MODELO A

EXAMEN PRIMER PARCIAL **Concurrencia y Sistemas Distribuidos**
Unidades Didácticas 1 a 6 - Prácticas 1, 2 y 3 **Fecha: 25 de Marzo de 2019**

Este examen tiene una duración total de 2 horas.

Este examen tiene una puntuación máxima de **10 puntos**, que equivalen a **3.5** puntos de la nota final de la asignatura. Consta tanto de preguntas de las unidades didácticas como de las prácticas.
Indique, para cada una de las siguientes **58 afirmaciones**, si éstas son verdaderas (**V**) o falsas (**F**).

Cada respuesta vale: correcta= 10/58, errónea= -10/58, vacía=0.

TEORÍA

Respecto a las diferencias entre la programación concurrente y la programación secuencial:

1. Los programas secuenciales pueden generar condiciones de carrera, al igual que los programas concurrentes.	F
2. Un proceso con varios hilos de ejecución, ejecutado con un procesador con un único núcleo que ofrece multiprogramación, es un ejemplo de aplicación concurrente.	V

Respecto a la programación concurrente en Java:

3. Si un método se etiqueta con `synchronized` y se invoca desde otro método de la misma instancia etiquetado también con `synchronized`, provoca que el hilo se espere a sí mismo (interbloqueo).	F

Sobre el problema de la sección crítica:

4. Una solución correcta para el problema de la sección crítica debe cumplir las propiedades de exclusión mutua, retención y espera, no expulsión y espera circular	F
5. El problema de la sección crítica aparece cuando varios hilos comparten un objeto inmutable (constante) al que todos los hilos necesitan acceder de forma simultánea.	F
6. El problema de la sección crítica se soluciona empleando correctamente mecanismos de sincronización, como por ejemplo monitores, semáforos y cerrojos.	V
7. Si empleamos un único cerrojo para proteger las secciones críticas de cierto objeto, se permitirá como máximo un hilo ejecutando dichas secciones críticas	V
8. En una sección crítica protegida por un `lock`, el protocolo de salida libera el `lock`, permitiendo que todos los hilos bloqueados en la entrada accedan al objeto	F

Sobre el concepto de monitor y sus variantes:

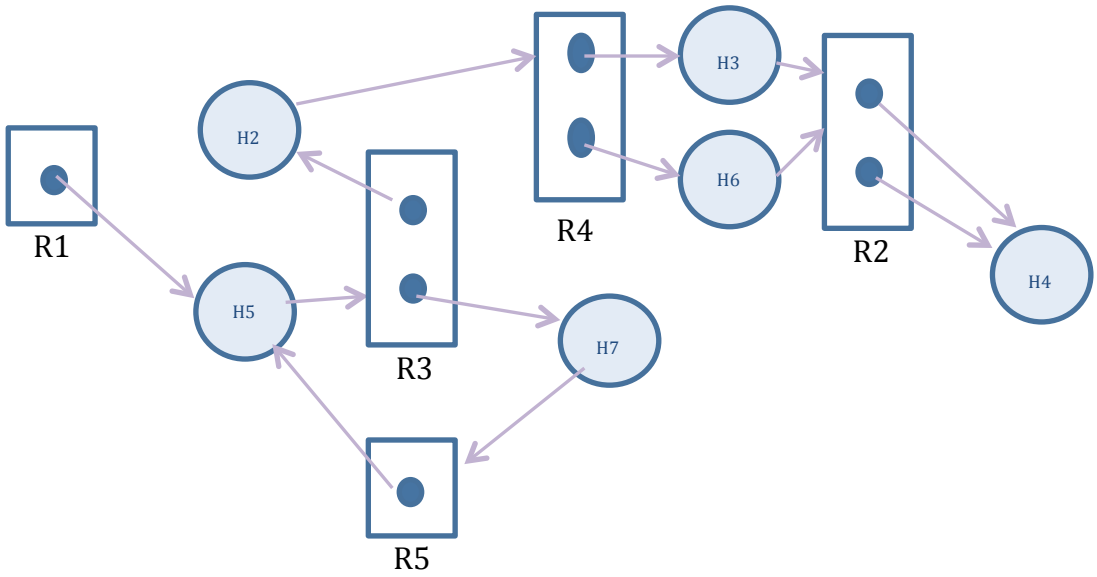
9. Suponiendo que haya algún hilo suspendido en la variable condición `c` , un monitor que siga el modelo de Hoare suspende al hilo que ha invocado a `c.notify()` y activa a uno de los hilos que llamó antes a `c.wait()`.	V
10. El lenguaje Java proporciona por defecto monitores de tipo "Lampson/Redell".	V
11. El monitor Lampson-Redell garantiza que tras una operación `notify`(), el hilo reactivado encuentra el estado del monitor exactamente igual que estaba cuando se ejecutó dicho `notify`.	F

Dado el siguiente programa Java:

```
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. Al ejecutarlo, veremos por pantalla al menos una línea con la sentencia `Task finished`	F
13. Al ejecutarlo, veremos por pantalla, entre otras cosas, la línea `Preparing worker 4 for: Chief4`	V
14. Con este código se crearán un total de 10 hilos, aparte de `main`.	F
15. Al ejecutarlo, veremos por pantalla, entre otras cosas, 5 líneas con la palabra `done`	V
16. Se requiere etiquetar el método `myAction()` con la etiqueta `synchronized` para evitar las condiciones de carrera.	F

Dado el siguiente grafo de asignación de recursos:



17. Se encuentran en interbloqueo los procesos H5 y H7.	F
18. El sistema presenta al menos una secuencia segura.	V
19. Si el proceso H4 solicita una instancia de cualquiera de los recursos del sistema (sin liberar las dos instancias que ya tiene asignadas del recurso R2), se producirá un interbloqueo y ningún proceso podrá acabar.	V
20. En este GAR se dan todas las condiciones de Coffman.	V

Sobre las condiciones de Coffman y las situaciones de interbloqueo:

21. Las condiciones de Coffman permiten diseñar sistemas que cumplan con todas ellas, para así garantizar que no se producirán interbloqueos.	F
22. Una de las condiciones de Coffman consiste en solicitar todos los recursos requeridos inicialmente, de modo que los hilos se bloquean (retención y espera) si existe un conflicto en las peticiones.	F
23. Las situaciones de interbloqueo pueden prevenirse asignando los recursos de manera que nunca se genere un ciclo dirigido.	V
24. Una de las condiciones de Coffman consiste en que los recursos asignados pueden ser expropiados.	F

MODELO A

Un taller de una joyería quiere montar collares de perlas solo blancas, solo azules o combinados de perlas blancas y azules. Para ello dispone de 5 encargados y 2 cestos, uno para cada color de perla, con capacidad limitada . Para organizar la producción se decide que un encargado será el proveedor de perlas blancas, otro encargado proveerá las perlas azules y el resto de encargados se destinará al montaje de cada tipo de collar. El monitor `GestorDePerlas`, gestiona el número de perlas almacenadas en los cestos. Hay un hilo asociado a cada encargado. Los encargados de proveer perlas, se encargan de obtener una perla y almacenarla en el cesto correspondiente utilizando los métodos `AñadirBlanca` ó `AñadirAzul`. El resto de encargados solicitan al monitor el número de perlas de cada color que necesitan para montar el collar utilizando el método `SolicitarPedido`. Analice la siguiente propuesta para el monitor `GestorDePerlas`

```
public class GestorDePerlas {

      final static private int NMaxBlancas  = 50;
      final static private int NMaxAzules = 50;

      private int NBlancas  = 0;
      private int NAzules = 0;

      private boolean PedidoEnCurso = false;

      public synchronized void AñadirBlanca() {
            NBlancas = NBlancas ++;
            notifyAll();
            while (NBlancas == NMaxBlancas)
              try {wait();} catch (InterruptedException e){};
      }

      public synchronized void AñadirAzul() {
            NAzules = NAzules ++;
            notifyAll();
            while (NAzules == NMaxAzules)
              try {wait();} catch (InterruptedException e){};
      }

      public synchronized void SolicitarPedido(int SolBlancas,
         int SolAzules) {

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

            PedidoEnCurso = true;

            while (SolBlancas > NBlancas || SolAzules > NAzules)
              try {wait();} catch (InterruptedException e){};

            NBlancas  = NBlancas  - SolBlancas;
            NAzules = NAzules - SolAzules;
            PedidoEnCurso = false;
            notifyAll();
      }
}
```

25. En esta solución se puede sobrepasar el máximo número de piezas blancas o azules en los cestos, puesto que se incrementan los contadores antes de comprobar si caben.	F
26. El atributo `PedidoEnCurso` es necesario para proporcionar exclusión mútua en el acceso al método `SolicitarPedido`.	F
27. La solución no es correcta porque la invocación al método `notifyAll` en `AñadirBlanca` y `AñadirAzul`, debería ser la última instrucción en ambos métodos.	F
28. El calificativo `synchronized` en los métodos `AñadirBlanca` y `AñadirAzul`, no es necesario ponerlo, ya que sólo hay un hilo que añade piezas blancas y un hilo que añade piezas azules.	F
29. El atributo `PedidoEnCurso` se utiliza para conseguir que cuando un pedido P1 está esperando a que se completen las piezas solicitadas, los nuevos pedidos no se atenderán hasta que se complete P1.	V
30. La solución propuesta para el monitor es correcta, y sincroniza adecuadamente según el enunciado, los proveedores de perlas y la gestión de los pedidos que realizan los montadores.	V

Un sistema de tiempo real crítico se compone de 5 tareas independientes cuyas características se describen en la siguiente tabla:

Tarea	Tiempo de cómputo	Periodo	Plazo
T1	3	15	8
T2	4	20	13
T3	6	30	18
T4	5	40	32
T5	6	50	50

Asumiendo que el sistema se planifica por prioridades fijas expulsivas con asignación de prioridades inversa a su plazo, es decir la tarea de menor plazo es la más prioritaria….

31. … el tiempo de respuesta de la tarea T4, R4 es 30.	F
32. … el hiperperiodo es 600.	V
33. … si las tareas siempre utilizan en todas sus activaciones el tiempo de cómputo indicado en la tabla, en el instante 605 la tarea T2 estará en ejecución en la CPU.	V
34. … el tiempo de respuesta de la tarea T2, R2 es 7.	V
35. … el tiempo de respuesta de la tarea T3 es menor o igual a su plazo.	V
36. … el sistema es planificable.	V

Asumiendo ahora que las tareas no son independientes, y que además las tareas utilizan tres semáforos M1, M2 y M3 para sincronizar el acceso a sus secciones críticas (SC) cuyas características se describen en la siguiente tabla, y que dichos semáforos utilizan el protocolo del techo de prioridad inmediato …

Tarea	Semáforo	Duración de la SC
T1	M1	1
T2	M1	2
T2	M2	4
T4	M2	3
T4	M3	2
T5	M3	1

37. … el factor de bloqueo de la tarea T1, B1 es 3.	F
38. … el factor de bloqueo de la tarea T3, B3 es 0.	F
39. … el techo de prioridad del semáforo M2 es igual a la prioridad de T4.	F
40. … el tiempo de respuesta de la tarea T2, R2 es 10.	V
41. … el tiempo de respuesta de la tarea T3 es menor o igual a su plazo.	F
42. … el tiempo de respuesta de la tarea T5 es igual al que tendría si todas las tareas fueran independientes (es decir no necesitasen sincronizarse).	V
43. … el sistema es planificable.	F

Sol:

Tarea	Tiempo de cómputo	Periodo	Plazo	Ri sin Bi	Bi	Ri con 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

Sobre el uso de las herramientas de la librería *java.util.concurrent*:

44. Los objetos `CountDownLatch` garantizan que el método `await()` suspenderá al hilo invocador, siempre y cuando la barrera no esté abierta.	V
45. Los objetos `CountDownLatch` se crean con un valor entero que indica el número de hilos que deberán quedarse esperando en la barrera para poder abrirla.	F

46. Dados 6 hilos de la clase A, 2 hilos de la clase B y 4 hilos de la clase C que comparten el mismo objeto "c" de tipo `CyclicBarrier` inicializado a 4, sabiendo que todos los hilos ejecutan `c.await()`, los hilos de la clase B podrán quedarse suspendidos indefinidamente.	F
47. Dados 6 hilos de la clase A, 2 hilos de la clase B y 4 hilos de la clase C que comparten los mismos objetos `sem1` y `sem2` de tipo `Semaphore` inicializados ambos a 0, sabiendo que cada hilo de la clase A ejecuta `sem1.acquire()`, los hilos B ejecutan `sem2.acquire()` y los hilos C ejecutan `sem1.release(); sem2.release();` algún hilo de la clase A podrá quedarse suspendido indefinidamente.	V
48. Cada cerrojo de tipo `ReentrantLock` puede crear una o varias variables condición, pasándole la referencia del cerrojo al constructor del objeto `Condition`.	F
49. Se puede implementar una solución correcta para el problema de la sección crítica mediante un `ReentrantLock rl`, utilizando como protocolo de entrada `rl.trylock()` y como protocolo de salida `rl.lock()`.	F
50. Si varios hilos hacen uso de un mismo objeto `AtomicLong`, se deben proteger con la etiqueta `synchronized` los métodos que dicho objeto ofrece, para evitar las condiciones de carrera.	F

PRÁCTICAS

Sobre la práctica 1 "Uso compartido de una piscina", donde teníamos los siguientes casos:

Pool0	Baño libre (no hay reglas)
Pool1	Los niños no pueden nadar solos (debe haber algún instructor con ellos)
Pool2	No se puede superar un determinado número de niños por instructor
Pool3	No se puede superar el aforo máximo permitido de nadadores
Pool4	Si hay instructores esperando salir, no pueden entrar niños a nadar

51. En la piscina Pool1 no es necesario suspender a ningún hilo, al no existir sincronización condicional.	F
52. Para implementar la piscina Pool3, es necesario el uso de dos monitores.	F
53. En la piscina Pool0 no es necesario poner `synchronized` en los métodos de esta clase, pues la piscina es un objeto sin estado.	V

Sobre la práctica 2 "Los cinco filósofos comensales", donde teníamos las siguientes versiones:

Versión 1	Asimetría (todos igual excepto el último)
Versión 2	Asimetría (pares/ impares)
Versión 3	Todo o nada
Versión 4	Capacidad de la mesa

54. En la solución basada en limitar el número de filósofos en la mesa, se garantiza la ausencia de interbloqueo con independencia del orden en el que los filósofos cojan los tenedores.	V

Queremos implementar la estrategia de coger los dos tenedores o ninguno para la práctica de "los cinco filósofos comensales". Dada la siguiente implementación de `takeLR`, en la clase `BothNoneTable`:

```
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. Este código es correcto.	F

Sobre la práctica 3 "Problema de las hormigas", donde teníamos las siguientes versiones:

Actividad 0	Lock intrínseco de Java
Actividad 1	ReentrantLock con una única variable condición asociada a todo el territorio
Actividad 2	ReentrantLock con una variable condición por cada celda del territorio

56. Utilizando los mismos tamaños de territorio y número de hormigas, si comparamos los resultados entre utilizar el lock intrínseco de Java (Actividad 0) y el uso de `ReentrantLock` con una variable condición por cada celda del territorio (Actividad 2), en término medio se produce un número significativamente mayor de reactivaciones "innecesarias" de hilos en la Actividad 0 respecto a la Actividad 2.	V
57. En la actividad 2 de la práctica, donde se asocia una variable condición por cada celda del territorio, necesitamos crear una matriz de objetos de tipo `ReentrantLock`, definiendo un lock y una variable condición por cada elemento del array.	F
58. En el código inicial suministrado se emplea una barrera cíclica que provoca que todas las hormigas comiencen a moverse por el territorio a la vez, tras estar todas las hormigas colocadas en el territorio en sus celdas respectivas.	F