read() \rightarrow 1$
5		$label[2].Read() \rightarrow 0$
6		$label[2] \leftarrow 2$
7		$flag[2] \leftarrow$ **true**
8		**while** *is* **false**
9	$flag[1] \leftarrow$ **true**	
10	**while** *is* **false**	
11		

Table 1: Possible execution for the **incorrect** Bakery algorithm.

a) The algorithm is incorrect. Suppose the execution presented in Table 1. At step 9, process 2 has entered the critical section before process 1, breaking the *FCFS* property. At step 11, both processes are in the critical section. Thus, the algorithm does not preserve *mutual exclusion*. In order to fix the algorithm, we simply swap lines 2 and 3 of the presented algorithm. The *correct* algorithm is the following:

uses: $label[1,\ldots,N]$ — shared array of registers, $flag[1,\ldots,N]$ — shared array of boolean registers
initially: $label[1,\ldots,N] \leftarrow 0$, $flag[1,\ldots,N] \leftarrow$ **false**

```
1 upon lock() do
2 |   flag[i] ← true
3 |   label[i] ← max(label[1], ..., label[N]) + 1
4 |   while (∃k ≠ i)(flag[k] and (label[k], k) ≪ (label[i], i))) do
5 |   |   ;
6 upon unlock() do
7 |   flag[i] ← false
```

b) The Bakery lock algorithm does not preserve mutual exclusion using *safe* registers. Suppose we have two processes, 1 and 2, both with labels equal to 1. Both processes proceed in parallel and process 2 enters its waiting loop first, reading $label[1]$ at the same time process 1 writes to it. Process 2 can read *any* arbitrary value (including a value larger than 1) and both processes could end up entering the critical section at the same time.

The algorithm ensures both the *FCFS* and *mutual exclusion* properties when using *regular* registers. Suppose processes 1 and 2 get labels j and j' with $j < j'$. Process 2 is faster and reaches the while loop and reads $label[1]$ in the while loop at the same time process 1 writes to it. Process 2 can only read the value being written or the previous one, both of which are less or equal to j', so it will not enter the critical section, maintaining the *FCFS* property. For the *mutual exclusion* property, suppose processes 1 and 2 are concurrently in the critical section. Let $labeling_1$ and $labeling_2$ be the last respective sequences of acquiring new labels prior to entering the critical section. Suppose that $(label[1], 1) \ll (label[2], 2)$. When process 2 successfully completed the test in its waiting section, it must have read that $flag[1]$ was **false** or that $(label[2], 2) \ll (label[1], 1)$. If process 2 read that $flag[1]$ was false, then that read preceded or overlapped process 1?s write to $flag[1]$, which preceded process 1?s read of $flag[2]$ and write to $flag[1]$, implying that $label[1] > label[2]$, a contradiction. So process 2 observed that $(label[2], 2) \ll (label[1], 1)$. Since process 1 never wrote such a value, process 2 must have read $label[1]$ at the time process 1 was updating it. But process 2 must have seen either the value being written, or the previous value, both of which are less than or equal to $label[2]$, a contradiction. Consequently, *regular registers* are the minimum required for the *correct* Bakery algorithm to ensure both properties

c) If we use *wraparound* registers for the *correct* Bakery algorithm, the algorithm does not satisfy *FCFS*, since a process can get a smaller label than a previous one, entering the loop first. It also does not satisfy *mutual exclusion*, since getting a label of 0 can cause a process to enter the critical section while the process with label v is already in it.

Problem 3

In the exercise sessions we showed that it is impossible to implement a *consensus* object using (any number of) base *Compare-and-Swap* (*C&S*) objects of which t can be non-responsive. Assume the **responsive** failure model in which if a base object fails (returns $\otimes$), the object will keep returning $\otimes$ for every invocation later. Is it possible to implement a *consensus* object using m base *C&S* objects of which t can fail in a responsive manner? Explain why.

We remind you that a *C&S* object provides one atomic operation *CAS* with the following sequential specification:

uses: *current_value* — local variable
initially: $current\_value \leftarrow \perp$

```
1 upon CAS(old_value, new_value) do
2     ret ← current_value
3     if old_value = current_value then
4         current_value ← new_value
5     return ret
```

Hint: Choose an m such that $m > t$.

Solution

To solve the problem we need (at least) $m = t + 1$ base *C&S* objects, so that there is at least one object that is non-faulty. Then, the idea behind the solution is to use all m *C&S* objects and according to the return value of each object, either adopt a new value, or continue with the previous one.

uses: $cas\_array[1, \ldots, m]$ — shared array of m *C&S* objects
initially: $cas\_array[1, \ldots, m] \leftarrow \perp$

```
1 upon propose(value) do
2     ret ← value
3     i ← 1
4     while i ≤ m do
5         r ← cas_array[i].CAS(⊥, ret)
6         if r ≠ ⊗ and r ≠ ⊥ then
7             ret ← r
8         i ← i + 1
9     return ret
```

The intuition of this algorithm is very simple: the value that will be put on the (for sure) non-faulty *C&S* will be the one decided, cause every process will adopt this value.

Problem 4

As seen in the course lectures, a *snapshot* shared object provides two operations, *update()* and *scan()* and maintains an array x of size n. The sequential specification of a snapshot object is the following:

uses: $x[1, \ldots, M]$ — array of registers,
initially: $x[1, \ldots, M] \leftarrow \perp$

```
1  upon update(i, v) do
2      x[i] ← v
3      return ok

4  upon scan() do
5      return x[1, ..., M]
```

We present below a wait-free implementation of the snapshot object, as presented in the course lectures:

uses: $x[1, \ldots, M]$ — array of registers,
initially: $x[1, \ldots, M] \leftarrow \perp$

```
1  upon update(i, v) do
2      ts ← ts + 1
3      x[i].write(v, ts, self.scan())
4      return ok

5  upon collect() do
6      for i ← 1 to M do
7          ret[i] ← x[i].read()
8      return ret[1, ..., M]

9  upon scan() do
10     t1 ← self.collect()
11     t2 ← t1
12     while true do
13         t3 ← self.collect()
14         if t3 = t2 then
15             return t3
16         for j ← 1 to M do
17             if t3[j, 2] ≥ t1[j, 2] + 2 then
18                 return (t3[j,3])
19         t2 ← t3
```

The condition on line 17 checks for a difference of at least 2 in the timestamps. Explain what happens if the condition is replaced by the following:

a) **if** $t3[j, 2] \geq t1[j, 2] + 1$ **then**

b) **if** $t3[j, 2] \geq t1[j, 2] + 3$ **then**

Solution

Step	Process 1	Process 2
1	$scan() \rightarrow (0,0,0)$	
2		$scan() \rightarrow (0,0,0)$
3	$x[1].write(5,1,(0,0,0))$	
4	$return(ok)$	
5		
6	$collect() \rightarrow (5,0,0)$	
7		$x[2].write(5,1,(0,0,0))$
8	$collect() \rightarrow (5,5,0)$	
9	$return(t3[2,3]) \rightarrow (0,0,0)$	

Table 2: Possible steps of an execution of the *snapshot* algorithm with [**if** $t3[j,2] \geq t1[j,2]+1$ **then**] as the condition in line 17.

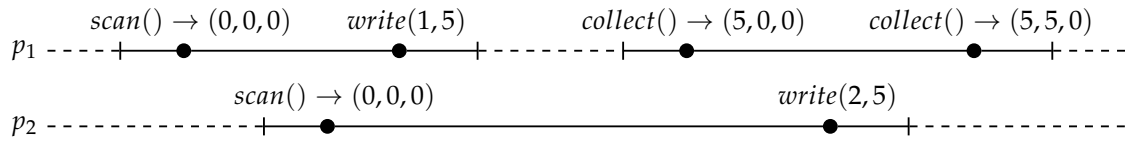

Figure 1: Possible execution of the *snapshot* algorithm with [**if** $t3[j,2] \geq t1[j,2]+1$ **then**] as the condition in line 17.

a) The implementation is incorrect. Consider the execution in Table 2. The $update$ operation of process 1 overlaps with the $update$ operation of process 2, and so the *scan* of process 2 does not capture the *write* of process 1. Then, process 1 invokes a *scan* operation, which does two *collect* operations. Between the two, process 2 does its *write* after the first *collect* of process 1. After the second *collect* of process 1 is not identical to the first, the algorithm reaches line 16 of the code. The arrays are not the same for index 2 and the timestamp difference is 1, so the process returns the *scan* of process 2. This result does not contain the $update$ that process 1 did in the previous operation, which is incorrect, since the two operations do not overlap. The execution is also shown in Figure 1.

b) The algorithm continues to be correct. Intuitively, it gives the process one more "opportunity" to try to get two consecutive *collect* results that match.