EXAMEN PRIMER PARCIAL **Concurrencia y Sistemas Distribuidos**
Unidades Didácticas 1 a 5 - Prácticas 1 y 2 **Fecha: 21 de Marzo de 2016**

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 **56 afirmaciones**, si éstas son verdaderas (**V**) o falsas (**F**).

Cada respuesta vale: correcta= 10/56, errónea= -10/56, vacía=0.

Importante: Los **primeros 3 errores no penalizarán**, de modo que tendrán una valoración equivalente a la de una respuesta vacía. A partir del 4º error (inclusive), sí se aplicará el decremento por respuesta errónea.

Dado el siguiente código:

```
public class MyThread extends Thread {
       protected String name;
       public MyThread(String name) {this.name = name;}

        public void myTask(){
         int num=Integer.parseInt(name);
         if (num<3){
              System.out.println("Doing my task");
              new Thread(new Runnable(){
                    public void run(){
                          System.out.println("Task is done"); }
                });
          }}

   public void run() {
              myTask();
              for (int i=0; i< 3; i++) {
                  System.out.println(Thread.currentThread().getName() + " is running");
                  try{Thread.sleep(i*1000);}
                     catch(InterruptedException ie){ie.printStackTrace(); };
              }
              System.out.println("Thread"+ name + " is done");
      }

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

1. Al ejecutarlo, veremos por pantalla al menos una línea con la sentencia "Task is done".	F
2. Al ejecutarlo, veremos por pantalla tres líneas con la sentencia "Doing my Task".	V
3. Con este código se crearán un total de 10 hilos, además de *main*.	F
4. De los hilos creados, sólo se ejecutarán 5 hilos, con nombres "MyThread 0"... "MyThread 4", aparte de *main*.	V
5. Veremos por pantalla al menos 3 líneas con "Thread MyThread X is done", siendo X un número.	F
6. En este código se pueden producir condiciones de carrera.	F

Sobre las características de la programación concurrente (vs. la programación secuencial):

7. Un programa concurrente requiere de máquinas con más de un procesador (o con varios núcleos) para poder ser ejecutado.	F
8. Un objeto constante (cuyo estado no cambia) puede compartirse de forma segura entre varios hilos de un programa concurrente, sin que puedan producirse condiciones de carrera.	V
9. La cooperación de las tareas en los programas concurrentes se realiza mediante la comunicación y la sincronización.	V
10. Un programa secuencial generalmente es determinista.	V

Dado el siguiente código Java:

```
public class Cantera {
   private int enanosWaiting;
   private boolean hayGigante;
   private int enanos;
   private int gigantes;

   public Cantera() {
       enanosWaiting=0;
       enanos=0;
       gigantes=0;
       hayGigante=false;
   }

   public synchronized void enano_entrar()
   {
    enanosWaiting++;
    while (hayGigante)
          try { wait(); }
            catch(Exception e) { };
    enanosWaiting--;
    enanos++;
   }
```

```
 public synchronized void enano_salir()
  {
       enanos--;
       notifyAll();
   }

 public synchronized void gigante_entrar()
  {
     gigantes++;
     while (hayGigante || enanos>0 )
       try { wait(); }
            catch(Exception e) { };
     hayGigante=true;
     bailando();
  }

public synchronized void gigante_salir ()
  {  hayGigante=false;
     notifyAll();
   }

public void bailando(){
    System.out.println("Gigante bailongo");
   }
}
```

Asumiendo que esta clase la utilizan hilos de tipo "Enano" y "Gigante" y que respetan siempre el protocolo de invocar el método "*enano/gigante_entrar()*" antes de acceder a la cantera y "*enano/gigante_salir()*" cuando salen de la cantera...

11. Esta clase implanta un monitor para controlar el acceso de los enanos y los gigantes a un recurso compartido, que en este caso es la cantera.	V
12. Podrá haber múltiples enanos accediendo simultáneamente a la cantera, pero en exclusión mutua con el gigante.	V
13. Podrá haber múltiples gigantes bailando a la vez dentro de la cantera.	F
14. Se requiere añadir la etiqueta "synchronized" al método "bailando" para establecer exclusión mutua entre los gigantes.	F
15. La variable "gigantes" indica el número de gigantes que hay bailando dentro de la cantera en todo momento.	F
16. Se utiliza la variable "enanosWaiting" para establecer que los gigantes tengan prioridad sobre los enanos en el acceso a la cantera.	F

Sobre el concepto de monitor y sus variantes:

17. Un monitor es una clase que permite definir objetos que se compartan de forma segura, al resolver la sincronización condicional y la exclusión mutua.	V
18. Los métodos de un monitor pueden utilizarse como protocolos de entrada/salida para controlar el acceso a un recurso compartido y así definir diferente semántica según el tipo de hilo (ejemplo: exclusión mutua entre lectores/escritores, concurrencia para lectores, exclusión mutua para escritores...)	V
19. En un monitor que siga la variante de Brinch-Hansen, el hilo que está en el monitor lo abandona al realizar un notify() sobre una variable condición del monitor, pues de lo contrario podría incumplirse la exclusión mutua.	V
20. En un monitor que siga la variante de Hoare, el hilo que está en el monitor, inmediatamente tras realizar un notify() sobre una variable condición continuará activo en el monitor, con independencia de que en dicha condición hubiera algún hilo suspendido.	F
21. En un monitor que siga la variante de Lampson-Redell, se garantiza que un hilo reactivado tras un *notify*() encuentre el estado del monitor conforme a la condición que requería.	F

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

22. Con cada cerrojo de tipo *ReentrantLock* se pueden crear tantas variables condición asociadas a dicho cerrojo como se requieran, utilizando para ello el método *newCondition*() de dicha clase.	V
23. Si se utiliza el cerrojo "ReentrantLock" para implementar un monitor en Java, los métodos donde se empleen los objetos "Condition" deben aparecer protegidos con el calificador "synchronized".	F
24. Para que M hilos de la clase A esperen a que otro hilo de la clase B les avise se puede utilizar un objeto "c" de la clase *CountDownLatch*. Para ello, inicializamos "c" a M, los hilos A usarán *c.await()* y el hilo B llamará a *c.countDown()* un total de M veces.	V
25. Para que un hilo A espere hasta que otros N hilos de la clase B finalicen, se puede utilizar un *Semaphore* S inicializado a N; el hilo A invoca *S.acquire()* mientras que los hilos B invocan *S.release()* antes de finalizar.	F
26. Entre otras propiedades, una *CyclicBarrier* se diferencia de una barrera *CountDownLatch* en que se puede especificar en el constructor de la barrera cíclica un método *run*() que se ejecutará justo cuando la barrera se abra.	V
27. Los objetos *CountDownLatch* garantizan que el método *await*() siempre suspenderá al hilo invocador.	F
28. Se puede implementar una solución correcta para el problema de la sección crítica mediante un *ReentrantLock* "rl", utilizando como protocolo de entrada "*rl.lock()*" y como protocolo de salida "*rl.unlock()*".	V
29. 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

Asuma que existe una clase Buffer "thread-safe" con capacidad para 100 elementos, cuyo método *put*() permite insertar un elemento de tipo "int" y cuyo método *get*() extrae un elemento. Se dispone del siguiente código:

```
public class Main{
  public static void main(String[] args){
    Buffer d=new Buffer();
    SENTENCIA A
    Worker w1=new Worker(c,d,1);
    Worker w2=new Worker(c,d,2);
    Worker w3=new Worker(c,d,3);
    w1.start();
    w2.start();
    w3.start();

    try{
      SENTENCIA B
    }catch(InterruptedException e){}
    System.out.println("This is a race!");
 }
}
```

```
public class Worker extends Thread{
  SENTENCIA C
  private Buffer buf;
  private int number;

  public Worker (SENTENCIA D, Buffer c, int d)
  {  obj=a;
     buf=c;
     number=id;
    }

  public void run(){
    for (int i=1; i<=100; i++)
    {  buf.put(number*100+i);
       if (i==50){
         try{   SENTENCIA E
            }catch(InterruptedException e){} };
       System.out.println(buf.get());
    }//for
  } //run
}
```

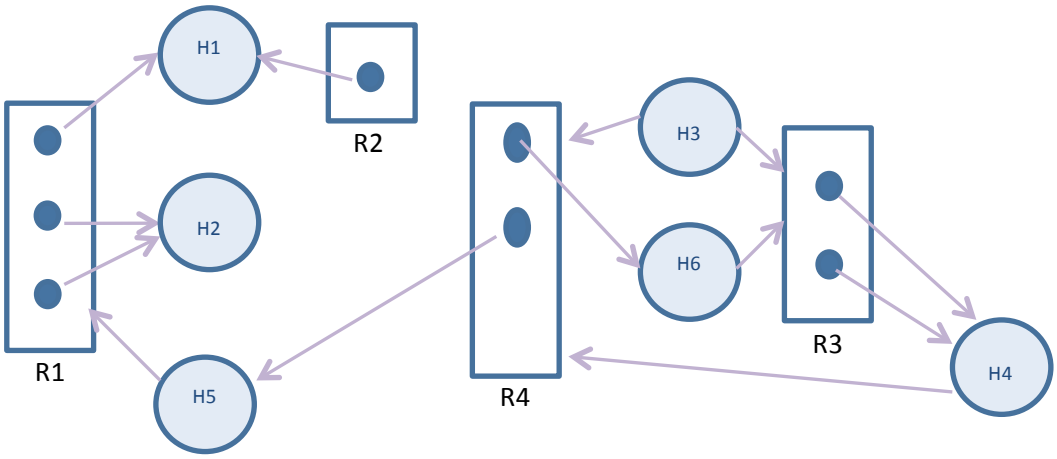
Y también se dispone de las siguientes etiquetas posibles para rellenarlo:

ID	ETIQUETA	ID	ETIQUETA
1	Semaphore c=new Semaphore(0);	2	Semaphore c=new Semaphore(4);
3	CyclicBarrier c=newCyclicBarrier(3);	4	CyclicBarrier c=new CyclicBarrier(4);
5	CountDownLatch c=new CountDownLatch(0);	6	CountDownLatch c=new CountDownLatch(3);
7	CountDownLatch c=new CountDownLatch(4);	8	int c=0;
9	c.await();	10	c.acquire();
11	c.release();	12	c.countDown();
13	c.acquire(); c.acquire(); c.acquire();	14	w1.join(); w2.join(); w3.join();
15	private Semaphore obj;	16	private CyclicBarrier obj;
17	private CountDownLatch obj;	18	private int obj;
19	Semaphore a	20	CyclicBarrier a
21	CountDownLatch a	22	int a
23	obj.release();	24	obj.acquire();
25	obj.await();	26	obj.countDown();

Si se desea que el hilo principal escriba su mensaje cuando cada uno de los hilos Worker mostrados en el código haya completado al menos 50 iteraciones:

30. Con independencia del objeto que se utilice para coordinar a los *Worker* entre sí, en SENTENCIA B se puede utilizar el método *join* de la clase *Thread*. Es decir, la etiqueta 14.	ANULADA
31. Si se emplean semáforos para coordinar a los *Worker* y al hilo principal, deberíamos utilizar la combinación de sentencias-etiquetas siguiente: A-1, B-13, C-15, D-19, E-11	**F**
32. Si se emplean barreras para coordinar a los *Worker* y al hilo principal, podríamos utilizar la combinación de sentencias-etiquetas siguiente: A-3, B-9, C-16, D-20, E-26.	F
33. Si se emplean barreras para coordinar a los *Worker*, podríamos utilizar la combinación de sentencias-etiquetas siguiente: A-7, B-12, C-17, D-21, E-25.	F
34. Si se emplean *CyclicBarriers*, deberíamos inicializarlas al valor 4 y utilizar para la SENTENCIA E la etiqueta 25, entre otros aspectos.	V

Dado el siguiente grafo de asignación de recursos:



35. Si los procesos H1 y H2 solicitan cada uno una instancia del recurso R4, se producirá un interbloqueo.	V
36. En el grafo de la figura existe una secuencia segura dada por: H1, H2, H5, H4, H3 y H6.	V
37. En el grafo de la figura existe un ciclo dirigido y como el recurso R2 tiene una única instancia, podemos afirmar que hay un interbloqueo.	F
38. En muchos sistemas operativos, como Unix o Windows, se utilizan grafos de asignación de recursos para evitar los interbloqueos.	F
39. En el grafo de la figura existe una secuencia segura dada por: H2, H5, H4, H6, H3 y H1	V

Sobre las Condiciones de Coffman:

40. Las condiciones de Coffman permiten diseñar sistemas que cumplan con todas ellas, para así garantizar que no se producirán interbloqueos.	F
41. Una de las condiciones consiste en que los recursos asignados no pueden ser expropiados.	V
42. 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
43. Las condiciones son necesarias pero no suficientes, de modo que en caso de interbloqueo se cumplen todas.	V
44. Si un recurso puede ser utilizado simultáneamente por varios hilos, sabemos aplicando las condiciones de Coffman que dicho recurso no podrá ser causante de interbloqueos entre los hilos de una aplicación distribuida.	ANULADA

Sobre la ejecución de la sección crítica:

45. Los protocolos de entrada/salida que controlan el acceso a una sección crítica deben garantizar que se respete la exclusión mutua, el progreso y la espera limitada.	V
46. Una solución correcta para el problema de la sección crítica debe constar siempre de un protocolo de entrada, aunque pueda faltarle el protocolo de salida.	F
47. Se debe garantizar que los hilos de un sistema sólo puedan acceder a estados intermedios de un objeto.	F
48. Las operaciones de "abrir lock" y "cerrar lock", en ese orden, permiten implementar los protocolos de entrada/salida de la sección crítica, garantizando que no aparezcan interbloqueos.	F

PREGUNTAS DE PRÁCTICAS

Sobre la práctica 1 "Uso compartido de una piscina", cuyas reglas se muestran a continuación:

Tipo de piscina	*Reglas para N niños e I instructores*
Pool0	Baño libre (no hay reglas) *(free access)*
Pool1	Los niños no pueden nadar solos (debe haber algún instructor con ellos) *(kids cannot be alone)*
Pool2	Pueden nadar un máximo de N/I niños por instructor *(max kids/instructor)*
Pool3	No puede haber más de (N+I)/2 nadadores nadando simultáneamente (máximo aforo permitido) *(max capacity)*
Pool4	Si hay instructores esperando salir, no pueden entrar niños a nadar *(kids cannot enter if there are instructors waiting to exit)*

49. Como la clase Pool0 es un objeto sin estado, no se requiere incluir la etiqueta "synchronized" en los métodos de dicha clase.	V
50. En la piscina Pool1 no se requiere aplicar sincronización condicional.	F

En Pool1 se ha implementado el método *instructorSwims(int id)* de la siguiente manera...:

```
private boolean inv(int k, int i)          {return (k==0 || i>0);}
public synchronized long instructorSwims(int id) {
          while (!inv(kids,inst+1)) wait();
          inst++;
          notifyAll();
}
```

51. En este método no es necesario realizar la instrucción *notifyAll*(), pues no hay que notificar cambios a ningún hilo.	F
52. El bucle *while* es innecesario porque no hay que esperar a ninguna condición.	V

Sobre la práctica 2 "Los cinco filósofos comensales"

53. La solución basada en la asimetría entre los filósofos requiere implementar un nuevo tipo de mesa, distinto a RegularTable, que permita a los filósofos coger los tenedores en orden contrario al original.	F
54. La solución basada en la asimetría entre los filósofos resuelve el problema de interbloqueos rompiendo la condición de espera circular.	V
55. La solución basada en "todos o nada" resuelve el problema de interbloqueos rompiendo la condición de no expropiación.	F
56. En la solución "last/but last" basada en la asimetría entre filósofos, la probabilidad de interbloqueo se incrementará conforme se aumenta el tiempo que tarda un filósofo entre coger su primer tenedor y su segundo tenedor.	F