Programación de Sistemas Concurrentes y Distribuidos 1ª Convocatoria curso 16/17

25 de enero de 2017

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Ejercicio 1 (2.5 ptos.)

Considérese el código C++ que se muestra en el Anexo I. El ejercicio pide:

- Presentar un modelo red de Petri que refleje el comportamiento del programa
- Obtener el grafo de estados alcanzables del modelo propuesto
- Razonando sobre el grafo obtenido dar respuestas convincentes a las siguientes cuestiones:
  - ¿Tiene el programa un comportamiento equitativo? En caso negativo, mostrar al menos una historia no equitativa.
  - ¿Presenta el programa problemas de bloqueo? En caso afirmativo, dar al menos una historia que lleve a un bloqueo. El número de historias que llevan a bloqueo ¿es finito o infinito?
  - Demostrar que se cumple el siguiente invariante: "En ningún momento pueden estar los dos procesos en las correspondientes secciones críticas simultáneamente".
  - Un lugar de una red de Petri se dice "implícito" si, al eliminarlo del modelo, las posibles secuencias de disparo de transiciones son las mismas (ambos modelos, el original y el resultante de la eliminación reconocerían el mismo lenguaje). En el caso de modelar un programa, puede indicar código redundante, cuya

eliminación no modificaría su comportamiento en ejecución. ¿Presenta tu modelo algún lugar implícito? Si es así, indícalo, razonando por qué es implícito, e indica la parte del código correspondiente que sería eliminable sin cambiar sus posibles historias de ejecución.

Ejercicio 2 (2.5 ptos.)

La figura 1 muestra el esquema de un cruce de vías. Los trenes que la usan circulan de izquierda a derecha (nótese que hay 4 posibles recorridos: AC, AE, BD y BF). El ejercicio pide diseñar cuatro tipos de procesos, uno por cada posible recorrido, que son coordinados para el acceso al cruce mediante un monitor (que también se ha de diseñar) para evitar colisiones. Para acceder al cruce, cada proceso debe ejecutar una operación de petición de entrada, y una de liberación cuando sale. El monitor debe asegurar que no haya dos trenes en el mismo tramo o en tramos con algún trozo de vía compartido, pero permitiendo el mayor paralelismo posible.

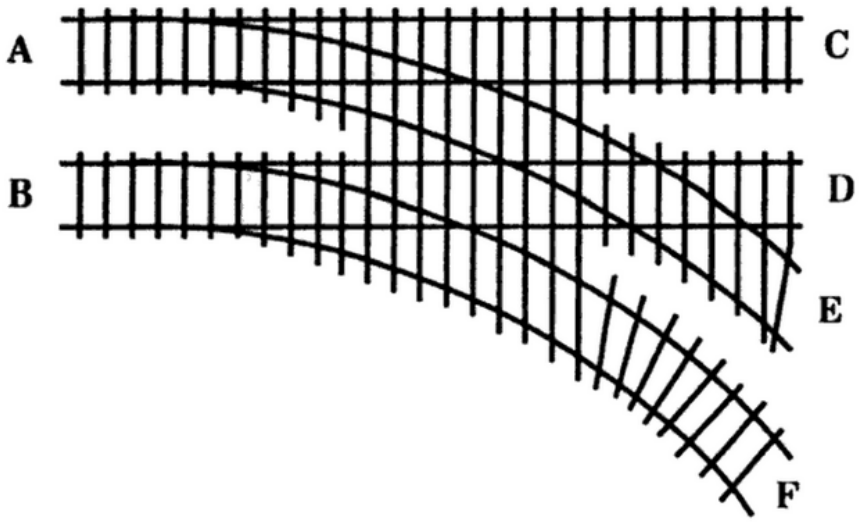

Figura 1: Esquema de un cruce de vías (C. R. Snow, "Concurrent Programming", Cambridge Computer Science Texts, 1992).

Ejercicio 3 (2.5 ptos.)

Consideremos un sistema con cuatro tipos de recursos, $R = \{R_i \mid i = 1, 2, 3, 4\}$, habiendo 10 unidades de cada uno de ellos. Consideremos también dos tipos de procesos, cuyos esquemas son los siguientes:

```
process P(i::1..10)

  while(true)
     //------------ fase 1
     //reserva de manera atómica, 2R_1+R_2
     //usa los recursos
     //libera los recuros
     //------------ fase 2
     //reserva de manera atómica, R_3+2R_4
     //usa los recursos
     //libera los recuros
  end while
end process

process Q(i::1..15)

  while(true)
     //------------ fase 1
     //reserva de manera atómica, R_1+2R_2
     //usa los recursos
     //libera los recuros
     //------------ fase 2
     //reserva de manera atómica, 2R_3+R_4
     //usa los recursos
     //libera los recuros
  end while
end process
```

El ejercicio pide desarrollar el programa, con la notación utilizada en clase y usando Linda para su coordinación, que implemente el sistema descrito, con dos restricciones adicionales: *no puede haber, en ningún momento, más de cuatro procesos en cada una de las fases.*

Ejercicio 4 (2.5 ptos.)

Se quiere programar un sistema distribuido que intercambie el valor de las componentes de una estructura tipo vector. El sistema consta de tres tipos de procesos diferentes que se comunicarán y coordinarán a través de paso síncrono de mensajes:

- *Procesos almacén*: Estos procesos son responsables de guardar, de forma distribuida, la estructura de datos y facilitar el acceso en lectura/escritura a los valores de sus componentes. La información del sistema consta de 2N datos de tipo entero, indexados del [1..2N]. La estructura de almacenamiento se compone de N procesos almacén, cuyos identificadores están en el rango [1..N]. El proceso con identificador "i" es responsable de almacenar y gestionar el acceso a las componentes [2i-1] y [2i] de la estructura de datos.
- *Procesos cliente*: Estos procesos intercambian un número aleatorio de veces el valor de dos componentes de la estructura de datos. Su comportamiento siempre es el mismo: solicitan al proceso servidor los índices de las dos componentes que deben intercambiar, e interaccionan con los procesos almacén que guardan estas componentes para proceder al intercambio de los correspondientes valores. El sistema tiene N procesos cliente que están ejecutándose en paralelo.
- *Proceso servidor*: Este proceso genera pares de enteros distintos y correctos (en el rango [1..2N]) que corresponden a índices de las componentes a intercambiar.

Se pide:

1. Diseñar y representar gráficamente la estructura de canales síncronos que facilitará la comunicación entre los procesos del sistema. También es necesario justificar de forma breve y concisa las decisiones adoptadas.
2. Explicar las cuestiones de sincronización involucradas en el sistema y cómo éstas van a ser resueltas.
3. Programar los procesos del sistema, utilizando la notación empleada en clase.

Anexo I

```
//========================= File: main.cpp
#include <iostream>
#include <thread>
#include <Semaphore.h>

using namespace std;
//---------------------------------------
void func1(Semaphore &s1,
           Semaphore &s2,
           Semaphore &s3){

    while(true){
        s1.signal();
        s2.wait();
        //SC1
        s2.signal();
        s3.wait();
    }
}
//---------------------------------------
void func2(Semaphore &s1,
           Semaphore &s2,
           Semaphore &s3){

    while(true){
        s1.wait();
        s2.wait();
        //SC2
        s2.signal();
        s3.signal();
    }
}
//---------------------------------------
int main(){
    Semaphore s1(0), s2(2), s3(0);

    thread th_1(&func1, ref(s1), ref(s2), ref(s3)),
           th_2(&func2, ref(s1), ref(s2), ref(s3));

    th_1.join();
    th_2.join();

    return 0;
}
```

```
//========================= File: Semaphore.h
#ifndef SEMAPHORE_H
#define SEMAPHORE_H

#include <mutex>
#include <condition_variable>
#include <assert.h>

using namespace std;

class Semaphore{
private:
    mutex mtx;         //ejecución de funciones en mutex
    condition_variable cv;
    int count;        //natural asociado al semáforo
    bool initialized;//para manejar dos constructores distintos

public:
    //------------------------- constructores
    //Pre:
    //Post: NOT initialized
    Semaphore();

    //Pre: n>=0
    //Post: count=n AND NOT initialized
    Semaphore(int n);

    //Pre: n>=0 AND NOT initialized
    //Post: initialized AND count=n
    void setInitValue(int n);
    //------------------------- operaciones estándar
    //Pre: initialized
    //Post: <count++>
    void signal();

    //Pre: initialized
    //Post: <await count>0 count-- >
    void wait();

    //------------------------- operaciones extendidas
    //Pre: n>0 AND initialized
    //Post: <count=count+n>
    void signal(int n);

    //Pre: n>0 AND initialized
    //Post: <await count>=n count=count-n >
    void wait(int n);
};
#endif
```