ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Ingeniería Informática + ADE

Universidad de Granada (UGR)

Autor: Ismael Sallami Moreno

Asignatura: Ejercicios Tema 3 SCD: Paso de mensajes

Índice

1 Ejercicio 77

1.1. Enunciado

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.

1.2. Solucion

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.

El controlador cuenta las peticiones que le llegan de los procesos:

- Las dos primeras peticiones no son respondidas, lo que provoca que los procesos que las envían queden bloqueados esperando una respuesta.

- La tercera petición produce el desbloqueo de los tres procesos pendientes, enviándoles una respuesta.

Una vez desbloqueados los tres procesos, el controlador inicializa la cuenta y procede cíclicamente de la misma forma para nuevas peticiones. Los clientes usan envío asíncrono seguro para realizar su petición y esperan con una recepción síncrona antes de realizar la tarea.

El código de los procesos clientes es el siguiente:

```
process Cliente[ i : 0..5 ] ;
begin
  while true do begin
    send( peticion, Controlador );   // Envía una petición al proceso
       controlador.
    receive( permiso, Controlador ); // Espera un permiso del controlador.
    Realiza_tarea_grupal();          // Realiza la tarea grupal una vez
       recibido el permiso.
  end
end
```

Listing 1: Código de los procesos clientes

El comportamiento del proceso controlador, que asegura la sincronización, se describe en pseudocódigo a continuación:

```
process Controlador ;
var
  contador : integer := 0;           // Cuenta las peticiones recibidas.
  buffer : array[0..2] of integer;   // Almacena los índices de los
     procesos en espera.

begin
  while true do begin
    select
      for i := 0 to 5                // Itera sobre las peticiones de los
         procesos clientes.
      when receive( peticion, Cliente[i] ) do
        buffer[contador] := i;       // Almacena el índice del proceso
           cliente en el buffer.
        contador := contador + 1;    // Incrementa el contador de
           peticiones.

        if contador = 3 then begin   // Si se han recibido tres peticiones
           ...
          for j := 0 to 2 do
            send( permiso, Cliente[buffer[j]] ); // Envía permiso a los
               tres procesos.
          contador := 0;             // Reinicia el contador.
        end
    end
  end
end
```

Listing 2: Pseudocódigo del proceso Controlador

Explicación del código del controlador:

- El proceso controlador mantiene un contador de peticiones y un buffer que almacena los índices de los clientes en espera.
- Por cada petición recibida, almacena el índice del cliente en el buffer y aumenta el contador.
- Cuando el contador alcanza tres, el controlador envía permisos a los tres procesos almacenados en el buffer y reinicia el contador.

Este enfoque asegura que los clientes se sincronizan correctamente antes de realizar la tarea grupal, tal como se requiere en el enunciado.

2 Ejercicio 78

2.1. Enunciado

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:

- Envía enteros al proceso consumidor siempre que su array local tenga al menos dos elementos disponibles.
- 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 del productor y del consumidor es el siguiente:

```
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

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

Se pide: describir en pseudocódigo el comportamiento del proceso buffer, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos.

2.2. Solución

```
Process Buffer;
var
  buffer : array[0..3] of integer; // Array local de 4 celdas enteras.
  count : integer := 0;             // Contador de elementos en el buffer.
  lastProducer : integer := -1;     // Índice del último productor que escribió en el buffer.

  begin

    while true do begin

      select
        when receive(dato, Productor[0]) do
          if(count <4 and lastProducer != 0) then
            buffer[count] := dato; // Almacena el dato en el buffer.
            count := count + 1;   // Incrementa el contador de elementos.
            lastProducer := 0; // Actualiza el índice del último productor.
          end

        when receive(dato, Productor[1]) do
          if(count <4 and lastProducer != 1) then
            buffer[count] := dato; // Almacena el dato en el buffer.
            count := count + 1;   // Incrementa el contador de elementos.
            lastProducer := 1; // Actualiza el índice del último productor.
          end

        when receive(dato, Productor[2]) do
          if(count <4 and lastProducer != 2) then
            buffer[count] := dato; // Almacena el dato en el buffer.
            count := count + 1;   // Incrementa el contador de elementos.
            lastProducer := 2; // Actualiza el índice del último productor.
          end

        when count >= 2 do // Si hay al menos dos elementos en el buffer...
          send(buffer[0], Consumidor); // Envía el primer elemento al consumidor.
          for i := 0 to 2 do
            buffer[i] := buffer[i + 1]; // Desplaza los elementos restantes.
          end
          count := count - 1; // Decrementa el contador de elementos.


```

```
41      end
42
43
44    end
```

Listing 3: Pseudocódigo del proceso Buffer

2.3. Solución usando select for

```
1     Process Buffer;
2     var
3       buffer : array[0..3] of integer; // Array local de 4 celdas enteras.
4       count : integer := 0;             // Contador de elementos en el
            buffer.
5       lastProducer : integer := -1;     // Índice del último productor que
            escribió en el buffer.
6
7     begin
8       while true do begin
9         select for
10          // Guardas indexadas para los productores
11          for i := 0 to 2
12          when (count < 4 and lastProducer != i) receive(dato, Productor[i
                ]) do
13            buffer[count] := dato;    // Almacena el dato en el buffer.
14            count := count + 1;      // Incrementa el contador de elementos
                  .
15            lastProducer := i;       // Actualiza el índice del último
                  productor.
16          end
17
18          // Caso para consumir datos si hay al menos 2 elementos en el
                buffer.
19          when count >= 2 do
20            send(buffer[0], Consumidor); // Envía el primer elemento al
                  consumidor.
21            for j := 0 to 2 do
22              buffer[j] := buffer[j + 1]; // Desplaza los elementos
                    restantes.
23            end
24            count := count - 1;           // Decrementa el contador de
                  elementos.
25
26        end
27      end
28    end
```

Listing 4: Pseudocódigo del proceso Buffer con guardas indexadas

3 Ejercicio 79

3.1. Enunciado

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 buffer mantendría, para cada proceso consumidor, el puntero de salida y el número de elementos que quedan en el buffer por consumir:

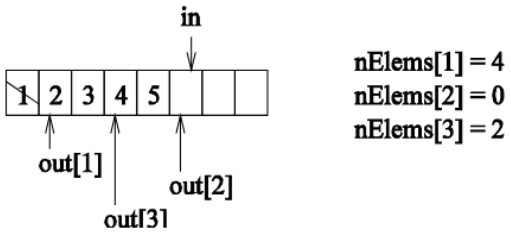

Figura 1: Esquema de interacción entre el productor y los consumidores.

3.2. Solución versión Profesor

```
process Buffer;

var
  out: array[0..2] of integer; // Punteros de salida para cada consumidor.
  nElems: array[0..2] of integer; // Número de elementos que quedan por consumir para cada consumidor.

  B: integer; // Tamaño del buffer.
  buf: array[0..B-1]; // Buffer acotado de tamaño B.
  begin
  select for i:=0 to 2 do

    when nElemens[i] != 0 do // Si el consumidor i tiene elementos por consumir...
```

```
13          send (buf[out[i]], consumidor[i]); // Envía el elemento al
                consumidor i.
14          out[i] + 1 mod B; // Actualiza el puntero de salida del consumidor
                i.
15          nElemens[i] = nElemens[i]-1; // Decrementa el número de elementos
                por consumir del consumidor i.
16        end do
17
18        when nElemens[0] != B or nElemens[1] != B or nElemens[2] != B do //
              Si el buffer no está lleno para algún consumidor...
19          receive(dato, productor); // Recibe un dato del productor.
20          bufer[i] = dato; // Almacena el dato en el buffer.
21          for j:=0 to 2 do
22            nElemens[j] = nElemens[j] + 1; // Incrementa el número de
                  elementos por consumir para cada consumidor.
23          end do
24        end do
25      end select
26    end
```

Listing 5: Pseudocódigo del proceso Buffer

Explicación del código del proceso Buffer

El proceso Buffer se encarga de gestionar la interacción entre un productor y tres consumidores que comparten un buffer acotado de tamaño B. Cada elemento depositado por el productor debe ser retirado por todos los consumidores para ser eliminado del buffer. La idea general del código es mantener un buffer acotado de tamaño B donde el productor puede depositar elementos y los consumidores pueden retirarlos. Cada consumidor tiene su propio puntero de salida y contador de elementos por consumir. El proceso Buffer asegura que cada elemento depositado por el productor sea retirado por todos los consumidores antes de ser eliminado del buffer. Además, se asegura de que el buffer no se llene completamente para ningún consumidor y que los consumidores retiren los elementos en el mismo orden en el que fueron depositados.

3.3. Solución versión Propia

```
1   process Buffer;
2
3   var
4     out: array[0..2] of integer; // Punteros de salida para cada consumidor
          .
5     nElems: array[0..2] of integer; // Número de elementos que quedan por
          consumir para cada consumidor.
6
7     B: integer; // Tamaño del buffer.
8     buf: array[0..B-1]; // Buffer acotado de tamaño B.
9     ocupadas: integer := 0; // Número de celdas ocupadas en el buffer.
10    begin
11    select for i:=0 to 2 do
12
```

```
13      when nElemens[i] != 0 do // Si el consumidor i tiene elementos por
           consumir...
14        send (buf[out[i]], consumidor[i]); // Envía el elemento al
             consumidor i.
15        out[i] + 1 mod B; // Actualiza el puntero de salida del consumidor
             i.
16        nElemens[i] = nElemens[i]-1; // Decrementa el número de elementos
             por consumir del consumidor i.
17      end do
18
19      when nElemens[0] != B or nElemens[1] != B or nElemens[2] != B do //
           Si el buffer no está lleno para algún consumidor...
20        receive(dato, productor); // Recibe un dato del productor.
21        bufer[ocupadas] = dato; // Almacena el dato en el buffer.
22        ocupadas++;
23        for j:=0 to 2 do
24          nElemens[j] = nElemens[j] + 1; // Incrementa el número de
               elementos por consumir para cada consumidor.
25        end do
26      end do
27
28    end select
29  end
```

Listing 6: Pseudocódigo del proceso Buffer

Explicación del código del proceso Buffer

El proceso Buffer se encarga de gestionar la interacción entre un productor y tres consumidores que comparten un buffer acotado de tamaño B. Cada elemento depositado por el productor debe ser retirado por todos los consumidores para ser eliminado del buffer. La idea general del código es mantener un buffer acotado de tamaño B donde el productor puede depositar elementos y los consumidores pueden retirarlos. Cada consumidor tiene su propio puntero de salida y contador de elementos por consumir. El proceso Buffer asegura que cada elemento depositado por el productor sea retirado por todos los consumidores antes de ser eliminado del buffer. Además, se asegura de que el buffer no se llene completamente para ningún consumidor y que los consumidores retiren los elementos en el mismo orden en el que fueron depositados.

Ahora añadimos los procesos Productor y Consumidor:

```
1   process Productor[ i : 0..2 ] ;
2   var dato : integer ;
3   begin
4     while true do begin
5       dato := Producir();
6       send( dato, Buffer );
7     end
8   end
9
10  process Consumidor ;
11  begin
12    while true do begin
13      receive ( dato, Buffer );
```

```
      Consumir( dato );
    end
  end
```

4 Ejercicio 80

4.1. Enunciado

Implementación de los procesos Salvajes y Cocinero

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
    var peticion : integer := ... ;
    begin
        while true do begin
            // esperar a servirse un misionero
            s_send( peticion, Olla );
            // comer:
            Comer();
        end
    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, utilizando 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.

4.2. Solución

```
process Olla;

var
    comida: integer := 0; // Cantidad de comida disponible en la olla.
    capacidad: integer := M; // Capacidad máxima de la olla.
    peticiones: queue of integer; // Cola para gestionar las peticiones de
       los salvajes.

begin
    select
        when not peticiones.empty() and comida > 0 do
            // Atender a un salvaje que quiere comer
            salvaje := peticiones.pop();
            comida := comida - 1;
            send("comer", salvaje);
        end

        when comida = 0 do
            // Solicitar al cocinero que rellene la olla
            send("rellenar", cocinero);
            receive("rellenado", cocinero);
            comida := capacidad;
        end
    end select
end
```

Listing 7: Pseudocódigo del proceso Olla

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

begin
    while true do
        // Solicitar comida a la olla
        send(i, Olla);
        receive("comer", Olla);
        Comer();
    end
end
```

Listing 8: Pseudocódigo del proceso Salvaje

```
process Cocinero;

begin
    while true do
        // Esperar solicitud para rellenar la olla
        receive("rellenar", Olla);
        // Rellenar la olla
        Rellenar();
        send("rellenado", Olla);
    end
end
```

Listing 9: Pseudocódigo del proceso Cocinero

Esta solución asegura:

- Sincronización adecuada entre salvajes y cocinero mediante paso de mensajes.
- Los salvajes pueden comer siempre que haya comida en la olla.
- El cocinero solo se despierta cuando la olla esté vacía.
- Evita el interbloqueo utilizando una cola para gestionar las peticiones de los salvajes.

5 Ejercicio 81

5.1. Enunciado

Considerar un conjunto de N procesos, $P[i]$, $(i = 0, \ldots, N-1)$ 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.

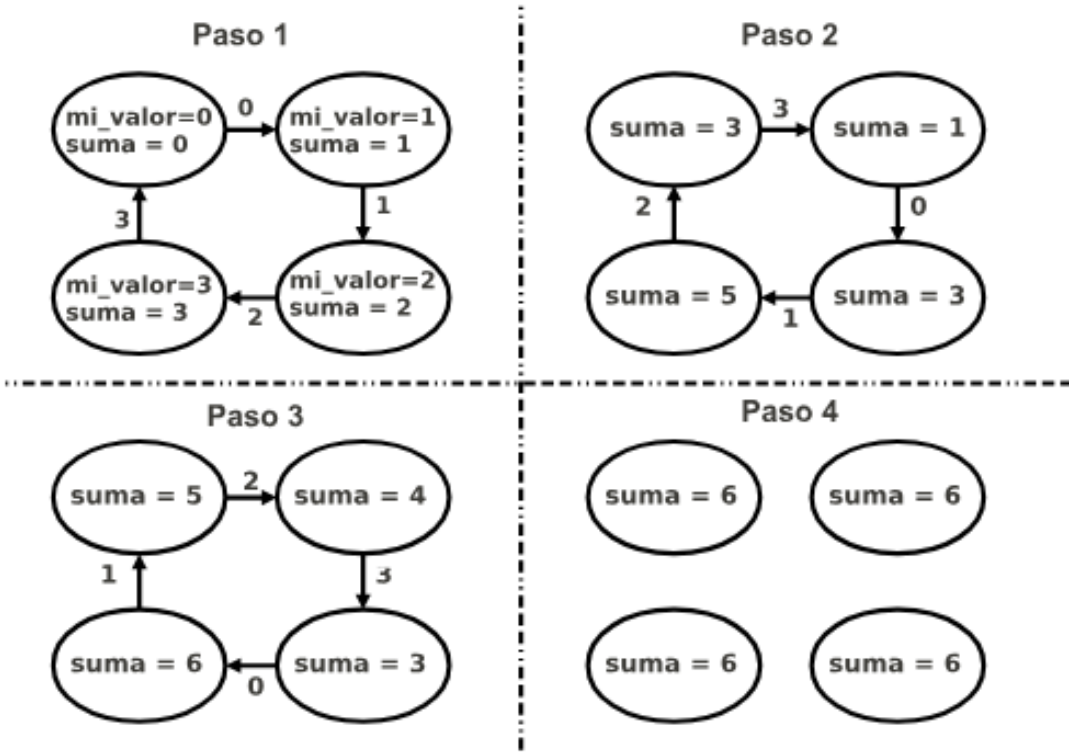

Figura 2: Esquema de interacción entre los procesos.

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
```

5.2. Solución

En este caso se afirma que se debe de seguir un estilo SPMD (Single Program Multiple Data), lo que significa que todos los procesos ejecutan el mismo código, pero con datos diferentes. En este caso, cada proceso $P[i]$ tiene un valor local *mi_valor* que se suma a la suma total.

```
  process P[ i : 0..N-1 ] ;
  var mi_valor : integer := i ; // 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
          send( mi_valor, P[ (i + 1) mod N ] );
          receive( valor_recibido, P[ (i - 1 + N) mod N ] );
          suma := suma + valor_recibido;
          mi_valor := valor_recibido;
      end
  end
```

6 Ejercicio 82

6.1. Enunciado

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
        // El estanquero coloca dos ingredientes aleatorios en el mostrador
        // y desbloquea al fumador que tiene el tercer ingrediente
        ...
    end
end

process Fumador[ i : 0..2 ] ;
begin
    while true do begin
        // El fumador espera hasta que sea desbloqueado
        // Toma los dos ingredientes del mostrador, lía un cigarro y fuma
        ...
    end
end
```

6.2. Solución

Usando envío asíncrono seguro y recepción síncrona. Para ello debemos de usar MPI_Isend y MPI_Recv.

```
  process Estanquero ;
  begin
      while true do begin
          // El estanquero coloca dos ingredientes aleatorios en el
             mostrador
          int ingrediente1, ingrediente2;
          ingrediente1 = generarIngrediente();
          do{
              ingrediente2 = generarIngrediente();
          }
          while(ingrediente1 == ingrediente2);
          // y desbloquea al fumador que tiene el tercer ingrediente
          MPI_Isend(ingrediente1, 1, MPI_INT, Fumador[3 - ingrediente1 -
             ingrediente2], 0, MPI_COMM_WORLD);
          MPI_Isend(ingrediente2, 1, MPI_INT, Fumador[3 - ingrediente1 -
             ingrediente2], 0, MPI_COMM_WORLD);
      end
  end

```

```
  process Fumador[ i : 0..2 ] ;
  begin
      while true do begin
          // El fumador espera hasta que sea desbloqueado
          int ingrediente1, ingrediente2;
          MPI_Recv(ingrediente1, 1, MPI_INT, Estanquero, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
          MPI_Recv(ingrediente2, 1, MPI_INT, Estanquero, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
          // Toma los dos ingredientes del mostrador, lía un cigarro y fuma
          LiarCigarro();
          Fumar();
      end
  end
```

En este caso no hemos usado select porque no es necesario, ya que el estanquero siempre coloca los ingredientes y los fumadores siempre esperan a recibirlos.

6.3. Solución 2

Usando `select` *para modelar la interacción entre el estanquero y los fumadores.*

En esta solución, utilizamos el mecanismo de `select`, que permite manejar múltiples condiciones de sincronización para coordinar a los procesos. A continuación, se presenta el código:

```
process Estanquero ;
begin
    while true do begin
        // El estanquero coloca dos ingredientes aleatorios en el mostrador
        int ingrediente1, ingrediente2;
        ingrediente1 = generarIngrediente();
        do {
            ingrediente2 = generarIngrediente();
        }
        while (ingrediente1 == ingrediente2);

        // Selecciona al fumador que necesita el tercer ingrediente
        int fumador = 3 - ingrediente1 - ingrediente2;
        enviar(ingrediente1, fumador);
        enviar(ingrediente2, fumador);
    end
end

process Fumador[ i : 0..2 ] ;
begin
    while true do begin
        select
            when recibir(ingrediente1, Estanquero) and  recibir(
                ingrediente2, Estanquero) do begin
                // Toma los dos ingredientes del mostrador
                LiarCigarro();
                Fumar();
        end // select
    end
end
```

En esta implementación:

- El `process Estanquero` utiliza una lógica similar para seleccionar aleatoriamente los ingredientes y desbloquear al fumador correspondiente.
- El `process Fumador` usa `select` para esperar a recibir los ingredientes necesarios. Este enfoque permite que el fumador gestione la recepción de ambos ingredientes de manera no bloqueante, mejorando la sincronización en sistemas distribuidos.

Ambas soluciones son válidas para el problema planteado, pero la elección entre ellas puede depender de las restricciones de diseño y el entorno de implementación.

7 Ejercicio 83

7.1. Enunciado

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.

7.2. Solución

```
1
2   process Controlador ;
3   var contador: integer;
4   begin
5       while true do begin
6           select
7             for i:= 0 to n do
8
9               when receive(pet_usar,Cli[i]) do
10                if contador < 13 then
11                  send(permiso,Cli[i]);
12                  contador := contador + 1;
13                end if
14              end do
15
16              when receive(pet_liberar,Cli[i]) do
17                send(permiso,Cli[i]);
18                contador := contador - 1;
19              end do
20
21          end select
22      end
23  end
```

Lógica de la solución

En este caso se ha pensado la solución de manera simple. Se trata de un controlador que recibe peticiones de los clientes para usar un recurso. Si el contador de procesos que están usando el recurso es menor que 13, el controlador envía un permiso al cliente para que pueda usar el recurso. Cuando un cliente termina de usar el recurso, envía una petición para liberarlo y el controlador disminuye el contador. De esta forma, se asegura que nunca haya exactamente 13 procesos usando el recurso al mismo tiempo. Es impotante **uso del if** para asegurarnos de que solo se usan los 13 procesos a la vez, y debemos de asegurarnos de que actualizamos de manera correcta la variable contador.

8 Ejercicio 84

8.1. Enunciado

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 de 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 dígitos 2 ya aceptados en dicho instante.

Cuando un productor envía un dígito que no se puede aceptar por el impresor, 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
```

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

Listing 10: Pseudocódigo del proceso Impresor

8.2. Solución

```
  process Impresor
  var
      contador_0_1 : integer := 0 ;
      contador_2 : integer := 0 ;
      turno : integer := 0 ; // 0: turno para aceptar 0, 1: turno para
         aceptar 1
  begin
      while true do begin
          select
            for i:= 0 to 2
              when receive(dato, Productor[i]) do
```

```
                if (dato == 0 and turno == 0 and contador_0_1 < 2 *
                    contador_2) then
                    imprime(dato);
                    contador_0_1 := contador_0_1 + 1;
                    turno := 1;
                else if (dato == 1 and turno == 1 and contador_0_1 < 2 *
                    contador_2) then
                    imprime(dato);
                    contador_0_1 := contador_0_1 + 1;
                    turno := 0;
                else if (dato == 2) then
                    imprime(dato);
                    contador_2 := contador_2 + 1;
                end if;
            end when
        end select
    end
end
```

Lógica de la solución

En este código del proceso `Impresor`, se maneja la sincronización entre los productores y el impresor usando un `select`, y además diseñandolo la solución usando la opción for para hacer más simple el código. Para cada productor, se controla la aceptación de los dígitos 0, 1 y 2, con las restricciones mencionadas en el enunciado. El proceso acepta un dígito según las condiciones del turno (alternancia entre 0 y 1) y el límite sobre el número de dígitos 0 y 1 que pueden aceptarse en comparación con los dígitos 2.

9 Ejercicio 85

9.1. Enunciado

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 imprimir "no hay mensajes, duermo"; después de 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 qué 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
```

9.2. Solución

Para resolver este problema, utilizamos un enfoque basado en `select`, ya que ni `receive` ni `i_receive` permiten detectar si no hay mensajes pendientes de forma directa. Esto se debe a que:

- `receive` es una operación bloqueante que espera hasta que haya un mensaje disponible, impidiendo que podamos comprobar si no hay mensajes sin bloquear el proceso.
- `i_receive` es una operación no bloqueante, pero simplemente devuelve un indicador de éxito o fallo y no permite implementar un mecanismo de espera como el solicitado.

Con `select`, podemos implementar un mecanismo de espera que permita al proceso receptor manejar mensajes cuando estén disponibles, o ejecutar una acción alternativa (como imprimir "no hay mensajes, duermo") si no hay mensajes. La solución se describe a continuación:

```
process Emisor[ i : 1..n ]
var dato : integer ;
begin
    while true do begin
        dato := Producir(); // Generar un nuevo dato
        send(dato, Receptor); // Enviar el dato al Receptor
    end
end
```

```
process Receptor()
var dato : integer ;
begin
    while true do begin
        select
            when receive(dato, Emisor[1..n]) do // Si hay mensajes
                imprime(dato); // Imprimir el mensaje recibido
            else // Si no hay mensajes pendientes
                imprime("no hay mensajes, duermo");
                sleep_for(10); // Bloquear durante 10 segundos
        end select
    end
end
```

Explicación de la solución

1. Emisor: Cada proceso `Emisor[i]` produce un valor entero con la función `Producir()` y lo envía al `Receptor` utilizando la operación `send`. Este proceso se ejecuta en un bucle infinito.

2. Receptor: El proceso `Receptor` implementa un bucle infinito que realiza las siguientes acciones:

- Usa una sentencia `select` para manejar dos escenarios:
  - Si hay mensajes disponibles, el receptor los consume con `receive` y los imprime utilizando `imprime(dato)`.
  - Si no hay mensajes pendientes (gracias al `else` del `select`), el receptor imprime "no hay mensajes, duermo" y entra en un estado de espera por 10 segundos usando `sleep_for(10)`.

3. Uso de `select`: La operación `select` permite al proceso `Receptor` manejar la recepción de mensajes de múltiples emisores (`Emisor[1..n]`), así como realizar una acción alternativa (`else`) cuando no hay mensajes pendientes.

Respuesta a las preguntas planteadas

1. ¿Por qué no se puede solucionar el problema con `receive` o `i_receive`?

- `receive` es bloqueante, lo que significa que el receptor esperará indefinidamente hasta que llegue un mensaje, impidiendo detectar que no hay mensajes.
- `i_receive` no es bloqueante, pero solo devuelve un indicador de éxito o fallo. No proporciona un mecanismo para realizar acciones alternativas cuando no hay mensajes pendientes, como el que se implementa con `select`.

2. ¿Por qué es útil `select`?

- `select` permite manejar múltiples condiciones de recepción de mensajes y, además, proporciona un mecanismo `else` para definir acciones alternativas cuando ninguna condición se cumple. Esto es crucial para implementar el comportamiento del receptor cuando no hay mensajes.

Con esta implementación, se garantiza que el `Receptor` cumpla con las especificaciones planteadas: procesar mensajes cuando estén disponibles o entrar en un estado de espera eficiente si no hay mensajes.

10 Ejercicio 86

10.1. Enunciado

En un sistema tenemos **N procesos emisores** que envían de forma segura un único mensaje cada uno de ellos a un proceso receptor. El mensaje contiene un entero con el número del proceso emisor. El proceso receptor debe 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 envío 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 recepción mediante una o varias instrucciones `select`.

10.2. Solución

A continuación, se analiza cada uno de los casos planteados y se proporciona una solución cuando es posible:

a) **Recepción exclusivamente mediante `receive`:**

Usar `receive` garantiza que el proceso receptor obtiene un mensaje completo antes de procesarlo, lo que permite determinar cuál fue el primer emisor. Sin embargo, `receive` no tiene orden garantizado cuando varios emisores envían simultáneamente. Para garantizar que se identifica correctamente al primer emisor, los mensajes deben llegar en el orden en que fueron enviados.

Solución:

```
process Receptor ;
var ganador : integer ;
    recibido : integer ;
    encontrado : boolean := false ;
begin
    while not encontrado do begin
        receive(recibido, *);  // Recibir de cualquier emisor
        if not encontrado then begin
            ganador := recibido;  // El primer mensaje recibido
            encontrado := true;
            print "El primer envío lo ha realizado: ", ganador;
```

```
        end
    end
end
```

En esta solución:

- El receptor procesa el primer mensaje recibido y almacena el identificador del emisor como `ganador`.
- Los demás emisores quedan bloqueados porque el receptor no solicita más mensajes.

b) **Recepción exclusivamente mediante `i_receive`:**

La operación `i_receive` permite verificar si un mensaje está disponible sin bloquearse. Sin embargo, `i_receive` no garantiza un orden, por lo que sería necesario iterar constantemente para determinar el primer emisor. Esto puede generar comportamiento indeterminado, ya que `i_receive` solo indica disponibilidad y no un orden temporal.

Conclusión: No es posible garantizar que el primer mensaje sea procesado correctamente usando solo `i_receive`.

c) **Recepción exclusivamente mediante `select`:**

La instrucción `select` permite manejar múltiples canales de recepción simultáneamente. Esto asegura que el receptor atienda el primer mensaje recibido en cualquiera de los canales, determinando así correctamente al primer emisor.

Solución:

```
process Receptor ;
var ganador : integer ;
    recibido : integer ;
    encontrado : boolean := false ;
begin
    while not encontrado do begin
        select
            for i := 1 to N do
                when receive(recibido, Emisor[i]) do begin
                    ganador := recibido;  // El primer mensaje
                        recibido
                    encontrado := true;
                    print "El primer envío lo ha realizado: ", ganador
                        ;
                end
            end for
        end select
    end
end
```

En esta solución:

- El receptor utiliza `select` para atender al primer mensaje recibido desde cualquier emisor.

- El bucle asegura que se identifica al emisor más rápido, y se imprime su número.
- Los emisores restantes quedan bloqueados, ya que no se procesan más mensajes después de encontrar al ganador.

Resumen:

a) Es posible resolver el problema usando `receive`, y la solución es viable.

b) No es posible garantizar una solución con `i_receive`, debido a la falta de orden y bloqueo.

c) Es posible resolver el problema con `select`, y la solución es eficiente y correcta.

11 Ejercicio 87

11.1. Enunciado

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).

- **Describe razonadamente si es o no posible hacer esto usando exclusivamente `s_send` para los envíos.** En caso afirmativo, escribe una solución.
- **Escribe una solución usando `send` y `receive`.**

11.2. Solución

- **1. Análisis del uso exclusivo de `s_send`:**

  El uso exclusivo de `s_send` (envío sin búfer y bloqueante) puede causar problemas de interbloqueo en este escenario. Dado que `s_send` requiere que el receptor esté listo para recibir el mensaje en el momento del envío, si todos los procesos están intentando enviar al mismo tiempo y ninguno está recibiendo, el sistema se bloqueará.

  Por lo tanto, **no es posible implementar esta solución utilizando exclusivamente `s_send`** debido a la naturaleza de la operación bloqueante.

- **2. Solución utilizando send y receive:**

  Para resolver este problema, se utiliza una combinación de send y receive, asegurando que cada proceso envíe sus caracteres a los demás procesos y, al mismo tiempo, reciba los caracteres enviados por otros procesos.

```
process P[ i : 1..N ] ;
var
    caract : char ;       // Carácter producido
    recibido : char ;     // Carácter recibido
begin
    // Enviar N-1 caracteres a los otros procesos
    for j := 1 to N do
        if j != i then begin
            caract := ProduceCaracter(); // Produce un carácter
            send(caract, P[j]);          // Enviar al proceso P[j]
        end
    end

    // Recibir N-1 caracteres de los otros procesos
    for j := 1 to N do
        if j != i then begin
            receive(recibido, P[j]);      // Recibir de proceso P[j]
            imprime(recibido);           // Imprimir el carácter
                recibido
        end
    end
end
```

Explicación de la solución:

- Cada proceso `P[i]` realiza dos bucles:
  - En el primer bucle, genera `N-1` caracteres con la función `ProduceCaracter` y los envía a los otros `N-1` procesos utilizando send.
  - En el segundo bucle, recibe los caracteres enviados por los otros `N-1` procesos utilizando `receive` y los imprime.
- Para evitar enviar mensajes a sí mismo, se incluye la condición `if j != i`.
- El uso de send y `receive` permite que los procesos se sincronicen correctamente sin riesgo de interbloqueo, ya que las operaciones de recepción permiten manejar los mensajes enviados.

12 Ejercicio 88

12.1. Enunciado

Escribe una nueva solución al problema anterior en la cual se garantice 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.

12.2. Solución

Para garantizar que los caracteres se impriman en el mismo orden en el que se inician los envíos, utilizamos la instrucción `select` en el proceso receptor. Esto permite gestionar de manera ordenada la recepción de los caracteres basándose en el orden de llegada de los mensajes.

```
process P[ i : 1..N ] ;
var
    caract : char ;       // Carácter producido
begin
    // Enviar N-1 caracteres a los otros procesos
    for j := 1 to N do
        if j != i then begin
            caract := ProduceCaracter(); // Produce un carácter
            send(caract, Receptor);      // Enviar al proceso Receptor
        end
    end
end

process Receptor ;
var
    caract : char ;       // Carácter recibido
    sender : integer ;    // ID del proceso emisor, se supone que se usa en
        imprime
    recibido[N] : integer := [0, ..., 0]; // Vector de contadores por
       emisor, se usa como extra
begin
    while true do begin
        select
            for i := 1 to N do
                when receive(caract, P[i]) do begin
                    imprime(caract, sender);        // Imprimir el carácter
                        recibido
                    //imprime(caract);
                    recibido[i] := recibido[i] + 1; // Actualizar contador
                end
            end
        end select
    end
end
```

Explicación de la solución:

- **Proceso `P[i]`:**
  - Cada proceso genera `N-1` caracteres usando la función `ProduceCaracter`.
  - Los caracteres se envían al proceso `Receptor` mediante `send`.
- **Proceso `Receptor`:**
  - Utiliza una instrucción `select` para manejar los mensajes recibidos desde los procesos `P[i]` en orden de llegada.
  - Cada mensaje contiene el carácter producido por el proceso emisor y su ID (implícito en la recepción).

- Los caracteres se imprimen en el orden de recepción, asegurando que se respeta el orden en el que se iniciaron los envíos.
- Un vector `recibido[i]` se utiliza para llevar un conteo del número de caracteres recibidos de cada proceso, si fuera necesario para el análisis.

13 Ejercicio 89

13.1. Enunciado

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.

13.2. Solución

Para resolver el problema en el que los procesos envían bloques de datos grandes y garantizar que no haya interbloqueo debido a limitaciones de memoria en el sistema de paso de mensajes, utilizamos operaciones inseguras de envío `i_send`. Además, aseguramos que los bloques se reciben en el mismo orden en que se inician los envíos mediante el uso de `select`.

```
process P[ i : 1..N ] ;
var
    bloque : TipoBloque;  // Bloque de datos producido
begin
    // Enviar N-1 bloques de datos a los otros procesos
    for j := 1 to N do
        if j != i then begin
```

```
            ProducirBloque(bloque);       // Producir un bloque de datos
            i_send(bloque, Receptor);    // Enviar de manera insegura al
               Receptor
        end
    end
end

process Receptor ;
var
    bloque : TipoBloque;   // Bloque de datos recibido
    sender : integer;      // ID del proceso emisor
    recibido[N] : integer := [0, ..., 0]; // Vector de contadores por
       emisor
begin
    while true do begin
        select
            for i := 1 to N do
                when receive(bloque, P[i]) do begin
                    ConsumirBloque(bloque);         // Consumir el bloque
                       recibido
                    recibido[i] := recibido[i] + 1; // Actualizar contador
                end
            end
        end select
    end
end
```

Explicación de la solución:

- **Proceso `P[i]`:**
  - Cada proceso genera `N-1` bloques de datos usando la función `ProducirBloque`.
  - Los bloques se envían al proceso `Receptor` utilizando `i_send`, que es una operación de envío no bloqueante.
  - Esto garantiza que los procesos emisores no se bloqueen entre sí debido a las limitaciones de memoria del sistema de paso de mensajes.
- **Proceso `Receptor`:**
  - Utiliza una instrucción `select` para manejar los mensajes recibidos desde los procesos `P[i]` en orden de llegada.
  - Cada mensaje contiene el bloque de datos producido por el proceso emisor.
  - Los bloques se consumen inmediatamente con `ConsumirBloque`, liberando la memoria del sistema de paso de mensajes.
  - Un vector `recibido[i]` se utiliza para llevar un conteo del número de bloques recibidos de cada proceso, si fuera necesario para análisis o depuración.

Garantías de esta solución:

- **Evita interbloqueos:** El uso de `i_send` permite que los procesos emisores no queden bloqueados si el sistema de paso de mensajes agota la memoria.

- **Orden de recepción:** La instrucción `select` garantiza que los bloques se procesen en el orden en el que llegan al receptor.
- **Consumo eficiente de memoria:** Los bloques se consumen tan pronto como son recibidos, liberando memoria en el sistema de paso de mensajes.

14 Ejercicio 90

14.1. Enunciado

En los tres problemas anteriores, cada proceso va esperando a recibir un ítem de datos de cada uno de los otros procesos, consume dicho ítem, y después pasa a 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 sistema de paso de mensajes (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 transmisión y la producción de bloques sean lo más simultáneas posible.

Suponer:

- Cada proceso puede almacenar como mínimo `2*N` bloques en su memoria local.
- El orden de recepción o de consumo de los bloques es indiferente.

14.2. Solución

Para implementar esta solución, cada proceso realiza los siguientes pasos:

1. Produce `N-1` bloques y los envía a los otros procesos utilizando `i_send`.
2. Recibe `N-1` bloques utilizando `receive`, asegurándose de completar todas las recepciones antes de comenzar a consumir los bloques.
3. Consume los bloques recibidos en cualquier orden.

```
process P[ i : 1..N ] ;
var
    bloquesEnviados[N-1] : TipoBloque;  // Bloques producidos para enviar
    bloquesRecibidos[N-1] : TipoBloque; // Bloques recibidos
    j : integer;                        // Iterador
begin
    // Paso 1: Producir y enviar bloques
    for j := 1 to N do
        if j != i then begin
            ProducirBloque(bloquesEnviados[j]); // Producir un bloque
            i_send(bloquesEnviados[j], P[j]);   // Enviar al proceso P[j]
```

```
12          end
13      end
14
15      // Paso 2: Recibir bloques
16      for j := 1 to N do
17          if j != i then
18              receive(bloquesRecibidos[j], P[j]); // Recibir bloque de P[j]
19          end
20      end
21
22      // Paso 3: Consumir bloques
23      for j := 1 to N do
24          if j != i then
25              ConsumirBloque(bloquesRecibidos[j]); // Consumir bloque
                    recibido
26          end
27      end
28  end
```

Explicación de la solución:

- **Producción y envío simultáneos:** Cada proceso genera los bloques para los otros `N-1` procesos de manera concurrente, enviándolos de inmediato mediante `i_send`. Esto permite que la producción y la transmisión sean simultáneas y aprovechen al máximo los recursos del sistema.
- **Recepción antes de consumo:** Cada proceso espera a recibir todos los bloques de los otros procesos antes de comenzar a consumirlos. Esto reduce la memoria ocupada en el sistema de paso de mensajes, ya que los mensajes en tránsito se procesan rápidamente.
- **Orden indiferente:** Dado que no se requiere un orden específico de consumo, los bloques pueden procesarse en cualquier secuencia tras completar las recepciones.

Ventajas de esta solución:

- **Minimización del uso de memoria del SPM:** Al completar las recepciones antes de comenzar el consumo, los mensajes en tránsito pendientes se reducen significativamente.
- **Simultaneidad:** Producción, transmisión y recepción ocurren de forma paralela, aprovechando la capacidad del sistema.
- **Simplicidad:** La solución es sencilla y asegura que no se producen interbloqueos, ya que las recepciones son bloqueantes y se manejan de manera ordenada.