Apellidos: **Nombre:** **Matrícula:**

Concurrencia (parte 1)/clave: a

Curso 2013-2014 - 2o. semestre (junio 2014)
Grado en Ingeniería Informática/Grado en Matemáticas e Informática
UNIVERSIDAD POLITÉCNICA DE MADRID

NORMAS: Este es un cuestionario que consta de 7 preguntas. Todas las preguntas son de respuesta simple excepto la pregunta 7 que es una pregunta de desarrollo. La puntuación total del examen es de 10 puntos. La duración total es de una hora. El examen debe contestarse en las hojas de respuestas. No olvidéis rellenar apellidos, nombre y número de matrícula 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*:

```
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;
       }
     }
   }
```

```
   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();
   }
}
```

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:

```
static class T extends Thread {
   volatile private Dato y;
   public T (Dato y) {
      this.y = y;
   }
   public void run() {
      Dato y = new Dato()
      y.m();
   }
}
```

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 continuación:[1]

C-TAD Circular
TIPO: $Circular = (a : \mathbb{B} \times b : \mathbb{B})$
INVARIANTE: $\neg \text{self}.a \vee \neg \text{self}.b$

INICIAL: self = (Falso, Falso)

CPRE: Cierto
Cero
POST: self = $(\text{self}^{pre}.a, \text{self}^{pre}.b)$

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

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

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

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* **en una ejecución determinada del programa**. (Sugerencia: usa el espacio en blanco de la última página para dibujar el grafo de estados.)

(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.

[1] No se muestra la declaración de operaciones.

Apellidos: **Nombre:** **Matrícula:**

(3 puntos) 7. **Se pide** completar la siguiente especificación de un recurso compartido que generaliza el comportamiento de semáforos contadores. Por una parte, las operaciones *await* y *signal* permiten hacer/liberar *n* reservas de permisos de manera indivisible. Por otro lado, existe una operación para dar un valor inicial al semáforo. Si un proceso invoca *await* sobre un semáforo no inicializado, debe esperar, pudiendo continuar si posteriormente es inicializado, pero no se permiten las llamadas a *signal* sobre semáforos sin inicializar ni las llamadas a *inicializar* sobre un semáforo ya inicializado (podéis usar para ello la cláusula PRE).

C-TAD SemaforoGen

OPERACIONES

ACCIÓN inicializar: $\mathbb{N}$*[i]*
ACCIÓN await: $\mathbb{N}$*[i]*
ACCIÓN signal: $\mathbb{N}$*[i]*

SEMÁNTICA
DOMINIO:

TIPO: *SemaforoGen* =

INVARIANTE:

INICIAL:

PRE:

CPRE:

inicializar(n)

POST:

CPRE:

await(n)

POST:

PRE:

CPRE:

signal(n)

POST:

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