Concurrent Algorithms 2019
Midterm Exam Solutions

December 20, 2021

Problem 1 (2 points)

Tasks.

1. Write a wait-free algorithm that implements a safe MRSW binary register using (any number of) SRSW safe binary registers.
2. Write a wait-free algorithm that implements a regular MRSW binary register using (any number of) safe MRSW binary registers.

Solution

The implementations can be found in the **Registers** lecture (slides $11 - 13$).

Problem 2 (2 points)

A *snapshot* object maintains an array of registers R of size n, has operations $scan()$ and $update_i()$ and the following sequential specification:

```
1 upon update_i(v) do
2 |  R_i ←v
3 upon scan do
4 |  return R
```

Figure 1: Sequential specification of the snapshot object.

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

```
1  upon update_i(v) do
2  |  ts ← ts + 1
3  |  R_i ← (v, ts, scan())
4  upon scan do
5  |  t_1 ← collect(), t_2 ← t_1
6  |  while true do
7  |  |  t_3 ← collect()
8  |  |  if t_3 = t_2 then return ⟨ t_3[1].val, ..., t_3[N].val ⟩
9  |  |
10 |  |  for k ← 1 to N do
11 |  |  |  if t_3[k].ts ≥ t_1[k].ts + 1 then return t_3[k].snapshot
12 |  |  |
13 |  |  t_2 ← t_3
14 upon collect do
15 |  for j ← 1 to N do
16 |  |  x_j ← R_j;
17 |  return x
```

Figure 2: Incorrect implementation of the snapshot object.

Task. 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.

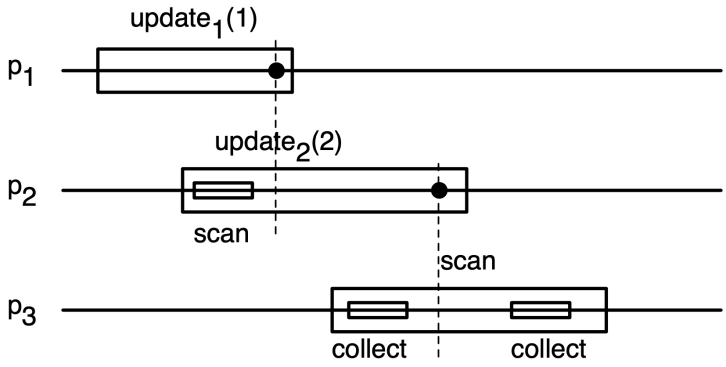

Figure 3: An execution violating atomicity

Problem 3 (3 points)

Consider the linearizable and wait-free log object. The log object supports two operations: append and getLog. The sequential specification of the log object is shown below:

```
1  Given:
2  Sequential linked list L that is initially empty.
3
4  procedure append(obj)
5      L.append(obj)
6
7  procedure getLog()
8      result[] ← ⊥
9      k ← length(L)
10     i ← 1
11     while i ≤ k do
12         i ← i + 1
13         result[i] ← element(L, i) // the element(L, i) function call returns the i-th element of list L
14     return result
```

Figure 4: Sequential specification of the log object.

Furthermore, consider a linearizable and wait-free fetch-and-increment object where its sequential specification is shown below:

```
1  Given:
2  Register R that is initially 0.
3
4  procedure fetchAndIncrement()
5      old ← R
6      R ← old + 1
7      return old
```

Figure 5: Sequential specification of the fetch-and-increment object.

Is it possible to implement the linearizable and wait-free log object by using any number of read-write registers and fetch-and-increment objects? Explain your answer.

Solution

There are two correct answers.

Yes: *In a system of two or fewer processes*, fetch-and-increment can implement an atomic log. We know from class that fetch-and-increment has consensus number 2, thus can be used to implement a universal construction in a system of 2 processes. That universal construction can then be used to implement the atomic log.

No: *In a system of 3 or more processes*, there is no wait-free atomic implementation of a shared log from fetch-and-increment objects and registers. The log object can be used to solve consensus in a system of n processes, where n can be arbitrarily large. To do so, upon invoking *propose(v)*, process p simply appends v to the shared log, then retrieves the log using *getLog()* and decides on the first value in the log. Thus, the log object has consensus number ∞. If it were possible to produce an implementation Imp of a linearizable and wait-free log object using atomic read-write registers and fetch-and-increment objects, Imp could then be used to solve consensus for more than 2 processes. This contradicts the fact that fetch-and-increment has consensus number 2.

Problem 4 (3 points)

An atomic shared counter maintains an integer x, initially 0, and has two operations `inc()` and `read()`. The sequential specification is as follows:

```
1  x integer, initially 0
2  upon read(x) do
3  |  return x
4  upon inc(x) do
5  |  x ← x + 1
```

Consider the following, *incorrect*, implementation of an obstruction-free consensus object from shared counters:

```
   uses: C_0, C_1 – atomic shared counters initialized to 0
1  upon propose(v) do
2  |  while true do
3  |  |  (x_0, x_1) ← readCounters()
4  |  |  if x_0 > x_1 then
5  |  |  |  v ← 0
6  |  |  else if x_1 > x_0 then
7  |  |  |  v ← 1
8  |  |  if |x_0 − x_1| ≥ 1 then
9  |  |  |  return v
10 |  |  C_v.inc()
11 upon readCounters() do
12 |  while true do
13 |  |  x_0 ← C_0.read()
14 |  |  x_1 ← C_1.read()
15 |  |  x'_0 ← C_0.read()
16 |  |  if x_0 = x'_0 then
17 |  |  |  return (x_0, x_1)
```

Give an execution of the above algorithm that shows that the algorithm is not a correct implementation of an obstruction-free consensus object, i.e. an execution in which some property (obstruction-freedom, validity, or agreement) of obstruction-free consensus is violated.

Solution

Consider a process p which is the only process taking steps. Because p is the only process taking steps and the value of C_v is incremented in the while loop, then eventually the value of C_v is going to be greater than the value of C_{v-1}. Therefore, process p will eventually decide a value, and consequently the algorithm satisfies obstruction-freedom.

The algorithm satisfies validity because if all processes propose the same value v (which could be either 0 or 1), then they increment the same counter C_v, and consequently the value of C_v is would be greater than the value of C_{v-1}.

Since the algorithm satisfies obstruction-freedom and validity, then the only property it violates is agreement. To show that it violates agreement consider the following execution in which process p_0 proposes value 0 and process p_1 proposes value 1:

- First, only process p_0 takes steps until it executes the step at line 8,

- then process p_0 stops and only process p_1 takes steps executing the first iteration of the while loop, where it increments C_1, and the second iteration of the loop, in which it decides 1 (because $C_1 = 1$ and $C_0 = 0$),
- then process p_0 resumes taking steps incrementing C_0 at line 10 and executing the second iteration of the loop.
- Because during the second iteration of the loop by p_0 the values of C_0 and C_1 are the same ($C_1 = 1$ and $C_0 = 1$), then p_0 increments C_0 again and executes the third iteration of the loop in which it decides 0 (because $C_1 = 1$ and $C_0 = 2$).

Problem 5 (2 points)

An atomic 0-**set-once** object is a shared object that has three states $\perp$, 0, and 1. $\perp$ is the initial state. It provides only one operation *set(v)* where $v \in \{0, 1\}$, such that:

- If the object is in state $\perp$, then *set(v)* changes the state of the object to v and returns v.
- If the object is in state s where $s \in \{0, 1\}$, then *set(v)* changes the state of the object to $s \wedge \neg v$ and returns the new state of the object (i.e., $s \wedge \neg v$).

Tasks.

1. Explain what it means for a shared object to have infinite consensus number.
2. Prove that the atomic 0-**set-once** object has infinite consensus number.

Solution

A shared object has infinite consensus number if and only if can solve wait-free consensus among *any* number of processes.

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}:

```
   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 ⊥.
 1 upon propose(v) for processes p_1, ..., p_n do
 2 |   val ← C_n.propose(v)
 3 |   R[0] ← val
 4 |   t ← S.set(0)
 5 |   if t = 0 then
 6 |   |   return R[0]
 7 |   else
 8 |   |   return R[1]
 9 upon propose(v) for process p_{n+1} do
10 |   R[1] ← v
11 |   t ← S.set(1)
12 |   if t = 1 then
13 |   |   return R[1]
14 |   else
15 |   |   return R[0]
```