Concurrent Algorithms 2014
Final Exam

January 15, 2015

Time: 16:15–19:15 (3 hours)

Instructions:
• This exam is closed book: no notes, electronics, or cheat sheets allowed.
• When solving a problem do not assume any known results from the lectures, unless the problem explicitly states that you might use some 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 you can initialize shared objects to any state.
• Unless otherwise stated we assume atomic multi-valued MRMW shared registers.
• Unless otherwise stated we assume a system of \( n \) asynchronous processes, \( n - 1 \) of which might crash.
• For every algorithm you write, provide a short explanation of why the algorithm is correct.

Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
Problem 1 (2 points)

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

Solution

Since SRMW atomic multi-valued registers are more powerful than SRSW atomic multi-valued registers we can use the following register transformations from the class:

1. Implement MRSW atomic multi-valued registers using SRSW atomic multi-valued registers,
2. Implement MRMW atomic multi-valued registers using MRSW atomic multi-valued registers,
Problem 2  (2 points)

You are given the following partial implementation of a wait-free stack object that uses an atomic Fetch&Add (F&A) object and an infinite array of registers that support an atomic swap operation. Given the implementation of the push operation, your task is to implement the corresponding atomic wait-free pop operation.

A stack is a shared object that implements a LIFO (last in, first out) data structure and provides the following operations:

1. push(v) that puts element v at the top of the stack,
2. pop() that returns the element on the top of the stack and removes it from the stack. The pop() operation returns the special value ⊥ if called on an empty stack.

The sequential specification of a Fetch&Add object is the following (x can be any integer number):

```plaintext
upon F&A(fa,x) do
  t ← fa;
  fa ← fa + x;
  return (t);
```

The sequential specification of a swap operation is the following:

```plaintext
upon swap(register,newvalue) do
  oldvalue ← register;
  register ← newvalue;
  return (oldvalue);
```

uses: range – atomic F&A object initialized to 1
uses: items[1,…] – an infinite array of registers supporting atomic swap operation, each initialized to ⊥

```plaintext
upon push(x) do
  i ← F&A(range,1);
  items[i] ← x;
```

```plaintext
upon pop() do
  t ← F&A(range,0) − 1;
  for i ← t down to 1 do
    x ← swap(items[i],⊥);
  if x ≠ ⊥ then
    return x
  return ⊥
```

Hint: items[range − 1] is not necessarily the topmost element of the stack.
**Problem 3 (2 points)**

A *k*-bounded set object is a set object which can hold up to *k* elements, where $1 \leq k \leq n$. It supports the following atomic operations to access it:

1. *find(e)* which looks for element *e* in the set and if it finds *e*, the operation returns *e*. If element *e* is not in the set, the operation returns *fail*;
2. *delete(e)* which deletes element *e* from the set and returns *ok*. If element *e* is not in the set, the operation returns *fail*;
3. *insert(e)* which inserts element *e* into the set and returns *ok* if the set contains less than *k* elements. If the set contains *k* elements, the operation does not do anything to the set and returns *fail*;

What is the consensus number of a *k*-bounded set object? Prove it.

**Solution**

A *k*-bounded set object has an infinite consensus number because it can be used to implement a consensus objects in a system with any number processes. The main idea of the implementation is to use a *k*-bounded set object which is not empty initially. We will use a *k*-bounded set object which is initialized with *k* − 1 elements, so that it has only one available slot for the *insert()* operation. Each process $p_i$ tries to win the available slot by invoking *insert(i)*. Since there is only one available slot, only one process succeeds in inserting its *id* and becomes the winner. The value proposed by the winner becomes the value decided.

uses: $S$ – shared atomic *k*-bounded set object initialized with *k* − 1 elements \{$\bot_1, \ldots, \bot_{k-1}$\}.
uses: $R[1, \ldots, n]$ – shared registers.

```
/* code of process $p_i$ */
upon propose($v$) do
    R[i] ← $v$;
    if $S$.insert($i$) $\neq$ fail then
        return $v$
    for $i \leftarrow 1$ to $n$ do
        if $S$.find($i$) $\neq$ fail then
            winner ← $i$
    return R[winner]
```
Problem 4  (1 point)

A weak counter is a weakening of a Fetch&Add object. A weak counter supports one operation \texttt{wInc()} which has the following sequential specification:

\begin{verbatim}
1  upon wInc() do
2        t ← countervalue;
3        countervalue ← countervalue + 1;
4  return (t)
\end{verbatim}

The operation satisfies the following correctness property: if an operation precedes another, then the second returns a value that is larger than the first one.

Below is the implementation of a weak counter using registers, as seen in the slides of the course.

\textit{uses}: \texttt{Reg[0,\ldots]} – an infinite array of shared registers initialized to 0.
\textit{uses}: \texttt{L} – a shared register initialized to 0.

\begin{verbatim}
1  upon wInc() do
2        l ← L.read(); t ← l; i ← 0; k ← 0;
3        while Reg[i].read() \neq 0 do
4            i ← i + 1;
5            if L.read() \neq l then
6                l ← L.read(); t ← max(t, l); k ← k + 1;
7            if k = n then
8                return (t)
9        L.write(i);
10       Reg[i].write(1);
11  return (i);
\end{verbatim}

Due to a typo, this implementation is incorrect. Your task is to give an execution of the above algorithm which shows that the algorithm is not a correct implementation of a weak counter object.

\textbf{Hint}: Show an execution in which some operation \texttt{op} precedes some operation \texttt{op}’ and the value returned by \texttt{op}’ is the same as the value returned by \texttt{op} (which violates correctness).

\textbf{Solution}

The problem with the implementation is that the write to \texttt{L}, at line 9, occurs before the write to \texttt{Reg[i]}, at line 10. In a correct implementation a process has to update register \texttt{Reg[i]} before announcing the change by updating register \texttt{L}.

To construct an execution which does not satisfy correctness we will use a system of 2 processes. First, we will let one of the processes set the value of the counter to 1 by allowing the process to execute a \texttt{wInc()} operation. After setting the counter value to 1 we let one of the two processes start executing a slow \texttt{wInc()} operation. We make the slow process to return value 2 at line 8 because of two concurrent \texttt{wInc()} operations by another fast process. The key idea is to suspend the second \texttt{wInc()} operation by the fast process right after it writes 2 to \texttt{L} and before it writes 1 to \texttt{R[2]}. That way, if the slow process executes another \texttt{wInc()} operation it will have to return the same value 2 again because \texttt{R[2]} is still 0.

Consider a system of \(n = 2\) processes.

1. Process \texttt{p1} executes a \texttt{wInc()} operation until the operation completes while \texttt{p2} is suspended. During the execution of the operation \texttt{p1} reads 0 from \texttt{R[0]} at line 3 (thus skipping the while loop), writes 0 to \texttt{L} at line 9, and writes 1 to \texttt{R[0]} at line 10. This results in the following state of shared memory: \(L = 0, R[0] = 1,\) and \(R[i] = 0\) for any \(i \geq 1.\)
2. Process $p_1$ starts executing another $wInc()$ operation. Process $p_1$ executes line 2, reading 0 from $L$ and setting $l$ to 0, and is suspended right before it proceeds to line 3. The state of the shared memory remains the same.

3. Process $p_2$ executes a $wInc()$ operation until the operation completes. During the execution of the operation $p_2$ reads 1 from $R[0]$, enters the first iteration of the while loop, increments $i$ to 1, observes that $L$ did not change (thus skipping assignments in the if statement), reads 0 from $R[1]$ (thus skipping the second iteration of the while loop), writes 1 to $L$ at line 9, writes 1 to $R[1]$ at line 10, completes the operation by returning 1, and is suspended. This results in the following state of shared memory: $L = 1$, $R[0] = 1$, $R[1] = 1$, and $R[i] = 0$ for any $i \geq 2$.

4. Process $p_1$ resumes taking steps. Process $p_1$ reads 1 from $R[0]$, enters the first iteration of the while loop, increments $i$ to 1, reads 1 from $L$ at line 5 and observes that $L$ changed since $l = 0$, reads 1 from $L$ at line 6 and sets $l$ to 1, $t$ to 1, and increments $k$ to 1, it does not execute line 8 since $k = 1$ and $n = 2$. Right before process $p_1$ is about to execute line 2, process $p_1$ is suspended. The state of the shared memory remains the same.

5. Process $p_2$ resumes taking steps by executing another $wInc()$ operation until the operation completes. During the execution of the operation $p_2$ reads 1 from $R[0]$, enters the first iteration of the while loop, increments $i$ to 1, observes that $L$ did not change (thus skipping assignments in the if statement), reads 1 from $R[1]$, enters the second iteration of the while loop, increments $i$ to 2, observes that $L$ did not change (thus skipping assignments in the if statement), reads 0 from $R[2]$ (thus skipping the third iteration of the while loop), writes 2 to $L$ at line 9, and is suspended before it is about to execute line 10. This results in the following state of shared memory: $L = 2$, $R[0] = 1$, $R[1] = 1$, and $R[i] = 0$ for any $i \geq 2$.

6. Process $p_1$ resumes taking steps. Process $p_1$ reads 1 from $R[1]$ at line 2, enters the second iteration of the while loop, increments $i$ to 2, reads 2 from $L$ at line 5 and observes that $L$ changed since $l = 1$, reads 2 from $L$ at line 6 and sets $l$ to 2, $t$ to 2, and increments $k$ to 2. Since $k = n = 2$ at line 7, process $p_1$ executes line 8 and returns value 2.

7. Process $p_1$ executes another $wInc()$ operation until the operation completes while $p_2$ is still suspended. During the execution of the operation $p_1$ reads 1 from $R[0]$, enters the first iteration of the while loop, increments $i$ to 1, observes that $L$ did not change (thus skipping assignments in the if statement), reads 1 from $R[1]$, enters the second iteration of the while loop, increments $i$ to 2, observes that $L$ did not change (thus skipping assignments in the if statement), reads 0 from $R[2]$ (thus skipping the third iteration of the while loop), writes 2 to $L$ at line 9, writes 1 to $R[2]$ at line 10, and executes line 11 returning value 2.
Problem 5  (1 point)

We can define an UpDownCounter object as follows. The state of the object is an integer c. The object implements 3 operations: read returns the state of the object c without changing c, inc increases c by 1 and returns ok, and dec decreases c by 1 and returns ok.

1. Here is a proposed (incorrect) implementation of an UpDownCounter for n processes, using n atomic registers (code for process pi):

   uses: A[1…n] – array of shared registers, each register is initialized to 0.
   uses: vi – local register at every process pi, initialized to 0.

   1) upon readi() do 
      2)     v_i ← 0 
      3)     for k ← 1 to n do 
            4)         v_i ← v_i + A[k].read_i() 
      5)     return v_i 

   6) upon inci() do 
      7)         v_i ← A[i].read_i() 
      8)         A[i].write_i(v_i + 1) 

   9) upon deci() do 
      10)        v_i ← A[i].read_i() 
            11)        A[i].write_i(v_i − 1) 

Show that this algorithm does not implement an atomic, wait-free UpDownCounter by giving an execution of the algorithm in which atomicity (linearizability) is violated.

2. Give an algorithm that implements an UpDownCounter object using only atomic registers and atomic snapshot objects. We remind you that an atomic snapshot object can be implemented using atomic registers. However, you do not need to write the implementation algorithm for the snapshot object.

3. Assuming the FLP result, for how many processes can one implement a consensus object using any number of (atomic, wait-free) UpDownCounter objects and atomic registers?

Solution

The algorithm is not a linearizable (atomic) implementation of an UpDownCounter. To prove it, consider the following execution of the algorithm:

<table>
<thead>
<tr>
<th>Step</th>
<th>Process p1</th>
<th>Process p2</th>
<th>Process p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>invokes read_1()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>reads A[1] = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>invokes inc_2()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>returns ok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>invokes dec_3()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>returns ok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>returns −1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. The execution is not linearizable because the operations inc_2() and dec_3() are not concurrent, and so the operation read_1() should have returned either 0 or 1. However, read_1() returns −1
2. The UpDownCounter object can be easily implemented from an atomic snapshot object, which can be implemented from registers (see the lecture slides). The algorithm would be the following:

uses: $S$ – shared atomic snapshot object. Each element is initialized to 0.
uses: $c_i$ – local register at every process $p_i$, initialized to 0.

1. **upon read$_i()$ do**
   2. $A \leftarrow S$.scan$_i()$
   3. $v \leftarrow 0$
   4. for $k \leftarrow 1$ to $n$ do
   5. $v \leftarrow v + A[k]$
   6. **return** $v$

2. **upon inc$_i()$ do**
   8. $c_i \leftarrow c_i + 1$
   9. $S$.update$_i(c_i)$

3. **upon dec$_i()$ do**
   11. $c_i \leftarrow c_i - 1$
   12. $S$.update$_i(c_i)$

3. Every object that can be implemented from registers can solve consensus among only one process (in an synchronous system, in which one process can crash; see lecture notes for the relevant proof).
Problem 6  (2 points)

Consider an extension of the *k*-bounded set of Problem 3, where the set associates each element of the set with a value and does not contain duplicate elements. The semantics of the set change to:

1. *find*(*e*) which looks for element *e* in the set and if it finds *e*, the operation returns the corresponding value *v*. If element *e* is not in the set, the operation returns *fail*;
2. *delete*(*e*) which deletes element *e* from the set and returns *ok*. If element *e* is not in the set, the operation returns *fail*;
3. *insert*(*e*, *v*) which inserts the pair of element *e* and value *v* into the set and returns *ok* if the set contains less than *k* elements. If the set contains *k* elements or if *e* is already in the set, the operation does not do anything to the set and returns *fail*.

We seek a practical implementation of the extended *k*-bounded set. The *delete* and *insert* operations should rely on locks, while the *find* operation should be wait-free. All three implementations should be atomic (linearizable).

Every insertion uses a unique value (i.e., a value that is not used by any other past, concurrent, or future insertion). The set should not contain duplicate elements (even with different values).

a. Implement with C-like pseudo-code the *lock*, *unlock*, *find*, *delete*, and *insert* functions of the extended *k*-bounded set (see below).

b. Explain why the *find* operation is wait-free and provide a sketch of correctness for your implementation of *find*, *delete*, and *insert* using linearization points.

```c
//Definitions of ok, fail, and empty. You can assume that 0 cannot be a value.
#define ok 1
#define fail 0
#define empty 0 //use empty when deleting a element (replace elems[i] with empty)

//You can use compare-and-swap (CAS) and reads and writes
int CAS(int* mem, int old, int new) //sequential specification of CAS
{
    int ret = *mem;
    if (ret == old)
        *mem = new;
    return ret;
}

struct kset
{
    int* lock; //initial value is 0
    int elems[k];
    int vals[k]; //initial values are empty
}

//Implement the following 5 functions:
void lock(int* lock); //grab a lock
void unlock(int* lock); //release a lock
int find(kset* set, int elem); //returns the corresponding value or fail. Wait-free
int delete(kset* set, int elem); //returns ok or fail. Lock-based
int insert(kset* set, int elem, int val); //returns ok or fail. Lock-based
```
Solution

a. The following code implements the k-bounded set:

```c
void lock(int* lock) { while (CAS(lock, 0, 1)); }
void unlock(int* lock) { *lock = 0; }

int find(kset* set, int elem)
{  
  for (int i = 0; i < k; i++)
  {  
    int val = set->vals[i];
    if (set->elems[i] == elem && set->vals[i] == val) // snapshot of elem/val pair
      return val;
  }
  return fail;
}

int delete(kset* set, int elem)
{  
  int ret = fail;
  lock(&set->lock);
  for (int i = 0; i < k; i++)
  {  
    if (set->elems[i] == elem)
    {  
      set->elems[i] = empty;
      ret = ok;
      break;
    }
  }
  unlock(&set->lock);
  return ret;
}

int insert(kset* set, int elem, int val)
{  
  int ret = fail, empty_idx = -1;
  lock(&set->lock);
  for (int i = 0; i < k; i++)
  {  
    if (set->elems[i] == elem)
    {  
      unlock(&set->lock);
      return fail;
    }  
    else if (set->elems[i] == empty)
    {  
      empty_idx = i;
    }  
    if (empty_idx >= 0)
    {  
      set->vals[empty_idx] = val;
      set->elems[empty_idx] = elem;
      ret = ok;
    }  
  }
  unlock(&set->lock);
  return ret;
}
```
b. The \textit{find} operation is obviously wait-free as it contains only a bounded for loop (i.e., the operation will take up to $k$ steps to complete and cannot be otherwise restarted).

The \textit{insert} and \textit{delete} operations are trivially correct because they are protected by a single lock. The linearization point of these operation can be at any point after the acquisition of the lock and before releasing the lock.

The \textit{find} operation is more tricky. Lines 8-9 implement a simple snapshot algorithm (remember that values are unique). The sequence that the insert operations write the element and the value (Lines 48-49) are important for the correctness of the snapshot. If we swap these lines, the \textit{elem}/\textit{val} snapshot is incorrect. Accordingly, if Line 10 is reached and \textit{val} is returned, we know that the \textit{elem}/\textit{val} pair is a correct pair.

If the “if statement” of Line 9 fails while the first clause is true ($\text{set} - > \text{elems}[i] == \text{elem}$), the \textit{find} operation must be operating concurrently with at least one deletion on \textit{elem}, otherwise, since an insertion first writes the \textit{val} and then the \textit{elem}, the \textit{find} would have found a correct snapshot. We can thus use a linearization point for the concurrent deletion at a point before the linearization point of \textit{find}.

Similarly, we can set the linearization points of insertions and deletions for other concurrent cases. For example, if \textit{find} is at $i = 3$ and an insertion for \textit{elem} happens at $i = 2$, we might need to put the linearization point for the insertion before the linearization point of \textit{find}.