Chapter 3

Problemas resueltos: Sistemas basados en paso de mensajes.

28

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`.

Respuesta (privada)

A continuación se da una posible solución. Esta solución usa un vector de valores lógicos (`recibido`) que indica, para cada proceso cliente, si el controlador ha recibido o no ya la petición de dicho cliente. Usando un contador, se determina cuando se han recibido 3 peticiones y por tanto cuando se puede dar paso a un grupo de tres procesos. En ese momento, el vector `recibido` almacena los procesos a los que hay que enviar respuesta.

```
process Controlador ;

var n         : integer := 6 ; {* numero de procesos, n >= 3 *}
    contador : integer := 0 ;
    peticion : integer ;
    permiso  : integer := .... ;
    recibido : array[0..n-1] of boolean := ( false, false, ..., false ) ;
begin
   while true do
      select
         for i := 0 to n-1 when receive( peticion, Cliente[i] )  do
            recibido[i] := true ;
            contador := contador + 1 ;
            if contador == 3 then begin
               contador := 0 ;
               for j := 0 to n-1 do
                  if recibido[j] then begin
                     send( permiso, cliente[j] ) ;
                     recibido[j] := false ;
               end
            end { if .. }
      end { select }
end
```

Otra variante (que usa las mismas variables locales) puede ser la que se incluye aquí abajo. En este caso, cuando el contador llega a 3 solo se puede ejecutar el segundo |when|, que se encarga de recibir los mensajes

```
while true do
   select
      for i := 0 to n-1 when contador < 3 receive( peticion, cliente[i] ) do
         contador := contador + 1 ;
         recibido[i] := true ;
      when contador == 3 do
         for j := 0 to n-1 do
            if recibido[j] then begin
               send( permiso, cliente[j] ) ;
               recibido[j] := false ;
            end
      end { select }
```

29

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
```

Respuesta (privada)

Una posible respuesta sería esta:

```
process Buffer ;

var tam      : integer :=  4 ; { capacidad del buffer }
    ultimo   : integer := -1 ; { indice del ultimo que escribio en buffer }
    contador : integer :=  0 ;
    dato     : integer ;
    buf      : array [0..tam-1] of integer ;
```



```
begin
   while true do
      select
         for i := 0 to 2
         when contador < tam and ultimo!=i receive( dato, Productor[i] ) do
            ultimo := i ;
            buf[contador]:=dato;
            contador := contador + 1 ;
         when contador >= 2 do
            contador:=contador-1;
            send( buf[contador], Consumidor );
      end { select }
end
```

El problema es que corresponde a una solución LIFO, y puesto que solo se envía cuando hay dos elementos, entonces el primer elemento insertado en el buffer |buf| (en la entrada 0) nunca sería enviado, por eso se necesita usar una solución FIFO, como se indica aquí:

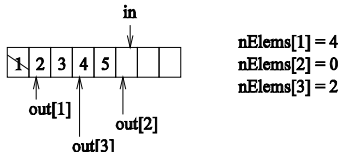
```
process Buffer ;

var tam      : integer :=  4 ; { capacidad del buffer }
    ultimo   : integer := -1 ; { indice del ultimo que escribio en buffer }
    contador : integer :=  0 ; { numero de entradas ocupadas }
    prim_ocu : integer :=  0 ; { primera entrada ocupada }
    prim_lib : integer :=  0 ; { primera entrada libre }
    dato     : integer ;
    buf      : array [0..tam-1] of integer ;

begin
   while true do
      select
         for i := 0 to 2
         when contador < tam and ultimo != i receive(&dato,Productor[i]) do
            ultimo := i ;
            buf[prim_lib]:= dato ;
            prim_lib := (prim_lib+1) mod tam ;
            contador := contador + 1 ;
         when contador >= 2 do
            prim_ocu := (prim_ocu+1) mod tam ;
            contador := contador - 1 ;
            send (&buf[contador], Consumidor);
      end { select }
end
```

30

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).



Respuesta (privada)

Se asumen operaciones con semántica bloqueante sin buffer. El código de los procesos consumidores y el productor es casi idéntico al de los productores y consumidores del ejercicio 2. Ahora una celda no está vacía si todavía queda algún consumidor por leer dicha celda. Por tanto, no se puede recibir del productor si algún consumidor tiene tantos valores pendientes de leer como entradas tiene el búfer.

```
process Buffer ;

var B       : integer  := ... ; { capacidad del buffer }
    in      : integer  := 0 ;
    dato    : integer;
    buf     : array [0..B-1] of integer ;
    nElems  : array [1..3]   of integer := (0,0,0) ;
    out     : array [1..3]   of integer := (0,0,0) ;
    max     : integer ; { máximo número de valores pendientes de leer }
begin
   while true do
      { hacer max := máximo valor almacenado en el array nElems }
      for j := 1 to 3 do
         if max < nElems[j] or j == 1 then
            max := nElems[j] ;

      { espera selectiva }
      select
         for i := 1 to 2 when nElems[i] > 0 do
            send( buf[out[i]], Consumidor[i] ); { enviar }
            out[i] := (out[i]+1) mod B;          { avanzar índice de salida de consum i }
```



```
            nElems[i] := nElems[i] - 1;          { decrementar pendientes de consum i }

         when max < B receive(dato,Productor) do
            buf[in] := dato ;                    { guardar dato }
            in := (in+1) mod B;                  { avanzar indice de entrada }
            for j := 1 to 3 do                   { para cada consumidor j }
               nElems[j] := nElems[j]+1 ;        {  incrementar pendientes de consum j }
   end
end
```

31

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.

Respuesta (privada)

Los salvajes deben de enviar sus mensajes a la olla con **s_send**, ya que deben de esperar a que quede algún misionero disponible antes de comérselo. La olla solo acepta los mensajes de los salvajes cuando hay misioneros. El cocinero debe esperar en un **receive** (síncrono) hasta que tiene que rellenar, después envía confirmación a la olla (esta confirmación puede hacerse con **send**)

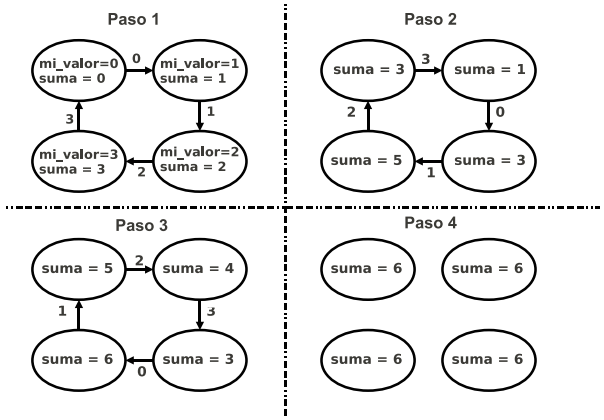
```
process Salvaje[ i : 0..2 ] ;
begin
   var peticion : integer := ... ;
begin
   while true do begin
      { esperar a servirse un misionero: }
      s_send( peticion, Olla );
      { comer: }
      Comer();
   end
end
```

```
process Cocinero ;
   var llenar       : integer ;
       confirmacion : integer := ...;
begin
   while true do begin
      { dormir esperando solicitud para llenar: }
      receive( llenar, Olla );
      { rellenar olla: }
      send( confirmacion, Olla );
   end
end
```

```
process Olla ;
   var contador   : integer :=  M ; { la olla está inicialmente llena }
       llenar     : integer := ...; { valor indiferente }
       esta_llena : integer ;
begin
   while true do
      select
         for i := 0 to 2 when contador > 0 receive( peticion, Salvaje[i] ) do
            contador := contador - 1 ;
         when contador == 0 do
            send( llenar, Cocinero);          { despertar al cocinero }
            receive( esta_llena, Cocinero); { esperar confirmación }
            contador := M;
      end { select }
end
```

32

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
```

Respuesta (privada)

El programa es sencillo, solo hay que tener en cuenta que, para evitar interbloqueos, cada proceso envía con **send** (asíncrono) y después, cuando la variable ya se ha leído, pero sin esperar la recepción, hace **receive** síncrono.

```
process P[ i : 0..N-1 ] ;

var mi_valor : integer := ... ;
    suma     : integer ;
```

```
begin
   for j := 0 to N-2 do begin
      send    ( mi_valor, P[ (i+1) mod N ] ) ;
      receive( mi_valor, P[ (i+N-1) mod N ] );
      suma := suma + mi_valor ;
   end
end
```

33

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
```

Respuesta (privada)


```
process Estanquero ;
   var ingredientes : integer := ...;
       confirmacion : integer ;
       i            : integer ;
begin
   while true do begin
      i := genera_ingredientes();
      send( ingredientes, Fumador[i] );
      receive( confirmacion, Fumador[i]
   end
end
```

```
process Fumador[ i : 0..2 ] ;
   var ingredientes : integer := ...;
       confirmacion : integer ;
begin
  while true do begin
    receive( ingredientes, Estanquero );
    send( confirmacion, Estanquero);
    Fumar();
  end
end
```

34

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`.

Respuesta (privada)

Una solución consiste en usar los valores (+1 y -1) asociados con la petición de uso y la petición de liberar, respectivamente. Asimismo, usamos un valor (`pendiente`) que indica si hay una petición de uso

pendiente (vale +1), o bien si hay una petición de liberar pendiente (vale −1), o bien no hay ninguna petición pendiente (vale 0). Finalmente, usamos una variable con el identificador de proceso que está esperando (**cliente_e**).

En esta solución hay que tener en cuenta que nunca puede haber más de una petición pendiente, ya que si hay una petición pendiente y llega otra, con seguridad podremos atender la nueva y la pendiente (o bien se compensan, si son de distinto signo, y el contador no cambia, o bien se acumulan, si son de igual signo, y el contador *se salta* el valor 13)

```
process Controlador ;
var
   permiso    : integer := ... ;
   peticion   : integer ;
   pendiente  : integer := 0 ;   { indica si no hay pendiente (0), o hay pte (−1 o +1)}
   cliente_e  : integer ;        { numero de cliente esperando, si 'pendiente' no es 0 }
   contador   : integer := 0  ; { numero de clientes usando el recurso }
begin
   while true do begin
      select
      for i := 0 to n when receive( peticion, Cli[i] ) do

         if contador + peticion + pendiente == 13 then  begin
            { no se puede atender (pendiente será  0 aquí con seguridad) }
            pendiente := peticion ; { registrar que clase de petición queda pendiente }
            cliente_e := i ; { registrar que cliente queda pendiente }
         end
         else begin
            { se puede atender, contador tomará un valor distinto de 13 }
            contador := contador + peticion + pendiente ;
            send( permiso, Cliente[i] );
            if pendiente != 0 then begin
               send( permiso, Cliente[cliente_e])
               pendiente := 0 ;
            end
         end
         else

      end { select }
   end { while true }
end { process }
```

35

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
```

Respuesta (privada)

```
Process Impresor ;
var
   num01     : integer := 0 ;    { numero de veces que se ha aceptado el 0 o el 1 }
   num2      : integer := 0 ;    { numero de veces que se ha aceptado el 2       }
   numero    : integer ;         { numero recibido                               }
   ultimo01  : integer := −1 ;   { ultimo digito 0 o 1 aceptado, −1 al principio  }
begin
   while true do begin
      select
         { si se puede aceptar un 0, recibirlo }
         when num01 < 2*num2 and ultimo01 != 0  receive( numero, Productor[0] ) do
            imprime( numero ) ;
```

```
            num01    := num01 + 1 ;
            ultimo01 := 0 ;
         { si se puede aceptar un 1, recibirlo }
         when num01 < 2*num2 and ultimo01 != 1  receive( numero, Productor[1] ) do
            print( numero ) ;
            num01    := num01 + 1 ;
            ultimo01 := 1 ;
         { se puede aceptar un 2 siempre: recibirlo }
         when receive( numero, Productor[2] ) do
            print( numero ) ;
            num2 := num2 + 1 ;
      end
   end
end
```

36

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." y 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
```

Respuesta (privada)

La solución no puede hacerse con **receive** o **i_receive**, ya que no se dispone de ninguna forma de saber si hay mensajes pendientes o no los hay, y necesitamos saber esto para decidir en el receptor si se debe dormir o se debe hacer **receive**, de acuerdo a los requerimientos.

Sin embargo, la sentencia **select** sí permite incluir guardas sin sentencia de entrada, guardas que solo

se considerarán para su ejecución en los casos en los que las guardas con sentencia de entrada ejecutables no tengan envíos pendientes que casen con ellas.

Usando esta característica de **select**, el código se escribiría como sigue:

```
process Receptor()
   var dato : integer ;
begin
   while true do
      select
         { si hay mensajes de un emisor, leer uno }
         for i := 1 to n when receive( dato, Emisor[i] ) do
            print "recibido: ", dato ;
         when true do  { siempre es ejecutable, pero no se ejecuta si hay mensajes pendientes }
            print "no hay mensajes, duermo." ;
            sleep_for(10) ;
      end
end
```

37

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**

Respuesta (privada)

(a) no es posible, ya que el orden en el que se reciben los mensajes es necesariamente el mismo orden en el que el receptor llama a **receive** para los distintos procesos emisores, orden que en general no puede coincidir con el orden en el que se llama a **s_send**, (este orden es desconocido en el receptor).

(b) no es posible, por el mismo motivo que antes. Ahora el orden en el que se reciben los mensajes no tiene porque coincidir con el orden en el que se hacen las llamadas a **i_receive** en el receptor, pero el orden de dichas llamadas tampoco coincide con el orden de envío, que sige siendo desconocido.

(c) en este caso sí es posible, ya que la sentencia **select**, en caso de que haya más de un mensaje iniciado que se pueda recibir, seleccionará el primero que comenzó a enviarse, por tanto podemos garantizar de forma sencilla que se reciben en el orden de envío. La solución sería:

```
process Receptor ;
begin
   select
      for i := 1 to N when receive(num,Emisor[i])
            ganador := i ;
   end
   print "El primer envio lo ha realizado: ....", ganador ;
end
```

38

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**

Respuesta (privada)

(a) Es imposible hacerlo con **s_send**, ya que se produciría interbloqueo. Cada proceso se quedaría bloqueado en su primer **s_send**, ya que ese proceso esperaría un recepción que el proceso destinatario no puede iniciar, al estar también bloqueado en el mismo **s_send**.

(b) La solución con **send**/**receive** es sencilla.


```
process P[ i : 1..N ] ;
   var c : char ;
begin
   { iniciar todos los envios }
   for j := 1 to N do
      if i != j then begin
         c := ProduceCaracter() ;
         send( c, P[j] );
      end
   { hacer todas las recepciones }
   for j := 1 to N do
      if i != j then begin
         receive( c, P[j] ) ;
         print( c );
      end
end
```

39

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).

Respuesta (privada)

La solución con **select** es sencilla: basta con ejecutar N-1 veces dicho **select** en un **for**. Cada vez se seleccionará el emisor que más tiempo lleve esperando, lo cual garantiza el orden que se pide en el enunciado.

El código queda así:

```
process P[ i : 1..N ] ;
   var c : char ;
begin
   { iniciar todos los envios }
   for j := 1 to N
      if i != j then begin
         c := ProduceCaracter() ;
         send( c, P[j] );
      end
   { hacer todas las N-1 recepciones }
   for k := 1 to N-1 do
      select
         for j := 1 to N when i != j receive( c, P[j] ) do
               print( c );
      end
end
```

40

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).

Respuesta (privada)

Una solución sencilla consiste en adoptar el mismo esquema que antes, pero sustituyendo **send** por **i_send** (que no se bloquea nunca, pues no espera), de forma que ahora la falta de memoria no puede bloquear los procesos. Quedaría asi:

```
process P[ i : 1..N ] ;
   var bloque  : TipoBloque ;
begin
   { iniciar todos los envios }
   for j:= 1 to N
      if i != j then begin
         ProducirBloque( bloque ) ;
         i_send( bloque, P[j] );
      end

   { hacer todas las N-1 recepciones }
   for k := 1 to N-1 do
      select
         for j:= 1 to N when i != j receive( bloque, P[j] ) do
               Consumirbloque( bloque );
      end
end
```

Claramente, este diseño es incorrecto, ya que no se garantiza la seguridad. La segunda llamada a **ProducirBloque** puede sobreescribir el primer bloque que podría no haber sido terminado de leer por el SPM para enviarlo. También puede ocurrir que un proceso acabe sin que se hayan leido sus datos.

Para evitarlo, se puede usar un array de N bloques, que sabemos que caben en la memoria de cada proceso, y no acabar hasta que no hayan terminado todos los envíos.

Por tanto, se usa un array de bloques (de nombre **bloque**). Se produce un bloque en cada entrada del array y se inicia el envío, sin esperar a que se complete ninguno de esos envíos. Una vez comenzado el envio de todos, se puede iniciar las recepciones y el consumo, que se pueden hacer igual que antes.

Finalmente, será necesario esperar a que se terminen todos los envios, antes de finalizar los procesos, ya que los bloques en proceso de envio deben permanecer en la memoria local del proceso. Para ello necesitamos un array de variables de resguardo, cada una de ellas asociada a uno de los **i_send** (array **estado**).

```
process P[ i : 1..N ] ;
   var bloque   : array[1..N]  of TipoBloque ;     { N-1 para envio, 1 (i) para recepcion }
   var estado   : array[1..N]  of TipoResguardo ; { estado de los envios }
begin

   { iniciar todos los envios }
   { (se usan todas las entradas de 'bloque', excepto la i-esima) }
   for j := 1 to N do
      if i != j then begin
         ProducirBloque( bloque[j] ) ;
         i_send( bloque[j], P[j], estado[j] );
      end

   { hacer las N-1 recepciones }
   { (se hacen todas en el bloque 'bloque[i ]') }
   for k := 1 to N-1 do
      select
         for j:= 1 to N when i != j receive( bloque[i], P[j] ) do
               ConsumirBloque( bloque[i] )
      end

  { esperar a que terminen todos los envios }
  for j := 1 to N do
     if i != j then
        wait_send( estado[j] ) ;
end
```

41

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.

Respuesta (privada)

Basta con hacer las recepciones ahora con **i_receive**, en lugar de **select** o **receive**, usando un vector de bloques en proceso de recepción (**bloque_rec**), adicional al vector que usamos para los envíos en proceso (**bloque_env**). Ahora se inician las recepciones al principio, de forma que ahora se facilita que los envíos encuentren una recepción que encaje con cada uno de ellos. Los consumos se podrán hacer cuando se hayan terminado todas las recepciones. Al igual que antes, el programa no puede acabar hasta que se hayan completado todos los envíos.

```
process P[ i : 1..N ] ;
   var bloque_env   : array[1..N]  of TipoBloque ;     { bloques producidos }
   var bloque_rec   : array[1..N]  of TipoBloque ;     { bloques recibidos  }
   var estado_env   : array[1..N]  of TipoResguardo ; { estado de los envios }
   var estado_rec   : array[1..N]  of TipoResguardo ; { estado de las recepciones }
begin
   { iniciar todas las recepciones }
   for j := 1 to N do
      if i != j then
         i_receive( bloque_rec[j], P[j], estado_rec[j] );

   { producir e iniciar todos los envios }
   for j := 1 to N do
      if i != j then begin
         ProducirBloque( bloque_env[j] ) ;
         i_send( bloque_env[j], P[j], estado_env[j] );
      end

   { esperar que terminen las recepciones  }
   for j := 1 to N-1 do
      if i != j then
         wait_recv( estado_rec[j] );

   { consumir todos los bloques }
   for j := 1 to N-1 do
      if i != j then
         ConsumirBloque( bloque_rec[j] );

  { esperar a que terminen todos los envios }
  for j := 1 to N do
     if i != j then
        wait_send( estado_env[j] ) ;
end
```

Chapter 4

Problemas resueltos: Sistemas de Tiempo Real.

42

Dado el conjunto de tareas periódicas y sus atributos temporales que se indica en la tabla de aquí abajo, determinar si se puede planificar el conjunto de dichas tareas utilizando un esquema de planificación basado en planificación cíclica. Diseña el plan cíclico determinando el marco secundario, y el entrelazamiento de las tareas sobre un cronograma.

Tarea	C_i	T_i	D_i
T1	10	40	40
T2	18	50	50
T3	10	200	200
T4	20	200	200

Respuesta (privada)

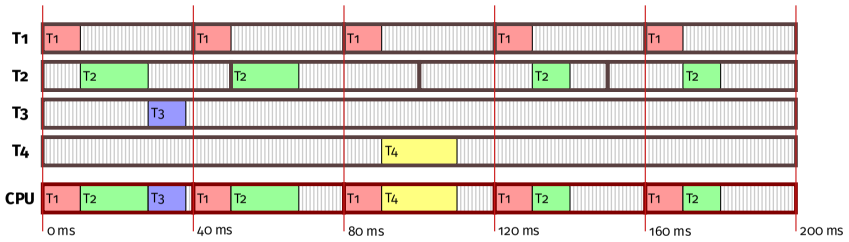
Para calcular la planificabilidad con ejecutivos cíclicos hay que calcular el hiperperiodo T_M que es $\text{mcm}(40,50,200,200) = 200$. Ahora calculamos el ciclo secundario, aplicando las siguientes condiciones según la teoría:

1. $T_S >= \max(10,18,10,20) = 20$.
2. $T_S \leq \min(40,50,200,200) = 40$.
3. T_S es divisor de $T_M = 200$.

Como consecuencia, el marco secundario puede valer 20, 25 o 40. Para diseñar el ejecutivo cíclico tenemos que distribuir la ejecución de las distintas tareas entre los marcos secundarios que se han establecido dentro

de un hiperperiodo, ya que luego el comportamiento se repite indefinidamente. Si suponemos que el marco secundario es $T_S = 40$, el cronograma podría ser:

Cada tarea tiene que cumplir con las restricciones temporales impuestas en el cuadro de parámetros temporales. Así, la tarea 1 tiene que ejecutarse 5 veces, una en el intervalo $[0,40]$, otra en el intervalo $[40,80]$, otra en el intervalo $[80,120]$, otra en el intervalo $[120,160]$, y por último en el intervalo $[160,200]$. Por ejemplo, la tarea 4 se tiene que ejecutar una vez en el intervalo $[0,200]$, por lo que se busca un hueco adecuado.



43

El siguiente conjunto de tareas periódicas se puede planificar con ejecutivos cíclicos. Determina si esto es cierto calculando el marco secundario que debería tener. Dibuja el cronograma que muestre las ocurrencias de cada tarea y su entrelazamiento. ¿Cómo se tendría que implementar? (escribe el pseudo-código de la implementación)

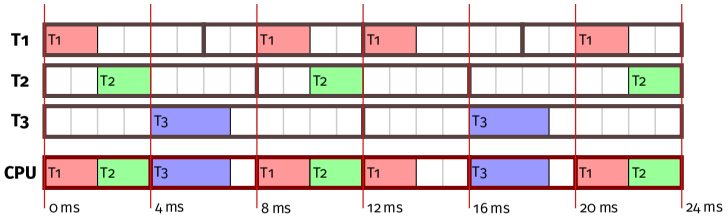
Tarea	C_i	T_i	D_i
T1	2	6	6
T2	2	8	8
T3	3	12	12

Respuesta (privada)

Para calcular la planificabilidad con ejecutivos cíclicos hay que calcular el hiperperiodo T_M, que es igual a $\text{mcm}(6,8,12) = 24$. Respecto a la duración del ciclo secundario T_S, tenemos en cuenta las restricciones que debemos o podemos aplicar, son estas:

1. $T_S >= \max(2,2,3) = 3$.
2. $T_S \leq \min(6,8,12) = 6$.
3. T_S es divisor de $T_M = 24$.

Por tanto, el valor T_S en principio puede ser 3, 4, o 6. Si seleccionamos $T_S = 4$, obtenemos esta posible solución:



Respecto a la implementación, podemos hacerla como se indica en este pseudo-código:

```
process EjecutivoCiclico ;
  var inicio : time_point := now() ; { instante inicio ciclo principal }
begin
  while true do begin { ciclo principal }
    { ejecutar cada una de las 4 iteraciones del ciclo secundario }
    T1 ; T2 ; sleep_until( inicio+4 );
    T3 ;      sleep_until( inicio+8 );
    T1 ; T2 ; sleep_until( inicio+12 );
    T1 ;      sleep_until( inicio+16 );
    T3 ;      sleep_until( inicio+20 );
    T1 ; T2 ; sleep_until( inicio+24 );
    inicio = inicio + 24 ; { actualizar instante de inicio de c.p. }
  end
end
```

44

Comprobar si el conjunto de procesos periódicos que se muestra en la siguiente tabla es planificable con el algoritmo RMS utilizando el test basado en el factor de utilización del tiempo del procesador. Si el test no se cumple, ¿debemos descartar que el sistema sea planificable?

Tarea	C_i	T_i
T1	9	30
T2	10	40
T3	10	50

Respuesta (privada)

Para comprobar la planificabilidad con RMS, en primer lugar calculamos el factor de utilización U:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{9}{30} + \frac{10}{40} + \frac{10}{50} = \frac{3}{4} = 0,75$$

También calculamos $U_0(n)$ con $n = 3$ y lo comparamos con U. Obtenemos:

$$U = 0,75 < 0,779 = 3\left(\sqrt[3]{2} - 1\right) = U_0(3)$$

Por tanto, vemos que para estos atributos temporales el test da **resultado positivo**, y como consecuencia podemos afirmar que el sistema **es planificable usando RMS**.

Respecto a la pregunta ¿debemos descartar que el sistema sea planificable?, la respuesta es:

Si el test no se hubiese cumplido, es decir, si hubieramos obtenido $U_0(3) < U \leq 1$, no podríamos descartar que el sistema sea planificable, ya que este test para RMS es **suficiente** (garantiza planificabilidad cuando $U \leq U_0(n)$), pero no es **necesario** (no descarta planificabilidad cuando $U_0(n) < U \leq 1$).

Finalmente, si hubiera ocurrido que $1 < U$, entonces el sistema no es planificable de ninguna forma con un solo procesador (se necesitarían más, en concreto uno más que la parte entera de U).

45

Considérese el siguiente conjunto de tareas compuesto por tres tareas periódicas:

Tarea	C_i	T_i
T1	10	40
T2	20	60
T3	20	80

Comprueba la planificabilidad del conjunto de tareas con el algoritmo RMS utilizando el test basado en el factor de utilización. Calcular el hiperperiodo y construir el correspondiente cronograma.

Respuesta (privada)

De nuevo calculamos el valor de U, que ahora es:

$$U = \frac{10}{40} + \frac{20}{60} + \frac{20}{80} = \frac{5}{6} = 0,833333$$

En este caso vemos que no se cumple la desigualdad requerida:

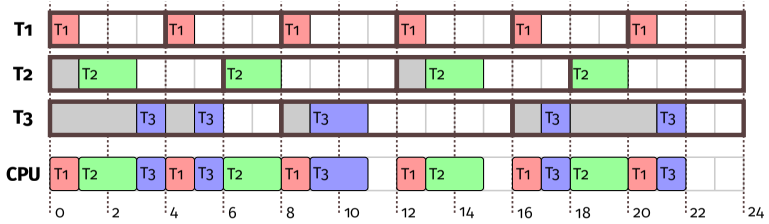
$$U = 0,8333 \not\leq 0,7779 = U_0(3)$$

y como consecuencia el test da un **resultado negativo**, luego **no permite afirmar ni negar** que este problema sea planificable con RMS.

El hiperperiodo T_M es el mínimo común múltiplo de los periodos de cada tarea del conjunto, y representa el intervalo de tiempo a partir del cual se repite el comportamiento temporal del sistema (cuando todas las tareas se activan de nuevo a la vez). En este caso es:

$$T_M = \text{mcm}(40, 60, 80) = 240$$

Si dibujamos el cronograma cubriendo el intervalo de tiempo desde 0 (inicio) hasta T_M (fin del ciclo principal), vemos que, aunque no pasa el test de planificabilidad, **el sistema sí es planificable**.



46

Comprobar la planificabilidad y construir el cronograma de acuerdo al algoritmo de planificación RMS del siguiente conjunto de tareas periódicas.

Tarea	C_i	T_i
T1	20	60
T2	20	80
T3	20	120

Respuesta (privada)

Para comprobar la planificabilidad con RMS, en primer lugar calculamos el factor de utilización U:

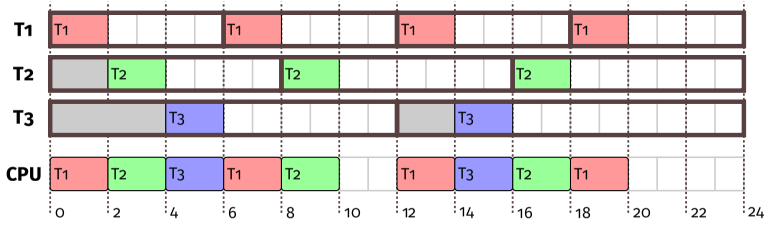
$$U = \frac{20}{60} + \frac{20}{80} + \frac{20}{120} = \frac{3}{4} = 0,75$$

También comparamos $U_0(3)$ con U. Obtenemos:

$$U = 0,75 < 0,779 = U_0(3)$$

Por tanto, vemos que para estos atributos temporales el test da **resultado positivo**, y como consecuencia podemos afirmar que el sistema **es planificable usando RMS**.

El hiperperiodo es $T_M = \text{mcm}(60,80,120) = 240$. Si se hace la simulación RMS en el intervalo de tiempo entre 0 y T_M, obtenemos el siguiente cronograma:



47

Determinar si el siguiente conjunto de tareas puede planificarse con la política de planificación RMS y con la política EDF, utilizando los tests de planificabilidad adecuados para cada uno de los dos casos. Comprobar tambien la planificabilidad en ambos casos construyendo los dos cronogramas.

Tarea	C_i	T_i
T1	1	5
T2	1	10
T3	2	20
T4	10	20
T5	7	100

Respuesta (privada)

En primer lugar calculamos el valor de U

$$U = \frac{1}{5} + \frac{1}{10} + \frac{2}{20} + \frac{10}{20} + \frac{7}{100} = \frac{97}{100} = 0,97$$

el valor de T_M

$$T_M = \text{mcm}(5,10,20,100) = 100$$

y el valor de $U_0(5)$

$$U_0(5) = 5\left(\sqrt[5]{2} - 1\right) = 0,74349$$

(1) Prioridades estáticas RMS

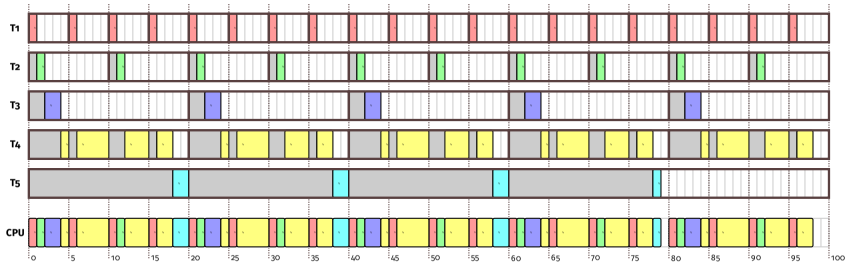
Pasamos el test de planificabilidad RMS basado en el factor de utilización:

$$U = 0,97 \not\leq 0,74349 = U_0(5)$$

Vemos que falla el test, por lo que no podemos afirmar ni negar la planificabilidad únicamente usando dicho test.

Realizamos el cronograma desde el inicio hasta $T_M = 100$ (se incluye a continuación). Las prioridades van en orden: la tarea 1 es de máxima prioridad, la 2 la siguiente, y así hasta la tarea 5 de mínima prioridad (consideramos que la 3 tiene más prioridad que la 4, también se puede hacer al revés sin que eso afecte a la planificabilidad, aunque produce una interfoliación distinta de las tareas 3 y 4).

Vemos que no hay ningún fallo en este intervalo de tiempo, por tanto no lo habrá nunca y podemos decir que **el sistema es planificable con RMS**.



(2) Prioridades dinámicas EDF

El valor de U es inferior a la unidad, luego podemos afirmar que el sistema **es planificable con EDF**.

Respecto al cronograma, en este caso, cada vez que actua el planificador debe de calcular, para cada tarea, a cual o cuales de ellas les resta un tiempo mínimo hasta el siguiente fin de su período (ya que los plazos coinciden con los períodos).

En este ejemplo, si consideramos la lista ordenada de los períodos distintos de las tareas, vemos que cada posible período en esa lista es múltiplo (el doble) del anterior. Esto implica que, si se evaluan en cualquier instante los tiempos hasta el siguiente fin de período, la tarea con el menor tiempo restante es siempre la tarea con el período menor. Como consecuencia, cada vez que se evaluan las prioridades (no importa en que instante) resulta que esas prioridades coinciden con la prioridades RMS. Por tanto, la interfolicación que se produce es la misma que con RMS.

48

Describe razonadamente si el siguiente conjunto de tareas puede planificarse o no puede planificarse en un sistema monoprocesador usando un ejecutivo cíclico o usando algún algoritmo basado en prioridades estáticas o dinámicas.

Tarea	C_i	T_i
T1	1	5
T2	1	10
T3	2	10
T4	10	20
T5	7	100