RETAKING EXAM 1st BLOCK **Concurrency and Distributed Systems**
Units 1 to 5 - Practices 1, 2 and 3 **Date: 15th June 2018**

This exam has a maximum duration of 90 minutes.

This exam has a maximum score of **10 points**, equivalent to **3.5 points** of the final grade for the course. It contains questions of theoretical units and lab sessions. Indicate, for each of the following **52 statements**, if they are true (T) or false (F). **Each answer is worth: right= 10/52, wrong= -10/52, empty=0.**

Important: **first 3 errors do not penalize**, so they will be equivalent to an empty answer. From the 4th error (inclusive), the decrement for wrong answers will be applied.

THEORY QUESTIONS

Regarding the usage of threads in Java:

1. The use of the synchronized label on a method converts that method into an atomic action with respect to the other synchronized methods of that object.	T
2. New threads can only be created in the code of the 'main' function.	F
3. The code that executes any thread must be implemented in a class that extends from the Thread class.	F
4. When we apply *interrupt* on a thread suspended in *wait*(), this thread goes to the ready-to-run state.	T

In relation to the definition of concurrent programming and the concept of critical section:

5. All concurrent programs are those that launch different activities with guarantees of simultaneous execution (real parallelism).	F
6. The concurrent execution of two or more Critical Sections that work on the same variables can lead to race conditions.	T

Regarding the Lock concept:

7. The protection of a Critical Section through a lock is a mechanism to prevent interferences in concurrent programs where objects are shared.	T
8. The input protocol to the Critical Section of a shared object blocks a thread that executes it if this Critical Section is already in use by another thread.	T
9. To achieve maximum concurrence, it is advisable to define a single lock to regulate access to all shared objects.	F
10. When a thread tries to open a lock that is already open, an exception is generated.	F

Regarding concurrent programming in Java:

11. In Java, every object has a lock, which can be used to prevent interferences between threads.	T
12. Every method with the synchronized label has its own implicit lock that closes as the first instruction and opens as the last instruction.	F
13. If a method of an object labelled with synchronized invokes another method of that same object also labelled with synchronized, it causes the thread to wait for itself (deadlock).	F

In a canal navigation system, there are floodgates to manage the gradient differences. Each floodgate has a limited capacity to N ships. It is necessary that the ships pass in groups of N, so that it is necessary to wait until a group is formed, then they will all go to the floodgate and immediately the passage to the following ships will be blocked until another group is formed. The management of the floodgate includes towing the ships from the entrance gate, filling / emptying the floodgate, towing the ships to the canal behind the exit gate and refilling / emptying the floodgate. In order to control the entrance of groups of ships into the floodgate, it has been decided to use the following program:

```
public class Ship extends Thread {
 private int id;
 private CyclicBarrier barrier;
 public Ship(int id, CyclicBarrier barrier) {
  this.id = id; this.barrier = barrier;
  setName(String.valueOf (id));
 }
 public void run() {
  try {Thread.sleep((long)(Math.random()*1000 ));
  } catch (InterruptedException e) {};
  System.out.println(id + ": arrives: " + new Date());
  try {barrier.await();
  } catch (BrokenBarrierException e1) {
  } catch (InterruptedException e2) {};
  System.out.println(id + ": continues sailing: " + new Date());
 }
}

public static void main(String[] args) {
 final int N=3;
 CyclicBarrier b=new CyclicBarrier(N,new Runnable() {
    public void run(){ System.out.println("management of the floodgate");
 }});
 for (int  i=1 ; i < 100 ; i++) {
  Ship ship =new Ship(i, b);
  ship.start();
}
}
```

14. In this solution N threads with name are executed, apart from the main one.	F
15. The program will not finish, because some thread will stay blocked without passing to the floodgate.	F
16. In this solution ships pass in groups of 3 to the floodgate.	T
17. Ships arrive to the floodgate entrance gate in the order in which they have been created.	F
18. For the solution to be correct, the barrier should be initialized to 0.	F
19. The program does not work correctly since each ship waits at a different barrier.	F
20. Before opening the barrier, the "management of the floodgate" message is displayed.	T
21. Ships access the floodgate in mutual exclusion.	F

For a thread X to wait until other N threads of the same class have executed a statement B within its code (being N> 1):

22. You can use a *CountDownLatch cl* initialized to N, calling thread X for *cl.await*() and the other N threads each to *cl.countdown*() after statement B.	T
23.You can use a *CyclicBarrier cb* initialized to N+1, calling thread X for *cb.await*() and the other N threads each to *cb.await*() after statement B.	T
24.You can use a *Semaphore s* initialized to 0, calling thread X for *s.acquire()* N times and the other N threads each to *s.release*() after statement B.	T

When executing the following Java code:

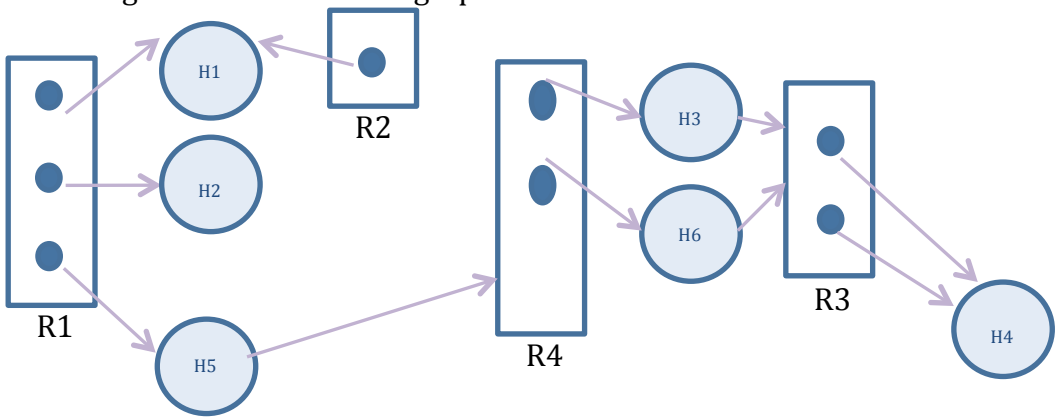
```
class Exam extends Thread {
    private int i;
    public Exam(int data) {i= data;}
    public void run() {
       for (int j=0; j<i; j++)
           if ((j%2)==0) new Exam(j).start();
       System.out.println("i:"+ i);
    }

    static public void main(String args[]){
       new Exam(6).start();
    }
}
```

25. We will see on the screen as many lines as threads are created, apart from the main one.	T
26. We will see four lines that contain "i:0".	T
27. We will see at least one line that contains "i:1".	F
28. We will see two lines that contain "i:2".	T

Regarding Coffman conditions:

29. They are necessary for the critical section to be executed in a mutually exclusive way.	F
30. If there is a deadlock, then all Coffman conditions are met.	T
31. If any of them does not hold, then there will not be a deadlock.	T
32. One of the Coffman's conditions is to request all the resources required initially, and if all the requests cannot be met then the thread releases all the required resources and gets blocked.	F

Given the following resource allocation graph:



33. The system presents at least one deadlock.	F
34. If process H4 requests for an instance of resource R4, there will be a deadlock.	T
35. If process H2 requests for the resource R2 there will be a deadlock.	F
36.If both process H1 and process H2 request for an instance of R4 each of them, there will be a deadlock.	F
37. This type of resource allocation graphs is used in the deadlock prevention strategy.	F

The following code aims to implement a monitor to control the access of readers (concurrent) and writers (exclusively) on a shared resource. Writers will use *pre-write*() before accessing and *post-write*() after having accessed. Readers will use *pre-read*() before accessing and *post-read*() after accessing:

```
Monitor controlReadersWriters {
  int readers, writers;
  condition  occupied;
  public controlReadersWriters () {
   readers=writers=0;}

  entry void pre-write() {
   if  (writers > 0 || readers > 0 )
      occupied.wait();
  writers = writers+1;
  }
  entry void post-write() {
   writers = writers-1;
   occupied.notify();
  }
```

```
  entry void pre-read() {
    if (writers>0)
       occupied.wait();
    readers = readers+1;
   }

  entry void post-read() {
   readers = readers-1;
   if (readers == 0) occupied.notify();
  }
}
```

38.If the monitor follows the Hoare variant, it provides access in mutual exclusion between readers and writers, and also for writers between them.	T
39.If the monitor follows the Hoare variant, it may be the case that some readers remain waiting in the "occupied" condition indefinitely, knowing that the condition queue is of FIFO type.	F
40. A single condition variable has been used because this representation of the monitors does not allow instantiation of more than one condition variable.	F
41. If the monitor follows the Lampson and Redell variant, the behaviour of the code complies with the access requirements indicated in the statement.	F

Regarding the monitor concept and its variants:

42. A monitor that follows the Lampson and Redell variant never suspends a thread in the *notify*() invocation.	T
43. A monitor is a high level synchronization mechanism integrated in some concurrent programming languages.	T
44. A monitor that follows the Hoare model suspends the thread that invokes *notify*(), and this thread is suspended in a special queue.	T

LAB PRACTICES

Regarding practice 1 "Shared use of a pool", where we have the following cases:

Pool0	Free access to the pool (no rules)
Pool1	Kids cannot swim alone (instructor must be with them in the pool)
Pool2	Rules of Pool1 and moreover there is a maximum of kids per instructor.
Pool3	Rules of Pool2 and moreover there is a maximum pool capacity.
Pool4	Rules of Pool3 and moreover if there are instructors waiting to exit the pool, kids cannot enter into the pool.

1. In Pool2, when a child leaves the pool by using *kidRests*(), it is necessary to invoke *notifyAll*() only if there is an instructor in the pool.	F
2. In Pool4, when a child leaves the pool by using *kidRests*(), a *notifyAll*() must always be invoked.	T

Regarding practice 2 "Dining philosophers", where we have the following versions:

Version 1	Asymmetry (all but last)
Version 2	Asymmetry (even/odd)
Version 3	Both or none
Version 4	Capacity of the table

3. In the problem of the dining philosophers, the table represents the monitor, the forks represent the resources and the philosophers represent the threads.	T
4. Interlocks will never occur in the solution for Version 2 (even/odd) even though the delay (N) for picking up the forks is very high.	T
5. In the original program, the waiting time from the moment a philosopher takes his first fork until he takes the second one influences the probability of deadlock.	T
6. In the solution based on taking both forks or none, the Coffman condition of "hold and wait" is broken.	T

Regarding practice 3 "The ants problem", where we have the following versions:

Activity 1	ReentrantLock with one condition variable related to the territory
Activity 2	ReentrantLock with a condition variable for each cell of the territory

7. It is necessary to use some synchronization mechanism as long as at least two ants are placed in the territory.	T
8. In the ants problem, the territory represents the monitor, the cells represent the resources and the ants represent the threads / processes that use these resources.	T