This exam has a maximum duration of 2 hours.

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 **58 statements**, if they are true (T) or false (F). **Each answer is worth: right= 10/58, wrong= -10/58, empty=0.**

THEORY QUESTIONS

Regarding the differences between concurrent programming and sequential programming:

1. Sequential programs can generate race conditions, as concurrent programs do.	F
2. A process with several threads of execution, executed with a processor with a single core that offers multiprogramming, is an example of a concurrent application.	T

Regarding concurrent programming in Java:

3. If a method is labeled with `synchronized` and it is invoked from another method of the same instance also labeled with `synchronized`, this causes the thread to wait for itself (deadlock).	F

About the problem of the critical section:

4. A correct solution to the problem of the critical section must comply with the properties of mutual exclusion, hold and wait, no preemption and circular wait.	F
5. The problem of the critical section appears when several threads share an immutable (constant) object that all the threads need to access simultaneously.	F
6. The problem of the critical section is solved using correctly synchronization mechanisms, such as monitors, semaphores and locks.	T
7. If we use a single lock to protect the critical sections of a certain object, a maximum of one thread will be allowed executing these critical sections	T
8. In a critical section protected by a `lock`, the output protocol releases the `lock`, allowing all threads blocked in the entry to access the object.	F

On the concept of monitor and its variants:

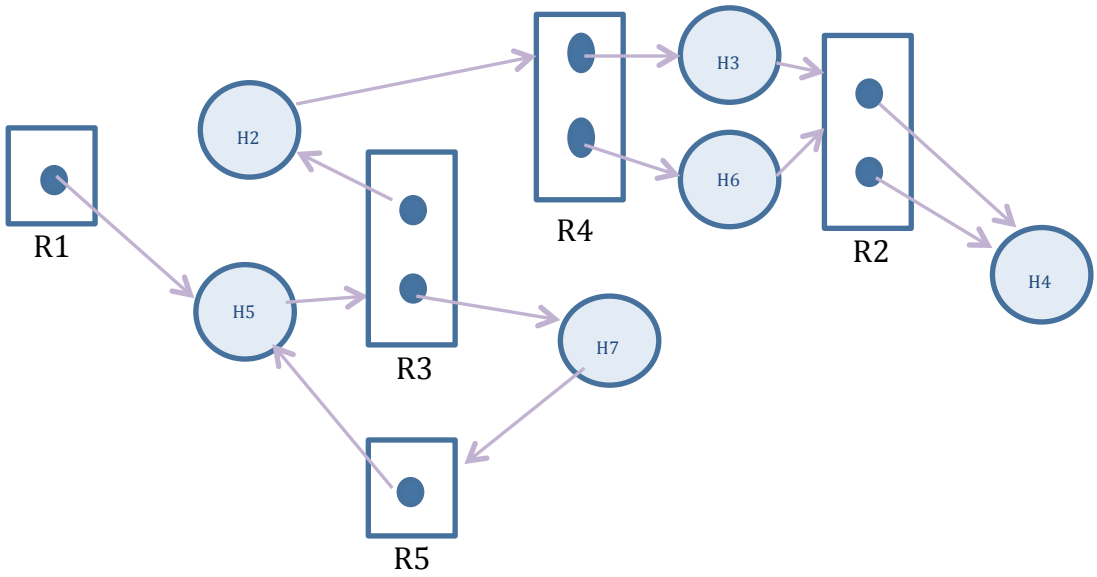
9. Assuming that there is some thread suspended in the condition variable `c`, a monitor that follows the Hoare model suspends the thread that invoked `c.notify()` and activates one of the threads that called before `c.wait()`.	T
10. The Java language provides "Lampson/Redell" monitors by default.	T
11. The Lampson-Redell monitor guarantees that after a `notify()` operation, the reactivated thread finds the status of the monitor exactly as it was when the `notify` was executed.	F

Given the following Java program:

```
public class GreatBoss extends Thread {
    protected int workers = 0;
    public GreatBoss(int workers) {this.workers = workers;}

    public void myAction(){
      for (int i=0; i<= workers; i++){
        System.out.println("Preparing worker " + i + " for: " +
             Thread.currentThread().getName());
        new Thread(new Runnable(){
            public void run(){
               System.out.println("Task finished"); }
        });
      }}

    public void run() {
       myAction();
       try{Thread.sleep(workers*1000);}
          catch(InterruptedException ie){ie.printStackTrace(); };
       System.out.println(Thread.currentThread().getName() +
            " done");

      }

    public static void main(String[] argv) {
       for (int i=0; i<10; i++){
           GreatBoss boss = new GreatBoss(i);
           if (i<5) {  boss.setName("Chief" + i);
                       boss.start();
                    }
           }
      }
  }
```

12. When executing it, we will see on screen at least one line with the sentence: `Task finished`	F
13. When executing it, we will see on the screen, among other things, the line: `Preparing worker 4 for: Chief4`	T
14. With this code a total of 10 threads will be created, apart from `main`.	F
15. When executing it, we will see on the screen, among other things, 5 lines with the word `done`	T
16. It is required to label `myAction`() method with the `synchronized` qualifier to avoid race conditions.	F

Given the following resource allocation graph (RAG):



17. Processes H5 and H7 are in a deadlock.	F
18. The system has at least one safe sequence.	T
19. If H4 process requests an instance of any of the system resources (without releasing the two instances that it has already assigned from the R2 resource), a deadlock will occur and no process will be able to finish.	T
20. In this GAR all Coffman's conditions hold.	T

About Coffman conditions and deadlock situations:

21. Coffman conditions allow designing systems that comply with all of them, in order to guarantee that deadlocks will not occur.	F
22. One of the Coffman conditions is to request all the resources required initially, so that the threads are blocked (hold and wait) if there is a conflict on the requests.	F
23. Deadlock situations can be prevented by allocating resources in a way that a directed cycle is never generated.	T
24. One of the Coffman conditions is that the assigned resources can be expropriated.	F

MODEL A

A jewellery workshop wants to assemble pearl necklaces only white, only blue or combined white and blue pearls. For this, it has 5 managers and 2 baskets, one for each pearl colour, with limited capacity. To organize the production it is decided that one manager will be the supplier of white pearls, another manager will supply the blue pearls and the rest of the managers will be allocated to the assembly of each type of necklace. The Monitor `ManagerOfPearls`, manages the number of pearls stored in the baskets. There is a thread associated with each manager. The managers responsible for supplying pearls are responsible for obtaining a pearl and store it in the basket using the methods `AddWhite` or `AddBlue`. The rest of the managers ask the monitor for the number of pearls of each colour they need to assemble the collar using the `RequestOrder` method. Analyse the following proposal for the `ManagerOfPearls` monitor.

```
public class ManagerOfPearls {

      final static private int NMaxWhites  = 50;
      final static private int NMaxBlues = 50;

      private int NWhites  = 0;
      private int NBlues = 0;

      private boolean OrderInProgress = false;

      public synchronized void AddWhite() {
            NWhites = NWhites++;
            notifyAll();
            while (NWhites == NMaxWhites)
              try {wait();} catch (InterruptedException e){};
      }

      public synchronized void AddBlue() {
            NBlues = NBlues++;
            notifyAll();
            while (NBlues == NMaxBlues)
              try {wait();} catch (InterruptedException e){};
      }

      public synchronized void RequestOrder(int RequestWhites,
         int RequestBlues) {

            while (OrderInProgress)
               try {wait();}
                    catch (InterruptedException e){};

            OrderInProgress = true;

            while (RequestWhites > NWhites || RequestBlues > NBlues)
              try {wait();} catch (InterruptedException e){};

            NWhites  = NWhites  - RequestWhites;
            NBlues = NBlues - RequestBlues;
            OrderInProgress = false;
            notifyAll();
      }
}
```

25. In this solution you can exceed the maximum number of white or blue pearls in the baskets, since the counters are increased before checking if they fit.	F
26. The `OrderInProgress` attribute is necessary to provide mutual exclusion in the access to the `RequestOrder` method.	F
27. The solution is not correct because the invocation to the `notifyAll` method in `AddWhite` and `AddBlue` should be the last instruction in both methods.	F
28. It is not necessary to include the `synchronized` qualifier in the methods `AddWhite` and `AddBlue`, since there is only one thread that adds white pieces and a thread that adds blue pieces.	F
29. The `OrderInProgress` attribute is used to ensure that when an order P1 is waiting for the requested pearls to be completed, new orders will not be served until P1 is completed.	T
30. The solution proposed for the monitor is correct, and synchronizes appropriately according to the problem formulation, the pearl suppliers and the management of the orders made by the assemblers.	T

A hard real-time system consists of 5 independent tasks whose characteristics are described in the following table:

Task	Computation Time	Period (T)	Deadline
T1	3	15	8
T2	4	20	13
T3	6	30	18
T4	5	40	32
T5	6	50	50

Assuming that the system is scheduled using fixed-priority pre-emptive scheduling, with task priority assignment inverse to their deadline, that is, the task with the shortest deadline has the highest priority...

31. … the response time for task T4, R4 is 30.	F
32. … the hyper period is 600.	T
33. ... if tasks always use the computation time indicated in the table in all their activations, then at time 605 the task T2 will be running on the CPU.	T
34. … the response time for task T2, R2 is 7.	T
35. … the response time of task T3 is lower than or equal to its deadline.	T
36. … the system is schedulable.	T

Assuming now that the tasks are not independent, and that they use three semaphores M1, M2 and M3 to synchronize the access to their critical sections (CS) whose characteristics are described in the following table, and that these semaphores use the Immediate Priority Ceiling Protocol...

Task	Semaphore	Duration of the CS
T1	M1	1
T2	M1	2
T2	M2	4
T4	M2	3
T4	M3	2
T5	M3	1

37. ... the blocking factor of task T1, B1 is 3.	F
38. ... the blocking factor of task T3, B3 is 0.	F
39. ... the priority ceiling of semaphore M2 is equal to the priority of T4.	F
40. ... the response time of task T2, R2 is 10.	T
41. ... the response time of task T3 is lower than or equal to its deadline.	F
42. ... the response time of task T5 is equal to the response time it will have if all tasks were independent (i.e. if they do not need to synchronize) .	T
43. ... the system is schedulable.	F

Sol:

Task	Computation Time	Period	Deadline	Ri without Bi	Bi	Ri with Bi
T1	3	15	8	3	2	5
T2	4	20	13	7	3	10
T3	6	30	18	13	3	19
T4	5	40	32	25	1	26
T5	6	50	50	40	0	40

Regarding the use of `java.util.concurrent` library tools:

44. The `CountDownLatch` objects guarantee that the `await()` method will suspend the invoking thread, as long as the barrier is not open.	T
45. The `CountDownLatch` objects are created with an integer value that indicates the number of threads that must remain waiting in the barrier to open it.	F

46. Given 6 threads of class A, 2 threads of class B and 4 threads of class C that share the same object "c" of type `CyclicBarrier` initialized to 4, knowing that all threads execute `c.await()`, the threads of class B may remain suspended indefinitely.	F
47. Given 6 threads of class A, 2 threads of class B and 4 threads of class C that share the same objects `sem1` and `sem2` of type `Semaphore` initialized both to 0, knowing that each thread of class A executes `sem1.acquire()`, threads B execute `sem2.acquire()` and threads C execute `sem1.release(); sem2.release();` some thread of class A may remain suspended indefinitely.	T
48. Each lock of type `ReentrantLock` can create one or several condition variables, passing the lock reference to the constructor of the `Condition` object.	F
49. You can implement a correct solution for the problem of the critical section with a `ReentrantLock rl`, using the `rl.trylock()` as input protocol and the `rl.lock()` as output protocol.	F
50. If several threads use the same `AtomicLong` object, the methods that this object offers must be protected with the `synchronized` label, in order to avoid race conditions.	F

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	There is a maximum of kids per instructor
Pool3	There is a maximum pool capacity
Pool4	If there are instructors waiting to exit the pool, kids cannot enter into the pool

51. In Pool1 it is not necessary to suspend any thread, since there is no conditional synchronization.	F
52. To implement the Pool3, it is necessary to use two monitors.	F
53. In Pool0 it is not necessary to put `synchronized` in the methods of this class, because the pool is an object without state.	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

54. In the solution based on limiting the number of philosophers on the table, the absence of deadlocks is guaranteed regardless of the order in which the philosophers take the chopsticks.	T

We want to implement the strategy of taking both chopsticks or none for the "Dining Philosophers" lab practice. Given the following implementation of `takeLR`, in the `BothorNoneTable` class:

```
public synchronized void takeLR(int id) throws InterruptedException{
      while (!state.rightFree(id)) { wait();}
      while (!state.leftFree(id)) {
          state.wtakeLR(id);
          wait();
      }
      state.takeLR(id);
}
```

55. This code is correct.	F

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

Activity 0	Java intrinsic lock
Activity 1	ReentrantLock with one condition variable related to the territory
Activity 2	ReentrantLock with a condition variable for each cell of the territory

56. Using the same sizes of territory and number of ants, if we compare the results between using the intrinsic lock of Java (Activity 0) and the use of `ReentrantLock` with a variable condition for each cell of the territory (Activity 2), on average there is a significantly greater number of "unnecessary" reactivations of threads in Activity 0 with respect to Activity 2.	T
57. In activity 2 of the practice, where a condition variable is associated for each cell of the territory, we need to create an array of objects of type `ReentrantLock`, defining a lock and a condition variable for each element of the array.	F
58. In the initial code supplied at the lab practice, a `CyclicBarrier` is used that causes all the ants to start moving through the territory at the same time, once they have all being placed in the territory in their respective cells.	F