EXAMEN RECUPERACIÓN PRIMER BLOQUE **Concurrencia y Sistemas Distribuidos**
Unidades Didácticas 1 a 6 - Prácticas 1, 2 y 3 **Fecha: 7 de Junio de 2019**
Este examen tiene una duración total de 90 minutos.

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

Cada respuesta vale: correcta= 10/57, errónea= -10/57, vacía=0.

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

1. La cooperación de las tareas en los programas concurrentes se realiza mediante la comunicación y la sincronización.	V
2. Un programa concurrente requiere de máquinas con más de un procesador (o con varios núcleos) para poder ser ejecutado.	F
3. La programación secuencial aprovecha mejor los recursos de la máquina.	F

Respecto a la programación concurrente en Java:

4. Un hilo empieza a ejecutar el código indicado en su método *run*() en el momento en que creamos dicho hilo (ej. *new Thread(..)*)	F
5. Una aplicación concurrente termina cuando finaliza su hilo principal.	F
6. Cuando creamos en Java un objeto que debe compartirse entre distintos hilos, utilizamos la etiqueta *'synchronized'* (ej. *synchronized Buffer b*)	F
7. Utilizamos los métodos *wait*, *notify*, *notifyAll*, de forma apropiada, para resolver la sincronización condicional.	V
8. Todo objeto Java posee un *lock* y una variable condición implícitas.	V
9. El lenguaje Java proporciona por defecto monitores de tipo "Brinch Hansen".	F

Sobre el concepto de monitor y sus variantes:

10. Un monitor es un mecanismo de sincronización que garantiza la exclusión mútua en el acceso a sus métodos y proporciona herramientas para resolver la sincronización condicional.	V
11. Un monitor que siga el modelo de Lampson y Redell en caso de que haya hilos suspendidos en una condición "c", suspende al hilo que ha invocado a *c.notify*(), activando a uno de los hilos que llamó antes a *c.wait*().	F
12. Un monitor que siga el modelo de Brinch Hansen obliga a que en los métodos del monitor etiquetados con entry haya una sentencia *notify*() puesto que si no la hay los hilos no podrán salir del monitor.	F
13. Un programador que utilice un monitor que siga el modelo de Hoare debe declarar una variable condition denominada cola especial, para suspender a los hilos que invoquen una sentencia *notify*().	F
14. En un monitor Java, cuando un hilo H1 invoca *notify*() activa a un hilo H2 que está esperando en *wait*(), y éste (H2) cuando entre al monitor, ejecutará la primera instrucción de su método *run*().	F

Analice la siguiente propuesta de código y responda a las afirmaciones con V/F:

```
public class joinClass {

    private int MaxProcesses;
    private int numProcesses = 0;
    private boolean leavingProcesses = false;

    public joinClass(int nProcesses) { this.MaxProcesses = nProcesses;}

    public synchronized void myjoin () throws InterruptedException {
        while (leavingProcesses) wait();

        if (numProcesses == MaxProcesses - 1) {
            leavingProcesses = true;
            notifyAll();
        }
        else {
            numProcesses ++;
            while (!leavingProcesses) wait();
            numProcesses --;
            if (numProcesses == 0) {
                leavingProcesses = false;
                notifyAll();
            }
        }
        System.out.println(Thread.currentThread().getName());
    }
}
```

```
class MyThread extends Thread {
 private int i;
 private joinClass j;
  public MyThread (int a, joinClass b) {i=a; j=b;};
  public void run () {
    try {j.myjoin();} catch (InterruptedException e) {};
    // ---- "Rest of instructions"
  }

  public static void main(String[] args) {
         joinClass S = new joinClass(10);
         MyThread h;
         for (int i=0; i<50; i++) {
               h= new MyThread(i,S); h.setName("Thread"+i); h.start();
         };
  }
}
```

15. Se crean 50 hilos aparte del principal, y la cadena "Thread" forma parte del nombre.	V
16. La clase *joinClass* es un monitor.	V
17. Todo los hilos invocan el método *myjoin* del mismo objeto compartido S.	V
18. La clase joinClass gestiona que los hilos se esperen hasta formar un grupo de 10 para poder continuar su ejecución con el "Rest of instructions".	V
19. La solución garantiza que los grupos de hilos que se forman, lo hacen en el mismo orden en que invocaron el método myjoin. Es decir el primer grupo está formado por los 10 primeros hilos que invocaron *myjoin*, el segundo grupo por los 10 hilos siguientes que invocaron myjoin, y así sucesivamente.	F
20. Veremos por pantalla la secuencia de mensajes Thread0, Thread1 …. hasta Thread49 siempre en ese orden.	F

Supongamos que cambiamos la clase *joinClass* por la siguiente *joinClass2*, que usarían los hilos y el programa principal anterior:

```
import java.util.concurrent.locks.*;
public class joinClass2 {

    private int MaxProcesses;
    private int numProcesses = 0;
    private boolean leavingProcesses = false;
    ReentrantLock l=new ReentrantLock(true);
    Condition queue1=l.newCondition();
    Condition queue2=l.newCondition();

    public joinClass2(int nProcesses) {
        this.MaxProcesses = nProcesses;
    }

    public void myjoin () throws InterruptedException {
        try {
        l.lock();
        while (leavingProcesses) queue2.await();

        if (numProcesses == MaxProcesses - 1) {
            leavingProcesses = true;
            queue1.signalAll();
        }
        else {
            numProcesses ++;
            while (!leavingProcesses) queue1.await();
            numProcesses --;
            if (numProcesses == 0) {
                leavingProcesses = false;
                queue2.signalAll();
            }
        }
        System.out.println(Thread.currentThread().getName());
    } finally {l.unlock();}
    }
}
```

21. En *queue1* esperan los hilos a que se forme un grupo.	V
22. Los hilos que han esperado en *queue2* nunca esperan en *queue1*.	F
23. *queue2* se utiliza para esperar a que salgan los hilos de un grupo ya formado.	V
24. La clase *joinClass2* gestiona que los hilos se esperen hasta formar un grupo de 10 para poder continuar su ejecución con el "Rest of instructions".	V
25. Veremos por pantalla la secuencia de mensajes Thread0, Thread1 …. hasta Thread49 en cualquier orden.	V

Si los hilos y el programa principal utilizasen la siguiente versión de la clase:

```
import java.util.concurrent.*;
public class joinClass3 {
    CyclicBarrier myBarrier;
    public joinClass3(int nProcesses) {
        myBarrier = new CyclicBarrier(nProcesses, new Runnable(){
           public void run(){System.out.println("wake up to "+ nProcesses+
                                                " processes");}});
    }

    public void myjoin () throws InterruptedException, BrokenBarrierException
{
        myBarrier.await();
    }
}
```

26. Veremos por pantalla 3 mensajes “wake up to 10 processes”.	F
27. La clase *joinClass3* gestiona que los hilos se esperen hasta formar un grupo de 10 para poder continuar su ejecución con el “Resto de instrucciones”.	V

Respecto a los interbloqueos:

28. Las condiciones de Coffman son necesarias y suficientes (si se cumplen todas hay interbloqueo; si hay interbloqueo se cumplen todas)	F
29. 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 6.	F
30. En un grafo de asignación de recursos si existe un ciclo dirigido podemos afirmar que existe interbloqueo.	F
31. En un grafo de asignación de recursos si existe una secuencia de finalización podemos afirmar que no hay interbloqueo.	V
32. Las condiciones de Coffman permiten diseñar sistemas que cumplan con todas ellas, para así garantizar que no se producirán interbloqueos.	F

Un sistema de tiempo real es un sistema informático en el que se cumplen, entre otras, estas características:

33. Es concurrente, pues está constituido por múltiples actividades o tareas.	V
34. Responde a estímulos de su entorno; debe hacerlo de manera determinista y dentro de un plazo.	V

Un sistema de tiempo real crítico se compone de 4 tareas que utilizan 4 semáforos para sincronizar el acceso a sus secciones críticas (SC), y cuyas características se describen en las siguientes tablas:

techo(S1) = 1.
techo(S3= = 1.
techo(S2) = 2.
techo(S4) = 4.

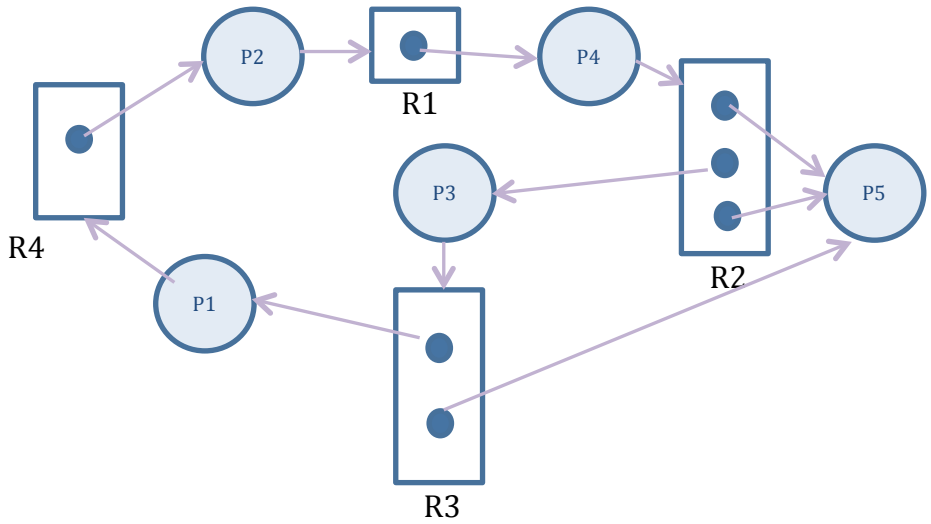
Tarea	Tiempo de cómputo	Periodo	Plazo
T1	4	20	15
T2	8	30	20
T3	12	40	28
T4	15	120	80

Tarea	Semáforo	Duración de la SC
T1	S1	3
T1	S3	4
T2	S2	6
T3	S2	5
T3	S3	1
T4	S1	2
T4	S4	7

Asumiendo que los semáforos utilizan el protocolo del techo de prioridad inmediato y el sistema se planifica por prioridades fijas expulsivas, con asignación de prioridades inversamente proporcional a su plazo, donde la tarea de menor plazo es la más prioritaria….

35. … el tiempo de respuesta de la tarea T1, R_1 es 6. R1 = 4 + B1 = 6 B1 = 2 (s1 de T4)	V
36. … los techos de los semáforos S1 y S3 son iguales Sí.	V
37. … el factor de bloqueo de T1, y T3 es 2. B3 = 2 (S1 de T4)	V
38. … el tiempo de respuesta de la tarea T3, R_3 es 28. R03=4+8+12+B3=24+2=26. R13 = 12 + [26/30]8 + [26/20]4 + B3 = 12 + 8 + 8 = 28+2 =30.	F
39. … el tiempo de respuesta de la tarea T4, R_4 es 79. r04 = 4+8+12+15+b4=39+0=39. r14 = 15 + [39/40]12 + [39/30]8 + [39/20]4 = 51. r24 = 15+[51/40]12 + [51/30]8 + [51/20]4 = 15+24+16+12=67. r34=79 = r44.	V
40. … todos los plazos están garantizados y, por tanto, el sistema es planificable. r3 = 30 y D3 = 28 -> NO es planificable.	F
41. … el factor de bloqueo de T2 es 5.	V

Dado el siguiente grafo de asignación de recursos….



42. Existe un interbloqueo entre varios procesos.	F
43. Como existe un ciclo dirigido en el que hay recursos con una única instancia, podemos afirmar que existe un interbloqueo.	F
44. Podemos encontrar la siguiente secuencia de terminación: < P5, P3, P4, P2, P1>.	V

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:

45. *CountDownLatch cl* inicializado a N; A invoca *cl.await*(), y H1..Hn invocan cada uno cl.*countDown*() tras la sentencia B.	V
46. *CyclicBarrier cb* inicializada a N; A invoca *cb.await*(), y H1..Hn invocan cada uno *cb.signal*() tras la sentencia B.	F
47. *CountDownLatch cl* inicializado a 0; A invoca *cl.countDown*(), y H1..Hn invocan cada uno *cl.await()* tras la sentencia B.	F
48. *Semaphore sem* inicializado a 0; A invoca *sem.acquire()*, y H1..Hn invocan cada uno *sem.release()* tras la sentencia B.	F
49. *Semaphore sem* inicializado a 0; A invoca N veces *sem.acquire()*, y H1..Hn invocan cada uno *sem.release()* tras la sentencia B.	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

50. No es necesario poner la etiqueta *synchronized* en los métodos de la clase Pool0 porque en esta piscina no hay reglas asociadas.	V
51. En la piscina Pool1 no es necesario suspender a ningún hilo, al no existir sincronización condicional.	F
52. Si escribimos *notifyAll*() al final de todos los métodos de la piscina Pool2, en algunos casos se despertará a hilos que no era necesario despertar.	V

Sobre la práctica 2 "Los cinco filósofos comensales":

53. Limitar el número de filósofos a la mesa es una solución incorrecta, pues puede producir inanición al filósofo que no se puede sentar.	F
54. En la solución basada en limitar el número de filósofos en la mesa, los filósofos deben necesariamente coger primero el tenedor de la derecha.	F
55. Hay una solución basada en que los filósofos pares cojan un tenedor primero (ej. derecho), y los impares empiecen cogiendo el otro tenedor (ej. izquierdo).	V

Sobre la práctica 3 "Problema de las hormigas":

56. En el problema de las hormigas, en una de las actividades del boletín de la práctica se solicita implementar un cerrojo ReentrantLock por celda.	F
57. En un terreno con muchas hormigas, es más eficiente disponer de una variable Condition por celda, respecto a tener una única variable Condition para todo el terreno.	V