Chalmers | Göteborgs Universitet

Concurrent Programming TDA384/DIT391

Saturday, 23 October 2021

Exam supervisor: N. Piterman (piterman@chalmers.se, 073 856 49 10)

(Exam set by N. Piterman, based on the courses given in September-October 2021)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes; English dictionary.

Grading: You can score a maximum of 70 points. Exam grades are:

points in exam	Grade Chalmers	Grade GU
28–41	3	G
42–55	4	G
56–70	5	VG

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows:

points in exam + labs	Grade Chalmers	Grade GU
40–59	3	G
60–79	4	G
80–100	5	VG

The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will receive no points!
- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.
- Answer each question on a new page. Glance through the whole paper first; five questions, numbered Q1 through Q5. Do not spend more time on any question or part than justified by the points it carries.
- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1.(14p). This question is concerned with implementing the token-ring protocol in Erlang.

The token-ring protocol coordinates a given number of processes by allowing them each to work in turn. It operates by transferring a token around the ring and only the process holding the token is allowed to operate. Thus, the token-ring protocol is a mutual-exclusion protocol giving a fixed turn to each of the participating processes. Here, instead of sending each process in the token ring a function to run as a parameter, we will send each process a (fixed) number that it has to print. At the end, the `token_start` function will spawn the entire ring and cause it to start operation (see further details below). For example, calling `token_start([1,2,3,4,5])` would result in:

```
<0.106.0>   5
<0.105.0>   4
<0.104.0>   3
<0.103.0>   2
<0.102.0>   1
```

repeating forever.

Do not worry about token-rings with less than two processes.

(Part a). Implement the `token_body` function. The function should have two parameters: a number and the PID of the next in the ring. When it receives a message with the atom `{ token }`, it outputs its number (in the above example also its PID) and then sends the token to the next process in the ring. It then calls itself recursively. *(4p)*

(Part b). Implement the `token_spawn` function. The function should have two parameters: (1) a list of the numbers to spawn token bodies with and (2) the PID of the first process in the ring. Notice that the first process in the ring already exists when `token_spawn` is called. The function should spawn the required number of token bodies connected to the given PID and to each other. It should return the PID of the last process in the ring (in order to close the ring).

(4p)

(Part c). Implement the `token_start` function. `token_start` gets a list of numbers (to be printed by the processes of the token ring; as above). It should initiate the process of spawning the entire ring and putting the ring into action. You probably need a `token_one` function that handles the head of the list in a special way in order to transfer the PID of the first process to `token_spawn` and to send the first token to the ring. You do not have to worry about token rings with less than 2 processes.

(6p)

Q2 (19p). The following is a paradigm of how to implement a monitor with multiple conditions with semaphores. The class has the following member variables:

- Semaphore `mutex` initialized to capacity 1.
- Semaphore `next` initialized to capacity 0.
- A `next_count` integer counting how many are waiting on `next`.
- For every condition a semaphore and counter of how many are waiting for it.

Each function should be enclosed as follows:

```
1  wait(mutex);
2  ...
3  function body
4  ...
5  if (next_count > 0)
6     signal(next);
7  else
8     signal(mutex);
```

In order to wait on a condition `x` you should do the following:

```
9  x_count++;
10 if (next_count > 0)
11    signal(next);
12 else
13    signal(mutex);
14 wait(x_sem);
15 x_count--;
```

In order to signal on a condition `x` you should do the following:

```
16 if (x_count > 0) {
17    next_count++;
18    signal(x_sem);
19    wait(next);
20    next_count--;
21 }
```

(Part a). Explain briefly how does this implementation satisfy mutual exclusion? *(4p)*

(Part b). What is the assumption on the behaviour of wait and signal? *(4p)*

(Part c). What is the signaling policy applied by this monitor? *(4p)*

(Part d). Suppose that every function itself is finite. Does this monitor guarantee lack of starvation? *(3p)*

(Part e). What would happen if the order of lines 17 and 18 is changed? *(4p)*

Q3 (11p). We revisit the first attempt at creating a lock-free set implementation.

The implementation assumes the usage of an atomic reference.

```
1 class AtomicReference<V> {
2
3    V get();             // return current reference
4    void set(V newRef);  // set reference to newRef
5
6    // if reference == expectRef, set newRef and return true
7    // otherwise, do not change reference and return false
8    boolean compareAndSet(V expectRef, V newRef);
9 }
```

This is the suggested implementation of remove:

```
1  public boolean remove(T item) {
2     boolean done;
3     do {
4        Node<T> pred,curr = find(head,item.key());
5        if (curr.key() >= item.key()) return false; // item not in set
6        else
7           // try to remove curr by setting pred.next using CAS
8           done = pred.next().compareAndSet(pred.next(),curr.next());
9     } while (!done);
10    return true;
11 }
```

(Part a). Give a scenario where remove fails. *(3p)*

Suppose that instead, after finding the item we want to remove we call another find with the key of pred and get hold of the item before pred. We then double check that pred has not been removed:

```
1  public boolean remove(T item) {
2     boolean done;
3     do {
4        Node<T> pred,curr = find(head,item.key());
5        if (curr.key() >= item.key()) return false; // item not in set
6
7        if (pred != head) {
8           Node<T> pre-pred,pred1 = find(head,pred.key());
9           if (pred1 != pred) continue; // some change before pred
10                                       // start again
11          // check first that pre-pred is still connected
12          // to pred
13          if (!pre-pred.next().compareAndSet(pred,pred))
```

```
14             continue;
15       }
16       // try to remove curr by setting pred.next using CAS
17       done = pred.next().compareAndSet(pred.next(),curr.next());
18    } while (!done);
19    return true;
20 }
```

(Part b). Does this new implementation work correctly? *(3p)*

We now introduce a super CAS that works on two references simultaneously:

```
boolean (ref1,ref2).compareAndSet(V expectedRef1, V newRef1,
                                  V expectedRef2, V newRef2);
```

This new super CAS, if `ref1` is `expectedRef1` and `ref2` is `expectedRef2`, it sets `ref1` to `newRef1`, sets `ref2` to `newRef2`, and returns `true`. Otherwise, it does no changes and returns `false`. All this is done *atomically*!

```
1  public boolean remove(T item) {
2     boolean done;
3     do {
4        Node<T> pre-pred,pred,curr = find(head,item.key());
5        if (curr.key() >= item.key()) return false; // item not in set
6
7        if (pred == head)
8           done = pred.next().compareAndSet(pred.next(),curr.next());
9        else {
10          Node<T> pre-pred,pred1 = find(head,pred.key());
11          if (pred1 != pred) continue; // some change before pred
12                                       // start again
13
14          // try to remove curr by checking pre-pred.next and setting
15          // pred.next using the new CAS
16          done =
17             (pre-pred.next(),
18                  pred.next()).compareAndSet(pred,pred,
19                                             pred.next(),curr.next());
20       }
21    } while (!done);
22    return true;
23 }
```

(Part c). Is this final implementation correct? *(5p)*

Q4 (13p). This program is a variant of Peterson's algorithm for mutual-exclusion for two threads. It uses a compare-and-swap operation.

The label p_i can mean the command that follows p_i, or the proposition that thread `p` is at p_i, and *the next command* `p` will execute is p_i.

boolean `turn`= false; **boolean** `flaga`= false; **boolean** `flagb`= false;			
	`p`		`q`
	while(true) {		**while**(true) {
p_1	*//NCS (non-critical section)*	q_1	*//NCS (non-critical section)*
p_2:	`flaga`= true;	q_2:	`flagb`= true;
p_3:	**while**(`flagb` &&	q_3:	**while**(`flaga` &&
	!`turn`.CAS(false,true)) { };		!`turn`.CAS(false,true)) { };
p_4	*//CS (critical section)*	q_4	*//CS (critical section)*
p_5:	`turn`= `flaga`= false;	q_5:	`turn`=`flagb`= false;
	}		}

For simplicity, we ignore the locations p_1 and p_4 and similarly q_1 and q_4. Process p moves directly from p_3 to p_5 and from p_5 to p_2 and similarly for q. We treat p_5 and q_5 as the critical section. Our programming language supports lazy evaluation of conditions. So in p_3, if `flagb` is false it does not evaluate `turn`.$CAS(false, true)$. Similarly for q_3.

You are going to construct the transition table of this program. A full state is of the form $(p_i, q_j,$ `flaga`, `flagb`, `turn`$)$, where i and j range over $\{2, 3, 5\}$, and `flaga`, `flagb`, and `turn` range over *true* and $false$. Only 11 states are reachable.

Here is a partial state transition table for the program above. As mentioned, only 11 states are reachable from the initial state $(p_2, q_2, 1)$.

state		new state if p moves	new state if q moves
s1	$(2, 2, f, f, f)$	$(3, 2, t, f, f)$ = s3	$(2, 3, f, t, f)$ = s2
s2	$(2, 3, f, t, f)$		$(2, 5, f, t, f)$ = s5
s3	$(3, 2, t, f, f)$	$(5, 2, t, f, f)$ = s6	
s4	$(3, 3, t, t, f)$		
s5	$(2, 5, f, t, f)$		$(2, 2, f, f, f)$ = s1
s6	$(5, 2, t, f, f)$	$(2, 2, f, f, f)$ = s1	
s7	$(5, 3, t, t, t)$		
s8	$(3, 5, t, t, t)$		
s9	$(3, 5, t, t, f)$		
s10	$(5, 3, t, t, f)$		
s11	$(5, 5, t, t, t)$		

(Part a) Fill in the blank entries in the table. *(6p)*

(Part b) Does the protocol maintain mutual exclusion? *(2p)*

(Part c) Does the protocol avoid starvation under fair scheduling? *(5p)*

Q5 (13p). Consider the following implementation of a barrier:

```
class BarrierWithManager {

   final int NumThreads = 3;
   static Semaphore start = new Semaphore(0), done = new Semaphore(0);

   static class Manager implements Runnable {

      public void run() {
         while (true) {
            start.release(NumThreads);

            while (true) {
               try {
                  done.acquire(NumThreads);
                  break;
               } catch (InterruptedException e) { }
      }  }  }
   }

   static class Worker implements Runnable {

      public void run() {
         while (true) {
            while (true) {
               try {
                  start.acquire();
                  break;
               } catch (InterruptedException e) { }
            }
            doWork();
            done.release();
      }  }
   }

   public static void main(String[] args) {
      for (int i = 0; i<NumThreads; i++) {
         new Worker().start();
      }
      new Manager().start();
   }
}
```

(Part a). Does this barrier work correctly? If no, give an example. If yes, explain how it is maintained. Yes/No answers with no explanation will be rejected. *(7p)*

(Part b). Replace the two semaphores with binary semaphores. You will have to increase the number of semaphores. You cannot use any other shared variables *(6p)*