Algorithmic Verification of Programs (Final Examination)

The subject of the exam contains two parts : Exo 1-2 and Exo 3-6. Write solutions for each part on a different exam sheet. The exam duration is 3 hours.

Exercice 1 : Multithreaded programs

Consider concurrent programs with dynamic creation of processes. These processes have local variables and a shared memory. In addition, they can communicate by message passing, either by rendez-vous (binary synchronous communication), or by broadcast (from one process to all processes).
We assume that all the processes are identical. They have the same set of local variables and the same set operations. The valuations of the local variables correspond to local states. Valuations of the shared variables are global states. We assume that the data domain is finite, which means that the set of local states is finite, and the same holds for the set of global states.
Then, the operations that a process can perform at each point are :

1. $g := g'$: update of the shared memory (change of the global state)
2. $\ell := \ell'$: update of its own local state
3. $async(\ell)$: spawn a new process at some initial local state : the new process is added to the set of processes
4. $send(a)$: sends a message a (we assume that there is a finite number of messages). This action is *blocking* and is executed only in a synchronous way with another process who can perfrom a $receive(a)$ action.
5. $receive(a)$: receives a message a (we assume that there is a finite number of messages). This action is *blocking* and is executed only in a synchronous way with another process who can perfrom a $send(a)$ action.
6. $broadcast(a)$: sends a message a to all processes who are ready to get it. This action is *not blocking*. When it is executed by a process at some program configuration, every process, if any, that can perform at this configuration an action $get(a)$ must execute this action to get the message a. Notice that it is possible that at some configuration no process gets the broadcasted message (if no process is ready to get the message at the moment it is broadcasted).

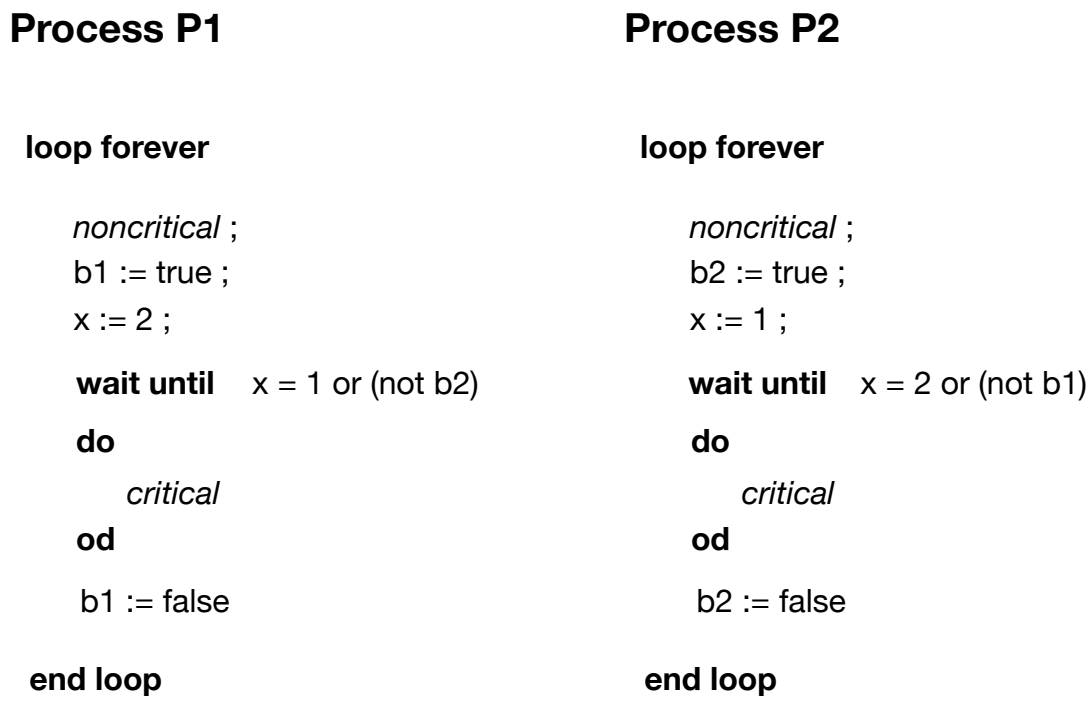

FIGURE 1 – Peterson algorithm

7. $get(a)$: gets a broadcasted message. This action is *blocking*. It is executed only at the moment of a broadcast of a by some process.

Question 1 : Provide an operational semantics to these programs, i.e., define a state machine that captures the computations of such program. Indication : use an extension of vector addition systems with states.

Question 2 : Prove that state reachability for these programs is decidable.

Exercice 2 : Weak Memory Models

Figure 1 shows a program implementing the Peterson mutual exclusion protocol. The program has two parallel processes P_1 and P_2, sharing the two boolean variables b_1 and b_2, and the variable x ranging over the set $\{1, 2\}$.

Question 1 : Verify that the program, when executed under the SC memory model, satisfies indeed mutual exclusion, which means that the two processes are never in the critical section in the same time.

Question 2 : Show that under the TSO memory model this program does not satisfy mutual exclusion, which means that there are behaviors of the program where both of the processes are in the critical section.

Question 3 : Add an irreducible set of memory fences to the program ensuring mutual exclusion. (Irreducible means that removing any fence from the set leads to a program that does not satisfy mutual exclusion, i.e., a program that have behaviors violating this property).

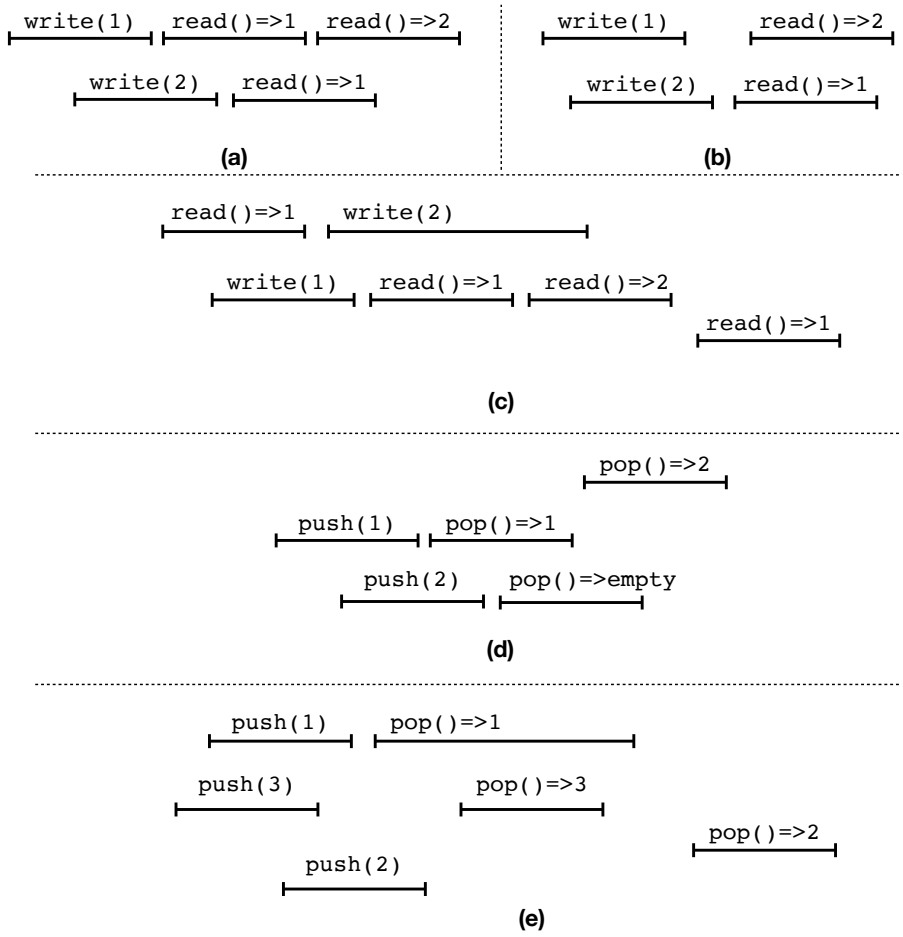

FIGURE 2 – Histories for Exercice 3

Exercice 3 : Consistency criteria

1. Figure 2(a)-(c) picture three histories of a concurrent *register* with methods `write(v)` writing the value `v` to the register and `read()` returning the current value of the register. Also, Figure 2(d)-(e) picture two histories of a concurrent stack. For each of these histories, give the correctness criteria they satisfy among quiescent consistency, sequential consistency, and linearizability. Provide a short justification for your answer.

2. Give an example of a history of a *concurrent queue* that is *linearizable but not sequentially consistent*, *sequentially consistent but not quiescent consistent*, and *quiescent consistent but not sequentially consistent*. Write a short justification in each case.

Exercice 4 : Quiescent consistency

Consider a memory object that encompasses two register components x and y. Therefore, the memory object has methods $x.write(v)$ and $y.write(v)$ for

writing some value v to x or y, and $x.read()$ and $y.read()$ that return the value of x and y, respectively.

1. Is it true that if both registers are quiescently consistent, then so is the memory ? More precisely, is it true that for every history/execution of the memory object, if each projection on operations of x and y, respectively, is quiescently consistent, then so is the entire history/execution ?

2. Does the converse hold ? If the memory is quiescently consistent, are the individual registers quiescently consistent ?

For both questions, outline a proof, or give a counterexample.

Exercice 5 : Stack object

Consider the problem of implementing a concurrent stack using an array indexed by a `top` counter, initially zero. To push an item, increment `top` to reserve an array entry (e.g., the first increment reserves position 0), and then store the item in that entry. To pop an item, decrement `top` if it is strictly greater than 0, and return the item at the previous `top` index. If `top` is 0 then return *empty_stack*. Assume that increment and decrements of `top` are done atomically in a single indivisible step.
Is this implementation linearizable ? Outline a proof, or give a counterexample. In case it is not linearizable, propose a fix that makes it linearizable and that does not involve locks.

Exercice 6 : Another Stack object

The following code presents an implementation of a concurrent stack (each statement is executed in a single atomic step) :

```
void push(int x) {
  i = range++;
  items[i] = x;
}
```

```
int pop() {
  t = range - 1;
  for i = t downto 0 {
    x = swap(items[i],null);
    if ( x != null )
      return x
  }
  return empty;
}
```

This stack stores the elements into an infinite array `items`, a shared variable `range` keeping the index of the first unused position in `items` (initially, `range`

is 0). The push method stores the input value in the array while also incrementing `range`. The pop method first reads `range` and then traverses the array backwards starting from this position, until it finds a position storing a non-null element (array cells can be nullified by concurrent pop invocations). It atomically reads this element and stores null in its place (represented by `swap(items[i],null)`). If the pop reaches the bottom of the array without finding non-null cells, then it returns that the stack is empty.
Does this implementation admit fixed linearization points (i.e., the linearization point of every push or pop in every execution corresponds to executing some fixed statement in the code) ? Justify your answer.
Is this implementation linearizable ? Justify your answer.