departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrency and Parallelism — 2nd Test — 2015-12-15 (Duration: 1h30m)

Number: _______________ Name: ____

Question	1	2 a)	2 b)	2 c)	3	4	5	6	7 a)	7 b i)	7 b ii)	7 b iii)	7 b iv)	7 b v)	8	9
Points	*0.5*	*0.5*	*1*	*1*	*1.5*	*2*	*2*	*0.5*	*2*	*1*	*1*	*1*	*1*	*1*	*2*	*2*

This test is written in English, but you should answer in Portuguese if possible, and in English otherwise.

1) Two or more concurrent processes may need to synchronize between themselves because they interact by competing or collaborating. Illustrate a basic mechanism for each of these interaction schemes and for each interaction scheme describe a situation where the mechanism is useful.

Competition: _________________ Situation: ____

Collaboration: _________________ Situation: ____

2) Assume that the initial value for 'x=0' and that methods M1 and M2 below execute in parallel.

M1: for (i=0; i<3; i++) { x=x+1; } M2: for (i=0; i<4; i++) { x=x+2; }

a) List **the smallest** and **the largest** possible values for 'x', knowing that the methods M1 and M2 are executed atomically.

Smallest: __________ Largest: __________

b) List the **three smallest** and **three largest** possible values for 'x', knowing that the instructions 'x=x+N' **is atomic**.

Smallest: __________ __________ __________ Largest: __________ __________ __________

c) List the **three smallest** and **three largest** possible values for 'x', knowing that the instructions 'x=x+N' **is not atomic**.

Smallest: __________ __________ __________ Largest: __________ __________ __________

3) Assume that the variables 'x' and 'y' have both the initial value '0', and that the methods M1, M2, and M3, are executed in parallel.

M1: x=x+1; M2: y=y+1; M3: y=3; x=y+1

Knowing that the execution keeps the relative order of the instructions (i.e. in M3, 'y=3' is always executed before 'x=y+1') and that the assignment is executed atomically, list all the final possible values for the variables 'x', 'y', and for the pairs '(x,y)' at the end of the execution of M1, M2 and M3.

Note 1: If the number of slots in ***larger*** *than necessary, just leave the remaining slots blank. If the number of slots in* ***smaller*** *than necessary, just fill it up to the available of available slots.*

Note 2: Showing wrong values for X, Y, or pair (X,Y) will reduce your grade in this question.

X: _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____

Y: _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____

(X,Y): (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__)

(__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__)

(__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__) (__,__)

4) The pseudo-code (incomplete) presented below implements a producer-consumer pattern by using a shared array *BUF* of size *N*. Knowing that synchronization is based in semaphores (*S1* and *S2*), complete the missing parts in the pseudo-code.

```
operation B.produce(v) is
  S1. ____________________;
  BUF[in].put(v);
  ________________________;
  S2. ____________________;
    return ();
end operation;
```

```
operation B.consume() is
  ________________________;
  r = BUF[out].____________;
  ________________________;
  ________________________;
    return (r);
end operation;
```

5) Mark each of the following statements as *true* (T or V) or false (F). *(Note: wrongly marking a statement will reduce your grade in this question).*

[ ] Obstruction free programs always exhibit progress of the program as a whole.
[ ] Lock-free programs may suffer from starvation.
[ ] Obstruction freedom provides more guaranties than lock freedom.
[ ] Lock freedom provides more guaranties than wait freedom.
[ ] Obstruction-free programs may deadlock.
[ ] Wait-free programs may deadlock.
[ ] Under low contention, lock-free programs behave as wait-free programs.
[ ] Wait-free programs may suffer from starvation.
[ ] Lock-free programs may deadlock.
[ ] Obstruction-free programs may suffer from starvation.
[ ] Under high contention, lock-free programs behave as obstruction-free programs.
[ ] All waif-free programs are also lock-free programs.
[ ] Obstruction-free programs always generate linearizable executions.
[ ] Wait-free programs always exhibit progress of the program as a whole.
[ ] All linearizable programs are sequential consistent.
[ ] Linearizability is a local property.
[ ] Sequential consistency is a global property.
[ ] The composition of linearized executions is linearizable.
[ ] The composition of linearized executions is linearizable.
[ ] All sequential consistent executions are also linearizable.

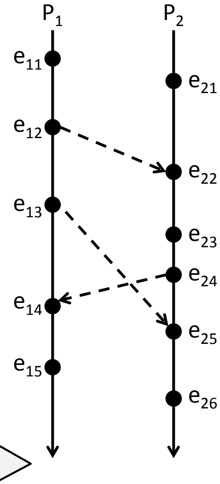
6) Consider a program with multiple threads. In which of the following situations **at least one thread may be progressing** in its intended computation? Mark each case with Progress (P) or No Progress (N).

[ ] deadlock [ ] livelock [ ] starvation [ ] busy-waiting

[ ] all threads simultaneously execute a *lock(X)* over the same lock variable *X*

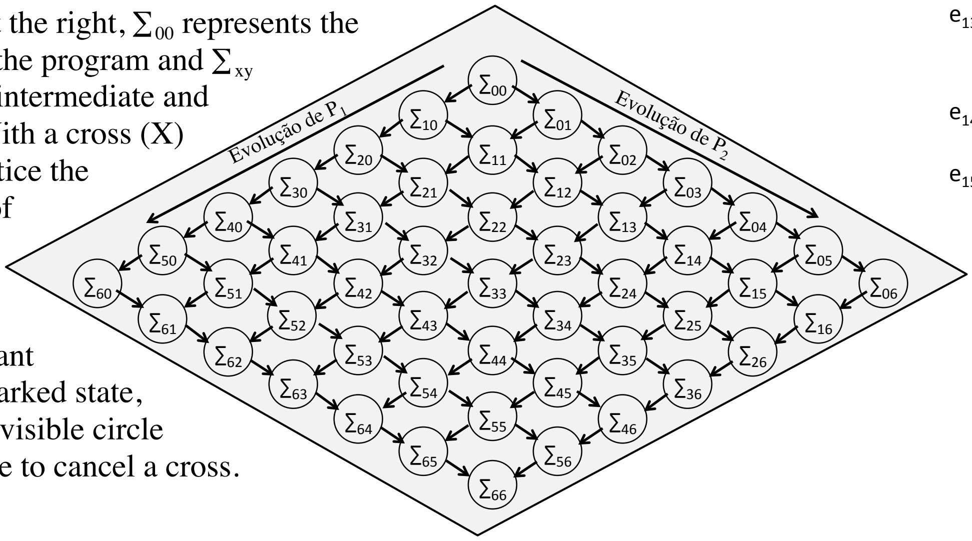
[ ] all threads simultaneously execute a *compareAndSwap(X,v)* over the same variable *X*

7) Consider that the processes P1 and P2 belong to the same program. Processes P1 and P2 interact by exchanging messages as shown in the figure at the right side of this question. The full line arrows denote time and the dashed line arrows denote communication events.



a) In the lattice at the right, Σ_{00} represents the initial state of the program and Σ_{xy} represents the intermediate and final states. With a cross (X) mark in the lattice the invalid states of this program.

If you make a mistake and want to *unmark* a marked state, draw a clearly visible circle around the state to cancel a cross.



b) Knowing that '$i(e_{nm})$' and '$r(e_{nm})$' correspond to the invocation and reply of na event 'e_{nm}' that occurred during the execution of the program, show:

i) A sequential history that is **complete and consistent**. If there is none, write "NONE" or "NÃO HÁ".

ii) Another sequential history that is **complete and consistent** and different form i), in which at least threeevents are in different positions than in i). If there is none, write "NONE" or "NÃO HÁ".

iii) Explain when two sequential histories are equivalent. If they cannot be equivalent, write "NONE" or "NÃO HÁ".

iv) A concurrent history that **is equivalent** to the history i) above. If there is none, write "NONE" or "NÃO HÁ".

v) A concurrent history that **is not equivalent** to the history i) above. If there is none, write "NONE" or "NÃO HÁ".

8) Consider the code shown at the right that presents the definition of the class *Node* and part of the class *LockFreeStack*, in particular its private fields and the code relevant to the implementation of the method *push()*. The method *backoff.backoff()* does a random pause.

The class *AtomicReference* has two relevant methods, namely *get()* that returns the reference and *compareAndSet(old, new)* with the usual semantics.

Fill the fields below with the code that should be present in lines 8, 9 and 10.

```
1   public class LockFreeStack<T> {
2     AtomicReference<Node> top = new AtomicReference<Node>(null);
3     static final int MIN_DELAY = ...;
4     static final int MAX_DELAY = ...;
5     Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
6
7     protected boolean tryPush(Node node){
8       Node oldTop =
9       node.next =
10      return(                                  ));
11    }
12    public void push(T value) {
13      Node node = new Node(value);
14      while (true) {
15        if (tryPush(node)) {
16          return;
17        } else {
18          backoff.backoff();
19        }
20      }
21    }
```

```
public class Node {
    public T value;
    public Node next;
    public Node(T value)
      value = value;
      next = null;
    }
}
```

8:
9:
10:

9) Considere um sistema de memória transacional, em que duas transações estão a executar concorrentemente. Considere ainda as seguintes transações:

T_1:	R(50, a);	R(100, b);	W(120, 8);	R(110, c)
T_2:	W(50, 3);	R(130, d);	W(110,8);	R(120, e)
T_3:	R(50, f);	R(100, g);	R(120, h)	

Knowing that *Ti(N) / Tj(M)* mean that transactions *Ti* and *Tj* execute in parallel starting respectively in time 'i' and 'j', fill the table below.

T_i // T_j	Read Set T_i	Read Set T_j	Write Set T_i	Write Set T_j	T_i commit or abort? Why?	T_j commit or abort? Why?
$T_1(5)$ // $T_2(7)$						
$T_1(8)$ // $T_3(4)$						
$T_2(2)$ // $T_3(8)$						

END!!!