Concurrent Algorithms 2016
Final Exam

February 1st, 2017

Time: 8h15 - 11h15 (3 hours)

Instructions:

- This exam 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 variable represents which shared object (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.
- 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	1	
3	2	
4	2	
5	2	
6	1	
Total	10	

Problem 1 (2 points)

Write an algorithm that implements a MRMW atomic multi-valued wait-free register using (any number of) MRSW atomic multi-valued wait-free registers.

Solution

The transformation is given in the slides on Registers.

Problem 2 (1 point)

Recall that base objects are *not* always correct and they may *fail*. In this problem, we assume that at most t base objects may fail. There are two types of object failures:

Responsive. The object only fails *once*; but when it fails, it fails forever. If a process calls an operation on a *responsive failed object*, it will return a specified value ($\perp$) and announce the process that it is faulty.
Non-responsive. In this type of failure, if a process calls an operation on a *non-responsive failed object*, the object will never reply to that process.

Your tasks:

1. Implement a failure-resilient SWMR register out of $t + 1$ SWMR base *responsive* failure-prone registers.
2. Implement a failure-resilient SWSR register out of $2t + 1$ SWSR base *non-responsive* failure-prone registers.

Solution

The transformations are given in the slides on faulty base objects.

Problem 3 (2 points)

An (m, n)-assignment object, where $n \geq m > 1$, has n fields (for instance, an n-element array) and two operations: `assign()` and `read()`. The `assign()` operation takes as arguments m values $v_1, ..., v_m$ and m indices $i_1, ..., i_m$ and atomically assigns value v_j to array element i_j, for $j = 1, ..., m$. The `read()` operation takes an index argument i and returns the i^{th} array element.

Your task is to provide an algorithm that solves consensus in a system of 2 processes using only atomic $(2, 3)$-assignment objects and atomic registers.

Solution

Please refer to Section 3.6 of the paper "Wait-free Synchronization" by Maurice Herlihy:

`https://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf`

Problem 4 (2 points)

Consider the following *incorrect* implementation of an obstruction-free consensus object from atomic multi-valued MRMW shared registers in a system of n processes. A process' id is known to itself as i.

```
Using: an array of atomic multi-valued MRMW shared registers T[1, 2, ..., n],
       initialized to 0;
Using: an array of atomic multi-valued MRMW shared registers V[1, 2, ..., n],
       initialized to (⊥, 0);

propose(v) {
    ts := i;
    while (true) do{
        T[i].write(ts);

        maxts := 0;
        val := ⊥;
        for j = 1 to n do
            (t, vt) := V[j].read();
            if maxts < t then
                maxts := t;
                val := vt;

         if val = ⊥ then val := v;

         maxts := 0;
         for j = 1 to n do
             t := T[j].read();
             if maxts < t then maxts := t;

         if ts = maxts then
             V[i].write(val, ts);
             return(val);
         ts := ts + n;
    }
}
```

Recall that obstruction-free consensus ensures the property of *obstruction-freedom* instead of *wait-freedom*. **Your tasks:**

1. Explain what is obstruction-freedom and what is the difference between obstruction-freedom, lock-freedom and wait-freedom.
2. Answer whether the implementation satisfies obstruction-freedom. Justify your answer.
3. Answer which property of obstruction-free consensus the implementation violates. Give an execution that shows the implementation indeed violates that property.

Solution

Let a correct process be a process that does not crash. Then obstruction-freedom stipulates the following:

- An implementation (of a shared object) is obstruction-free if any of its operations returns a response if it is eventually executed without concurrency by a correct process.

Wait-freedom is stronger: any correct process that executes an operation eventually returns a response. The difference is concurrency. Obstruction-freedom ensures termination in an obstruction-free execution, i.e., assuming that eventually at most one process is taking steps. However, in other executions, an obstruction-free implementation can never termiante.

The implementation is obstruction-free. Suppose that eventually only process P is taking steps. Then eventually P finds its local timestamp ts is the highest among all the values in the registers in array T, and then returns a value.

Now we give an example execution where the implementation violates agreement, which shows the implementation is incorrect. Figure **??** illustrates the example execution. Assume two processes P_1 and P_2.

1. P_1 proposes some value v_1. P_1 executes until the condition $ts = maxts$. P_1 checks the condition to be true. Then P_1 is suspended.
2. P_2 proposes some value v_2. P_2 executes to the end. We note that in the first loop, P_2 sees that each cell of an array V is $(\perp, 0)$ and thus P_2 assigns v_2 to val after the first loop. Then P_2 decides v_2.
3. P_1 now continues and decides v_1.

The example execution breaks agreement as P_1 and P_2 returns their own proposals, which can be different.

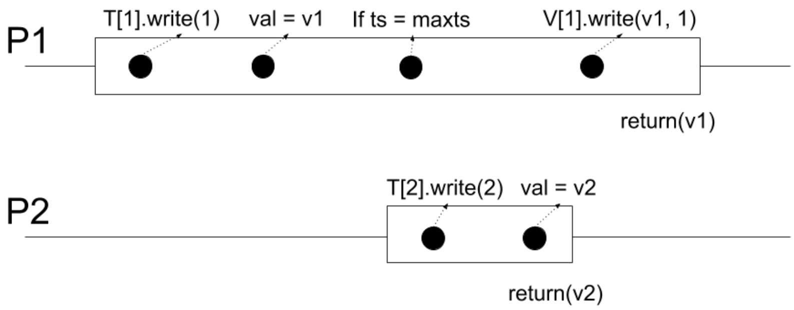

Figure 1: Example execution of an incorrect implementation of obstruction-free consensus

Problem 5 (2 points)

Recall that a *weak counter* is a shared object that provides a single operation *wInc* which returns an integer. Operation *wInc* has the following **weak increment** property:

- If one operation $wInc_1$ precedes another $wInc_2$ (i.e., $wInc_1$ ends before $wInc_2$ starts), the value returned by the later operation $wInc_2$ must be larger than the value returned by the earlier one $wInc_1$.

We note that two concurrent *wInc* operations may return the same value.

For this problem, we examine an *incorrect* implementation of the weak counter object with anonymous processes. The pseudocode is as follows.

```
Using: an infinite array of atomic binary MRMW shared registers R[1, 2, ...],
       initialized to 0;
Using: an atomic multi-valued MRMW shared register L, initialized to 0;

wInc() {
    k := 1;
    l := L.read();
    t := l;
    while (R[k].read() ≠ 0) do
        if (L.read() ≠ l) then
           l := L.read();
           t := max(t,l);
           return(t);
        k := k + 1;
    R[k].write(1);
    L.write(k);
    return(k);
}
```

The number of processes is n and known to every process. Assume that $n \geq 2$. Explain why the implementation is incorrect. **Your tasks:**

1. Answer whether the implementation above is an **anonymous** implementation or not. Justify your answer.
2. Answer which property of the weak counter shared object with anonymous processes the implementation violates. Give an execution that shows the implementation indeed violates that property.

Solution

The implementation violates weak increment. We show a counter-example later.

The pseudocode is an anonymous implementation. Recall that in an anonymous system, a collection of n processes execute identical algorithms; in particular, the processes do not have identifiers.

Now we give an example execution where the implementation violates weak increment, which shows the implementation is incorrect. Figure **??** illustrates the example execution. Assume two processes P_1 and P_2.

1. P_1 reads $R[1]$ and finds it 0. P_1 skips the loop. Then P_1 is suspended.

2. P_2 reads $R[1]$ and also finds it 0. P_2 executes $wInc$ to the end and returns 1.

3. P_2 reads $R[1]$, finds it 1, and then reads $R[2]$ and finds it 0. P_2 executes $wInc$ to the end and returns 2.

4. P_2 reads L to l and thus $t = l = 2$. P_2 reads $R[1]$ and finds it 1. P_2 enters the loop. Then P_2 is suspended.

5. P_1 now continues and writes 1 to $R[1]$ and L.

6. P_2 now continues. P_2 checks the condition L.read() $\neq l$ and finds it true. Then P_2 reads L to l and thus $t = 2, l = 1$. P_2 returns 2.

The example execution breaks weak increment as P_2 has two operations, one preceding the other, which return the same value.

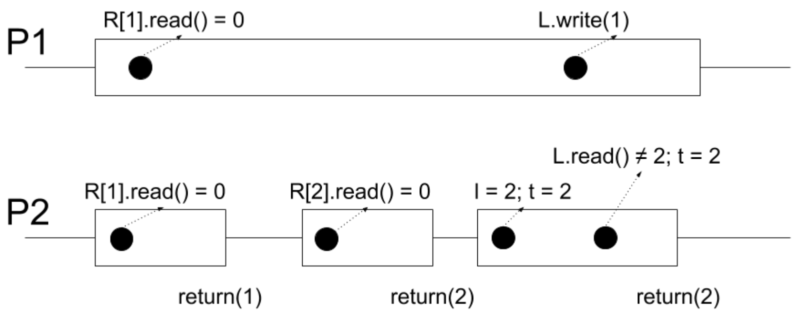

Figure 2: Example execution of an incorrect anonymous implementation of weak counter

Problem 6 (1 point)

Recall that *consensus* is a shared object that has a single operation *propose* and satisfies agreement, validity, and termination. In this problem, we consider a shared object called *weak agreement*.

Weak agreement also has a single operation *propose*. Each process p proposes a value v. If the operation returns d to p, then we say that p decides d and decision d consists of a pair (dec, val) where *dec* can be either *commit* or *suggest*. Weak agreement satisfies the following properties:

- **Validity**: any *val* in a decision is a value proposed by some process.
- **Weak agreement**: if any process decides $(commit, v)$, then every process (that does not crash) decides $(commit, v)$ or $(suggest, v)$.
- **Commitment**: if every process proposes the same value v and every process decides, then at least one process decides $(commit, v)$.
- **Termination**: every process (that does not crash) eventually decides.

Your tasks:

1. Give the sequential specification of a *snapshot* shared object.
2. Give an algorithm that implements weak agreement using (any number of) snapshot shared objects and (any number of) atomic multi-valued MRMW shared registers. Justify your answer. (**Hint**: there is a solution using only two snapshot shared objects.)

Solution

A snapshot object can be seen as a vector of n elements. It has two operations: *update(i, v)* and *snapshot()*. Operation *update* takes a position i and a value v as arguments and updates the ith element of the vector to v. Operation *snapshot* returns a vector of n values. The sequential specification of the snapshot object is defined as a set of sequential histories of *update* and *snapshot* operations. In every such sequential history, each position i of the vector returned by every *snapshot* operation contains the argument of last preceding *update* operation (if any, or the initial value $\perp$ otherwise).

Here is a possible algorithm that implements weak agreement using only two snapshot shared objects.

```
Using two snapshot shared objects: S_1 and S_2 of size n, each element of which is initialized to ⊥;
Using two local array of registers: ai and bi of size n;

propose(v){
    S_1.update(i,v);
    ai := S_1.snapshot();
    if every non-⊥ value in ai is v or every value in ai is ⊥ then
        x := (commit, v);
    else
        x := (suggest, v);
    S_2.update(i,x);
    bi := S_2.snapshot();
    if every value in bi is equal to (commit, v) then
        return (commit, v);
    if some value in bi is equal to (commit, v) then
```

```
        return (suggest, v);
    return (suggest, v);
}
```

It is easy to see that termination and validity are satisfied. For weak agreement, if some process P decides $(commit, v)$, then P finds that every value in *bi* is $(commit, v)$, which means that every process updates S_2. Then for any other process Q, when Q decides, Q sees at least one $(commit, v)$ in *bi*, which is updated by Q itself into S_2. Thus Q decides either $(commit, v)$ or $(suggest, v)$. For commitment, if every process proposes the same value v and every process decides, then every process eventually updates S_1 with $(commit, v)$. Then the process that checks the condition "every value in *bi* is equal to $(commit, v)$" must find the condition to be true and thus returns $(commit, v)$.