Programación de Sistemas Concurrentes y Distribuidos 1ª Convocatoria curso 18/19

5 de febrero de 2019

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Ejercicio 1 (2.5 ptos.)

Hay tres tipos de procesos, A, B y C, que entran y salen, bajo determinadas condiciones, de una habitación:

- Un proceso A solo pueden salir si, en algún momento, ha coincidido en la habitación, simultáneamente, con un proceso B y un proceso C.
- Un proceso B solo pueden salir si, en algún momento, ha coincidido en la habitación con un proceso B o un proceso C.
- Un proceso C puede dejar la habitación en cualquier momento, sin ninguna restricción

Se pide completar el siguiente diseño, de manera que la sincronización entre los procesos se lleva a cabo mediante el monitor *controlHab*.

```
integer N = 10
Process A(i: 1..N)::
   ...
end
Process B(i: 1..N)::
   ...
end
Process C(i: 1..N)::
   ...
end
monitor controlHab
   ...
end
```

Ejercicio 2 (2.75 ptos.)

Considérese el entorno de datos que se muestra debajo. Se trata de diseñar un programa concurrente, con el esquema máster-trabajador, en el que NW procesos trabajadores son coordinados por un proceso máster con el objetivo de encontrar el máximo de una matriz de $N \times N$ enteros, usando semáforos para los aspectos relacionados con la sincronización,

```
integer NW=5    --num de trabajadores
integer N=100   --dim de la matriz
integer array[1..N,1..N] m := ... --ya inicializada

Queue<integer> fP;
integer maxG //el máximo global
...
Process master::
   //insertar índices de filas a procesar
   //esperar todas hayan sido procesadas
   //avisar a trabajadores para que finalicen
   write(maxG)
end

Process worker(i: 1..NW)::
   bool fin := false
   ...
   while not fin
      //tomar índice siguiente fila a procesar, o
      //indicación de terminación
      //actuar en consecuencia
   end
end
```

Nota 1: Podéis usar el TAD *cola* acorde a su especificación en el Anexo-I

Nota 2: Es necesario comenzar con un diseño basado en instrucciones $< await... >$, lo que facilita un diseño correcto. Posteriormente se debe aplicar la técnica de paso del testigo para dar una solución basada en semáforos.

Ejercicio 3 (2.0 ptos.)

Cuatro generales bizantinos **A**, **B**, **C** y **D** deciden **A**, **A**, **R**, y **A**, respectivamente. Considérese la siguiente situación:

- El general **B** envía a **A** y **C** su decisión inicial y miente a **D**. Además, antes de comenzar la segunda vuelta, **B** cae.
- Durante la segunda vuelta, el general **C** miente a **A** en todos los mensajes que le envía.

Se pide completar las tablas de los cuatro generales y explicar de forma concisa y justificada qué generales son traidores y si se alcanza un consenso.

Ejercicio 4 (2.75 ptos.)

Se pide completar el código que se muestra debajo usando Linda para la coordinación de los procesos. Se trata de completar tanto el proceso *init* como los procesos *P*, de manera que estos últimos, en cada iteración, se esperan en una barrera común antes de realizar la siguiente. Cada proceso ejecuta un total de 100 iteraciones.

```
Process init::
   integer N := ...        //num. de procesos involucrados
   integer N_ITER = 100    //num. de iteraciones
   //poblar linda con lo necesario
   //esperar a que los procesos P acaben
   //limpiar linda
end

Process P(id: 1..N)::
   integer N_ITER = 100
   integer nI := 0 //nI: num. de iteraciones que he ejecutado
   ...
   while nI<=N_ITER
      //ejecutar mi tarea (sea lo que sea)
      //esperar a que todos estén en la barrera
      nI++
   end
end
```

Anexo-I

Especificación de un TAD **cola**

```
// Una cola "c" se representará como "c=< d_1 ... d_n ]"
// siendo "d_1" el elemento más antiguo, "d_2" el siguiente, etc.
// "n" es el número de elementos; "n=0" es equivalente a que esté vacía

template <class T>
class Queue {
public:
    //-------------------------------------------------------
    //Pre:
    //Post: this=< ]
    //Com:  constructor
    Queue();
    //-------------------------------------------------------
    //Pre:
    //Post:
    //Com:  destructor
    ~Queue();
    //-------------------------------------------------------
    //Pre:
    //Post: this=< ]
    void empty();
    //-------------------------------------------------------
    //Pre:  this=< d_1 ... d_n ]
    //Post: this=< d_1 ... d_n d ]
    void enqueue(const T d);
    //-------------------------------------------------------
    //Pre:  this=< d_1 ... d_n ] AND 0<n
    //Post: this=< d_2 ... d_n ]
    void dequeue();
    //-------------------------------------------------------
    //Pre:  this=< d_1 ... d_n ] AND 0<n
    //Post: first()=d_1
    T first();
    //-------------------------------------------------------
    //Pre:  this=< d_1 ... d_n ]
    //Post: length()=n
    int length();
    //-------------------------------------------------------
    ...
};
```