ACTIVITIES UNIT 3

QUESTION 1. Given the following statements, modify them accordingly to make them **true**. Note that some statements might be already true.

1. A monitor is a high-level synchronization mechanism integrated in some concurrent programming languages.
2. The concept of monitor avoids the requirement for threads to share memory.
3. A monitor is a class that solves mutual exclusion and conditional synchronization
4. A monitor always has an entry queue where those threads that want to use the monitor when it is being used by another thread have to wait.
5. Race conditions may occur within a monitor. When this happens, the thread that executes code inside the monitor must execute *c.wait()* to leave it.
6. A monitor that follows the Brinch-Hansen model requires the notify() invocation to be the last sentence in all methods of the monitor in which this invocation appears..
7. The notifyAll() sentence cannot be implemented in a Brinch-Hansen monitor, because this monitor cannot ensure mutual exclusion if two or more threads are reactivated.
8. Brinch-Hansen and Hoare monitors ensure that after a notify() sentence the reactivated thread finds the state of the monitor exactly the same as it was when this notify() was executed.
9. The Lampson-Redell monitor employs a special queue (with priority over the entry queue), where all threads that execute notify() must wait.
10.A cross-invocation between the methods of two monitors will rarely produce a deadlock.

QUESTION 2. There is a territory, formed by a matrix of cells, which is shared by a set of ants. Each cell can be occupied at most by an ant. Each ant, which is modeled by a thread, moves along the territory, moving from its current cell to another neighboring cell. If the neighboring cell is busy, the ant will have to wait.

Given the following initial situation: the ant H1 in the cell (1,1); the H2 ant in the cell (1,2). Analyze what happens in the following situations (always starting from the same initial situation):

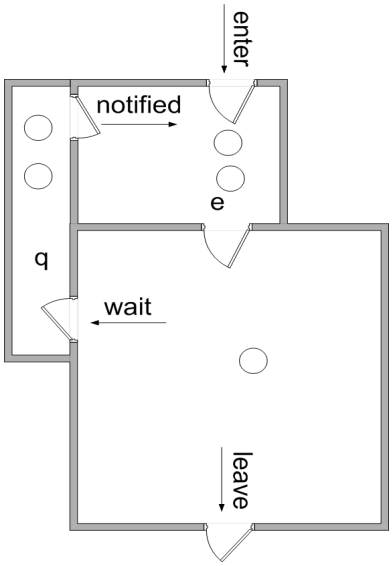
a) H2 wants to move to (1,3); H1 wants to move to (2,1).

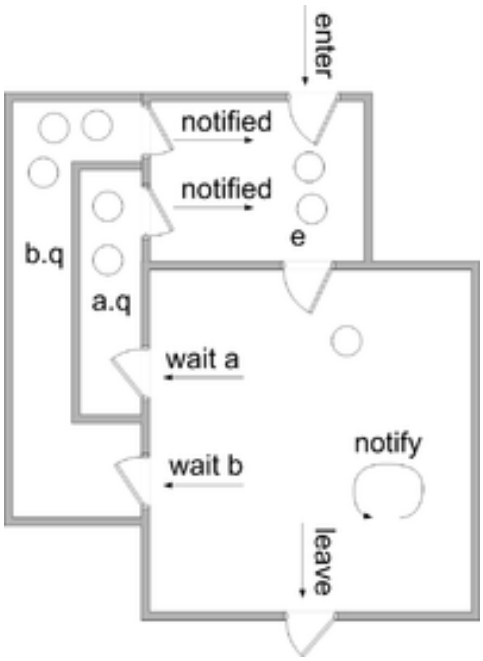
b) H1 wants to move to (1,2); H2 wants to move to (1,3).

Below there is part of the code of the Ant class, providing the pseudocode of the *move()* method:

```
void move(int FromX, int FromY, int ToX, int ToY){
  //The ant wants to move form cell (FromX,FromY) to cell (ToX,ToY)

  close_lock;   //to protect the critical section
  //if occupied[ToX, ToY] it must wait until it is free
  while (occupied[ToX, ToY]) {
      sleep(100);

   //cell free. Ant moves
   occupied[FromX,FromY]=false;
   occupied[ToX,ToY]=true;

  open_lock;
}
```

QUESTION 3. The following drawings are different representations of the monitor concept.





2.1 Explain, for each of them, what do they represent: (i) the balls or circles; (ii) the "rooms" or areas that appear (eg q, e, b.q, a.q); (iii) the actions (eg wait, notified ....)

2.2. A monitor must guarantee mutual exclusion and conditional synchronization. How is this achieved?

2.3 Explain, for each monitor, how a thread can pass from one "room" to another, until leaving the monitor.

2.4 According to the studied monitor variants, could you indicate which variant corresponds to each of the previous monitor representations?

QUESTION 4. Complete the following summary table about the characteristics of each type of monitor, by checking the corresponding box.

Features	Brinch Hansen	Hoare	Lampson/Redell
The notifier thread...			
• Leaves the monitor			
• Goes to a special queue			
• Continues its execution			
The notified thread...			
• Goes to the entry queue			
• Continues its execution			

CUESTION 5. We want to program a monitor that models the behavior of a **barrier** that is usually closed, blocking the passage of the threads that invoke its **wait**() method. When another thread invokes its **open**() method, the barrier momentarily opens and reactivates all blocked threads. Once this is done, it will remain closed until the next call to open. Please, indicate what the code of this monitor should be.

ACTIVITY 1 OBJECTIVES: To understand the model of concurrent programming provided by monitors.

FORMULATION: We have a program in which there is a monitor X and 4 tasks T1, T2, T3 and T4. The system scheduler employs a priority algorithm with preemptive static priorities, being:

Priority(T1)>Priority(T2)>priority(T3)>priority(T4)

At the initial instant, task T4 is executing an operation of monitor X, and tasks T1, T2 and T3 are suspended.

Please, fill in the following table with the schedule of the system from the initial instant, having the next sequence of events:

1) T1 becomes active, and its next operation is to invoke an operation of monitor X.

2) T2 becomes active, and its next operation is to execute code that is outside monitor X.

3) The running process invokes an operation of monitor X.

4) T3 becomes active and its next operation is to execute code that is outside monitor X.

Events	**CPU**	**ACTIVES outside monitor X**	**ACTIVES inside monitor X**	**QUEUE of monitor X**	**Suspended**
Initially	T4	---	T4	---	T1,T2,T3
Event 1)					
Event 2)					
Event 3)					
Event 4)					

ACTIVITY 2. Let us assume a street with a **crosswalk** whose correctness condition is that cars and pedestrians cannot cross it simultaneously. The crosswalk is governed by the monitor shown below. Its methods ***enterX()*** are invoked by threads of type X (C=Car or P=Pedestrian) when they arrive to the crosswalk. Its methods ***leaveX()*** are invoked by threads of type X when they leave the crosswalk.

```
monitor Crosswalk {
   condition OKcars, OKpedestrians;
   int  c, c_waiting, p, p_waiting;

   public Crosswalk() {
     c = c_waiting = p = p_waiting = 0;
   }

   entry void enterC() {
     c_waiting++;
     while (p>0) OKcars.wait();
     c_waiting--;
     c++;
     OKcars.notify();
    }

    entry void leaveC() {
        c--;
        OKcars.notify();
       OKpedestrians.notify();
    }

   entry void enterP() {
      p_waiting++;
      while ((c>0) || (c_waiting>0)
            OKpedestrians.wait();
      p_waiting--;
      p++;
      OKpedestrians.notify();
    }

    entry void leaveP() {
         p--;
         OKcars.notify();
        OKpedestrians.notify();
    }
}
```

Using the (Hoare / Lampson Redell) variant:

a) Write the evolution of the state of each one of the monitor attributes if we have the following sequence of invocations to methods of the monitor.

Method invoked	Entry Queue	c_waiting	c	Queue OKcars	p_waiting	p	Queue OKpedestrians
(Initial)		0	0	empty	0	0	empty
P1:M.enterP();							
P2:M.enterP();							
C1:M.enterC();							
P3:M.enterP();							
C2:M.enterC();							
P1:M.leaveP();							
C3:M.enterC();							
P2:M.leaveP();							
Continue trace.....							

b) This monitor grants priority to one type of threads. To which one?

ACTIVITY 3 Rewrite the original code of the monitor of Activity 2 so that it now implements the Brinch Hansen model. Remember that in the Brinch Hansen model there can only be a call to *notify*() of a certain condition as the last sentence of those methods where *notify*() should be used. That is, **two notify()** calls, one after another as in the example of Activity 2, cannot be executed.

ACTIVITY 4 Write a monitor that implements a communication link with null capacity. This implies that when the receiver is the first to be ready, it should wait for the sender; i.e., until the latter sends a message. On the other hand, when the sender is the first to be ready, it should also wait until the receiver calls its associated receive() method. Note that "null capacity" means that the operating system does not use any buffer to keep temporarily the pending undelivered messages. This behavior is implemented by the monitor shown below (Note that null-capacity links only interconnect a pair of processes. A single server and a single receiver will interact with this monitor):

```
monitor SynchronousLink {
  condition OKsender, OKreceiver;
  int senders_waiting, receivers_waiting;
  Message msg;

  public SynchronousLink() {
     senders_waiting = receivers_waiting = 0;
     msg = null;
  }

  entry void send(Message m) {
     if (receivers_waiting > 0) {
        msg = m;
        OKreceiver.notify();
     } else {
        senders_waiting++;
        OKsender.wait();
        senders_waiting--;
        msg = m;
     }
   }
```

```
 entry Message receive() {
      if (senders_waiting > 0) {
         OKsender.notify();
      } else {
         receivers_waiting++;
         OKreceiver.wait();
         receivers_waiting--;
      }
      return msg;
   }
}
```

a) Justify whether this monitor provides the expected behaviour for a Brinch Hansen monitor. Show traces that justify your answer. If this monitor does not work, please indicate why (explain motives) and try to give a solution.

b) Justify whether this monitor provides the expected behavior for a Hoare monitor implementation.

c) Justify whether this monitor provides the expected behavior for a Lampson/Redell implementation.

d) Explain which modifications to the code you should do to implement this monitor in Java.

ACTIVITY 5 Given the following program:

```
public class TestIt {
  public class Test {
    public synchronized void test (Test t) {
      t.SayHello ();
    }
    public synchronized void SayHello () {
      System.out.println ("Hello");
    }
  }

  public TestIt () {
    Test t1 = new Test();
    Test t2 = new Test();
    new Thread (new Runnable () {public void run() {
      t1.test(t2);
    }}).start();
    new Thread (new Runnable () {public void run() {
      t2.test(t1);
    }}).start();
  }
  public static void main (String args[]) {
    new TestIt();
  }
}
```

Answer if the following statements are true or false, and justify your response.

1. If we see "Hello" once on the screen, it is sure that we will see "Hello" twice.
2. It is a correct program, free of race conditions and free of deadlocks, because the methods are "synchronized".
3. In every execution, at least we will see "Hello" on the screen once.
4. It can present race conditions in access to variables t1 and t2.

ACTIVITY 6 Next you have the last proposal for the activity of Producer-Consumer problem shown in Unit 2. Modify it appropriately, using Java syntax and the Java monitor model, so that the resultant class is a correct monitor.

```
public class Consumer extends Thread
{
     private Box box;
     private int cname;
     public Consumer(Box c, int name)
     {
         box = c;
         cname = name;
     }
     public void run()
     {
         int value = 0;
         for (int i = 1; i < 11; i++)
         {
            value = box.get();
            System.out.println("Consumer #" + cname + " gets: " + value);
            try
            {               sleep((int)(Math.random() * 100));           }
            catch (InterruptedException e) { }
         }
     }
}
```

```
public class RaceCondition
{
     public static void main(String[] args)
     {
          Box c = new Box();
          Consumer c1 = new Consumer(c, 1);
          Producer p1 = new Producer(c, 1);

          c1.start();
          p1.start();

     }
}
```

```
public class Producer extends Thread
{
    private Box box;
    private int prodname;
    public Producer(Box c, int name)
    {
         box = c;
         prodname = name;
    }
    public void run()
    {
         for (int i = 1; i < 11; i++)
         {
             box.put(i);
             System.out.println("Producer #" + prodname + " puts: " + i);
             try
             {
                 sleep((int)(Math.random() * 100));
             } catch (InterruptedException e) { }
         }
    }
}
```

```
public class Box
{
   private int content =0;
   private boolean full = false;

   public synchronized int get()
   {
         while (!full) Thread.yield();
         int value = content;
         content = 0;
         full = false;
         return value;
   }
   public synchronized void put(int value)
   {
         while (full) Thread.yield();
         full = true;
         content = value;
   }
}
```

ACTIVITY 7 OBJECTIVES: Build monitors in concurrent programming languages, avoiding their potential problems.

FORMULATION: Review the code that is shown in the following figures (these classes are available in PoliformaT as a BoundedBuffer.zip file, in the folder associated to the "Material Aula" of Unit 3). This code tries to solve the problem of the bounded buffer using two "conditions" using the Java programming language. This activity will explain why this solution does not make sense (we will see its potential problems).

Download that ".zip" file, uncompress and compile it and check its functionality. To this end, modify the classes Main (e.g., generating more threads of the Producer and Consumer classes) and BoundedBuffer (e.g., tagging its methods put() and get() with "synchronized"). Analyze the results of these modifications executing the resulting program. Answer the questions that can be found at the end.

```
public class BoundedBuffer {
    private long numItems;
    private int first,last,capacity;
    private long items[];
    private Object notFull;
    private Object notEmpty;

    public BoundedBuffer(int size){
       capacity = size;
       items = new long[size];
       numItems = first = last = 0;
       notFull = new Object();
       notEmpty = new Object();
    }
    public void put(long item) {
        if (numItems == capacity) try {
                synchronized(notFull) {
                    notFull.wait();
                }
             }catch(Exception e){};
       items[last] = item;
       last = (last + 1) % capacity;
       numItems++;
       synchronized(notEmpty){
           notEmpty.notify();
       }
    }
```

```
    public long get() {
       long valor;
       if (numItems == 0) try {
               synchronized(notEmpty){
                   notEmpty.wait();
               }
            }catch(Exception e){};

       valor = items[first];
       first = (first + 1) % capacity;
       numItems--;
       synchronized(notFull){
           notFull.notify();
       }
       return valor;
    }
}
```

```
public class Main {
    static public void main(String[] args) {
        BoundedBuffer buf = new BoundedBuffer(4);
        Consumer c = new Consumer(buf,1);
        Producer p = new Producer(buf,1);

        p.start();
        c.start();
    }
}
```

```
public class Producer extends Thread {
    private BoundedBuffer buf;
    private int id;

    public Producer( BoundedBuffer buffer, int ident ) {
        buf = buffer;
        id = ident;
    }

    public void run() {
        long i;

        for (i=0; i<10; i++) {
            System.out.println("Producer "+id+
                               ": inserting element "+i+
                               "...");
            buf.put(i);
            System.out.println("Producer "+id+": OK!");
        }
        System.out.println("End of producer "+id+".");
    }
}
```

```
public class Consumer extends Thread {
    private BoundedBuffer buf;
    private int id;

    public Consumer( BoundedBuffer buffer, int ident ) {
        buf = buffer;
        id = ident;
    }

    public void run() {
        int i;
        long el;

        for (i=0; i<10; i++) {
            System.out.println("Consumer "+id+
                               ": obtaining element " +
                               "...");
            el = buf.get();
            System.out.println("Consumer "+id+
                               ": Element "+el+".");
        }
        System.out.println("End of consumer "+id+".");
    }
}
```

Answer these questions about the BoundedBuffer class:

a) Does it appropriately implement the conditional synchronization? Why? Do we need to change the “if” of these sentences by “while” loops?

b) Do we need to label put() and get() methods of the BoundedBuffer class with “synchronized”? What would imply this change?

c) Explain whether we can consider that this class is a correct monitor. If not, please explain the reasons and provide a trace in which the monitor is generating a “race condition”.

ACTIVITY 8 OBJECTIVE: Evaluate existing monitor variants

FORMULATION: There is a company that uses a mixed bathroom where both men and women can go inside, but with the condition that simultaneously there can only be people of one sex. In addition, the bathroom has a limited capacity of 3 people.

```
monitor Bathroom {
    condition full, opposite_sex;
    int occupants=0, capacity=3;
    boolean women=FALSE;
    entry enters_adult(boolean isWoman){
       if (occupants+1 > capacity) full.wait();
       if (occupants>0 && women!= isWoman){
          full.notify();
          opposite_sex.wait();
          opposite_sex.notify();
     }
    occupants++;
    women = isWoman;
   }

   entry exits_adult(){
         occupants--;
         if (occupants+1 == capacity)
               full.notify();
         else  if (occupants==0)
                  opposite_sex.notify();
    }
}
```

Taking into account that both men and women invoke the methods of the monitor following the protocol *enters_adult() ... exits_adult(),* and knowing that the monitor must control that the capacity of the bathroom is not exceded and that it can only be used by people of the same sex at the same time...

a) With the Hoare variant, could adults of different sex enter the bathroom? Could the capacity of the bathroom be exceeded? Show some trace(s) that justifies your answers.

b) With the Hoare variant, would there be a difference in the execution of the monitor if instead of the "if" statements of the *enter_adult* method we would have "while" sentences?

c) Can we say that the monitor proposed here works correctly for the Hoare variant?

d) Perform the same analysis for the Lampson-Redell variant. That is, with Lampson-Redell can adults of different sex enter the bathroom? Can the capacity of the bathroom be exceeded? Show a trace that justifies your answers.

e) Could this monitor be used with the Brinch-Hansen variant? Why?

f) Is "waterfall reactivation" used in this monitor? If so, how could it have been implemented in Java?

ACTIVITY 9 OBJECTIVE: Evaluate existing monitor variants

FORMULATION: In a forest there are 10 wolves and 10 lambs that share a river where they go to drink. The access to the river is managed by the following code:

```
monitor River {
  condition enter,exit;
  int inside=0;

  entry void wolf_enter()
  {
     // the wolf is hungry and thirsty
     if ( inside == 1) eat_lamb();
  }

  entry public void wolf_exit()
  {   // the wolf stops drinking }

  public comer_cordero(){
   //eating
  }
```

```
   entry void lamb_enter()
   { // the lamb is thirsty
      enter.notify();
      if ( inside == 0) enter.wait();
      inside++;
      exit.notify();
    }

   entry void lamb_exit()
   {   exit.notify();
      if ( inside == 2) exit.wait();
      inside--;
      // the lamb stops drinking
    }
} // end monitor
```

Assuming that both wolves and lambs always respect the protocol for invoking the method *wolf/lamb_enter*() before accessing the river to drink there and the method *wolf/lamb_exit()* when they leave the river...

a) If the variant of Hoare monitors is used, do the wolves and the lambs access the river in exclusion? That is, if there are wolves drinking there can not be lambs drinking and vice versa? Can wolves eat lambs? Could starvation occur for the wolves, if lambs do not stop getting to the river? Justify your answers with a trace.

b) If we use the Brinch Hansen monitor variant, can the lambs be extinguished when the wolves eat them? Justify your answer with a trace.

c) If we use the Lampson Redell monitor variant, can lambs be eaten by wolves? Justify your answer with a trace.

d) What is the variable "inside" used for? What does it represent?

ACTIVITY 10 – Ant Problem

Given the "ant problem", in which there are many "ant threads" that want to move from one cell to another, complete the code of the monitor, assuming the following options.

Option 1) There is one condition variable for each cell of the territory.

```
Monitor Territory{
      boolean [N][N] occupied;



      entry void moves(int x, y, x', y'){



            occupied[x',y']=true;
            occupied[x,y]=false;



      }
}
```

Option 2) There is only one condition variable for all the territory.

```
Monitor Territory{
      boolean [N][N] occupied;


      entry void moves(int x, y, x', y'){



            occupied[x',y']=true;
            occupied[x,y]=false;



      }
}
```

Questions:

- Which of these two solutions is more efficient? Why?

- Which one requires "waterfall awake"?

- Which one can be implemented using the basic monitor model in Java?

ACTIVITY 11. The following code aims to implement a monitor to control the access of readers and writers on a shared resource. Writers will use *pre-writing*() before accessing and *post-writing*() after having accessed. Readers will use *pre-reading*() before accessing and *post-reading*() after accessing:

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

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

  entry void pre-reading() {
     read_waiting ++;
     SENTENCE-2
     read_waiting --;
     readers = readers+1;
     read.notify();
  }

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

Consider the following values for the missing sentences and answer the questions:

```
SENTENCE-1: if (read_waiting > 0) read.notify(); else write.notify();
```

```
SENTENCE-2: if (writers > 0) read.wait();
```

1. If the monitor used follows the Hoare variant, it provides access in mutual exclusion between readers and writers, and also for writers among them.	
2. If the monitor used follows the Brinch Hansen variant, the code is wrong, because it does not allow multiple readers to access simultaneously the shared resource.	
3. If the monitor used follows the Hoare variant, readers are given priority, and this implies starvation for writers. Therefore, in case there are already readers accessing the shared resource and other readers continue to arrive, the writers will have to wait.	
4. If the monitor used follows the Brinch Hansen variant, the code will not meet the access requirements, since in this variant it is not allowed to define multiple "condition" attributes in a single monitor.	
5. If the monitor used follows the Lampson-Redell variant, the code complies with the access requirements indicated in the problem.	