STiDC'08: Example Final Exam Questions

January 12, 2009

Problem 1

We can define an `UpDownCounter` object as follows. The state of the object is an integer c. The object implements 3 operations: *read* returns the state of the object c without changing c, *inc* increases c by 1 and returns *ok*, and *dec* decreases c by 1 and returns *ok*.

1. Here is a proposed (incorrect) implementation of an `UpDownCounter` for n processes, using n atomic registers (code for process p_i):

   uses: $A[1, \ldots, n]$ – atomic registers
   initially: $A[1, \ldots, n] = 0$

   upon $read_i()$ **do**
   $v \leftarrow 0$
   for $k \leftarrow 1$ **to** n **do**
   $v \leftarrow v + A[k].read_i()$
   return v

   upon $inc_i()$ **do**
   $v \leftarrow A[i].read_i()$
   $A[i].write_i(v + 1)$

   upon $dec_i()$ **do**
   $v \leftarrow A[i].read_i()$
   $A[i].write_i(v - 1)$

   Show that this algorithm does not implement an atomic, wait-free `UpDownCounter` by giving an execution of the algorithm in which atomicity (linearizability) is violated.

2. Give an algorithm that implements an `UpDownCounter` object using only atomic registers.

3. For how many processes one can implement a consensus object using any number of (atomic, wait-free) `UpDownCounter` objects and atomic registers?

Solution. The algorithm is not a linearizable (atomic) implementation of an `UpDownCounter`. To prove it, consider the following execution of the algorithm:

Step	Process p_1	Process p_2	Process p_3
1.	invokes $read_1()$		
2.	reads $A[1] = 0$		
3.	reads $A[2] = 0$		
4.		invokes $inc_2()$	
5.		writes $A[2] \leftarrow 1$	
6.		returns *ok*	
7.			invokes $dec_3()$
8.			writes $A[3] \leftarrow -1$
9.			returns *ok*
10.	reads $A[3] = -1$		
11.	returns -1		

The execution is not linearizable because the operations $inc_2()$ and $dec_3()$ are not concurrent, and so the operation $read_1()$ should have returned either 0 or 1. However, $read_1()$ returns -1.

The `UpDownCounter` object can be easily implemented from an atomic snapshot object, which can be implemented from registers (see the lecture slides). The algorithm would be the following:

uses: S – atomic snapshot (other variables are local)
initially: $c_i = 0$ at every process p_i, and the value of each element of S is 0

upon $read_i()$ **do**
  $A \leftarrow S.scan_i()$
  $v \leftarrow 0$
  for $k \leftarrow 1$ **to** n **do**
    $v \leftarrow v + A[k]$
  return v

upon $inc_i()$ **do**
  $c_i \leftarrow c_i + 1$
  $S.update_i(c_i)$

upon $dec_i()$ **do**
  $c_i \leftarrow c_i - 1$
  $S.update_i(c_i)$

As atomic snapshot can be implemented from registers, also an `UpDownCounter` can be implemented from registers. Every object that can be implemented from registers can solve consensus among only one process (in an asynchronous system, in which one process can crash; see the lecture notes for the relevant proof).

Problem 2

An SB object is a shared object that has three states, $\perp$, 0 and 1, and one operation, called $set(b)$, where $b \in \{0,1\}$. If the object is in state $\perp$, then $set(b)$ operation changes the state to b and returns b. If the object is in state s (where $s \in \{0,1\}$), the $set(b)$ operation changes the state to $s \wedge \neg b$ and returns the new state of the object (i.e., $s \wedge \neg b$).

- Prove the following lemma: If there is a wait-free consensus algorithm for n processes that use k registers and j SB objects, then there is a wait-free consensus algorithm for $n+1$ processes that uses $k+2$ registers and $j+1$ SB objects.
- For how many processes one can implement a wait-free consensus object using (any number of) SB objects and atomic registers?

Solution. To prove the lemma, we show an algorithm that implements $(n+1)$-consensus (i.e., consensus for $n+1$ processes) using n-consensus (e.g., implemented using SB objects and registers), two registers and one SB object. The algorithm is the following:
For processes $p_1, \ldots, p_n$:

upon $propose(v_i)$ **do**
  $v \leftarrow nCons.propose(v_i)$
  $R[0] \leftarrow v$
  $w \leftarrow SB.set(0)$
  if $w = 0$ **then return** v
  else return $R[1]$

For process p_{n+1}:

upon $propose(v_{n+1})$ **do**
  $R[1] \leftarrow v_{n+1}$
  $w \leftarrow SB.set(1)$
  if $w = 1$ **then return** v_{n+1}
  else return $R[0]$

Clearly, the above lemma almost immediately proves that an SB object can implement consensus for any number of processes (hint: implement 1-consensus, then use induction and the above lemma).