Programación de Sistemas Concurrentes y Distribuidos 1ª Convocatoria curso 12/13

5 de febrero de 2013

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Ejercicio 1 (1.5 ptos.)

Considérese el siguiente programa concurrente, en el que tres procesos iteran escribiendo distinta información en la salida estándar. Se pide completar el código con lo que se considere necesario para que se cumplan simultáneamente los dos invariantes siguientes: 1) `num(A) >= num(C);` 2) `num(D) > num(E)` (donde, para un carácter dado <c>, `num(c)` representa el número de veces que se muestra por la salida estándar).

```
process P1
   while TRUE
      write("A")
      write("B")
   end while
end process
```

```
process P2
   while TRUE
      write("C")
      write("D")
   end while
end process
```

```
process P3
   while TRUE
      write("E")
      write("F")
   end while
end process
```

Ejercicio 2 (2.50 ptos.)

La figura 1 representa de manera esquemática el patrón de diseño *supervisor-worker*. Este esquema propone una organización sencilla en la que un proceso supervisor suministra tareas a los trabajadores a través de un medio común para el intercambio de

información. Los procesos trabajadores van tomando las solicitudes, obtienen los resultados y los envían al supervisor. Esto se repite hasta que el supervisor le indique que puede terminar. El supervisor toma los resultados y actúa en función a cuál sea su objetivo.

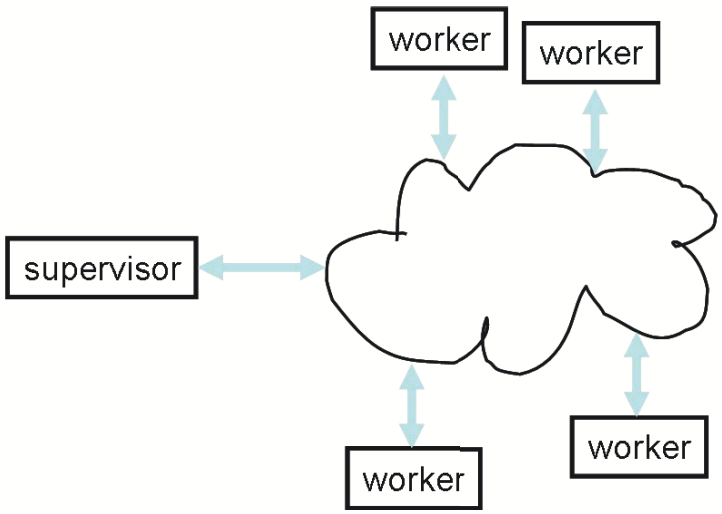

Figura 1: Esquema del patrón de diseño *supervisor-worker*

Se pide desarrollar un programa concurrente que ordene 100 ficheros de datos, de manera que un proceso supervisor suministra los nombres de los ficheros, y cuatro procesos trabajadores van tomando las órdenes depositadas por el supervisor y ordenan el fichero cuyo nombre se les ha suministrado. Cada trabajador va iterando ese proceso hasta que recibe del supervisor la orden de terminar. Dependiendo del entorno de ejecución, es posible que distintos procesos trabajador tengan distintas velocidades, por lo que no es adecuado solicitar un cuarto de las ordenaciones a cada proceso trabajador. La sincronización entre el proceso supervisor y los trabajadores se llevará a cabo mediante el uso de monitores.

```
-------------------------------------------------------------
    constant natural NUM_WORKERS := 4
    constant natural NUM_FICH := 100

monitor control


end monitor

process supervisor
     string array[NUM_FICH] ficheros := ...
                          --lo que sea; ya inicicializado

    --completar de acuerdo al enunciado del ejercicio
end process
```

```
process worker(i:1..4)::
    operation ordena_fichero(string nom_fich)
       --Pre: el fichero en <nom_fich> existe
       --Post: el fichero en <nom_fich> está ordenado
       --Com: Asumimos que está implementada

         ...
    end operation

    --completar de acuerdo al enunciado del ejercicio
end process
-------------------------------------------------------
```

Ejercicio 3 (2.50 ptos.)

Un jardín tiene dos puertas de acceso. Cada puerta tiene instalado un torno responsable de gestionar las entradas y salidas del jardín. Debido a su reducido tamaño no puede haber más de 50 personas simultáneamente visitándolo. Inicialmente, hay 100 personas interesadas en realizar la visita. Cada vez que una persona quiere acceder al jardín debe comprar un ticket de entrada. En el ticket se le especifica la puerta concreta por la que debe entrar y salir del jardín.

Todas las personas se comportan de la misma manera, repitiendo la siguiente secuencia de acciones durante un número aleatorio de veces: primero, compran un ticket; segundo, solicitan a la puerta asignada permiso para entrar en el jardín y esperan a que ésta les dé acceso; tercero, visitan el jardín durante un cierto tiempo; y, finalmente, solicitan salir y esperan a que el torno les autorice la salida. Por otro lado, el funcionamiento de los tornos es muy simple: reciben las solicitudes de los usuarios (que ya tienen su ticket correspondiente) y les dan los permisos tanto de entrada como de salida.

Se pide implementar el sistema concurrente anteriormente descrito. El sistema consta de tres tipos de procesos: el proceso *expendedor de tickets*, los dos procesos *puerta*, y los 100 procesos *persona*. La comunicación y coordinación de los procesos debe ser programada utilizando Linda.

Ejercicio 4 (2.0 ptos.)

Ricart y Agrawala propusieron un algoritmo distribuido para el acceso a una sección crítica basado en el paso de un testigo (se adjunta en el Anexo-II). El sistema dispone de un único testigo que es compartido por todos los procesos. En todo momento, sólo el

proceso que tiene el testigo tiene derecho a entrar en su sección crítica. Mientras tanto, el resto de los procesos interesados deben esperar.

Ejercicio 4.1. Se pide modificar el algoritmo Ricart-Agrawala para que funcione con procesos con prioridades. La prioridad de cada proceso será representada por un número entero. Antes de que comience la ejecución del programa, el proceso con identificador 1 será el responsable de generar aleatoriamente las prioridades de todos los procesos. Estas prioridades serán estáticas durante la ejecución del programa. La nueva versión del algoritmo determinará a qué proceso se le pasa el testigo según las siguientes reglas: 1) el testigo será pasado al proceso que tenga la prioridad maxima de entre los que hubieran declarado su interés; y, 2) si hubiera más de un proceso con la misma prioridad, el testigo será pasado al que tenga el identificador menor. Justificar que la solución propuesta es libre de bloqueos.

Ejercicio 4.2. Analizar el comportamiento de la solución propuesta desde el punto de vista de la propiedad de equidad. Si hubiera problemas de equidad, esbozar de forma concisa y breve una posible solución.

Ejercicio 5 (1.5 ptos.)

Dado el siguiente pseudocódigo correspondiente a un programa concurrente en el que están involucradas nP procesos, se pide implementar el sistema en lenguaje Java utilizando el paquete de semáforos *j*ava.util.concurrent.Semaphore.

```
-------------------------------------------------------------
    constant integer n := 50       --num. de datos
                     nP := 200     --num. de procesos
    integer array[1..n] datos := ... --lo que sea
    semaphore array[1..n] mutex_dato := (1..n,1)
                 --evitar interferencias en accesos a datos

Process P(i: 1..nP)::
    integer i1,i2,temp

    loop forever
        i1 := número aleatorio entre 1 y n
        i2 := número aleatorio entre 1 y n
                                      --1<=i1<=n AND 1<=i2<=n
        if i1<>i2  --el caso i1=i2 no hace nada
            if i1>i2
               temp := i1
               i1 := i2
               i2 := temp
            end if
            --i1 tiene el índice menor, i2 el mayor
            --acceso en orden ---> no hay bloqueos
            wait(mutex_dato[i1])
            wait(mutex_dato[i2])
            temp := datos[i1]
            datos[i1] := datos[i2]
            datos[i2] := temp
            signal(mutex_dato[i1])
            signal(mutex_dato[i2])
        end if
    end loop
-------------------------------------------------------------
```

Anexo I: Especificación del TAD cola genérica

```
=======================================================
espec colasGenéricas usa booleanos, naturales
   parámetro formal
      género elemento
   fpf género cola --valores del TAD cola representan
                   --secuencias de elementos con acceso FIFO
                   --(first in, first out): el primer elemento
                   --añadido será el primero en ser borrado
     --ejemplo: una cola de strings
     --    cola of string c_s := crear()
  operaciones
     función crear() devuelve cola
        --Pre: TRUE
        --Post: devuelve una cola vacía, sin elementos

     procedimiento encolar(in/out c:cola in e:elemento)
        --Pre: TRUE
        --Post: "c" es  la cola resultante de añadir "e" a "c"

     función esVacía?(c:cola) devuelve booleano
        --Pre: TRUE
        --Post: devuelve TRUE si "c" vacía; FALSE en caso
        --      contrario

     función primero(c:cola) devuelve elemento
        --Pre: "c" es no vacía
        --Post: devuelve el elemento más antiguo de "c"

     procedimiento desencolar(in/out c:cola)
        --Pre: "c" es no vacía
        --Post: "c" es la cola resultante de eliminar de "c"
        --      su primer elemento

     función longitud(c:cola) devuelve natural
        --Pre: TRUE
        --Post: devuelve el num. de elementos de "c"
fespec
=======================================================
```

Anexo II: Ricart-Agrawala por paso del testigo