Apellidos: **Nombre:** **Matrícula:**

Concurrencia (parte 1)/clave: a (Solución)

Curso 2018/2019 - Convocatoria Ordinaria (Mayo 2019)
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 **mismas hojas**. No olvidéis rellenar **apellidos, nombre y número de matricula** en cada hoja. 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

(1½ puntos) 1. Suponiendo que hay tres procesos $t1$, $t2$ y $t3$, que cada uno ejecuta repetidamente las operaciones *uno*, *dos*, *tres* del CTAD que se especifica a continuación:

TIPO: MiCTAD $= (a : \mathbb{B} \times b : \mathbb{B})$
INICIAL: $self = (\text{Falso}, \text{Falso})$
INVARIANTE: $\neg \text{self}.a \vee \neg \text{self}.b$

CPRE: $\neg self.a$
uno()
POST: $self^{pre} = (c, d) \wedge self = (c, \neg d)$
CPRE: $self.a \wedge \neg self.b$
dos()
POST: $self^{pre} = (c, d) \wedge self = (c, \neg d)$
CPRE: $\neg \text{self}.b$
tres()
POST: $self^{pre} = (c, d) \wedge self = (\neg c, d)$

Se pide marcar la afirmación correcta:

(a) ☐ El programa podría acabar en interbloqueo pero nunca se violaría la invariante

(b) ☐ El programa podría violar la invariante pero nunca acabaría en interbloqueo

(c) ☑ El programa podría violar la invariante y además acabar en interbloqueo

(Sugerencia: Dibuja aquí el grafo de los estados por los que puede pasar el recurso.)

(1 punto) 2. Dadas las clase `MiHilo` y el siguiente `main`:

```
class MiHilo extends Thread {
    private static int n = 0;

    public void run () {
        n ++;
    }
}
```

```
public static void main (String [] args) {
    Thread t1 = new MiHilo ();
    Thread t2 = new MiHilo ();

    t1.start();
    t1.join();
    t2.start();
    t2.join();
    t1.run();
}
```

Se pide marcar la afirmación correcta:

(a) ☑ No se producirán condiciones de carrera en el acceso a `n` y el valor de `n` al final de `main` será 3

(b) ☐ No se producirán condiciones de carrera en el acceso a `n` y el valor de `n` al final de `main` será 1

(c) ☐ Se pueden producir condiciones de carrera en el acceso a `n`

(1 punto) 3. Dado el `main` de la pregunta anterior ¿cuántos hilos *como máximo* habría simultáneamente ejecutando en el programa?

(a) ☐ 4 hilos.

(b) ☑ 2 hilos.

(c) ☐ 3 hilos.

(1½ puntos) 4. Dado un programa concurrente en la que tres *threads* instancias de las clases `A`, `B` y `C` comparten una variable `n`:

```
volatile int n = 0;
static Semaphore s1 = new Semaphore(2);
static Semaphore s2 = new Semaphore(0);
static Semaphore s3 = new Semaphore(0);
```

```
static class A
extends Thread {
  public void run() {
    s2.await();
    n = n * 2;
    s1.signal();
  }
}
```

```
static class B
extends Thread {
  public void run() {
    s1.await();
    s3.await();
    n = n * n;
    s2.signal();
  }
}
```

```
static class C
extends Thread {
  public void run() {
    s1.await();
    n = n + 2;
    s3.signal();
  }
}
```

¿Cuáles son los posibles valores de `n` tras **terminar** los tres threads?

(a) ☐ 0 y 8

(b) ☐ 4

(c) ☑ 8

(1 punto) 5. Si en el código anterior el semáforo `s1` se inicializa a 1:

(a) ☑ No está garantizada la terminación de las tres tareas.

(b) ☐ No está garantizada la exclusión mutua en el acceso a `n`.

Apellidos: **Nombre:** **Matrícula:**

(1 punto) 6. Dado el siguiente código que trata de implementar exclusión mutua en los accesos a la variable compartida `n`:

```
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++) {
            en_sc_dec = true;
            while (en_sc_inc) { }
            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 marcar la afirmación correcta:

(a) ☐ El programa no garantiza la exclusión mutua

(b) ☑ El programa puede acabar en un interbloqueo.

(3 puntos) 7. Se está desarrollando un sistema concurrente de ordenación por burbuja (*bubblesort*). En este, $N-1$ procesos permutadores se encargan de comparar y, si es necesario, permutar pares de elementos adyacentes de un vector de tamaño N. Para evitar situaciones de carrera en el acceso al vector se va a disponer de un recurso con operaciones para reservar o liberar el acceso a posiciones consecutivas en el vector.

Así, el proceso *permutador*(i) ejecutaría, en bucle, la siguiente secuencia:

reservar(i); $\langle$ **si** $v[i] > v[i+1]$ **entonces** *permutar*$(v[i], v[i+1])$ $\rangle$; *liberar*(i);

La operación *reservar*(i) impide que otro proceso pueda acceder a las posiciones i e $i+1$. Para ejecutarla, es necesario que ambas posiciones no estén siendo usadas por otro proceso permutador (la llamada bloquearía hasta que esa condición se dé). La operación *liberar*(i) vuelve a dejar disponibles las posiciones i e $i+1$. No se permite que un proceso invoque *liberar* sobre posiciones que no estén reservadas (podéis usar la clásula PRE para expresar esta prohibición).

Se pide: completar el siguiente CTAD, de acuerdo con el comportamiento arriba explicado.[1]

C-TAD Burbuja

OPERACIONES

ACCIÓN reservar: *Indice[e]*
ACCIÓN soltar: *Indice[e]*

SEMÁNTICA
DOMINIO:
TIPO: $Burbuja = Indice \nrightarrow \mathbb{B}$
TIPO: $Indice = \{0 \ldots N-1\}$
INVARIANTE: $| \{i \in Indice \bullet self(i)\} | \bmod 2 = 0$

INICIAL: $\forall\, i \in Indice \bullet \neg self(i)$

CPRE: $\neg \text{self}(i) \wedge \neg \text{self}((i+1) \bmod N)$
reservar(i)
POST: $\text{self} = \text{self}^{pre} \oplus \{i \mapsto \text{Cierto}\} \oplus \{(i+1) \bmod N \mapsto \text{Cierto}\}$

PRE: $\text{self}(i) \wedge \text{self}((i+1) \bmod N)$
CPRE: Cierto
liberar(i)
POST: $\text{self} = \text{self}^{pre} \oplus \{i \mapsto \text{Falso}\} \oplus \{(i+1) \bmod N \mapsto \text{Falso}\}$

Se ha tenido en cuenta la representación de la información (1 punto), las postcondiciones (0.5), las condiciones de sincronización (1) y el aspecto sintáctico y formal (0.5).

[1] En un sistema de ordenación real, habría que añadir funcionalidad al recurso para que los procesos se puedan detener cuando el vector esté ordenado. Todo esto ha sido omitido para simplificar el ejercicio.