Programación de Sistemas Concurrentes y Distribuidos
1ª Convocatoria curso 19/20. 7 de febrero de 2020
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Examen teórico-práctico

- Hay que dejar sobre la mesa, en lugar visible, un **documento de identificación** provisto de fotografía y escribir el nombre y dos apellidos en cada una de las hojas de papel que haya sobre la mesa.
- Cada ejercicio debe resolverse en una **hoja independiente**.
- Todos los ejercicios deben ir acompañados de una **clara y concisa explicación** de la solución propuesta.

Ejercicio 1 (2.0 ptos.)

Considérese el esquema de programa concurrente que se muestra a continuación. El objetivo es que N procesos se repartan el trabajo de encontrar el máximo de un vector, de manera que todos trabajan sobre el mismo número de componentes. Se trata de completar el código para que, usando semáforos como mecanismo de sincronización, se muestre al terminar el programa el valor máximo del vector. En el programa no pueden intervernir más procesos que los declarados, ni deberían usarse más de tres semáforos.

Es necesario explicar brevemente los aspectos fundamentales de la sincronización, la finalidad de cada semáforo, etc.

```
const integer N = 10
const integer ND = 1000
integer array[1,ND] datos
integer maxVect
...

// Carga los datos del fichero "nomFich" al vector "vD"
// Se asume ya implementada. Deberá ejecutarla uno de los procesos
operation cargaVect(string nomFich, REF integer array[1,ND] vD)

Process P(i: 1..N)::
   ...
end
```

Ejercicio 2 (1.5 ptos.)

El problema a resolver es el mismo que en el ejercicio anterior, pero usando monitores para gestionar los aspectos de sincronización.

Ejercicio 3 (2.25 ptos.)

1. Un sistema consta de tres procesos distribuidos que se coordinan a través de `linda`. El proceso `ProdNum` genera y escribe un número natural aleatorio en el espacio de tuplas y espera a que algún proceso lector lo consuma. Este comportamiento lo repite indefinidamente. Por otro lado, los otros dos procesos, llamados `ConsPar` y `ConsImpar`, actúan como lectores de números pares e impares, respectivamente. Cada número producido se lee una única vez, por el proceso lector adecuado, mostrándolo por su correspondiente salida estándar.

   Se pide escribir el código de los tres procesos que forman el sistema, representando el estado del sistema y la información intercambiada con una única tupla `linda`. Es necesario explicar de forma breve y concisa el formato de la tupla y la semántica de cada uno de sus elementos.

2. El sistema anterior constaba de un proceso escritor y dos lectores. Se pide programar una nueva versión del sistema en la que hay 10 procesos `ConsPar` y 10 procesos `ConsImpar`. En este caso, la solución debe garantizar que cada vez que se escriba un nuevo número par en el espacio de tuplas, éste sea leído una sola vez por todos los procesos `ConsPar` (similar comportamiento para los números impares y los procesos `ConsImpar`).

   Se pide escribir el código de los procesos involucrados, representando el estado del sistema y la información intercambiada con una única tupla `linda`. Es necesario explicar de forma breve y concisa el formato de la tupla y la semántica de cada uno de sus elementos.

Ejercicio 4 (1.75 ptos.)

Considérese un sistema distribuido compuesto por N procesos que comunican mediante *canales síncronos*. Se pide completar el siguiente esquema con el código necesario para asegurar que ningún proceso P ejecute dos iteraciones consecutivas sin que cada uno de los otros $N-1$ procesos ejecute la suya.

```
Process P(i: 1..N)::
   loop
       //mis tareas
       //sincronizar con Barrera
   end
end

Process Barrera::
   loop
       //implementa una barrera para los P(i: 1..N)
   end
end
```

Ejercicio 5 (1.0 ptos.)

La red de Petri que aparece a continuación corresponde a un programa concurrente, de acuerdo a la forma de modelado desarrollada en las clases de teoría, en que dos procesos comparten dos variables de tipo booleano (0 y 1 representan *false* y *true*, respectivamente). Se pide escribir una versión del código de dicho programa, con la notación utilizada en clase.

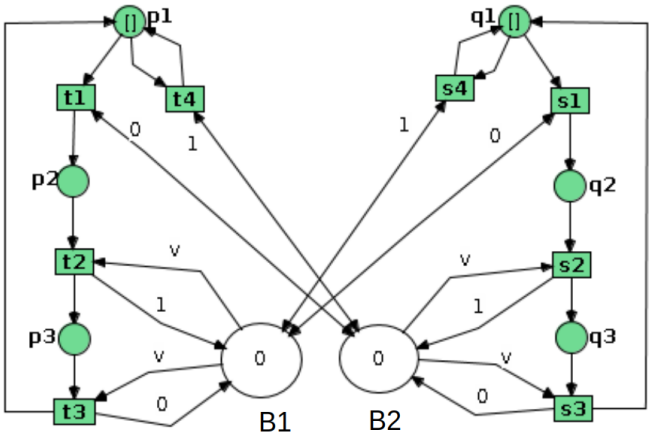

Figura 1: Modelo de un programa concurrente

Ejercicio 6 (1.5 ptos.)

Complétese el código C++ de los procedimientoes `wait` y `signal` usados en prácticas.

```
//Semaphore.hpp
class Semaphore {
private:
    mutex mtx;
    condition_variable_any cv;
    int count;
public:
    ...
    Semaphore(...);
    ~Semaphore();
    void signal();
    void wait();
};
//Semaphore.cpp
void Semaphore::wait() {
    ...
}
void Semaphore::signal() {
    ...
}
```