SLR206: Solutions for Quiz 5

1 Uninitialized queues

Recall that if a queue initially stores $\{winner, loser\}$, then two processes can solve consensus by performing a dequeue operation and deciding on your own value if you are the winner, or the value of the other process otherwise.

Suppose that we can only use *empty* queues. The trick is to use *two* queues, one for each process. Each process p_i first initializes its queue $Q[j]$, then registers its input in $T[i]$, and then for $j = 0, 1$ (in this order) runs the consensus algorithm $Cons_j$ using the initialized queue $Q[j]$ and proposing the value decided in the first consensus to the second one. If for some $j = 0, 1$, $T[j] = \perp$ (the input of p_j is not yet registered), p_i simply skips the corresponding consensus (Algorithm 1).

The proof of correcteness is left as an exercise.

Hint: To prove that Algorithm 1 indeed solves consensus, assume that p_i $(i = 01,)$ was the first process to write in $T[i]$. Show first that if both processes return, then they both go through consensus $Cons_i$ and, thus, they must retun the same value (returned by $Cons_i$).

Note that the approach can be used for any set of uninitialized base objects and any algorithm that solves consensus among any number of processes assuming a specific initialization of these base objects.

Algorithm 1 2-process consensus using empty queues

Shared variables:
  registers $T[0, 1] = \{\perp\}$
  queues $Q[0, 1] = \{\}$

propose(v_i) **performed by** p_i $(i = 0, 1)$**:**
  $Q[i].enq(winner)$;
  $Q[i].enq(loser)$;
  $T[i].write(v_i)$;
  $v = v_i$;
  for $j = 0..1$ **do**
    if $T[j] \neq \perp$ **then**
      $v = Cons_j(v)$ (using queue $Q[j]$);
  $return(v)$;

2 Lock-free universal construction

If we only lock-freedom is required, processes may avoid sharing their operations with each other. Instead, whenever a process has a new operation *op* to be performed, it may go to the next (not yet accessed) consensus objects in the series $C[1], C[2], \ldots$, let it be $C[k]$, and propose (op, i) to it by invoking $C[k].propose(op, i)$. The value (op', j) returned by $C[k]$ is appended to the local list *decided*. If $(op', j) = (op, i)$, the process returns the response of *op* computed, based on *decided*, using the sequential specification of the implemented object. Please check that the algorithm is correct.

3 Consensus numbers of TAS

By contradiciton, suppose that an algorithm A solves *binary* 3-process consensus (for processes p_0, p_1, p_2) using registers and TAS objects.

Recall that any input configuration C_0 in which some process p proposes 0 and another process q proposes 1 is *bivalent*: p running solo from C_0 must decide 0 and q running solo from C_0 must decide 1.

We show that C_0 must have a critical descendant: a configuration C reachable from C_0 by a finite execution such that:

- C is bivalent;
- for each p_i ($i = 0, 1, 2$), $C.p_i$ (the configuration obtained after p_i takes one more step of A after C) is *monovalent* (0-valent or 1-valent).

Indeed, suppose, by contradiction, that C_0 has no critical descendants. Thus, for every bivalent descendant of C_0 (including C_0 itself) has a bivalent descendant.

Now we construct an infinite execution that only goes through bivalent configurations as follows. Let C_1 be the bivalent one-step extension of C_0 (it must exist by our assumption), C_2 - the bivalent one-step extension of C_1, etc. We denote the resulting infinite execution by E. Recall that no process can decide in a bivalent configuration - otherwise, the agreement property of consensus is violated in some extension of this configuration. Thus, no process can decide in E—a violation of the termination property of consensus.

Thus, A has a critical configuration C. Without loss of generality let $C.p_0$ (the extension of C with one step of p_0) be 0-valent, and $C.p_1$ be 1-valent.

We observe that the steps of p_0 and p_1 enabled in C must be on the same base object X: otherwise they *commute*, i.e., configurations $C.p_0.p_1$ and $C.p_1.p_0$ are indistinguishable (the process and base-object states are identical), but have opposite valences.

Moreover, as we have shown in the class, X cannot be a register (to see this, consider the cases of read and write operations performed by p_0 and p_1 and show that in each case we can find indistinguishable configurations of opposite valences).

> Note that until now we have not used the assumption that A uses only registers and TAS objects. The claims above hold for any wait-free consensus algorithm using base objects.

Thus, X must be a TAS object. But then $C.p_0.p_1$ and $C.p_1.p_0$ only differ in the local states of p_0 and p_1: only these two processes "know" who won the TAS object and who lost it, and all base objects have identical states in the two configurations.

Thus, p_2 running solo from $C.p_0.p_1$ must decide the same value as it would decide running solo from $C.p_1.p_0$—a contradiction with the assumption that $C.p_0$ (and, thus, $C.p_0.p_1$) is 0-valent and $C.p_1$ (and, thus, $C.p_1.p_0$) is 1-valent.

Hence, TAS and registers cannot be used to solve consensus among 3 processes, which, combined with the 2-process consensus algorithm using TAS and registers discussed in class, implies that the consensus number of TAS is 2.