Pseudónimo:

Concurrencia (parte 1, salvo recursos)

Curso 2017–2018 - Prueba de autoevaluación (6 de abril de 2018)
Grado en Ingeniería Informática
UNIVERSIDAD POLITÉCNICA DE MADRID

NORMAS: Este es un cuestionario que consta de 7 preguntas. Todas las preguntas son de respuesta simple. La puntuación total del examen es de $7\frac{1}{2}$ puntos. La duración total es de 45 minutos. Esta es una prueba de autoevaluación *anónima*. Podéis usar un pseudónimo si queréis (útil en caso de que entreguéis y publique los resultados).

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. ¿Cuál es el método de `java.lang.Thread` que pone en marcha un thread?

(a) `run`.

(b) `start`.

(1 punto) 2. Dadas las siguientes clases `MiHilo` y `Dato`:

```
class MiHilo extends Thread {
    private Dato d;

    public class MiHilo(Dato d) {
        this.d = d;
    }
    public void run () {
        d.m();
    }
}
```

```
class Dato {
    private int n;

    public class Dato () {
        n = 0;
    }
    public void m () {
        n++;
    }
}
```

Se pide marcar la afirmación correcta:

(a) La ejecucíón del método `m()` puede ser una sección crítica

(b) La ejecucíón del método `m()` nunca será una sección crítica

(1 punto) 3. Dado el siguiente `main` ¿cuántos hilos *como máximo* habría simultáneamente ejecutando en el programa?:

```
public static void main(String[] args) {
  MiHilo t1 = new MiHilo (new Dato());
  t1.start();
  t1.run();
}
```

Se pide marcar la afirmación correcta:

(a) 1 hilo.

(b) 2 hilos.

(c) 3 hilos.

($1\frac{1}{2}$ puntos) 4. Dado un programa concurrente en el que dos *threads* instancias de las clases `A` y `B`, `C` comparten una variable `n`:

```
static int n = 0;
static Semaphore s1 = new Semaphore(1);
static Semaphore s2 = new Semaphore(0);
```

```
static class A
  extends Thread {
    public void run() {
        s2.await();
        n = 3 * n;
        s1.signal();
    }
}
```

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

```
static class C
  extends Thread {
    public void run() {
        s1.await();
        n = n + 4;
        s2.signal();
    }
}
```

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

(a) 6, 10, 14 y 18

(b) 14 y 6

(c) 10 y 14

(1 punto) 5. Si en el código anterior el semáforo `s1` se inicializa a 2:

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

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

(1 punto) 6. Se desea modelar con semáforos una barrera para hilos con la cual se bloquearán todos los hilos hasta que haya llegado el último, momento en el cual permitiremos seguir a todos los hilos.

```
static class Hilos{
  static final int MAX_HILOS = 5;
  static class MiHilo
  extends Thread{
    static volatile int cont = 0;
    static Semaphore mutex = new Semaphore(1);
    static Semaphore barrera = new Semaphore(0);
    public void run(){
      tarea1();

      mutex.await();
      cont = cont + 1;
      mutex.signal();

      if (cont == MAX_HILOS)
        barrera.signal();

      barrera.await();
      tarea2();
    }
  }
}
```

Supongamos que lanzamos `MAX_HILOS` hilos instancias de `MiHilo`. Asumiendo que los métodos `tarea1()` y `tarea2()` siempre terminan, **se pide** marcar la afirmación correcta.

(a) Se pueden producir condiciones de carrera en la variable `cont`.

(b) El sistema siempre acabará en un interbloqueo.

(c) El sistema podría comportarse como esperamos si los hilos ejecutan sus instrucciones en un orden determinado.

Pseudónimo:

(1 punto) 7. 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) {
               en_sc_inc = false;
               en_sc_inc = true;
            }
            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) {
               en_sc_dec = false;
               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 marcar la afirmación correcta:

(a) El programa no garantiza la exclusión mutua

(b) El programa puede acabar en un interbloqueo.

(c) El programa no cumple la propiedad de ausencia de inanición