Prueba objetiva final - Clave a

Concurrencia

2009-2010 - Extraordinaria de Julio

Lenguajes y Sistemas Informáticos e Ingeniería de Software

Normas

Este es un cuestionario tipo test que consta de **5 preguntas** en **4 páginas**. 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 DNI** en cada hoja de respuesta.

Sólo hay una respuesta válida por pregunta. 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

(2 puntos) 1. Dado el siguiente **CTAD** (sólo se muestran las partes necesarias):

TIPO: Factoría $= \mathit{Área} \nrightarrow \mathbb{N}$
DONDE: $\mathit{Área} = \{0, 1\}$
INVARIANTE: $\forall f \in \mathit{Factoría}.\ (\forall\, a \in \mathit{Área}.\, f(a) \leq 10) \wedge \sum_{a \in \mathit{Área}} f(a) < 20$

INICIAL(f)**:** $\sum_{a \in \mathit{Área}} f(a) = 0$

CPRE: $f(a) < 10 \wedge \sum_{a \in \mathit{Área}} f(a) < 19$
Entrar(f,a)
POST: $f^{sal} = f^{ent} \oplus \{a \mapsto f^{ent}(a) + 1\}$

CPRE: Cierto
Salir(f,a)
POST: $f^{sal} = f^{ent} \oplus \{a \mapsto f^{ent}(a) - 1\}$

Supóngase un programa concurrente en el que los procesos respetan el siguiente protocolo (o esquemas de llamada): *Entrar*(f, a); ...; *Salir*(f, a).

Se pide señalar la respuesta correcta (asumir, obviamente, que la invariante se cumple antes de la invocación de cada operación).

(a) El programa puede violar la invariante del recurso compartido sólo en la operación *Entrar*.

(b) El programa puede violar la invariante del recurso compartido sólo en la operación *Salir*.

(c) El programa puede violar la invariante del recurso compartido tanto en *Entrar* como en *Salir*.

(d) Ninguna de las otras respuestas es correcta.

(2 puntos) 2. Dado el siguiente **CTAD** (sólo se muestran las partes necesarias):

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

INICIAL(r)**:** $r = (cierto, cierto)$

CPRE: *cierto*
S(r)
POST: $r^{ent} = (pe, se) \wedge r^{sal} = (ps, \neg se) \wedge ps = (\neg pe \wedge se) \vee (pe \wedge \neg se)$

CPRE: $r \neq (cierto, falso)$
M(r)
POST: $r^{sal} = (\neg r^{ent}.a, r^{ent}.b)$

Se pide marcar la afirmación correcta:

(a) El recurso puede pasar, a lo sumo, por 3 estados diferentes.

(b) El recurso podría estar en un estado dado y, tras ejecutarse una sola de sus acciones, continuar en el mismo estado.

(c) Si el sistema solo contiene (además de un recurso de este tipo) un único proceso que intenta ejecutar *S* indefinidamente, el sistema no puede acabar en interbloqueo.

(d) Si el sistema consta de un recurso de este tipo, un proceso que invoca a *S* una sola vez y otro que intenta ejecutar *M* indefinidamente, el sistema no puede terminar en interbloqueo.

PISTA: Dibujar el grafo de estados en hoja aparte.

(2 puntos) 3. La instrucción TST (*Test and set*) es típica de algunas arquitecturas. Su comportamiento se basa en la existencia de una variable `c` común a varios procesos. Al ejecutar `x = TST();`, donde `x` debería ser una variable local al proceso, se puede asumir que se realiza **atómicamente** la siguiente ejecución: `x = c; c = 1`. El siguiente programa concurrente hace uso de dicha instrucción para regular el acceso a una sección crítica:

```
public static final
    void main(final String[] args) {
  Thread t1, t2;
  t1 = new T(); t2 = new T();
  t1.start(); t2.start();
}
```

```
static class T extends Thread {
  public T () { }
  public void run() {
    int x;
    while (true) {
      Sec_No_Critica();
      do x = TST(); while (x != 0);
      Sec_Critica();
      c = 0;
    }
  }
}
```

Se pide marcar la afirmación correcta:

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

(b) Se puede producir interbloqueo.

(c) Puede producirse inanición de un proceso.

(d) Se garantiza la exclusión mutua y la ausencia de inanición sin que haya posibilidad de interbloqueo.

(2 puntos) 4. A continuación se muestran las partes relevantes (cliente y servidor) de una implementación del recurso del problema 1 utilizando paso de mensajes.

```
// Canales de comunicación
private Any2OneChannel chEntrar = Channel.any2one();
private Any2OneChannelInt chSalir = Channel.any2oneInt();
```

```
// Ejecutado por el cliente
public void entrar(int a) {
  One2OneChannel sincro =
    Channel.one2one();
  Object[] pet =
    {new Integer(a),sincro};
  chEntrar.out().write(pet);
  sincro.in().read();
}

public void salir(int a) {
  chSalir.out().write(a);
}

//  Código del servidor
public void run() {
  // Estado del recuros compartido
  int dentro[] = {0, 0};
  int total = 0;

  // Cola de bloqueados
  Queue<Object[]> esperanEntrar =
    new NodeQueue<Object[]>();

  // Preparando la recepción
  // no determinista
  final int ENTRAR = 0;
  final int SALIR = 1;
  Guard[] entradas =
    {chEntrar.in(), chSalir.in()};
  Alternative servicios =
    new Alternative(entradas);
  boolean[] sincCond =
    new boolean[2];

  // Variables auxiliares
  int a;
  Object[] pet;
  One2OneChannel chResp;
```

```
  // Bucle principal del servidor
  while (true) {
    sincCond[ENTRAR] = total < 19;
    sincCond[SALIR] = true;

    // Aceptamos y almacenamos peticiones
    switch (servicios.fairSelect()) {
    case ENTRAR:
      pet = (Object[])chEntrar.in().read();
      a = ((Integer)pet[0]).intValue();
      chResp = ((One2OneChannel)pet[1]);
      if (dentro[a] < 10) {
        dentro[a]++; total++;
        chResp.out().write(null);
      }
      else
        esperanEntrar.enqueue(pet);
      break;
    case SALIR:
      a = chSalir.in().read();
      dentro[a]--;  total--;
      break;
    default:
      break;
    }
    int n = esperanEntrar.size();
    for (int i = 0; i < n; i++) {
      pet = esperanEntrar.front();
      a = ((Integer)pet[0]).intValue();
      chResp = ((One2OneChannel)pet[1]);
      if (dentro[a] < 10) {
        dentro[a]++; total++;
        chResp.out().write(null);
      }
      else {
        esperanEntrar.dequeue();
        esperanEntrar.enqueue(pet);
      }
    }
  }
}
```

Se pide señalar la respuesta correcta.

(a) La expresión `chResp.out().write(null)` no compila.

(b) Es una implementación correcta del recurso compartido.

(c) La operación `entrar` puede atenderse sin que se cumpla su *CPRE*.

(d) Pueden quedar procesos bloqueados en `esperanEntrar` a pesar de que su *CPRE* se cumple.

(2 puntos) 5. Un recurso que encapsula una cuenta bancaria compartida va a ser implementado con *Locks* y *Conditions*. La operación *Reintegro* se especifica de la manera siguiente:

CPRE: $q.Saldo >= c$
Reintegro(q, c)
POST: $q^{sal}.Saldo = q^{ent}.Saldo - c$

Al depender la *CPRE* del parámetro de entrada se ha optado por declarar una clase *Peticion* que contiene la cantidad pedida y una variable *condition* y usar una cola para guardar las peticiones pendientes.

El código de la operación *Reintegro* es el siguiente (dejando aparte el manejo de la exclusión mutua):

```
public void reintegro(int c) {
   // traduccion CPRE
   if (c > this.saldo) {
      Peticion pet = new Peticion(c);
      pendientes.enqueue(pet);
      pet.condition.await();
      pendientes.dequeue();
   }
   // traduccion POST
   this.saldo = this.saldo - c;
   // codigo desbloqueos
   if (!pendientes.isEmpty() &&
      (pendientes.front().cantidad <= this.saldo)) {
           pendientes.front().condition.signal();
   }
}
```

Se pide señalar la respuesta correcta:

(a) Bastaba haber usado una sola variable *condition*.

(b) Ese código puede provocar inanición.

(c) El problema hay que implementarlo con métodos *synchronized* de Java.

(d) Es un desbloqueo en *cascada* que sigue la metodología vista en clase.