Programación de Sistemas Concurrentes y Distribuidos
2ª Convocatoria curso 14/15

4 de septiembre de 2015

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Ejercicio 1 (2.5 ptos.)

Considérese el código Java que aparece en el Anexo I, en el que se lanzan dos procesos concurrentes que se coordinan mediante semáforos.

El ejercicio pide:

1. Construir un modelo red de Petri para el programa considerado.
2. A partir del modelo anterior, construir su grafo de alcanzabilidad.
3. Analizando el grafo construído:
   a) Determinar si el programa tiene problemas de bloqueo o de falta de equidad.
   b) Demostrar que se cumple el siguiente invariante: $(x \geq 0) \wedge (y \leq 2) \wedge (x+y \leq 2)$.
   c) Demostrar que es posible encontrar un estado en el que $x + y > 1$, y dar una ejecución que lo alcance.
4. Si del código se eliminaran los semáforos y las instrucciones que los usan, ¿habrías sido capaz de responder a las cinco preguntas anteriores?

Ejercicio 2 (2.5 ptos.)

La figura 1 representa un sistema distribuido que consta de dos tipos de procesos: los procesos *Cliente* y los procesos *Recurso*. Más concretamente, existen 50 procesos cliente y 10 procesos recurso. Estos procesos se comunican entre sí a través de un espacio de tuplas compartido. Además, tienen acceso y comparten un repositorio de ficheros.

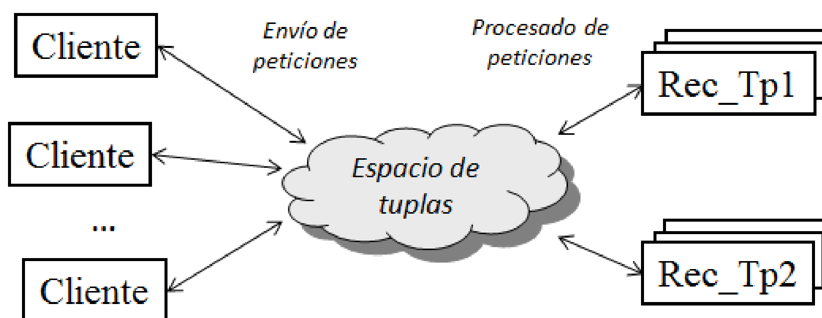

Figura 1: Descripción del sistema

El código del proceso *Cliente* se muestra y detalla a continuación. Un cliente solicita a un recurso cualquiera que procese el contenido de un fichero concreto y, posteriormente, espera a que el recurso le notifique que finalizó su trabajo. Este comportamiento lo repite durante un número aleatorio de veces. La elección del fichero a procesar en cada iteración es determinada por una función *nextFileToProcess(...)*, que devuelve el nombre del fichero y su tamaño en *megabytes*. Por otro lado, desde el punto de vista de la comunicación, la tupla de solicitud de procesado que envía un cliente consta de cuatro campos: el identificador único del proceso cliente, la acción a realizar ("PROCESAR") y el nombre y tamaño del fichero a procesar. La tupla de la correspondiente respuesta enviada por el recurso que lo procese constará de dos campos: el identificador del proceso cliente destinatario de la notificación y la marca de que la acción ha sido realizada ("PROCESADO"). Por último, una vez todas las peticiones han sido atendidas, el cliente indica su finalización enviando una tupla con el nombre de fichero vacío y un tamaño de fichero cualquiera.

```
process Cliente (idC: 1..50)
    string name
    integer size, numVeces

    numVeces:= Random()
    for i:=1..numVeces
        nextFileToProcess(name,size)
        postnote([idC,"PROCESAR",name,size])
        removenote([idC=,"PROCESADO"])
    end for
    postnote([idC,"PROCESAR","",0])
end process
```

Los procesos *Recurso* pueden ser de dos tipos distintos: tipo 1, llamados *Rec_Tp1* y los de tipo 2, *Rec_Tp2* (por simplicidad, existe el mismo número de recursos de cada tipo). Ambos tipos de recursos son capaces de procesar ficheros, pero los de tipo 1 son más eficientes para procesar ficheros de tamaño pequeño (menos de 50 *megabytes*), mientras que los de tipo 2 son más adecuados para ficheros grandes. En cualquier caso, el comportamiento de un recurso es siempre el mismo: está a la espera de recibir una petición de procesado y, cuando la recibe, la ejecuta (función *processFile(...)*) y notifica su finalización al cliente que la envió.

Para gestionar de forma eficiente las peticiones de los clientes, es necesario un proceso *Gestor* que actúe de intermediario entre clientes y recursos: las solicitudes de procesado de ficheros pequeños deben ser asignadas a los recursos de tipo 1 y las de ficheros grandes a recursos de tipo 2, respectivamente. Además, este gestor también será el responsable de garantizar que el sistema finaliza de manera controlada: una vez todos los clientes hayan notificado el final de su ejecución, los recursos también deben finalizar su actividad.

Se pide diseñar el sistema descrito y programarlo utilizando la notación presentada en las sesiones de aula.

Ejercicio 3 (2.5 ptos.)

Como se esquematiza en la figura 2, una empresa ha desplegado en la red un nuevo servicio en línea. Este servicio ofrece la operación *addInteger(name,value)*, que incrementa la variable entera *name* en *value* unidades. Por simplicidad, internamente el servicio consta sólo de dos variables enteras (*V1* y *V2*) y se puede suponer que las peticiones que efectuarán los potenciales clientes están libres de errores. Desde un punto de vista de diseño, el servicio consta de un proceso *Servidor* que atiende las peticiones de los clientes y de tres procesos internos P_i que actualizan el valor de las variables compartidas y avisan al cliente cuando la operación se ha realizado. Cuando una petición concreta es recibida, el servidor determina si existe un proceso P_i que esté desocupado y, en caso de que exista, le asigna el correspondiente trabajo. Obviamente, en todo momento, el servidor debe garantizar el mayor paralelismo posible a la hora de atender las peticiones de servicio recibidas. La comunicación entre el proceso servidor y los procesos ejecutores es por medio de paso de mensajes síncronos.

Por otro lado, el sistema consta de 10 procesos *Cliente* que están continuamente realizando peticiones al servicio. Para cada petición eligen al azar la variable a modificar y su incremento. La comunicación entre los clientes y el servidor será también programada por medio de paso de mensajes síncronos.

Se pide programar el sistema descrito utilizando la notación presentada en las sesiones de aula. La solución deberá resolver de forma correcta todos aquellos aspectos de concurrencia y sincronización involucrados en el sistema.

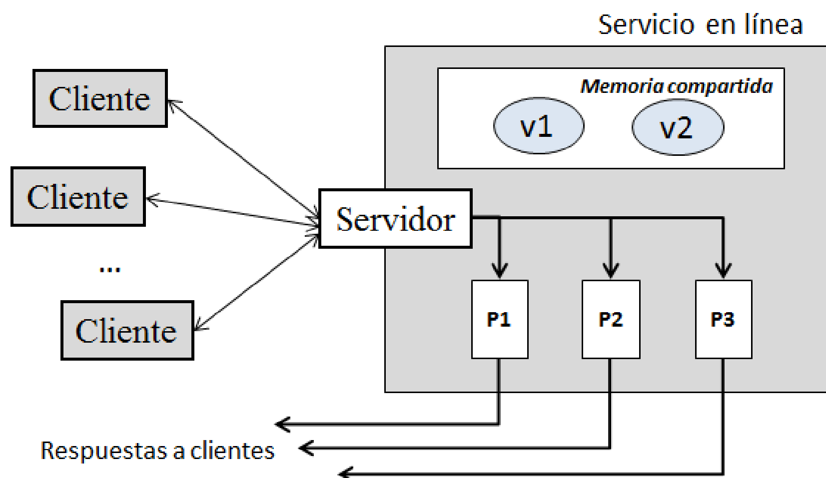

Figura 2: Descripción del sistema

Ejercicio 4 (2.5 ptos.)

Considérese el esqueleto de programa concurrente que se muestra en el Anexo II, que se utiliza para resolver un problema mediante aproximaciones sucesivas. A partir de un valor inicial de un vector de reales de dimensión N se obtiene un nuevo vector en base a aplicar una función f que depende del índice del valor: el nuevo valor para un índice k se obtiene mediante la invocación $f(k, D[k])$. Se considerará que se obtiene una solución suficientemente buena cuando la diferencia (en valor absoluto) entre un valor actual, $D[k]$, y el siguiente, $f(k, D[k])$, es menor que una tolerancia dada (valor de la constante `TOLERANCIA`), para al menos una de la componentes del vector.

Para ganar en eficiencia, se lanza un proceso por cada una de las dimensiones del vector, de acuerdo al esquema que establece el esqueleto de programa. La sucesivas iteraciones de los procesos funcionan de manera sincronizada: todos los procesos realizan una iteración; una vez hayan terminado todos, se puede llevar a cabo la actualización de la componente con el nuevo valor obtenido, si ninguno ha alcanzado la condición de terminación, pudiendo iniciarse la siguiente iteración. Y así sucesivamente hasta que alguno de ellos llegue a la condición de terminación. Cuando esto ocurra, un proceso `informador` deberá mostrar la media de los valores finales en el vector una vez todos los procesos involucrados han terminado la iteración en marcha.

Se pide completar el programa suministrado, en el que la concurrencia se gestiona mediante el monitor `control`.

Anexo I

```java
//fichero: Ejercicio.java

public class Ejercicio{
    static Semaphore S1 = new Semaphore(0);
    static Semaphore S2 = new Semaphore(0);

    public static void main(String[] args){
        Datos d = new Datos(0,1);
        Thread proc1 = new Thread(new Proc1(d,S1,S2));
        Thread proc2 = new Thread(new Proc2(d,S1,S2));

        proc1.start();
        proc2.start();
    }
}
//---------------------------------------------
//fichero: Datos.java
class Datos{
    private int x,y;

    Datos(int xx, int yy){
        x = xx;
        y = yy;
    }

    public synchronized void incX(int v){
        x = x+v;
    }

    public synchronized void incY(int v){
        y = y+v;
    }
}
//---------------------------------------------
//fichero: Proc1.java

class Proc1 implements Runnable{
    private Datos losDatos;
    private static Semaphore S1,S2;

    public Proc1(Datos d, Semaphore S1, Semaphore S2){
        this.losDatos = d;
        this.S1 = S1;
        this.S2 = S2;
    }

    public void run(){
        while(true){
            try{
                losDatos.incX(1);
```

```java
                S1.release();
                S2.acquire();
                losDatos.incY(1);
            }catch(InterruptedException ex){
                ex.printStackTrace();
            }
        }
    }
}
//---------------------------------------------
//fichero: Proc2.java
import java.util.concurrent.Semaphore;

class Proc2 implements Runnable{
    private Datos losDatos;
    private static Semaphore S1,S2;

    public Proc2(Datos d, Semaphore S1, Semaphore S2){
        this.losDatos = d;
        this.S1 = S1;
        this.S2 = S2;
    }

    public void run(){
        while(true){
            try{
                S1.acquire();
                losDatos.incY(-1);
                losDatos.incX(-1);
                S2.release();
            }catch(InterruptedException ex){
                ex.printStackTrace();
            }
        }
    }
}
```

Anexo II

```
constant integer N := 817
constant float TOLERANCIA := 10E-8
float array[1..N] D := ... //vector de N reales, inicializado

operation f(integer k, float d) return float
//se supone implementada

//----------------------------------------
monitor control
    ...
end monitor
//----------------------------------------
process P(i: 1..N)::
   float nuevoD

   while control.seguir()
        nuevoD := f(i, D[i])
        //esperar a que todos acaben iteración
        //procesar el dato

        ...
   end while
end process
//----------------------------------------
process informador::
    float media := 0.0

    //esperar a que los cálculos se den por terminados
    //calcular la media de los valores en D
    ...
    write('Valor␣obtenido␣=␣')
    write(media)
end process
```