EXAMEN FINAL DE RECUPERACIÓN – Concurrencia y Sistemas Distribuidos
Unidades Didácticas 1 a 11 **23 de Junio de 2014**

Modelo A

NOMBRE:		**DNI:**	

Este bloque tiene una puntuación máxima de **10 puntos**, que equivalen a 5 puntos de la nota final de la asignatura. Indique, para cada una de las siguientes 50 afirmaciones, si éstas son verdaderas (**V**) o falsas (**F**).

Cada respuesta vale: correcta= 0.2, errónea= -0.2, 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.

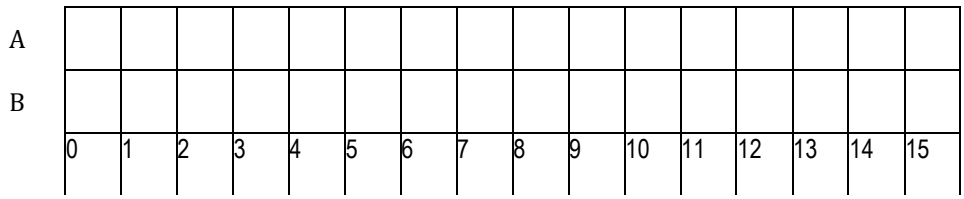
Atención: La columna V/F que aparece en este documento sólo sirve a efectos de facilitar el proceso de realización del examen, de modo que no se tendrá en cuenta lo ahí puesto en la corrección del examen. **Solamente se corregirá la hoja de respuestas, que se proporciona aparte.**

Sean dos tareas A, y B, con prioridad(A)>prioridad(B); y dos semáforos Q y R que utilizan el protocolo del techo de prioridad inmediato, tales que:

Tarea	Prioridad	Instante Activación	Patrón Ejecución
A	1	2	ERRRQQE
B	2	0	EQQRRRE

E representa una unidad de ejecución sin acceso a sección crítica alguna, Q una unidad de ejecución con bloqueo del semáforo Q, R una unidad de ejecución con bloqueo del semáforo R; cada tarea bloquea el semáforo una única vez.

Para contestar a las cuestiones puede ayudarse utilizando la cuadrícula adjunta, donde se ilustre gráficamente la ejecución de las tareas.



	V/F
1. La tarea B está en ejecución en el intervalo [0,2).	V
2. La tarea A está en ejecución en el intervalo [2,6).	F
3. En el instante t=6 se produce un interbloqueo entre A y B.	F
4. En el intervalo [1,6) la prioridad de la tarea B es 1.	V
5. La tarea B finaliza su ejecución en el instante 14.	V

Dado el siguiente código:

```
1.1  public class Buffer {
1.2    private int store = 0;
1.3    private boolean full = false;
1.4
1.5    public int get()   {
1.6        int value = store;
1.7        store = 0;
1.8        full = false;
1.9        return value;
1.10   }
1.11
1.12   public void put(int value)    {
1.13       full = true;
1.14       store = value;
1.15   }
1.16 }

2.1  public class MainThread {
2.2     public static void main(String[] args)  {
2.3        Buffer c = new Buffer();
2.4        Consumer c1 = new Consumer(c, 1);
2.5        Producer p1 = new Producer(c, 2);
2.6        c1.start();
2.7        p1.start();
2.8        System.out.println("Main Thread finishes.");
2.9     }
2.10 }
```

```
3.1  public class Consumer extends Thread {
3.2     private Buffer b;
3.3     private int number;
3.4     public Consumer(Buffer ca, int id)  {
3.5        b = ca;
3.6        number = id;
3.7     }
3.8
3.9     public void run()  {
3.10       int value = 0;
3.11       for (int i = 1; i < 101; i++)
3.12       {
3.13          try {
3.14                sleep((int)(Math.random() * 100));
3.15           } catch (InterruptedException e) { }
3.16        value = b.get();
3.17        System.out.println("Consumer #" + number  +
                " gets: " + value);
3.18     }
3.19    }
3.20 }

4.1  public class Producer extends Thread {
4.2     private Buffer b;
4.3     private int number;
4.4     public Producer(Buffer ca, int id) {
4.5        b = ca;
4.6        number = id;
4.7     }
4.8     public void run()  {
4.9        for (int i = 1; i < 101; i++)
4.10       {
4.11          b.put(i);
4.12          System.out.println("Producer #" + number +
              " puts: " + i);
4.13       try {
4.14            sleep((int)(Math.random() * 100));
4.15       } catch (InterruptedException e) { }
4.16       }
4.17    }
4.18 }
```

6. En este código se ha aplicado la sincronización condicional haciendo uso del método sleep.	F
7. Este código es determinista pues, como el consumidor se queda dormido inicialmente, nunca se podrán producir condiciones de carrera.	F
8. Al ejecutar este código, puede ocurrir que alguno de los valores producidos por el hilo "Producer" no sea consumido por el hilo "Consumer".	V
9. Para implantar la sincronización condicional, entre otras modificaciones, se requiere añadir el siguiente código entre las líneas 1.8 y 1.9: **while (full) try {** **wait();** **} catch (InterruptedException e) {};**	F
10. Para evitar que se produzcan condiciones de carrera, se debe añadir la etiqueta "synchronized" a todos los métodos de las clases Consumer y Producer.	F
11. Si se implanta la exclusión mutua y la sincronización condicional en la clase Buffer, entonces dicha clase actuará como un monitor.	V

Dado el código anterior, se desea que el hilo principal del programa escriba su mensaje cuando tanto el hilo consumidor como el hilo productor hayan finalizado su trabajo.

12. Una solución correcta consiste en añadir el siguiente código antes de la línea 2.8: *p1.join(); c1.join();*	V
13. Se puede hacer uso de un Semaphore "sem", inicializado a 1, utilizando como protocolo de entrada "*sem.acquire*()" y como protocolo de salida "*sem.realease*()".	F
14. Si en vez de la clase Buffer se hubiese empleado una colección concurrente, del tipo "BlockingQueue", dicho problema estaría resuelto.	F
15. Se puede implementar una solución con un CountDownLatch "cdl" iniciado a 2, de modo que tanto Producer como Consumer realizan "*cdl.countDown*()" al final de su método *run*(), mientras que el hilo principal realiza "*cdl.await*()" justo antes de escribir su mensaje.	V
16. Se puede implementar una solución con un CyclicBarrier "cb" iniciado a 3, de modo que tanto Producer como Consumer realizan "*cb.await*()" al final de su método *run*(), mientras que el hilo principal realiza "*cb.await*()" justo antes de escribir su mensaje.	V
17. Se puede implementar una solución con un ReentrantLock "rl", de modo que tanto Producer como Consumer realizan "*rl.unlock*()" al final de su método *run*(), mientras que el hilo principal realiza "*rl.lock*()" justo antes de escribir su mensaje.	F
18. Se puede implementar una solución con una variable Condition "cond", de modo que tanto Producer como Consumer realizan "*cond.signal*()" al final de su método *run*(), mientras que el hilo principal realiza dos veces "*cond.await*()" (dentro de un bloque "try") antes de escribir su mensaje.	F

Respecto a los monitores:

19. Un monitor es un mecanismo de sincronización proporcionado directamente por el sistema operativo.	F
20. Un monitor que siga el modelo de Lampson y Redell en caso de que haya hilos suspendidos en alguna condición, suspende al hilo que ha invocado a *notify*(), activando a uno de los hilos que llamó antes a *wait*().	F
21. Un monitor que siga el modelo de Hoare obliga a que toda invocación a notify() sea la última sentencia en los métodos del monitor en los que aparezca tal invocación.	F
22. Un monitor que siga el modelo de Hoare suspende (y deja en una cola especial de entrada) al hilo que ha invocado a *notify*(), activando a uno de los hilos que llamó antes a *wait*().	V

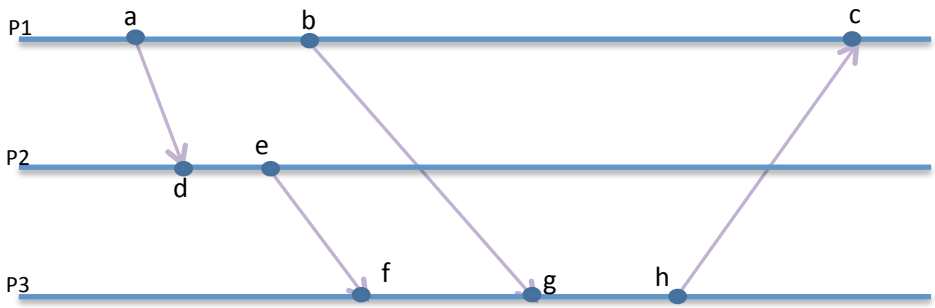
En un grafo de asignación de recursos:

23. Se da un interbloqueo cuando existe un ciclo dirigido y los recursos tienen más de una instancia.	F
24. Si en dicho grafo existe un ciclo dirigido, podemos afirmar que hay interbloqueo si existe una secuencia segura.	F
25. Cuando varios procesos se encuentran esperando porque no existe ninguna instancia libre del recurso que necesitan, podemos afirmar que existe interbloqueo.	F

Sobre los relojes lógicos de Lamport:

26. Si dos eventos tienen el mismo valor de reloj, entonces son concurrentes.	V
27. Si dos eventos son concurrentes, entonces tienen el mismo valor de reloj.	F
28. Si un evento ocurre antes que otro, el primero tendrá un valor de reloj menor.	V
29. Requieren que se envíen mensajes entre los nodos de forma periódica para mantener dichos relojes sincronizados.	F

Dado el siguiente diagrama temporal de eventos… (se asume que no hay eventos previos):



30. El valor del reloj lógico de Lamport del evento “g” es 5.	V
31. El valor del reloj vectorial del evento “h” es [2,2,3]	V
32. Podemos observar que los eventos “b” y “d” son concurrentes.	V
33. Podemos deducir que el valor del reloj lógico de “c” en ocasiones será 7 y en ocasiones tendrá otro valor.	F
34. Se cumple que f -> c y que e||b	V

Sobre la localización de entidades en sistemas distribuidos:

35. El mecanismo de punteros hacia delante escala peor que el mecanismo de difusiones.	F
36. ARP es un ejemplo de localización por punteros hacia delante.	F
37. Un problema del mecanismo de localización por punteros hacia delante radica en que se deben hacer difusiones para encontrar partes de la cadena.	F
38. Nunca es necesaria cuando disponemos de un servicio de nombres.	F

Sobre la invocación a objetos remotos (ROI):

39. El proxy ofrece la misma interfaz que el esqueleto.	F
40. Existe un esqueleto por cada método del objeto remoto.	F
41. El proxy empaqueta los argumentos del método llamado antes de que el esqueleto invoque al objeto remoto.	V
42. El esqueleto desempaqueta los argumentos enviados por el proxy antes de invocar al objeto remoto.	V

Respecto a los algoritmos de sincronización de relojes:

43. En el algoritmo de Berkeley no se asume que exista un ordenador con un reloj preciso.	V
44. Cuando se utiliza el algoritmo de Cristian, en la expresión Cs + (T1 -T0)/2 , T1 es el instante en el que el cliente pide el valor del reloj del servidor.	F
45. Permiten asignar valores numéricos a los eventos de envío y recepción de mensajes a partir de la relación “ocurre-antes”.	F
46. En general es problemático hacer que el reloj de un ordenador retroceda.	V

Sobre los sistemas distribuidos:

47. La transparencia de ubicación oculta el hecho de que un recurso esté ubicado en memoria volátil o en memoria persistente.	F
48. Para mejorar la escalabilidad administrativa hay que conseguir que todos los ordenadores del sistema pertenezcan a una única organización.	F
49. La capa de middleware puede integrar algunos mecanismos de comunicación de alto nivel que faciliten la programación de aplicaciones distribuidas; por ejemplo: RPC.	V
50. Las arquitecturas de sistema para sistemas distribuidos representan la implementación física de los componentes software del sistema en máquinas reales.	V