EXAMEN PRIMER PARCIAL **Concurrencia y Sistemas Distribuidos**
Unidades Didácticas 1 a 6 - Prácticas 1, 2 y 3 **Fecha: 3 de Abril de 2017**

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.

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.

Un monitor...:

1. ... permite resolver problemas que no podemos resolver con los semáforos.	F
2. ... proporciona los mecanismos necesarios para garantizar exclusión mutua.	V
3. ... proporciona mecanismos para resolver la sincronización condicional.	V

Un monitor que siga el modelo de Hoare...:

4. ... no reactiva a ningún hilo suspendido en *wait*() mientras el hilo notificador no finalice el método que está ejecutando sobre el monitor.	F
5. ... obliga a que en los métodos donde se invoca a *notify*(), ésta sea la última sentencia	F
6. ... evita que múltiples hilos lleguen a compartir memoria.	F
7. ... suspende (y deja en una cola especial de entrada) al hilo que ha invocado a *notify*(), activando a uno de los hilos suspendidos por *wait*().	V

En un grafo de asignación de recursos…:

8. ... podemos asegurar que existe interbloqueo si todos los recursos que participan en un ciclo dirigido tienen una única instancia.	V
9. ... si existe una secuencia de finalización podemos afirmar que no hay interbloqueo.	V
10. ... se da una situación de interbloqueo siempre que no existe ninguna instancia libre en ningún recurso.	F
11. ... se da una situación de interbloqueo si todos los recursos que participan en un ciclo dirigido tienen más de una instancia.	F
12. ... si existe un ciclo dirigido podemos afirmar que existe interbloqueo.	F

Sobre la utilización de hilos en Java:

13. Para asignar nombre a los hilos se puede pasar una cadena como argumento en su constructor.	V
14. En Java solo se puede crear hilos si se utiliza el paquete *java.util.concurrent*.	F
15. El código a ejecutar por cada hilo debe estar contenido en su método *start*().	F

El juego de la vida es en realidad un juego de cero jugadores, lo que quiere decir que su evolución está determinada por el estado inicial y no necesita ninguna entrada de datos posterior. El "tablero del juego" es una malla formada por cuadrados ("células") que se extiende por el infinito en todas las direcciones. Cada célula tiene 8 células vecinas, que son las que están próximas a ella, incluidas las diagonales. Las células tienen dos estados: están "vivas" o "muertas" (o "encendidas" y "apagadas"). El estado de la malla evoluciona a lo largo de unidades de tiempo discretas (se podría decir que por turnos). El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente. Todas las células se actualizan simultáneamente.

Las transiciones dependen del número de células vecinas vivas:

- Una célula muerta con exactamente 3 células vecinas vivas "nace" (al turno siguiente estará viva).
- Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso muere o permanece muerta (por "soledad" o "superpoblación").

A continuación se muestra una implementación del juego de la vida, en Java:

```
public class LifeGame {
        public int N = 20, M = 20;
        public boolean[][] cells = new boolean [N][M];
        public int population = 0, generation = 0;
        private CyclicBarrier generationStart = null;
        private CyclicBarrier generationEnd = null;

        public class Cell extends Thread {
                private int x, y;
                public Cell (int x, int y) {
                        this.x=x; this.y=y;
                        cells [x][y] = (Math.random() < 0.4);
                }
                private int nextTo (int b, int a, int m) {
                        return b+a<0 ? m+a : (b+a) % m;
                }
                public void run () {
                        while (true) {
                           if (cells[x][y]) population++;
                           try {
                                   generationStart.await ();
                           } catch (Exception e) {}

                           int nAlive = 0;
                           for (int i=-1; i<2; i++)
                                   for (int j=-1; j<2; j++)
                                           if (cells [nextTo(x,i,N)] [nextTo (y,j,M)]) nAlive++;
                           if (cells[x][y]) nAlive--;

                           boolean nextTimeAlive = (nAlive == 3) ? true :
                                   (nAlive != 2) && cells[x][y] ? false : cells[x][y];
                           try {
                                   generationEnd.await();
                           } catch (Exception e) {}
                           cells [x][y] = nextTimeAlive;
                        }
                }
        }
```

```
public static void main (String args[]) {
        LifeGame game = new LifeGame ();
        game.go ();
}

public void go () {
        generationStart = new CyclicBarrier (N*M, new Runnable () {
          public void run () {
                System.out.println ("Generation: " +generation+ ". Population: "+population);
                population = 0; generation ++;
          });
        generationEnd = new CyclicBarrier (N*M);

        for (int i=0; i<N; i++)
                for (int j=0; j<M; j++)
                        new Cell (i,j).start();
}}
```

16. Cada vez que una célula del juego de la vida muere, el hilo correspondiente termina.	F
17. Se pueden producir condiciones de carrera en la variable "cells".	F
18. Además del hilo principal, se crean y se ejecutan N*M hilos.	V
19. Se pueden producir condiciones de carrera en la variable "population".	V
20. La barrera "generationEnd" está mal implementada, pues es necesario proporcionar una instancia "Runnable" como argumento.	F

Suponemos un sistema con tres actividades A,B,C y prioridad A>B>C.

	A	B	C
tiempos de cómputo (C)	1	3	4
periodos (T)	10	5	15
plazos (D)	8	4	15

21. El hiperperiodo es 30.	V
22. El tiempo de respuesta de C es 15.	V
23. El sistema es planificable.	V
24. El tiempo de respuesta de B es 4.	V
25. La actividad C tiene un tiempo de respuesta mayor que su plazo.	F

Suponemos un sistema que utiliza el protocolo de techo de prioridad inmediata.
Se dispone de tres actividades A,B,C con prioridad A>B>C, y semáforos S1,S2,S3.

A utiliza S1 y S3 para secciones críticas de longitud 2 y 3 respectivamente.
B utiliza S2 y S3 (longitudes 2 y 1)
C utiliza S2 y S3 (longitudes 3 y 2)

26. El techo de prioridad de S1 y S2 es el mismo.	F
27. El techo de prioridad de S1 y S3 es el mismo.	V
28. El factor de bloqueo de B es 2.	F
29. Siempre que C está en ejecución y llega A al sistema, A expulsa a C	F
30. El techo de prioridad de S2 es igual a la prioridad de A.	F

Deseamos que un hilo (A) espere hasta que otros N hilos (H1..Hn) hayan ejecutado una sentencia B dentro de su código. Para resolverlo correctamente, podemos utilizar un:

31. Semaphore inicializado a N; A invoca *acquire*(), y H1..Hn *release*() tras la sentencia B.	F
32. Semaphore inicializado a N; A invoca *release*(), y H1..Hn *acquire*() tras la sentencia B.	F
33. CountDownLatch inicializado a N; A invoca *await*(), y H1..Hn *countDown*() tras la sentencia B.	V
34. CyclicBarrier inicializada a N; A invoca *await*(), y H1..Hn *signal*() tras la sentencia B.	F
35. CountDownLatch inicializado a 1; A invoca *countDown*(), y H1..Hn *await*() tras la sentencia B.	F

Sobre el estado de planificación de los hilos en Java:

36. *Thread.yield*() hace que un hilo pase del estado 'preparado' al estado 'suspendido'.	F
37. Si un hilo utiliza el método *wait*() de un objeto pasa del estado 'en ejecución' al 'preparado'.	F
38. Un hilo Java no puede pasar directamente del estado 'en ejecución' al estado 'preparado'.	F

Respecto a los interbloqueos:

39. Las situaciones de interbloqueo pueden prevenirse asignando los recursos de manera que nunca se genere una espera circular.	V
40. En un sistema disponemos de dos tipos de recursos (R1,R2) y 3 instancias en cada uno. Si todos los procesos piden una instancia de cada recurso, el máximo número de procesos para garantizar ausencia de interbloqueos es 2.	F
41. Las condiciones de Coffman son necesarias y suficientes (si se cumplen todas hay interbloqueo; si hay interbloqueo se cumplen todas).	F
42. Las situaciones de interbloqueo pueden evitarse si el sistema monitoriza las peticiones de recursos, denegando las que generen peligro de interbloqueo.	V

Suponemos que el código de todos los hilos de un determinado programa esta protegido por un mismo 'lock' (cada hilo lo cierra al empezar y lo abre justo antes de terminar)

43. Obtenemos paralelismo real.	F
44. Cumple trivialmente la propiedad de exclusión mutua (no es necesario reforzarlo en el código).	V
45. No hay condiciones de carrera.	V

Sobre el uso de variables Condition en Java:

46. En lugar de *wait()* y *notify()* utilizamos los métodos *await()* y *signal().*	V
47. Para crear un objeto Condition en un monitor protegido con ReentrantLock se invoca el método *newCondition*() del lock.	V
48. Antes de crear un objeto Condition hay que asegurarse de que no haya otra instancia de la clase Condition declarada en el mismo monitor.	F

Para construir monitores en Java...:

49. ... utilizamos los métodos *sleep*() e *interrupt*() (heredados de la clase Thread) para implantar sincronización condicional.	F
50. ... necesitamos utilizar la biblioteca *java.util.concurrent*	F
51. .. utilizamos los métodos *wait()*, *notify()* y *notifyAll()* (heredados de Object) para implantar exclusión mutua.	F

PREGUNTAS DE PRÁCTICAS

Sobre la práctica 3 "Problema de las hormigas":

1. Los métodos *lock*() y *unlock*() se utilizan para proporcionar exclusión mutua, lo que en la implementación inicial se consigue con la etiqueta *synchronized* de los métodos del Territory.	V
2. En la solución basada en una variable condición por cada celda, cada variable condición está asociada a un objeto ReentrantLock diferente.	F

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

3. Todos las versiones vistas en la práctica que solucionan los interbloqueos rompen alguna de las condiciones de Coffman.	V
4. Una de las soluciones planteadas consiste en que el filósofo espere a que sus dos tenedores estén libres antes de cogerlos.	V

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

5. En la piscina Pool1, si un niño quiere nadar y no hay ningún instructor nadando, invocará el método *notifyAll()* para avisar a los instructores.	F
6. En los casos de Pool1 a Pool4, cuando el método *kidSwims* retorna de su ejecución puede que el niño esté esperando a nadar.	F
7. En el caso de Pool2, un niño tiene que comprobar si hay instructores antes de invocar el método *kidSwims* de la piscina.	F