Concurrent Algorithms 2016
Midterm Exam

November 29th, 2016

Time: 1h45

Instructions:

- This midterm is "closed book": no notes, electronics, or cheat sheets 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 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.
- Read through each problem before starting to solve it.

Good luck!

Problem	Max Points	Score
1	3	
2	4	
3	3	
Total	10	

,

Problem 1 (3 points)

Your tasks:

1. Explain the difference between a regular register and an atomic register. Provide an example execution that is allowed for a regular register but not allowed for an atomic register.
2. Write an algorithm that implements an SRSW atomic register using (any number of) SRSW regular registers.
3. In your algorithm, if you replace the base registers (SRSW regular registers) by MRSW regular registers, does your algorithm yield an MRSW atomic register? Justify your answer. (I.e., give a counterexample if your algorithm does not work, or prove the correctness if it does.)

Solution

1. When using a regular register, a read that is concurrent with a write returns either the value being written or the previous value; a read that is not concurrent with a write returns the most recently written value. When using an atomic register, every operation appears to execute instantaneously at some point between the operation invocation and response. Figure 1 shows an execution that is regular but not atomic.
2. The transformation is given in slide 17 of the lecture on *Registers*.
3. The transformation does not work for multiple readers (the result is not an atomic register). The non-atomic execution in Figure 1 is possible in this case. Since the register is regular, the read by R1 may read the value 2 being concurrently written by W. Since this is R1's first read operation, the timestamp it obtains for value 2 is higher than its local timestamp. Later, the read by R2 (also concurrent with *Write(2)*) may read the previous value of the register (1). Since this is R2's first read operation, the timestamp it obtains for value 1 is also higher than its local timestamp.

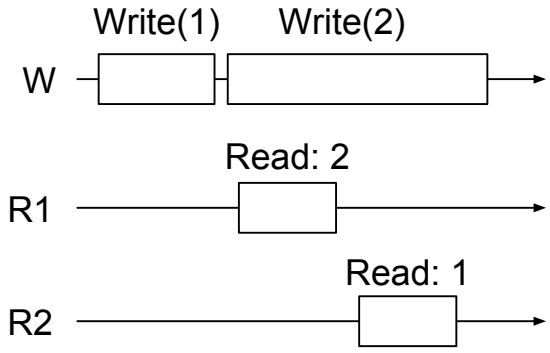

Figure 1: An execution that is possible with a regular register but not with an atomic register. There are three processes: a writer (W) and two readers (R1 and R2). The two reads are concurrent with the second write. The read by R1 completely precedes the read by R2. The execution is not atomic because it is impossible to assign linearization points to all operations: if the linearization point of *Write(2)* is before that of the read by R1, then the read by R2 cannot have a linearization point; if the linearization point of *Write(2)* is after that of the read by R1, then that read cannot have a linearization point.

,

Problem 2 (4 points)

A LIFO *stack* is a shared object that provides the following atomic operations:

- *push(e)* that pushes element e at the beginning of the stack, and
- *pop()* that returns the element from the beginning of the stack and removes it from the beginning. If the stack is empty, then *pop()* returns $\perp$. (Symbol $\perp$ can only be returned if the stack is empty.)

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

- *Agreement*: No two processes decide different values.
- *Validity*: The value decided is one of the values proposed.

Your tasks:

1. Devise an algorithm that implements a wait-free consensus object using (any number of) atomic stacks that are *initially empty* (uninitialized) and atomic registers in a system of 2 processes.
2. Explain why your algorithm satisfies the **Agreement** and **Validity** properties.

Solution

The main idea is to use the algorithm that implements a consensus in a system of 2 processes using initialized stacks and to add a new "initialization" phase.

The algorithm (denoted by Algorithm 1 later) that implements a consensus in a system of 2 processes using initialized stacks works as follows.

```
Using an atomic shared stack S, initialized to {loser, winner}.
Using two atomic shared registers R[2], initialized to 0.

cons(val)
    R[i] = val;
    s = S.pop();
    if s = winner then return val
    else return R[3-i]
```

Now we use the *cons* method of Algorithm 1 as a procedure *cons* in our algorithm (denoted by Algorithm 2 later). As a procedure in Algorithm 2, *cons* takes an atomic shared stack S and two atomic shared registers $R[2]$ as an argument.

```
Using two atomic shared stacks S[2], initially empty.
Using two atomic shared registers R[2], initialized to 0.
Using two atomic shared registers ready[2], initialized to false.

propose(val)
    S[i].push(loser);
    S[i].push(winner);
```

```
ready[i] = true;
for k = 1, 2 do
    if ready[k] = true then val = cons(S[k], R, val);
return val
```

In Algorithm 2, both processes first "initialize" their stacks and then they execute Algorithm 1 on all stacks that have been initialized using the last decided value.

Validity is clearly satisfied, because local variable *val* is only assigned with a value proposed. **Agreement** can be shown by contradiction. Suppose two processes do not agree. Clearly, if they both find $ready[2] = true$, then they agree on the same value; if they both find $ready[1] = true$, they also agree on the same value (as the value they use as an argument in *cons* for $k = 2$ is a result of $k = 1$). Then the only possibility is that (1) when P_1 returns, P_1 finds $ready[1] = true, ready[2] = false$, and (2) when P_2 returns, P_2 finds $ready[1] = false, ready[2] = true$. This means that before P_2 enters the loop, P_1 has already invoked procedure *cons* at least once and has assigned *true* to $ready[1]$. As a result, when doing the loop, P_2 would find $ready[1] = true$, a contradiction.

,

Problem 3 (3 points)

A FIFO *queue* is a shared object that provides the following atomic operations:

- *enq(e)* that puts element *e* at the end of the queue, and
- *deq()* that returns the element from the beginning of the queue and removes it from the beginning. If the queue is empty, then *deq()* returns $\perp$. (Symbol $\perp$ can only be returned if the queue is empty.)

For this problem we will examine a queue implementation whose *enqueue* method does not have a linearization point. The implementation uses the following shared objects:

- *tail*, an atomic *fetch&increment* shared object, initially equal to `0`.
- *items*, an infinite array of atomic *swap* shared objects, all initially equal to $\perp$.

An atomic *fetch&increment* object stores an integer value *x* and provides two methods, *read* and *fetch&increment*, with the following sequential specification:

```
read() {
  return x
}

fetch&increment() {
  oldVal = x
  x = x+1
  return oldVal
}
```

An atomic *swap* object stores an integer value *x* and provides two methods, *write* and *swap*, with the following sequential specification:

```
write(v) {
  x = v
}

swap(v) {
  oldVal = x
  x = v
  return oldVal
}
```

(please see next page)

The pseudocode for the queue is as follows:

```
enq(x) {
  i = tail.fetch&increment()
  items[i].write(x)
}

deq() {
  while (true) {
    range = tail.read()
    for i = 0 to range {
      value = items[i].swap(⊥)
      if (value != ⊥) {
        return value
      }
    }
  }
}
```

Your tasks:

1. Give an example execution showing that the linearization point for *enq* cannot be the *fetch&increment* on the first line (`i = tail.fetch&increment()`). Hint: give an execution where two *enq* calls are not linearized in the order they execute this line.
2. Give another example execution showing that the linearization point for *enq* cannot occur at the *write* on the second line (`items[i].write(x)`).
3. Since these are the only two shared memory accesses in *enq*, we must conclude that *enq* has no single linearization point. Does this mean *enq* is not linearizable? Justify your answer.

Solution

Here is an execution where two enq() calls are not linearized in the order they execute the first line. See also Figure 2.

1. P calls *fetch&increment* and returns `0`.
2. Q calls *fetch&increment* and returns `1`.
3. Q stores item q at array index `1`.
4. R finds array index `0` empty.
5. R finds array index `1` full and dequeues q.
6. P stores item p at array index `0`.
7. R finds array index `0` full and dequeues p.

Here is an execution where two enq() calls are not linearized in the order they execute the second line. See also Figure 3.

1. P calls *fetch&increment* and returns `0`.
2. Q calls *fetch&increment* and returns `1`.
3. Q stores item q at array index `1`.

4. P stores item p at array index `0`.
5. R finds array index `0` full and dequeues p.
6. R finds array index `1` full and dequeues q.

These examples do not mean the method is not linearizable, they just illustrate that we cannot define a single linearization point that works for all method calls.

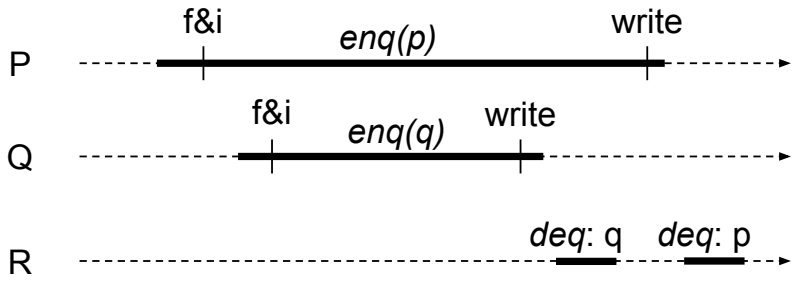

Figure 2: Execution showing that the first line of the *enq()* method cannot be its linearization point.

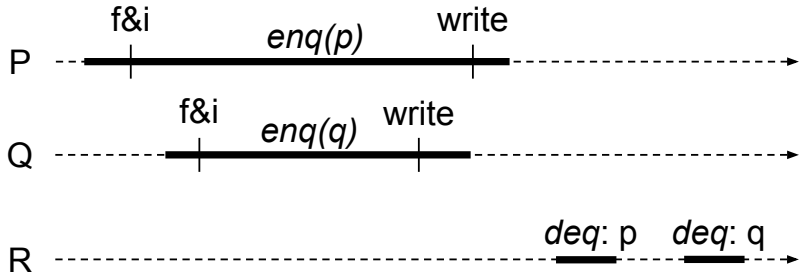

Figure 3: Execution showing that the second line of the *enq()* method cannot be its linearization point.

,