Sistemas basados en paso de mensajes.

29

Supongamos que tenemos un programa distribuido con tres procesos de forma que queremos que cada uno pase un dato (el valor de una variable) al siguiente para que el siguiente lo imprima, siendo indiferente el orden en el que se realizan las operaciones de paso de mensaje, y también siendo indiferente el orden en el que se imprimen los valores. Esto se ha programado usando el siguiente esquema usando un paso de mensajes síncrono:

```
Process P0 ;
   var  x,y : integer;
begin
   x := ….. ;
   s_send( x, P1 );
   receive( y, P2 );
   imprime( y );
end
```

```
Process P1 ;
   var  x,y : integer;
begin
   x := ….. ;
   s_send( x, P2 );
   receive( y, P0 );
   imprime( y );
end
```

```
Process P2 ;
   var  x,y : integer;
begin
   x := ….. ;
   s_send( x, P0 );
   receive( y, P1 );
   imprime( y );
end
```

Contesta a las siguientes cuestiones:

(a) Este programa produce interbloqueo. Describe brevemente a qué se debe esto.

(b) Si el envío de los mensajes es asíncrono seguro, ¿se podría producir un interbloqueo? Razonar brevemente.

(c) Describe brevemente los cambios que harías en los procesos para cumplir los requisitos del enunciado y evitar el interbloqueo manteniendo un paso de mensajes síncrono.

30

Dado el siguiente ejemplo de paso de mensajes entre dos procesos,

```
Process PA ;
   var env : integer;
begin
   env := 40 ;
   ENVIAR( env, PB );
   env := 20;
end
```

```
Process PB ;
   var rec : integer;
begin
   rec := 30 ;
   RECIBIR( rec, PA );
   imprime( rec );
end
```

Para cada uno de los siguientes casos, indica qué valor o valores se pueden transferir por el SPM, y que valor o valores puede imprimir **PB**:

(a) **ENVIAR** es **send** y **RECIBIR** es **i_receive**.

(b) **ENVIAR** es **i_send** y **RECIBIR** es **i_receive** .

(c) **ENVIAR** es **s_send** y **RECIBIR** es **receive** .

31

En un sistema distribuido, 6 procesos clientes necesitan sincronizarse de forma específica para realizar cierta tarea, de forma que dicha tarea sólo podrá ser realizada cuando tres procesos estén preparados para realizarla. Para ello, envían peticiones a un proceso controlador del recurso y esperan respuesta para poder realizar la tarea específica. El proceso controlador se encarga de asegurar la sincronización adecuada. Para ello, recibe y cuenta las peticiones que le llegan de los procesos, las dos primeras no son respondidas y producen la suspensión del proceso que envía la petición (debido a que se bloquea esperando respuesta) pero la tercera petición produce el desbloqueo de los tres procesos pendientes de respuesta. A continuación, una vez desbloqueados los tres procesos que han pedido (al recibir respuesta), inicializa la cuenta y procede cíclicamente de la misma forma sobre otras peticiones.

El código de los procesos clientes aparece aquí abajo. Los clientes usan envío asíncrono seguro para realizar su petición, y esperan con una recepción síncrona antes de realizar la tarea.

```
process Cliente[ i : 0..5 ] ;
begin
   while true do begin
      send( peticion, Controlador );
      receive( permiso, Controlador );
      Realiza_tarea_grupal( );
   end
end
```

```
process Controlador ;
begin
   while true do begin

      ...

   end
end
```

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo `select for i=... to ...` para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice `i`.

32

En un sistema distribuido, 3 procesos productores producen continuamente valores enteros y los envían a un proceso buffer que los almacena temporalmente en un array local de 4 celdas enteras para ir enviándoselos a un proceso consumidor. A su vez, el proceso buffer realiza lo siguiente, sirviendo de forma equitativa al resto de procesos:

a) Envía enteros al proceso consumidor siempre que su array local tenga al menos dos elementos disponibles.

b) Acepta envíos de los productores mientras el array no esté lleno, pero no acepta que cualquier productor pueda escribir dos veces consecutivas en el búfer.

El código de los procesos productor y consumidor es el siguiente, asumiendo que se usan operaciones síncronas.

```
process Productor[ i : 0..2 ] ;
   var dato : integer ;
begin
   while true do begin
      dato := Producir();
      send( dato, Buffer );
   end
end
```

```
process Consumidor ;
begin
   while true do begin
      receive ( dato, Buffer );
      Consumir( dato );
   end
end
```

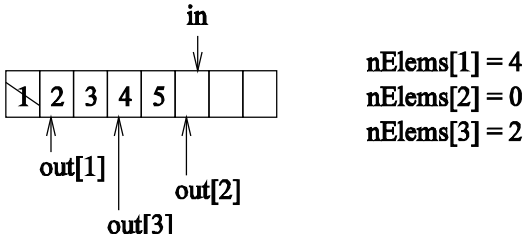
Describir en pseudocódigo el comportamiento del proceso Buffer, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos.

```
process Buffer ;
begin
   while true do begin
      ...
   end
end
```

33

Suponer un proceso productor y 3 procesos consumidores que comparten un buffer acotado de tamaño B. Cada elemento depositado por el proceso productor debe ser retirado por todos los 3 procesos consumidores para ser eliminado del buffer. Cada consumidor retirará los datos del buffer en el mismo orden en el que son depositados, aunque los diferentes consumidores pueden ir retirando los elementos a ritmo diferente unos de otros. Por ejemplo, mientras un consumidor ha retirado los elementos 1, 2 y 3, otro consumidor puede haber retirado solamente el elemento 1. De esta forma, el consumidor más rápido podría retirar hasta B elementos más que el consumidor más lento.

Describir en pseudocódigo el comportamiento de un proceso que implemente el buffer de acuerdo con el esquema de interacción descrito usando una construcción de espera selectiva, así como el del proceso productor y de los procesos consumidores. Comenzar identificando qué información es necesario representar, para después resolver las cuestiones de sincronización. Una posible implementación del bufffer mantendría, para cada proceso consumidor, el puntero de salida y el número de elementos que quedan en el buffer por consumir (ver figura).



34

Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya rellenado la olla con otros M misioneros.

```
process Salvaje[ i : 0..2 ] ;
begin
   while true do begin
       { esperar a servirse un misionero: }
       .....
       { comer }
       Comer();
   end
end
```

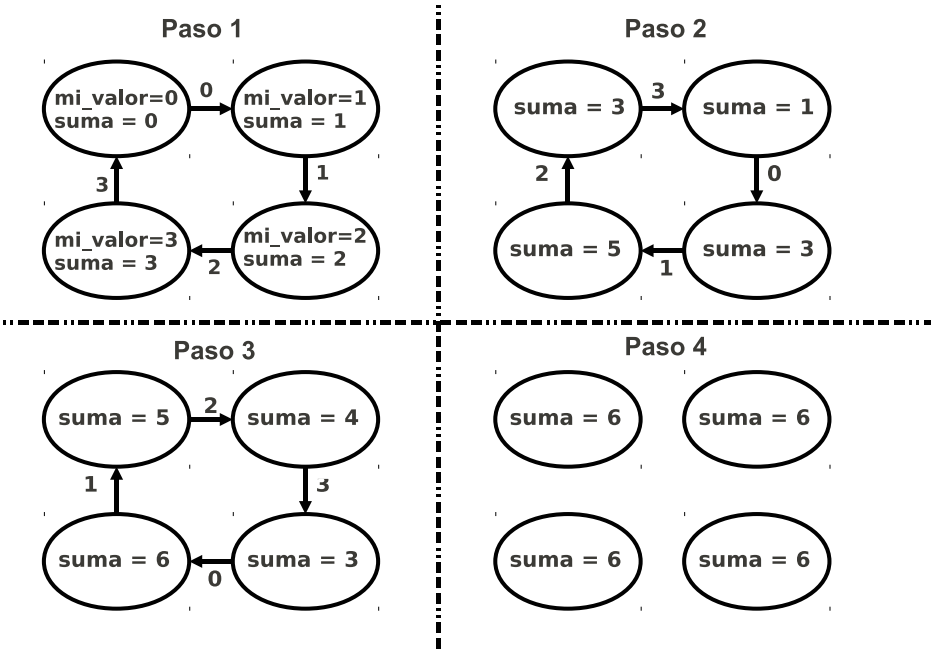
```
process Cocinero ;
begin
   while true do begin
       { dormir esperando solicitud para llenar: }
       ......
       { confirmar que se ha rellenado la  olla }
       ......
   end
end
```

Implementar los procesos salvajes y cocinero usando paso de mensajes, usando un proceso olla que incluye una construcción de espera selectiva que sirve peticiones de los salvajes y el cocinero para mantener la sincronización requerida, teniendo en cuenta que:

- La solución no debe producir interbloqueo.
- Los salvajes podrán comer siempre que haya comida en la olla,
- Solamente se despertará al cocinero cuando la olla esté vacía.

35

Considerar un conjunto de N procesos, `P[i]`, $(i = 0, ..., N-1)$ que se pasan mensajes cada uno al siguiente (y el primero al último), en forma de anillo. Cada proceso tiene un valor local almacenado en su variable local `mi_valor`. Deseamos calcular la suma de los valores locales almacenados por los procesos de acuerdo con el algoritmo que se expone a continuación.



Los procesos realizan una serie de iteraciones para hacer circular sus valores locales por el anillo. En la primera iteración, cada proceso envía su valor local al siguiente proceso del anillo, al mismo tiempo que recibe del proceso anterior el valor local de éste. A continuación acumula la suma de su valor local y el recibido desde el proceso anterior. En las siguientes iteraciones, cada proceso envía al siguiente proceso siguiente el valor recibido en la anterior iteración, al mismo tiempo que recibe del proceso anterior un nuevo valor. Después acumula la suma. Tras un total de $N-1$ iteraciones, cada proceso conocerá la suma de todos los valores locales de los procesos.

Dar una descripción en pseudocódigo de los procesos siguiendo un estilo SPMD y usando operaciones de envío y recepción síncronas.

```
process P[ i : 0..N-1 ] ;
   var mi_valor : integer := ... ; { valor arbitrario (== i en la figura, por ejemplo) }
       suma     : integer := mi_valor ; { suma inicializada a 'mi_valor'}
begin
   for j := 0 to N-1 do begin
      ...
   end
end
```

36

Considerar un estanco en el que hay tres fumadores y un estanquero. Cada fumador continuamente lía un cigarro y se lo fuma. Para liar un cigarro, el fumador necesita tres ingredientes: tabaco, papel y cerillas. Uno de los fumadores tiene solamente papel, otro tiene solamente tabaco, y el otro tiene solamente cerillas. El estanquero tiene una cantidad infinita de los tres ingredientes.

- El estanquero coloca aleatoriamente dos ingredientes diferentes de los tres que se necesitan para hacer un cigarro, desbloquea al fumador que tiene el tercer ingrediente y después se bloquea. El fumador seleccionado, se puede obtener fácilmente mediante una función `genera_ingredientes` que devuelve el índice (0,1, ó 2) del fumador escogido.
- El fumador desbloqueado toma los dos ingredientes del mostrador, desbloqueando al estanquero, lía un cigarro y fuma durante un tiempo.
- El estanquero, una vez desbloqueado, vuelve a poner dos ingredientes aleatorios en el mostrador, y se repite el ciclo.

Describir una solución distribuida que use envío asíncrono seguro y recepción síncrona, para este problema usando un proceso `Estanquero` y tres procesos fumadores `Fumador(i)` (con `i`=0,1 y 2).

```
process Estanquero ;
begin
   while true do begin
      ....
   end
end
```

```
process Fumador[ i : 0..2 ] ;
begin
   while true do begin
      ....
   end
end
```

37

En un sistema distribuido, un gran número de procesos clientes usa frecuentemente un determinado recurso y se desea que puedan usarlo simultáneamente el máximo número de procesos. Para ello, los clientes envían peticiones a un proceso controlador para usar el recurso y esperan respuesta para poder usarlo (véase el código de los procesos clientes). Cuando un cliente termina de usar el recurso, envía una solicitud para dejar de usarlo y espera respuesta del Controlador. El proceso controlador se encarga de asegurar la sincronización adecuada imponiendo una única restricción por razones supersticiosas: nunca habrá 13 procesos exactamente usando el recurso al mismo tiempo.

```
process Cli[ i : 0....n ] ;
var pet_usar    : integer := +1 ;
    pet_liberar : integer := -1 ;
    permiso     : integer := ... ;
begin
   while true do begin
      send( pet_usar, Controlador );
      receive( permiso, Controlador );

      Usar_recurso( );

      send( pet_liberar, Controlador );
      receive( permiso, Controlador );
   end
end
```

```
process Controlador ;
begin
   while true do begin
     select

       ...

   end
end
```

Describir en pseudocódigo el comportamiento del proceso controlador, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos. Es posible utilizar una sentencia del tipo `select for i=... to ...` para especificar diferentes ramas de una sentencia selectiva que comparten el mismo código dependiente del valor de un índice `i`.

38

En un sistema distribuido, tres procesos **Productor** se comunican con un proceso **Impresor** que se encarga de ir imprimiendo en pantalla una cadena con los datos generados por los procesos productores. Cada proceso productor (**Productor**`[`i`]` con $i = 0, 1, 2$) genera continuamente el correspondiente entero i, y lo envía al proceso **Impresor**.

El proceso **Impresor** se encarga de ir recibiendo los datos generados por los productores y los imprime por pantalla (usando el procedimiento **imprime** $(entero)$) generando una cadena dígitos en la salida. No obstante, los procesos se han de sincronizar adecuadamente para que la impresión por pantalla cumpla las siguientes restricciones:

- Los dígitos 0 y 1 deben aceptarse por el impresor de forma alterna. Es decir, si se acepta un 0 no podrá volver a aceptarse un 0 hasta que se haya aceptado un 1, y viceversa, si se acepta un 1 no podrá volver a aceptarse un 1 hasta que se haya aceptado un 0.
- El número total de dígitos 0 o 1 aceptados en un instante no puede superar el doble de número de digitos 2 ya aceptados en dicho instante.

Cuando un productor envía un digito que no se puede aceptar por el imprersor, el productor quedará bloqueado esperando completar el **s_send**.

El pseudocódigo de los procesos productores (**Productor**) se muestra a continuación , asumiendo que se usan operaciones bloqueantes no buferizadas (síncronas).

```
process Productor[ i : 0,1,2 ]
while true do begin
   s_send( i, Impresor ) ;
end
```

Escribir en pseudocódigo el código del proceso **Impresor**, utilizando un bucle infinito con una orden de espera selectiva **select** que permita implementar la sincronización requerida entre los procesos, según este esquema:

```
Process Impresor
var
  .....
begin
  while true do begin
     select
        .....
     end
  end
end
```

39

En un sistema distribuido hay un vector de `n` procesos iguales que envían con **`send`** (en un bucle infinito) valores enteros a un proceso receptor, que los imprime.

Si en algún momento no hay ningún mensaje pendiente de recibir en el receptor, este proceso debe de imprimir "no hay mensajes. duermo.z después bloquearse durante 10 segundos (con **`sleep_for`**`(10)`), antes de volver a comprobar si hay mensajes (esto podría hacerse para ahorrar energía, ya que el procesamiento de mensajes se hace en ráfagas separadas por 10 segundos).

Este problema no se puede solucionar usando **`receive`** o **`i_receive`**. Indica a que se debe esto. Sin embargo, sí se puede hacer con **`select`**. Diseña una solución a este problema con **`select`**.

```
process Emisor[ i : 1..n ]
    var dato : integer ;
begin
   while true do begin
        dato := Producir() ;
        send( dato, Receptor );
   end
end
process Receptor()
   var dato : integer ;
begin
   while true do
     ......
end
```

40

En un sistema tenemos N procesos emisores que envían de forma segura un único mensaje cada uno de ellos a un proceso receptor, mensaje que contiene un entero con el número de proceso emisor. El proceso receptor debe de imprimir el número del proceso emisor que inició el envío en primer lugar. Dicho emisor debe terminar, y el resto quedarse bloqueados.

```
process Emisor[ i : 1.. N ]
begin
   s_send(i,Receptor);
end
process Receptor ;
   var ganador : integer ;
begin
   { calcular 'ganador' }
   ....
   ....
   print "El primer envio lo ha realizado: ....", ganador ;
end
```

Para cada uno de los siguientes casos, describir razonadamente si es posible diseñar una solución a este problema o no lo es. En caso afirmativo, escribe una posible solución:

(a) el proceso receptor usa exclusivamente recepción mediante una o varias llamadas a **receive**

(b) el proceso receptor usa exclusivamente recepción mediante una o varias llamadas a **i_receive**

(c) el proceso receptor usa exclusivamente recepcíon mediante una o varias instrucciones **select**

41

Supongamos que tenemos **N** procesos concurrentes semejantes:

```
process P[ i : 1..N ] ;
   ....
begin
   ....
end
```

Cada proceso produce N-1 caracteres (con N-1 llamadas a la función **ProduceCaracter**) y envía cada carácter a los otros N-1 procesos. Además, cada proceso debe imprimir todos los caracteres recibidos de los otros procesos (el orden en el que se escriben es indiferente).

(a) Describe razonadamente si es o no posible hacer esto usando exclusivamente **s_send** para los envíos. En caso afirmativo, escribe una solución.

(b) Escribe una solución usando **send** y **receive**

42

Escribe una nueva solución al problema anterior en la cual se garantize que el orden en el que se imprimen los caracteres es el mismo orden en el que se inician los envíos de dichos caracteres (pista: usa **select** para recibir).

43

Supongamos de nuevo el problema anterior en el cual todos los procesos envían a todos. Ahora cada item de datos a producir y transmitir es un bloque de bytes con muchos valores (por ejemplo, es una imagen que puede tener varios megabytes de tamaño). Se dispone del tipo de datos **TipoBloque** para ello, y el procedimiento **ProducirBloque**, de forma que si `b` es una variable de tipo **TipoBloque**, entonces la llamada a **ProducirBloque**`(b)` produce y escribe una secuencia de bytes en `b`. En lugar de imprimir los datos, se deben consumir con una llamada a **ConsumirBloque**`(b)`.

Cada proceso se ejecuta en un ordenador, y se garantiza que hay la suficiente memoria en ese ordenador como para contener simultáneamente al menos hasta N bloques. Sin embargo, el sistema de paso de mensajes (SPM) podría no tener memoria suficiente como para contener los $(N-1)^2$ mensajes en tránsito simultáneos que podría llegar a haber en un momento dado con la solución anterior.

En estas condiciones, si el SPM agota la memoria, debe retrasar los **`send`** dejando bloqueados los procesos y en esas circunstancias se podría producir interbloqueo. Para evitarlo, se pueden usar operaciones inseguras de envío, **`i_send`**. Escribe dicha solución, usando como orden de recepción el mismo que en el problema anterior (3).

44

En los tres problemas anteriores, cada proceso va esperando a recibir un item de datos de cada uno de los otros procesos, consume dicho item, y después pasa recibir del siguiente emisor (en distintos órdenes). Esto implica que un envío ya iniciado, pero pendiente, no puede completarse hasta que el receptor no haya consumido los anteriores bloques, es decir, se podría estar consumiendo mucha memoria en el SPM por mensajes en tránsito pendientes cuya recepción se ve retrasada.

Escribe una solución en la cual cada proceso inicia sus envíos y recepciones y después espera a que se completen todas las recepciones antes de iniciar el primer consumo de un bloque recibido. De esta forma todos los mensajes pueden transferirse potencialmente de forma simultánea. Se debe intentar que la transimisión y las producción de bloques sean lo más simultáneas posible. Suponer que cada proceso puede almacenar como mínimo 2N bloques en su memoria local, y que el orden de recepción o de consumo de los bloques es indiferente.