

Prueba final 1 - Clave a
Concurrencia
 2013-2014 - Segundo semestre
 Dpto. de Lenguajes, Sistemas Informáticos e Ingeniería de Software
 Universidad Politécnica de Madrid

Normas

Este es un cuestionario que consta de **7 preguntas** en **4 páginas**. Todas las preguntas son **preguntas de respuesta simple** excepto la pregunta 7 que es una **pregunta de desarrollo**. La puntuación total del examen es de **11 puntos**. La duración total es de **una hora**. El examen debe contestarse en las **hojas de respuestas**. No olvidéis rellenar **apellidos, nombre, DNI y clave** en cada hoja de respuesta.

Sólo hay una respuesta válida a cada pregunta de respuesta simple. Toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada. Toda pregunta incorrectamente contestada restará del examen una cantidad de puntos igual a la puntuación de la pregunta dividido por el número de alternativas ofrecidas en la misma.

Cuestionario

- (1 punto) 1. Supóngase que `T` es una clase correctamente implementada que hereda de `java.lang.Thread`. Supóngase que un programa en ejecución con un solo thread ejecuta las sentencias `T t = new T(); t.run();`. ¿Cuántos threads en ejecución tendrá el programa inmediatamente después de ejecutar estas sentencias?
- (a) 1.
(b) 2.
- (1 punto) 2. El siguiente código, una variación del algoritmo de Peterson, pretende garantizar la exclusión mutua en el acceso a las secciones críticas `n++`; y `n--`; desde, respectivamente, dos *threads*:

<pre>static class MutexEA { static final int N_PASOS = 1000000; // Variable compartida volatile static int n = 0; // Variables para asegurar mutex volatile static boolean en_sc_inc = false; volatile static boolean en_sc_dec = false; static class Incrementador extends Thread { public void run () { for (int i = 0; i < N_PASOS; i++) { en_sc_inc = true; while (en_sc_dec) { } n++; en_sc_inc = false; } } } }</pre>	<pre>static class Decrementador extends Thread { public void run () { for (int i = 0; i < N_PASOS; i++) { while (en_sc_inc) { } en_sc_dec = true; n--; en_sc_dec = false; } } } public static final void main(final String[] args) throws InterruptedException { Thread t1 = new Incrementador(); Thread t2 = new Decrementador(); t1.start(); t2.start(); t1.join(); t2.join(); }</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Se pide señalar la respuesta correcta.

- (a) No se garantiza la propiedad de exclusión mutua.
 (b) Se garantiza la propiedad de exclusión mutua.

- (1 punto) 3. Dada la siguiente clase *thread* en la que `Dato` es una clase con atributos no estáticos de tipo `int` y `m` un método que accede y modifica sólo dichos atributos:

<pre>static class T extends Thread { volatile private Dato y; public T (Dato y) { this.y = y; } }</pre>	<pre>public void run() { Dato y = new Dato(); y.m(); }</pre>
-----------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------

Se pide marcar la afirmación correcta.

- (a) `y.m()` nunca es una sección crítica.
 (b) `y.m()` puede ser una sección crítica.
- (1 punto) 4. Supongamos un programa concurrente con cuatro tipos de thread T_0, T_1, T_2 y T_3 que ejecutan repetidamente operaciones $r.Cero, r.Uno, r.Dos$ y $r.Tres$, respectivamente. Suponiendo que existe al menos un thread de cada tipo y que r es un recurso compartido del tipo especificado a continuación¹:

C-TAD Circular

TIPO: $Circular = (a : \mathbb{B} \times b : \mathbb{B})$

INVARIANTE: $\neg self.a \vee \neg self.b$

INICIAL: $self = (Falso, Falso)$

CPRE: Cierto

Cero

POST: $self = (self^{pre}.a, self^{pre}.b)$

CPRE: $\neg self.a \wedge \neg self.b$

Uno

POST: $self = (self^{pre}.a, \neg self^{pre}.b)$

CPRE: $\neg self.a \wedge \neg self.b$

Dos

POST: $self = (\neg self^{pre}.a, self^{pre}.b)$

CPRE: $self.a \vee self.b$

Tres

POST: $self = (\neg self^{pre}.a, \neg self^{pre}.b)$

Se pide señalar la respuesta correcta.

- (a) En el programa se puede violar la invariante de r .
 (b) En el programa no se puede violar la invariante de r .
- (1½ puntos) 5. Supóngase que se eliminan los threads de tipo T_3 (aquellos que ejecutaban $r.Tres$) del programa concurrente descrito en la pregunta 4. Se pide señalar el número máximo de estados por los que puede pasar el recurso r para una ejecución determinada del programa:
- (a) 2
 (b) 3
 (c) 4

- (1½ puntos) 6. Supóngase un programa concurrente con un semáforo inicializado a 0. Tres threads invocan el método `await` sobre dicho semáforo y posteriormente otros dos threads invocan el método `signal` sobre el mismo semáforo. Supóngase que el programa se detiene sin realizar más invocaciones de métodos sobre dicho semáforo.

Se pide marcar la afirmación correcta.

- (a) El valor del semáforo es 0 al detener el programa.
 (b) El valor del semáforo es 1 al detener el programa.
 (c) El valor del semáforo es 2 al detener el programa.

¹No se muestra la declaración de operaciones.

- (4 puntos) 7. Está a punto de inaugurarse el primer centro comercial en el que la compra la realizan robots: e -QEA. Los compradores envían sus listas de la compra y sus robots inician el recorrido. e -QEA lo componen un número fijo de tres de secciones adyacentes. Los robots inician la compra entrando en la sección 0, avanzan a la sección adyacente (0, 1, 2) cuando han acumulado los productos requeridos de esa sección y terminan su compra saliendo de la sección 2.

El centro e -QEA tiene un problema estructural (literalmente): cada sección soporta un peso máximo (P). Esto significa que cuando un robot con un determinado peso (p) quiere avanzar, debe esperar hasta que el incremento de peso que provoca no ponga en peligro la estructura (si el peso actual de la sección adyacente es $peso$ entonces un robot con peso p no puede avanzar si $peso + p > P$).

Se desea crear un recurso compartido para gestionar el movimiento de robots por e -QEA. Los *threads* que controlan los robots ejecutan la operación $avanzar(s, p)$ cuando el robot quiere avanzar de la sección $s - 1$ a la sección s portando un peso p (con $s = 0$ se indica la entrada al centro y con $s = 2$ la salida del mismo).

Se pide completar la especificación de un recurso compartido *Controle-QEA* que controle el avance de los robots por el centro e -QEA:

C-TAD *Controle-QEA*

OPERACIONES

ACCIÓN avanzar: $TipoSección[e] \times \mathbb{N}[e]$

SEMÁNTICA

DOMINIO:

TIPO: *Controle-QEA* =

TIPO: *TipoSección* = $\{0, 1, \dots, N\}$

INVARIANTE:

INICIAL:

PRE:

CPRE:

avanzar(s,p)

POST:

(Página intencionadamente en blanco, puede usarse como hoja en sucio).