Concurrent Algorithms 2015
Final Exam

January 21, 2016

Time: 12:15–15:15 (3 hours)

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.

Good luck!

Problem	Max Points	Score
1	1	
2	2	
3	2	
4	2	
5	1	
6	2	
Total	10	

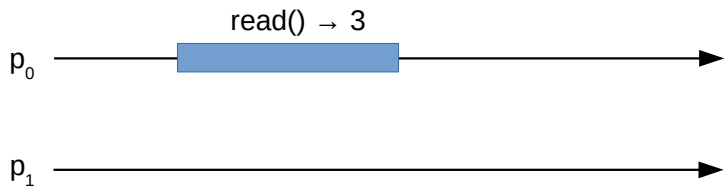
Problem 1 (1 point)

Give the definitions of *non-safe* (i.e. weaker than safe), *safe*, *regular*, and *atomic* registers. Additionally, show four register executions:

- One that is non-safe;
- One that is safe but not regular;
- One that is regular but not atomic, and
- One that is atomic.

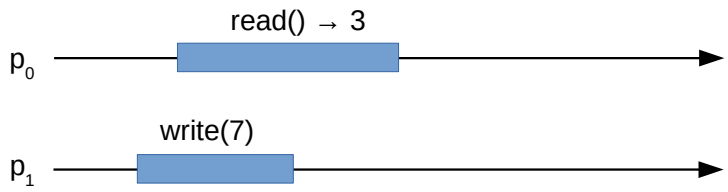
Solution

Non-safe register. A read operation can return any arbitrary value.

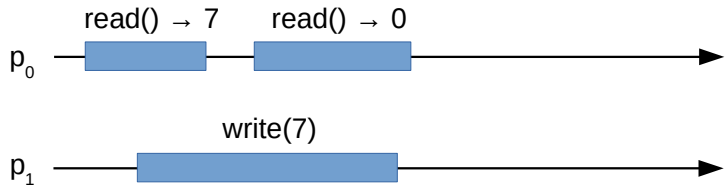


(Safe and regular registers are defined for a single writer.)

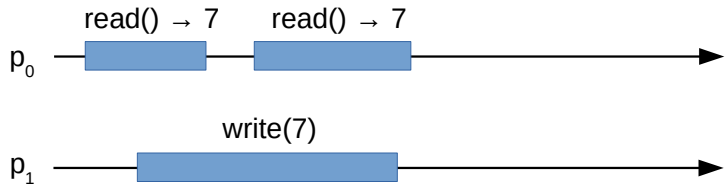
Safe register. A read operation on a safe register returns the last written value in the absence of concurrent writes. In the presence of concurrent writes, a read can return any arbitrary value.



Regular register. A regular register is a safe register with the additional following property: A read that is concurrent with one or several writes returns the value written by a concurrent write or the value written by the last preceding write.



Atomic register. An atomic register is a multiple-reader multiple-writer register whose execution histories are *linearizable*. It is possible to totally order all its read and write operations in such a way that this total order S respects their real-time occurrence order and each read returns the value written by the last write operation that precedes it in S.



Problem 2 (2 points)

In the lectures of the course, we introduced the *splitter* object. The splitter object can return any of three values: *stop*, *down* and *right*. The splitter object ensures the following:

1. If a single process executes *DIRECTION*, then the process returns *stop*;
2. If two or more processes execute *DIRECTION*, then not all of them return the same value; and
3. At most one process returns *stop*.

For this problem, we introduce the *fair-splitter* object. Every fair-splitter object ensures the following:

1. If a single process executes *DIRECTION*, then the process returns *stop*;
2. If n processes execute *DIRECTION*, then **exactly one** of them returns *stop*. From the rest of the processes, **half** of them return *right* and **half** return *down*. If n is an even number, the processes are distributed as $n/2$ and $((n/2)-1)$ to *right* and *down*, or the opposite (it does not matter which of the two return values gets the extra process).

Your tasks are the following:

1. Give two implementations for a *fair-splitter* object using:
   - Any number of atomic Compare&Swap objects (i.e., objects that provide the *CAS* operation) and atomic registers.
   - Any number of atomic Fetch&Increment objects (i.e., objects that provide the *FAI* operation) and atomic registers.
2. Construct an algorithm that uses any of the two fair-splitter objects you developed and atomic registers to solve the renaming problem. What is the largest name that can be given by the renaming algorithm? Is it *adaptive*? Explain your answer.

The sequential specifications of the *CAS* and *FAI* operations are the following:

```
1 upon FAI() do
2     temp ← value;
3     value ← value + 1;
4     return (temp);
```

```
1 upon CAS(oldvalue, newvalue) do
2     temp ← value;
3     if value == oldvalue then
4         value ← newvalue
5     return temp
```

Solution

1. The algorithms are the following:

   - Here is a simple implementation using Compare&Swap objects (the implementation does not use the minimum number of objects required, but does however get all points of the exercise):

     uses: $choice[0, \ldots, \lceil n/2 \rceil - 1]$ — shared array of atomic Compare&Swap objects
     initially: $choice[0, \ldots, \lceil n/2 \rceil - 1] \leftarrow [0, \ldots, 0]$

     ```
     1 upon DIRECTION() do
     2     value ← choice[0].CAS(0, 1)
     3     if value == 0 then
     4         return(stop)
     5     for i ← 1; i up to ⌈n/2⌉ − 1 do
     6         value ← choice[i].CAS(0, 1)
     7         if value == 0 then
     8             return(right)
     9     return(down)
     ```

   - Here is an implementation using Fetch&Increment objects:

     uses: *choice* — shared atomic Fetch&Increment object
     initially: $choice \leftarrow 0$

     ```
     1 upon DIRECTION() do
     2     value ← choice.FAI()
     3     if value == 0 then
     4         return(stop)
     5     else if (value % 2) == 1 then
     6         return(down)
     7     else
     8         return(right)
     ```

2. The simplest answer to this question relies on the fact that a *fair-splitter* object returns *stop* **exactly once**. We use a one-dimentional array of fair-splitter objects. Each process goes through the array, until it gets stop. It then returns the index of the array. The the largest name that can be returned is p (the number of participating processes) and the algorithm is adaptive, since the largest name depends only on the number of participating processes.

   uses: $names[0, \ldots, n]$ — shared fair-splitter object array

   ```
   1 upon NEW_NAME(id) do
   2     j ← 0
   3     while True do
   4         if names[j].DIRECTION() == STOP then
   5             return(j)
   6         j ← j + 1
   ```

Note: there are different solutions to this problem depending on the algorithm devised in the first question.

Problem 3 (2 points)

Recall that if a shared object has *consensus number* $+\infty$, then it solves consensus in a system of n processes for any integer n. In the lecture, we show that atomic registers have consensus number 1. This problem discusses registers augmented with register-to-register primitives.

1. Let R_1 and R_2 be two atomic registers. The register-to-register *move* primitive $R_1.move(R_2)$ moves atomically the content of R_2 into R_1 and sets the value of R_2 to 0.
2. Let R_1 and R_2 be two atomic registers. The register-to-register *copy* primitive $R_1.copy(R_2)$ copies atomically the content of R_2 into R_1 (the content of R_2 is not modified).

Your task is to prove that both objects have a consensus number of $+\infty$.

Hint: Use arrays whose size is the number of processes n.

Solution

First, we give the construction based on the register-to-register copy primitives. The processes share two arrays: $prefer[1,\ldots,n]$ and $r[1,\ldots,n][1,2]$, where $r[i,1]$ is initialized to 1 and $r[i,2]$ to 0 for each $i, 1 \leq i \leq n$.

uses: $prefer[1,\ldots,n]$ – shared atomic registers.
uses: $r[1,\ldots,n][1,2]$ – shared atomic registers augmented with register-to-register copy primitives.

```
1 upon propose_i(v_i) do
2     prefer[i] ← v_i
3     r[i,2].copy(r[i,1])
4     for j := i + 1, ..., n do
5         r[j,1] ← 0
6     for j := n, ..., 1 do
7         if r[j,2] = 1 then
8             return prefer[j]
```

Second, we give the construction based on the mem-to-mem move primitives. The processes share two arrays: $prefer[1:n]$ and $winner[1:n]$, and an extra register x, where x is initialized to 1 and $winner[i]$ is initialized to 0 for each $i, 1 \leq i \leq n$.

uses: $prefer[1,\ldots,n]$ – shared atomic registers.
uses: $winner[1,\ldots,n]$ and x – shared atomic registers augmented with register-to-register copy primitives.

```
1 upon propose_i(v_i) do
2     prefer[i] ← v_i
3     winner[i].move(x)
4     for j := 1, ..., n do
5         if winner[j] = 1 then
6             return prefer[j]
```

Problem 4 (2 points)

Recall that base objects are *not* always correct and they may *fail*. In this problem, we assume that at most t base registers might 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*, it will never reply to that process.

In the lecture, we presented an implementation of a failure-free MRSW atomic register from $(t+1)$ MRSW responsive base atomic registers. The write (read) operation of this implementation writes (reads) base registers in a specific order, which prevents parallelization.

You tasks are the following:

1. Re-implement a failure-free SRSW atomic register from $(t+1)$ MRSW responsive base atomic objects. Your implementation should write (read) base registers without a specific order, which can thus be issued in parallel.

2. Explain why your solution produces a failure-free SRSW atomic register, but **not** a MRSW register.

Hint: Consider the base atomic registers that are potentially unbounded; i.e., those registers can accommodate an arbitrarily large value.

Solution

The construction is based on sequence numbers. It consequently requires base atomic registers that are potentially unbounded. The $t+1$ registers are denoted $R[1,\ldots,(t+1)]$. Each register $R[i]$ is made up of two fields denoted $R[i].sn$ (sequence number part) and $R[i].val$ (value part). Each base register $R[i]$ is initialized to the pair $(v_{init}, 0)$ where v_{init} is the initial value of the constructed register. sn, $last$ and aux are local variables. $last$ is initialized to $(v_{init}, 0)$.

uses: $R[1,\ldots,(t+1)]$ – shared atomic registers, $last$ — local variable
initially: $R[1,\ldots,(t+1)] \leftarrow [(0,\perp),\ldots,(0,\perp)]$, $last \leftarrow (0,\perp)$

```
1  upon write do
2      sn ← sn + 1
3      invoke write(v, sn) on all R[1, ..., t + 1]
4      for j ∈ {1, ..., t + 1} do
5          R[j] ← (v, sn)
6  upon read do
7      invoke read() on all R[1, ..., t + 1]
8      wait for t + 1 responses
9      if any pair (v, sn) has sn > last.sn then
10         last ← (v, sn)
11     return last.val
```

Problem 5 (1 point)

Recall that *anonymous* processes do not have *id*'s although they may know the total number of processes.

In the lecture, we presented an implementation of a wait-free weak counter from wait-free atomic registers among anonymous processes. This problem discusses the relation between two different counters.

A *mod-m* counter has state set $\{0, 1, \ldots, m-1\}$. It has two operations: *inc* increments the current state from x to $(x+1)$ *mod* m; *read* returns the current state.

An m-valued counter has state set $\{0, 1, \ldots, m\}$. It also has two operations: *inc* and *read*. However, it can only increment *normally* m times. Normally, *inc* increments the current state from x to $x+1$ and *read* returns the current state, but after m increments, i.e., in state m, *read* may return nondeterministally any of the vaules $0, 1, 2, \ldots, m-1$.

Your task is to implement an $(m+1)$-valued counter from a mod-m counter and atomic registers among n anonymous processes. Briefly explain how it is linearized.

Hint: Focus on the corner cases.

Solution

We can implement an $(m+1)$-valued counter from a mod-m counter C and an atomic register R. Assume that C and R are both initialized to o. The variables x and y are local variables.

```
uses: C – shared mod-m counter.
uses: R – shared atomic register.
1  upon inc do
2      C.inc()
3      R ← 1
4  upon read do
5      y ← R
6      x ← C
7      if y = 0 then
8          return 0
9      else
10         if x = 0 then
11             return m
12         else
13             return x
```

Linearize all *inc* operations whose accesses to C occur before the first write to R in the execution at the first write to R (in an arbitrary order). Linearize all remaining *inc* operations when they access C.

Linearize any *read* that reads 0 in R at the moment it reads R. Since no *inc* are linearized before this, the *read* is correct to return 0. Linearize each other *read* when it reads counter C. If at most m *inc* are linearized before the *read*, the *read* is allowed to return any result.

Problem 6 (2 points)

Consider the `CASn` compare-and-swap operation that atomically compare-and-swaps n given memory locations. The sequential specification of `CASn` is:

```
// n : number of addresses
// addr[n] : the n target addresses. An address can appear in addr[n] only once
// old[n] : the n old values
// new[n] : the n new values
boolean CASn(n, addr[n], old[n], new[n]) //sequential specification of CASn
{
  for (i = 0; i < n; i++)
    if (*addr[i] != old[i])
      return false;

  for (i = 0; i < n; i++)
    *addr[i] = new[i]

  return true;
}
```

Your tasks are to:

1. Give the definition of the *opacity* property of software transactional memories (STMs). Explain which use cases opacity covers while traditional safety properties do not.
2. Implement the `CASn` operation using STM. You can consider that you have access to an STM system with the following interface:
   - `tx-begin()` – for starting a new transaction;
   - `tx-end()` – for ending a transaction;
   - `tx-read(addr)` – for loading some shared data within a transaction;
   - `tx-write(addr, val)` – for storing a new value to some shared memory within a transaction.
3. Design and highlight the implementation of an STM algorithm using `CASn`.

Solution

a. See slides for a definition of opacity. Opacity disallows executions that could potentially result in irrevocable operations, such as a division by zero.

b. We can easily implement `CASn` with STM:

```
boolean CASn(n, addr[n], old[n], new[n]) //CASn implementation using STM
{
  tx-begin();
  for (i = 0; i < n; i++)
    if (tx-read(addr[n]) != old[n]) {
      tx-end();
      return false;
    }

  for (i = 0; i < n; i++)
    tx-write(addr[n], new[n]);
```

```
  tx_end();
  return true;
}
```

c. The three main "operations" that we have to design for an STM algorithm (if we disregard contention management) are: (i) the transactional read, (ii) the transactional write, and (iii) the commit of a transaction.

The `CASn` operation pinpoints to an STM design that tries to atomically perform the whole set of writes of a transaction in the commit phase using a single `CASn` operation. Accordingly, transactional writes must be buffered. Regarding transactional loads, the simplest approach that fits with the aforementioned design is to employ version numbers and validation of the read set on every transactional read.

Overall, if we assume that we can read a $< version, value >$ pair of an address at once:

Transactional read:

1. Try to find the address in the write set of the transaction. If found, return the corresponding value.
2. Add the $\{address, < version, value >\}$ tuple for the target address in the read set if it is not already in there.
3. Validate that none of the entries of the read set has been modified. This can be performed either address-by-address, or using `CASn`. With `CASn`, we can create two arrays–one with the addresses (`addr`) and one with the $< version, value >$ pairs (`vv`)–and then call `CASn(read-set-size, addr, vv, vv)`. If this invocation returns true, then the read set has been validated, otherwise we need to restart the transaction.
4. Return the corresponding value.

Transactional write:

1. Add the $\{address, < version, value >, newvalue\}$ tuple for the target address in the write set. If an entry for the target address exists in the write set, update the *newvalue* field.

Transactional commit:

1. Validate for a last time the read set.
2. Create the following arrays from the write set: (i) `addr` which includes the target addresses, (ii) `old` which includes the $< version, value >$ pairs stored while writing, and (iii) `new` which includes $< version + 1, newvalue >$ pairs for the corresponding address.
3. Perform `CASn(write-set-size, addr, old, new)`. If true, then the new values and the new version numbers of the addresses have been establish and the transaction is committed. If false, the transaction must be restarted.