Sistemas Concurrentes y Distribuidos:
Problemas.Resueltos.

Curso 2013-14

Contents

Chapter 1

Problemas resueltos: Introducción

1

Considerar el siguiente fragmento de programa para 2 procesos P_1 y P_2:

Los dos procesos pueden ejecutarse a cualquier velocidad. ¿ Cuáles son los posibles valores resultantes para x ?. Suponer que x debe ser cargada en un registro para incrementarse y que cada proceso usa un registro diferente para realizar el incremento.

```
{ variables compartidas }
var x : integer := 0 ;
```

```
process P1 ;
   var i : integer ;
begin
   for i := 1 to 2 do begin
      x := x+1 ;
   end
end
```

```
process P2 ;
   var j : integer ;
begin
   for j := 1 to 2 do begin
      x := x+1 ;
   end
end
```

Respuesta

Los valores posibles son 2, 3 y 4. Suponemos que no hay optimizaciones al compilar y que por tanto cada proceso hace dos lecturas y dos escrituras de x en memoria. La respuesta se basa en los siguientes tres hechos:

- el valor resultante no puede ser inferior a 2 pues cada proceso incrementa x dos veces en secuencia partiendo de cero, la primera vez que un proceso lee la variable lee un 0 como mínimo, y la primera vez que la escribe como mínimo 1, la segunda vez que ese mismo proceso lee, lee como mínimo un 1 y finalmente escribe como mínimo un 2.
- el valor resultante no puede ser superior a 4. Para ello sería necesario realizar un total de 5 o más incrementos de la variable, cosa que no ocurre pues se realizan únicamente 4.

- existen posibles secuencias de interfoliación que producen los valores 2,3 y 4, damos ejemplos de cada uno de los casos:

  resultado 2: se produce cuando todas las lecturas y escrituras de un proceso i se ejecutan completamente entre la segunda lectura y la segunda escritura del otro proceso j. La segunda lectura de j lee un 1 y escribe un 2, siendo esta escritura la última en realizarse y por tanto la que determina el valor de x

  resultado 3: se produce cuando los dos procesos leen y escriben x por primera vez de forma simultánea, quedando x a 1. Los otros dos incrementos se producen en secuencia (un proceso escribe antes de que lea el otro), lo cual deja la variable a 3.

  resultado 4: se produce cuando un proceso hace la segunda escritura antes de que el otro haga su primera lectura. Es evidente que el valor resultado es 4 pues todos los incrementos se hacen secuencialmente.

2

¿ Cómo se podría hacer la copia del fichero `f` en otro `g`, de forma concurrente, utilizando la instrucción concurrente **`cobegin-coend`** ? . Para ello, suponer que:

- los archivos son secuencia de items de un tipo arbitrario `T`, y se encuentran ya abiertos para lectura (`f`) y escritura (`g`). Para leer un ítem de `f` se usa la llamada a función **`leer`**`(f)` y para saber si se han leído todos los ítems de `f`, se puede usar la llamada **`fin`**`(f)` que devuelve verdadero si ha habido al menos un intento de leer cuando ya no quedan datos. Para escribir un dato x en `g` se puede usar la llamada a procedimiento **`escribir`**`(g,`x`)`.
- El orden de los ítems escritos en `g` debe coincidir con el de `f`.
- Dos accesos a dos archivos distintos pueden solaparse en el tiempo.

Respuesta

Los ítems deben ser escritos en secuencia para conservar el orden, así que la lectura y la escritura puede hacerse en un bucle secuencial. Sin embargo, se puede solapar en el tiempo la escritura de un ítem leído y la lectura del siguiente, y por tanto en cada iteración se usará un **`cobegin-coend`** con la lectura solapada con la escritura.

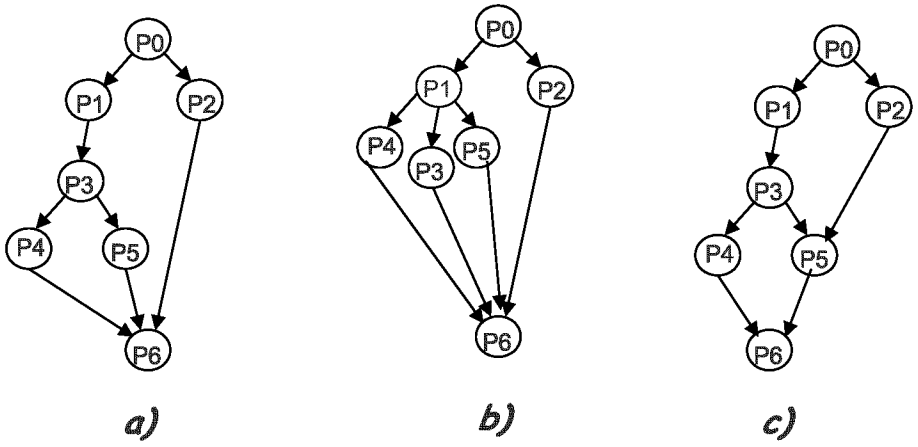
La solución más obvia sería usar una variable `v` (compartida entre la lectura y la escritura) para esto, es decir, usar en cada íteración la solución que aparece en la figura de la izquierda. El problema es que en esta solución la variable `v` puede ser accedida simultáneamente por la escritura y la lectura concurrentes, que podrían interferir entre ellas, así que es necesario usar dos variables. El esquema correcto quedaría como sigue aparece en la figura de la derecha.

```
process Incorrecto ;
   var v  : T ;
begin
   v := leer(f) ;
   while not fin(f) do
   cobegin
      escribir(g,v);
      v := leer(f) ;
   coend
end
```

```
process Correcto ;
   var v_ant, v_sig : T ;
begin
   v_sig := leer(f) ;
   while not fin(f) do begin
      v_ant := v_sig ;
      cobegin
         escribir(g,v_ant);
         v_sig := leer(f) ;
      coend
   end
end
```

3

Construir, utilizando las instrucciones concurrentes **cobegin-coend** y **fork-join**, programas concurrentes que se correspondan con los grafos de precedencia que se muestran a continuación:



Respuesta

A continuación incluimos, para cada grafo, las instrucciones concurrentes usando **cobegin-coend** (izquierda) y **fork-join** (derecha)

(a)

```
begin
   P0 ;
   cobegin
      begin
         P1 ; P3 ;
         cobegin
            P4 ; P5 ;
         coend
      end
      P2 ;
   coend
   P6 ;
end
```

```
begin
   P0 ; fork P2 ;
   P1 ; P3 ; fork P4 ; fork P5 ;
   join P2 ; join P4 ; join P5 ;
   P6 ;
end
```

(b)

```
begin
   P0 ;
   cobegin
      begin
         P1 ;
         cobegin
            P3 ; P4 ; P5 ;
         coend
      end
      P2 ;
   coend
   P6 ;
end
```

```
begin
   P0 ; fork P2 ;
   P1 ; fork P3 ; fork P4 ; fork P5 ;
   join P2 ; join P3 ;
   join P4 ; join P5 ;
   P6 ;
end
```

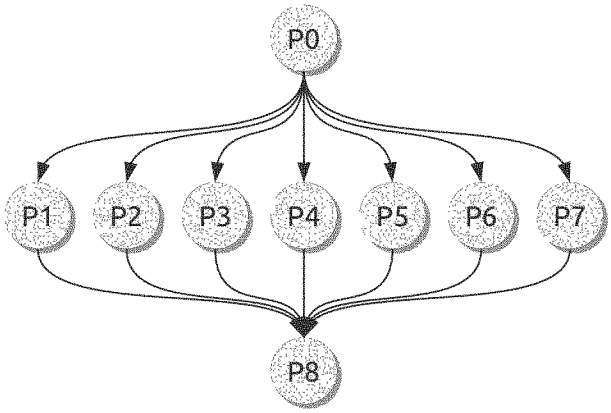
(c) en este caso, **cobegin–coend** no permite expresar el simultáneamente el paralelismo potencial que hay entre `P4` y `P2` y el que hay entre `P4` y `P5`, mientras **fork–join** sí permite expresar todos los paralelismos presentes (es más flexible).

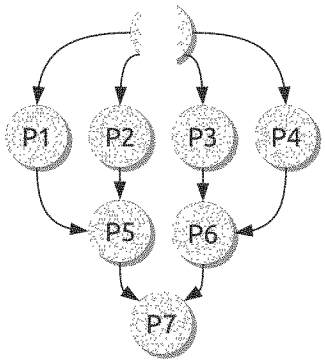
```
begin
   P0 ;
   cobegin
      begin
         P1 ; P3
      end
      P2 ;
   coend
   cobegin
      P4 ; P5 ;
   coend
   P6 ;
end
```

```
begin
   P0 ;
   cobegin
      begin
         P1 ; P3 ; P4 ;
      end ;
      P2 ;
   coend
   P5 ; P6 ;
end
```

```
begin
  P0 ; fork P2 ;
  P1 ;
  P3 ; fork P4 ;
  join P2 ;
  P5 ;
  join P4 ;
  P6 ;
end
```

```
                                    begin
                                       P0 ;
                                       cobegin
                                          begin
                                            cobegin
                                              P1;P2;
.                                           coend
 ; P4 ; P5 ; P6 ;                          P5;
                                          end
                                          begin
                                            cobegin
                                              P3;P4;
                                            coend
                                           P6;
                                          end
                                       coend
                                          ;
```





5

Suponer un sistema de tiempo real que dispone de un captador de impulsos conectado a un contador de energía eléctrica. La función del sistema consiste en contar el número de impulsos producidos en 1 hora (cada Kwh consumido se cuenta como un impulso) e imprimir este número en un dispositivo de salida. Para ello se ha de escribir un programa concurrente con 2 procesos: un proceso acumulador (lleva la cuenta de los impulsos recibidos) y un proceso escritor (escribe en la impresora). En la variable común a los 2 procesos n se lleva la cuenta de los impulsos. Suponiendo que el sistema se encuentra en un estado correspondiente al valor de la variable $n = N$ y en estas condiciones se presentan simultáneamente un nuevo impulso y el final del periodo de 1 hora, obtener las posibles secuencias de ejecución de los procesos y cuáles de ellas son correctas.

Suponer que el proceso acumulador puede invocar un procedimiento `Espera_impulso` para esperar a que llegue un impulso, y que el proceso escritor puede llamar a `Espera_fin_hora` para esperar a que termine una hora.

En el enunciado se usan sentencias de acceso a la variable `n` encerradas entre los símbolos < y >. Esto significa que cada una de esas sentencias se ejecuta en exclusión mutua entre los dos procesos, es decir, esas sentencias se ejecutan de principio a fin sin entremezclarse entre ellas.

Respuesta

El código de los procesos podría ser el siguiente:

```
{ variable compartida: }
var n : integer; { contabiliza impulsos }
```

```
process Acumulador ;
begin
   while true do begin
      Espera_impulso();
      <n:=n+1>; { (1) }
   end
end
```

```
process Escritor ;
begin
   while true do begin
      Espera_fin_hora();
      write( n ); { (2) }
      <n:=0>;     { (3) }
   end
end
```

Suponemos que se cumple lo siguiente:

$$n == N \quad \wedge \quad \text{Hay nuevo impulso} \quad \wedge \quad \text{Fin periodo hora}$$

Las secuencias de interfoliación posibles y su traza, en ese caso, se muestran a continuación, donde la variable OUT designa la salida del contador.

1. $\{n == N\}$ **(1)** $\{n == N+1\}$ **(2)** $\{n == OUT == N+1\}$ **(3)** $\{n == 0 \wedge OUT == N+1\}$

   Esta secuencia sería correcta ya que el nuevo impulso se contabiliza en el periodo que termina.

2. $\{n == N\}$ **(2)** $\{n == OUT == N\}$ **(1)** $\{n == N+1 \wedge OUT == N\}$ **(3)** $\{n == 0 \wedge OUT == N\}$

   Esta secuencia sería incorrecta ya que el nuevo impulso no se contabiliza en ningún periodo.

3. $\{n == N\}$ **(2)** $\{n == OUT == N\}$ **(3)** $\{n == 0 \wedge OUT == N\}$ **(1)** $\{n == 1 \wedge OUT == N\}$

   Esta secuencia sería correcta ya que el nuevo impulso se contabiliza en el periodo que comienza.

6

Supongamos que tenemos un vector `a` en memoria compartida (de tamaño par, es decir de tamaño $2n$ para algún $n > 1$), y queremos obtener en otro vector `b` una copia ordenada de `a`. Podemos usar una llamada a procedimiento de la forma **sort**`(s,t)` para ordenar un segmento de `a` (el que va desde `s` hasta `t`, ambos incluidos) de dos formas: (a) para ordenar todo el vector de forma secuencial con **sort**`(1,2n);b:=a` o bien (b) para ordenar las dos mitades de `a` de forma concurrente, y después mezclar dichas dos mitades en un segundo vector `b` (para mezclar usamos un procedimiento **merge**). A continuación se encuentra una posible versión del código:

```
var a,b : array[1..2*n] of integer ;  { n es una constante predefinida }
var uc  : array[1..2] of integer ;    { ultimo completado de cada mitad }
```

```
procedure Secuencial() ;
   var i : integer ;
begin
   Sort(1,1,2n) ; {ord. a}
   for i := 1 to 2*n do {copia a en b}
      b[i] := a[i] ;
end
procedure Concurrente() ;
begin
  cobegin
     Sort(1,1,n-1);
     Sort(2,n,2*n);
  coend
  Merge(1,n+1,2*n);
end
```

```
{ ordena el segmento de a entre s y t }

procedure Sort( mitad, s, t : integer )
   var i, j : integer ;
begin
  for i := s to t do begin
     for j:= s+1 to t do
        if a[i] <= a[j] then
           swap( a[i], b[j] ) ;
     uc[mitad] := i ;
  end
end
```

El código de **Merge** se encarga de ir leyendo las dos mitades y en cada paso seleccionar el menor elemento de los dos siguientes por leer en `a` (uno en cada mitad), y escribir dicho menor elemento en el vector mezclado `b`. El código es el siguiente:

```
procedure Merge( inferior, medio, superior: integer ) ;
   var k, c1, c2, ind1, ind2 : integer;
begin
   { k es la siguiente posicion a escribir en b }
   k:=1 ;
   { c1 y c2 indican siguientes elementos a mezclar en cada mitad }
   c1 := inferior ;
   c2 := medio ;
   { mientras no haya terminado con la primera mitad }
   while c1 < medio do
   begin
      if a[c1] < a[c2] then begin { minimo en la primera mitad }
         b[k] := a[c1] ;
         k    := k+1 ;
         c1   := c1+1 ;
         if c1 >= medio then { Si fin prim. mitad, copia la segunda }
            for ind2 := c2 to superior do begin
               b[k] := a[ind2] ;
               k    := k+1 ;
            end
      end
      else begin { minimo en la segunda mitad }
         b[k] := a[c2] ;
         k    := k+1 ;
         c2   := c2+1 ;
         if c2 >= superior then begin { Si fin segunda mitad, copia primera }
            for ind1 := c1 to medio do begin
               b[k] := a[ind2] ;
               k:=k+1;
            end
            c1 := medio ; { fuerza terminacion del while }
         end
      end
    end
end
```

Llamaremos $T_s(k)$ al tiempo que tarda el procedimiento **Sort** cuando actua sobre un segmento del vector con k entradas. Suponemos que el tiempo que (en media) tarda cada iteración del bucle interno que hay en **Sort** es la unidad (por definición). Es evidente que ese bucle tiene $k(k-1)/2$ iteraciones, luego:

$$T_s(k) = \frac{k(k-1)}{2} = \frac{1}{2}k^2 - \frac{1}{2}k$$

El tiempo que tarda la versión secuencial sobre $2n$ elementos (llamaremos S a dicho tiempo) será evidentemente $T_s(2n)$, luego

$$S = T_s(2n) = \frac{1}{2}(2n)^2 - \frac{1}{2}(2n) = 2n^2 - n$$

con estas definiciones, calcula el tiempo que tardará la versión paralela, en dos casos:

(1) Las dos instancias concurrentes de **Sort** se ejecutan en el mismo procesador (llamamos P_1 al tiempo que tarda).

(2) Cada instancia de **Sort** se ejecuta en un procesador distinto (lo llamamos P_2)

escribe una comparación cualitativa de los tres tiempos (S,P_1 y P_2).

Para esto, hay que suponer que cuando el procedimiento **Merge** actua sobre un vector con p entradas, tarda p unidades de tiempo en ello, lo cual es razonable teniendo en cuenta que en esas circunstancias **Merge** copia p valores desde a hacia b. Si llamamos a este tiempo $T_m(p)$, podemos escribir

$$T_m(p) = p$$

.

Respuesta (privada)

(1) Sobre un procesador el coste total de la versión paralela (P_1) sería el de dos ordenaciones secuenciales de n elementos cada una, (es decir $2T_s(n)$), más el coste de la mezcla secuencial (que es $T_m(2n)$), esto es:

$$P_1 = 2T_s(n) + T_m(2n) = \left(n^2 - n\right) + 2n = n^2 + n$$

Si comparamos $P_1 = 2n^2 - n$ con $S = n^2 + n$, vemos que, aun usando un único procesador en ambos casos, para valores de n grandes la versión potencialmente paralela tarda la mitad de tiempo que la secuencial.

(2) Sobre dos procesadores, el coste de la versión paralela (P_2) será el de la ejecución concurrente de dos versiones de **Sort** iguales sobre n elementos cada una, por tanto, será igual a $T_s(n)$. Después, la mezcla se hace en un único procesador y tarda lo mismo que antes, $T_m(2n)$, luego:

$$P_2 = T_s(n) + T_m(2n) = \left(\frac{1}{2}n^2 - \frac{1}{2}n\right) + 2n = \frac{1}{2}n^2 + \frac{3}{2}n$$

ahora vemoa que (de nuevo para n grande), el tiempo P_2 es aproximadamente la mitad de P_1, como era de esperar (ya que se usan dos procesadores), y por supuesto P_2 es aproximadamente la cuarta parte de S.

7

Supongamos que tenemos un programa con tres matrices (a,b y c) de valores flotantes declaradas como variables globales. La multiplicación secuencial de a y b (almacenando el resultado en c) se puede hacer mediante un procedimiento **MultiplicacionSec** declarado como aparece aquí:

```
var a, b, c : array[1..3,1..3] of real ;

procedure MultiplicacionSec()
   var i,j,k : integer ;
begin
   for i := 1 to 3 do
      for j := 1 to 3 do begin
         c[i,j] := 0 ;
         for k := 1 to 3 do
            c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
      end
end
```

Escribir un programa con el mismo fin, pero que use 3 procesos concurrentes. Suponer que los elementos de las matrices `a` y `b` se pueden leer simultáneamente, así como que elementos distintos de `c` pueden escribirse simultáneamente.

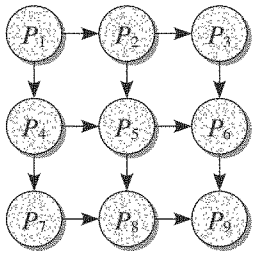
Respuesta (privada)

Para implementar el programa, haremos que cada uno de esos 3 procesos concurrentes (llamados **`CalcularFila`**) calcule y escriba un conjunto distinto de entradas de `c`. Por simplicidad (y equidad entre los procesos), lo más conveniente es hacer que cada uno de ellos calcule una fila de `c` (o cada uno de ellos una columna)

```
var a, b, c : array [1..3,1..3] of real ;

process CalcularFila[ i : 1..3 ] ;
   var j, k : integer ;
begin
   for j := 1 to 3 do begin
      c[i,j] := 0 ;
      for k := 1 to 3 do
         c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
   end
end
```

8

Un trozo de programa ejecuta nueve rutinas o actividades ($P_1, P_2, \ldots, P_9$), repetidas veces, de forma concurrentemente con **`cobegin coend`** (ver la figura de la izquierda), pero que requieren sincronizarse según determinado grafo (ver la figura de la derecha):



;

```
procedure EsperarPor( i : integer )
begin
   while not finalizado[i] do
     begin end
end
```

```
procedure Acabar(
var j : integer ;
begin
   if i < 9 then
      finalizado[i] := true ;
   else
      for j := 1 to 9 do
         finalizado[j] := false ;
end
```

Chapter 2

Problemas resueltos: Sincronización en memoria compartida.

9

¿Podría pensarse que una posible solución al problema de la exclusión mutua, sería el siguiente algoritmo que no necesita compartir una variable **Turno** entre los 2 procesos?

(a) ¿Se satisface la exclusión mutua?

(b) ¿Se satisface la ausencia de interbloqueo?

```
{ variables compartidas y valores iniciales }
var c0, c1 : integer ; { los valores iniciales no son relevantes }
```

```
1  Process P0 ;
2  begin
3     while true do begin
4        c0 := 0 ;
5        while c1 = 0 do begin
6           c0 := 1 ;
7           while  c1 = 0 do begin end
8           c0 := 0 ;
9        end
10       { seccion critica }
11       c0 := 1 ;
12       { resto sentencias }
13    end
14 end
```

```
1  process P1 ;
2  begin
3     while true do begin
4        c1 := 0 ;
5        while c0 = 0 do begin
6           c1 := 1 ;
7           while c0 = 0 do begin end
8           c1 := 0 ;
9        end
10       { seccion critica }
11       c1 := 1 ;
12       { resto seccion }
13    end
14 end
```

Respuesta (privada)

(a) ¿ Se satisface la exclusión mutua ?

Sí se satisface. Para verificar si se cumple, supongamos que no es así e intentemos llegar a una contradicción. Por tanto, supongamos que ambos procesos están en la sección crítica en un instante t. La última acción de ambos antes de acceder a SC es leer (atómicamente) la variable del otro, y ver que está a 0 (en la línea 5). Sin pérdida de generalidad, asumiremos que el proceso 0 realizó esa lectura antes que el 1 (en caso contrario se intercambian los papeles de los procesos, ya que son simétricos). Es decir, el proceso cero tuvo que leer un 0 en `c1`, en un instante que llamaremos s, con $s < t$. A partir de s, la variable `c0` contiene el valor 0, pues el proceso 0 es el único que la escribe y entre s y t dicho proceso está en SC y no la modifica.

En s el proceso 1 no podía estar en RS, ya que entonces no podría haber entrado a SC entre s y t (ya que `c0`=0 siempre después s), luego concluimos que en s el proceso 1 estaba en el PE. Más en concreto, el proceso 1 estaba (en el instante s) forzosamente en el bucle de la línea 6, ya que en otro caso `c1` sería 0 en s, cosa que no ocurrió.

Pero si el proceso 1 estaba (en s) en el bucle de la línea 7, y a partir de s `c0`=0, entonces el proceso 1 no pudo entrar a SC después de s y antes de t, lo cual es una contradicción con la hipótesis de partida (ambos procesos en SC), que por tanto no puede ocurrir.

(b) ¿ Se satisface la ausencia de interbloqueo ?

No se satisface. Para verificarlo, veremos que existe al menos una posible interfoliación de intrucciones atómicas en la cual ambos procesos quedan indefinidamente en el protocolo de entrada.

Entre las líneas 5 y 9, cada proceso i permite pasar a SC al otro proceso j. Sin embargo, para garantizar exclusión mútua, cada proceso i cierra temporalmente el paso al proceso j mientras i está haciendo la lectura de la línea 4. Por tanto, puede ocurrir interbloqueo si ambos proceso están en PE, pero cada uno de ellos comprueba siempre que puede pasar justo cuando el otro le ha cerrado el paso temporalmente.

Esto puede ocurrir partiendo de una situación en la cual ambos procesos están en el bucle de la línea 7. Como ambas condiciones son forzosamente falsas, ambos pueden abandonarlo, ejecutando ambos la asignación de la línea 8 y la lectura de la línea 5 antes de que ninguno de ellos haga la asignación de la línea 6. Por tanto las dos condiciones de la línea 5 se cumplen cuando se comprueban y ambos vuelven a entrar en el bucle de la línea 7. A partir de aquí se repite la interfoliación descrita en este párrafo, lo cual puede ocurrir indefinidamente.

10

Al siguiente algoritmo se le conoce como solución de Hyman al problema de la exclusión mutua. ¿Es correcta dicha solución?

```
{ variables compartidas y valores iniciales }
var c0    : integer := 1 ;
    c1    : integer := 1 ;
    turno : integer := 1 ;
```

```
1  process P0 ;
2  begin
3     while true do begin
4        c0 := 0 ;
5        while turno != 0 do begin
6           while c1 = 0 do begin end
7           turno := 0 ;
8        end
9        { seccion critica }
10       c0 := 1 ;
11       { resto sentencias }
12    end
13 end
```

```
1  process P1 ;
2  begin
3     while true do begin
4        c1 := 0 ;
5        while turno != 1 do begin
6           while c0 = 0 do begin end
7           turno := 1 ;
8        end
9        { seccion critica }
10       c1 := 1 ;
11       { resto sentencias }
12    end
13 end
```

Respuesta (privada)

No es correcta.

Este algoritmo fue publicado[1] por Hyman en 1966, en la creencia que era correcto, y como una simplificación del algoritmo de Dijkstra. Después se vio que no era así. En concreto, no se cumple exclusión mutua ni espera limitada:

- Exclusión mutua: existe una secuencia de interfoliación que permite que ambos procesos se encuentren en la sección crítica simultáneamente. Llamemos I a un intervalo de tiempo (necesariamente finito) durante el cual el proceso 0 ha terminado el bucle de la línea 6 pero aún no ha realizado la asignación de la línea 7. Supongamos que, durante I, **turno** vale 1 (esto es perfectamente posible). En este caso, durante I el proceso 1 puede entrar y salir en la SC un número cualquiera de veces sin espera alguna y en particular puede estar en SC al final de I. En estas condiciones, al finalizar I el proceso 0 realiza la asignación de la línea 7 y la lectura de la línea 5, ganando acceso a la SC al tiempo que el proceso 1 puede estar en ella.
- Espera limitada: supongamos que **turno**=1 y el proceso 0 está en espera en el bucle de la línea 6. Puede dar la casualidad de que, en esas circunstancias, el proceso 1 entre y salga de SC indefinidamente, y por tanto el valor de **c1** va alternando entre 0 y 1, pero puede ocurrir que lo haga de forma tal que siempre que el proceso 0 lea **c1** lo encuentre a 0. De esta manera, el proceso 0 queda indefinidamente postergado mientras el proceso 1 avanza.

11

Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (señalan el turno en que ha de ser atendido el cliente), los números de los tickets se representan por dos variables `n1` y `n2`

[1] http://dx.doi.org/10.1145/365153.365167

que valen inicialmente 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener 2 números iguales se procesa primero el proceso número 1.

a) Demostrar que se verifica la ausencia de bloqueo y la ausencia de inanción.

b) Demostrar que las asignaciones `n1:=1` y `n2:=1` son ambas necesarias.

```
{ variables compartidas y valores iniciales }
var n1 : integer := 0 ;
    n2 : integer := 0 ;
```

```
process P1 ;
begin
   while true do begin
      n1 := 1 ;                          { 1.0 }
      n1 := n2+1 ;                       { 1.1 }
      while n2 != 0 and                  { 1.2 }
            n2 < n1 do begin end;        { 1.3 }
      { seccion critica }
      n1 := 0 ;                          { 1.4 }
      { resto sentencias }
   end
end
```

```
process P2 ;
begin
   while true do begin
      n2 := 1 ;                          { 2.0 }
      n2 := n1+1 ;                       { 2.1 }
      while n1 != 0  and                 { 2.2 }
            n1 <= n2 do begin end;       { 2.3 }
      { seccion critica }
      n2 := 0 ;                          { 2.4 }
      { resto sentencias }
   end
end
```

Respuesta (privada)

apartado (a)

Demostraremos la ausencia de interbloqueo (progreso) y la ausencia de inanición (espera limitada)

(a.1) ausencia de interbloqueo

El interbloqueo es imposible. Supongamos que hay interbloqueo, es decir que los dos procesos están en sus bucles de espera ocupada de forma indefinida en el tiempo. Entonces siempre se cumplen las dos condiciones de dichos bucles (ya que las variables no cambian de valor), y por tanto siempre se cumple la conjunción de ambas (que es `n1!=0 y n2!=0 y n2<n1 y n1<=n2`). Por tanto se cumple `n2<n1 y n1<=n2`, lo cual es imposible.

(a.2) ausencia de inanición

Supongamos que un proceso está en espera ocupada (en el bucle) durante un intervalo T y comprobemos cuantas veces puede entrar el otro a SC durante T.

Supongamos que el proceso i está en el bucle durante T, luego en ese intervalo se cumple $n_i > 0$. El proceso j (con $i \neq j$) puede entrar a SC una vez. Si dicho proceso intenta entrar a SC una segunda vez, durante T, antes de hacerlo tiene que ejecutar $n_j{:=}n_i+1$ lo que forzosamente hace cierta la condición $n_j > n_i$, y como se sigue cumpliendo $n_i > 0$, vemos que el proceso j no puede entrar de nuevo a SC.

Esto implica que la cota que exige la propiedad de progreso es la unidad (la mejor posible).

apartado (b)

Para resolver esto, daremos dos pasos: en primer lugar, veremos que, sin esas asignaciones, no se cumple exclusión mutua (dando un contraejemplo, es decir, una interfoliación que permite a ambos procesos acceder). A continuación, demostraremos que, con las asignaciones, no puede haber dos procesos en SC.

(b.1) ausencia de EM sin las asignaciones

Supongamos que no están las asignaciones `n1:=1` ni `n2:=1`. Ambas variables están a cero y comienzan los dos procesos.

Supongamos que el proceso 2 comienza y alcanza SC en el intervalo de tiempo que media entre la lectura y la escritura de la asignación 1.1. Entonces, el proceso 1 también puede alcanzar SC mientras el 2 permanece en SC. Más en concreto, la secuencia de interfoliación (a partir del inicio), sería la siguiente:

1. el proceso 1 lee un `0` en `n2` (en 1.1)
2. el proceso 2 lee un `0` en `n1` (en 2.1)
3. el proceso 2 escribe un `1` en `n2` (en 2.1)
4. el proceso 2 lee `0` en `n1` (en 2.2)
5. el proceso 2 ve que la condición `n1!=0` no se cumple y avanza hasta SC
6. el proceso 1 escribe `1` en `n1` (en 1.1), en este momento, ambas variables están a `1`.
7. el proceso 1 ve que la condición 1.3 (`n2<n1`) es falsa, y avanza a la SC

(b.2) EM en el programa con las asignaciones

Ahora supondremos que las asignaciones sí están. Supongamos que en un instante t ambos procesos están en SC, y consideremos para cada proceso la última vez que accedió al PE, cuando logró entrar a SC

Es imposible que uno de los dos procesos logrará ejecutar el PE de entrada completo y verificara que podía acceder a SC estando el otro en RS durante todo ese tiempo, ya que en este caso el segundo claramente no habría podido entrar después. Por tanto, los procesos forzosamente ejecutan 1.0 y 2.0 (es decir, comienzan el PE) antes de que ninguno verifique que puede entrar a SC.

Llamemos q al instante en que se finaliza la segunda y última escritura (atómica) de 1.1 o 2.1. A partir de q, ninguna de las dos variables cambia de valor. Es imposible que las lecturas en 1.2 y 2.2 sean ambas posteriores a q, ya que en ese caso ambos, en el bucle, ven la misma combinación de valores de las dos variables, siendo ambos valores mayores que cero, y cualquier combinación de valores de estas características permite entrar en SC a uno de los dos procesos como mucho, nunca a ambos. Por tanto q separa los momentos en que los que se hacen las lecturas 1.2 y 2.2. Sea i el índice del proceso que hace esas lectura antes y j el del que la hace después.

Todo lo anterior implica que la secuencia de algunos eventos relevantes previos a t es forzosamente la siguiente:

1. Se ejecuta la última de las dos asignaciones 1.0 y 2.0

   Justo después, se cumple $n_1 > 0$ y $n_2 > 0$, y esto ocurre desde aquí hasta t, ya que nada puede hacer que esas variables disminuyan de valor antes de t.

2. El proceso i llega al bucle de espera ocupada, y hace la lectura de n_j en $i.2$

   Esto tiene que ocurrir forzosamente antes de que j haga su escritura en $j.1$ (ya hemos visto que si ocurriese después, no podrían entrar los dos). Como $j.0$ ya ha ocurrido y la escritura de $j.1$ no, el valor leído de n_j debe ser 1.

   Llamamos x al valor de n_i en este instante, se cumple $x > 0$. El valor de n_i es x hasta después de t.

   Como el proceso i determina que puede entrar, al leer n_j se cumple $n_i \leq n_j$ (por eso puede entrar i a SC). Luego $x \leq 1$, pero como $x > 0$ entonces sabemos que $x = 1$.

   Por tanto, el proceso i ve que puede entrar siendo $n_i = n_j = 1$, luego el proceso i es realmente el proceso 1 (el proceso 2 no puede entrar con los dos valores iguales).

3. El proceso j (el 2) hace la escritura de n_2 en 2.1 (esto es en el instante q)

   Llamamos z al valor escrito en n_2, puesto que es el resultado de incrementar en una unidad un valor anterior de n_1, y dicho valor anterior no puede ser nunca negativo, entonces concluimos que forzosamente $z > 0$.

4. El proceso 2 hace la lectura de n_1 en 2.2

   El proceso tiene que haber leído el valor x en n_1 y después el valor z en n_2. Pero sabemos que $x = 1$ y $z > 0$. Por tanto, en esa lectura se cumple $x \leq z$ (es decir $n_1 \leq n_2$), con lo cual el proceso 2 no podría entrar a SC hasta t.

Así que en el paso 4 concluimos que forzosamente el proceso 2 no pudo entrar a SC antes de t. Como esto es una contradicción, la hipótesis de partida no puede darse, es decir, no puede haber ningún instante de tiempo con ambos procesos en SC, es decir: se cumple exclusión mútua

12

El siguiente programa es una solución al problema de la exclusión mutua para 2 procesos. Discutir la corrección de esta solución: si es correcta, entonces probarlo. Si no fuese correcta, escribir escenarios que demuestren que la solución es incorrecta.

```
{ variables compartidas y valores iniciales }
var c0 : integer := 1 ;
    c1 : integer := 1 ;
```

```
1  process P0 ;
2  begin
3     while true do begin
4        repeat
5           c0 := 1-c1 ;
6        until c1 != 0 ;
7        { seccion critica }
8        c0 := 1 ;
9        { resto sentencias }
10    end
11 end
```

```
1  process P1 ;
2  begin
3     while true do begin
4        repeat
5           c1 := 1-c0 ;
6        until c0 != 0 ;
7        { seccion critica }
8        c1 := 1 ;
9        { resto sentencias }
10    end
11 end
```

Respuesta (privada)

No se cumple exclusión mutua.. Hay interfoliaciones que permiten a los dos procesos acceder a la SC. Supongamos que **`c1`** y **`c0`** valen ambas 1 (inicialmente ocurre esto), y los dos procesos acceden al PE. A continuación:

1. ambos procesos ejecutan las asignaciones de la línea 5, y las lecturas de la 6 (ambos procesos escriben y después leen el valor 0), antes de que ninguno de los dos repita las asignaciones de la línea 5.
2. Se repiten las asignaciones de la línea 5 y las lecturas de la 6 (ambos procesos escriben y después leen el valor 1) antes de que ningún proceso alcance la línea 8.

por tanto, tras las lecturas del paso 2, ambos pueden acceder a la SC.

13

Considerar el siguiente algoritmo de exclusión mutua para `n` procesos (*algoritmo de Knuth*). Escribir un escenario en el que 2 procesos consiguen pasar el bucle de las líneas 14 a 16, suponiendo que el turno lo tiene inicialmente el proceso 0.

```
{ variables compartidas y valores iniciales }
var c     : array[0..n-1] of (pasivo,solicitando,enSC) := [pasivo,..,pasivo];
    turno: integer := 0 ;
```

```
1  process P[ i : 0..n-1 ] ;
2  begin
3     while true do begin
4        repeat
5           c[i] := solicitando ;
6           j := turno;
7           while j <> i do begin
8              if c[j] <> pasivo then
9                 j := turno ;
10             else
11                j := (j-1) mod n ;
12          end
13          c[i] := enSC ;
14          k := 0;
15          while  k<=n and ( k=i or c[k]<>enSC ) do
16             k := k+1;
17       until k > n ;
18       turno := i ;
19       { seccion critica }
20       turno := (i-1) mod n ;
21       c[i] := pasivo ;
22       { resto sentencias }
23    end
24 end
```

Respuesta (privada)

En el bucle de las líneas 7 a 12, el i-ésimo proceso espera hasta que **turno**==i o bien que todos todos los procesos entre i y **turno** esten pasivos (ni en SC ni en PE). Si suponemos que i y **turno** no coinciden, lo anterior se comprueba haciendo una búsqueda secuencial en el vector de estados, comenzando en **turno** y terminando en i, en sentido descendente de los índices (y considerando el vector como circular). Suponemos que **turno** vale 0.

Supongamos que un proceso i_1 (con $0 < i_1$) está realizando esa búsqueda secuencial y en un instante esta mirando en una entrada e (forzosamente $i_1 < e$). El proceso ya ha mirado todas las entradas desde 0 hacia abajo hasta $e+1$ (ambas incluidas) de forma descendente (el buffer es circular). En ese momento, otro proceso puede también entrar al PE, teniendo ese otro proceso un índice i_2 con $e < i_2$.

En las circunstancias descritas, el proceso i_2 no examina la entrada del vector correspondiente a i_1 (ya que $i_1 < i_2$). Por otro lado el procesó i_1 sí examina la entrada de i_2, pero cuando lo hace esa entrada tiene el valor pasivo (ya que i_2 aún no ha llegado al PE). Luego los dos procesos observan todas las entradas que examinan al valor pasivo.

Por tanto, dos procesos distintos pueden alcanzar la asignación c[i]:=enSC. Ambos, por tanto, pueden alcanzar el bucle **while** k<=n .... (puede ocurrir lo mismo con más procesos). En este bucle se examinan todas entradas en forma ascendente y se comprueba si alguna otra está al valor enSC. Por tanto, ambos procesos pueden acabar el bucle (sin examinar todas las entradas) pues ambos pueden ver que el otro está

en estado `enSC`. Entonces, para ambos se cumple `k<=n` y vuelven al **`repeat...`** inicial, y vuelven a poner su estado a `intentando`.

14

Supongamos que tres procesos concurrentes acceden a dos variables compartidas (`x` e `y`) según el siguiente esquema:

```
var x, y : integer ;
```

```
{ accede a 'x' }
process P1 ;
begin
 while true do begin
   x := x+1 ;
   { .... }
 end
end
```

```
{ accede a 'x' e 'y' }
process P2 ;
begin
 while true do begin
   x := x+1 ;
   y := x ;
   { .... }
 end
end
```

```
{ accede a 'y' }
process P3 ;
begin
 while true do begin
   y := y+1 ;
   { .... }
 end
end
```

con este programa como referencia, realiza estas dos actividades:

1. usando un único semáforo para exclusión mutua, completa el programa de forma que cada proceso realice todos sus accesos a `x` e `y` sin solaparse con los otros procesos (ten en cuenta que el proceso 2 debe escribir en `y` el mismo valor que acaba de escribir en `x`).

2. la asignación `x:=x+1` que realiza el proceso 2 puede solaparse sin problemas con la asignación `y:=y+1` que realiza el proceso 3, ya que son independientes. Sin embargo, en la solución anterior, al usar un único semáforo, esto no es posible. Escribe una nueva solución que permita el solapamiento descrito, usando dos semáforos para dos secciones críticas distintas (las cuales, en el proceso 2, aparecen anidadas).

Respuesta

(1) en este caso la solución es sencilla, basta englobar los accesos en pares **`sem_wait-sem_signal`**. El proceso 2 debe ejecutar las dos asignaciones de forma atómica, ya que si hace las asignaciones de forma atómica cada una (pero por separado), el valor escrito en `y` podría ser distinto al escrito antes en `x`, ya que el proceso 1 podría acceder en mitad. La solución es esta:

```
var x,y   : integer ;
    mutex : semaphore := 1 ;
```

```
process P1 ;
begin
 while true do begin
   sem_wait(mutex);
   x := x+1 ;
   sem_signal(mutex);
   { .... }
 end
end
```

```
process P2 ;
begin
 while true do begin
   sem_wait(mutex);
     x := x+1 ;
     y := x ;
   sem_signal(mutex);
   { .... }
 end
end
```

```
process P3 ;
begin
 while true do begin
   sem_wait(mutex) ;
     y := y+1 ;
   sem_signal(mutex);
   { .... }
 end
end
```

(2) en este caso usamos dos semáforos, uno (`mutex_x`) para los accesos a `x` y el otro (`mutex_y`) para los accesos a `y`, anidando las secciones críticas en el proceso 2:

```
var x,y     : integer ;
    mutex_x : semaphore := 1 ;
    mutex_y : semaphore := 1 ;
```

```
process P1 ;
begin
 while true do begin
   sem_wait(mutex_x);
     x := x+1 ;
   sem_signal(mutex_x);
   { .... }
 end
end
```

```
process P2 ;
begin
 while true do begin
   sem_wait(mutex_x);
     x := x+1 ;
     sem_wait(mutex_y);
       y := x ;
     sem_signal(mutex_y);
   sem_signal(mutex_x)
   { .... }
 end
end
```

```
process P3 ;
begin
 while true do begin
   sem_wait(mutex_y) ;
     y := y+1 ;
   sem_signal(mutex_y);
   { .... }
 end
end
```

15

En algunas aplicaciones es necesario tener exclusión mutua entre procesos con la particularidad de que puede haber como mucho n procesos en una sección crítica, con n arbitrario y fijo, pero no necesariamente igual a la unidad sino posiblemente mayor. Diseña una solución para este problema basada en el uso de espera ocupada y cerrojos. Estructura dicha solución como un par de subrutinas (usando una misma estructura de datos en memoria compartida), una para el protocolo de entrada y otro el de salida, e incluye el pseudocódigo de las mismas.

Respuesta (privada)

Usaremos una variabe compartida, llamada `plazas` que indica cuantos procesos pueden entrar en la sección crítica (se iniciaiza a n). Los procesos esperan en el protocolo de entrada a que dicha variable

sea mayor que cero, entonces la decrementan y entran a SC. En el protocolo de salida, dicha variable se incrementa. Para que los accesos a `plazas` sean correctos, se hacen en exclusión mutua, usando un cerrojo, que llamamos `mutex`.

```
var mutex   : boolean  := false ; { cerrojo de acceso a 'plazas'        }
    plazas  : integer  := n     ; { numero de plazas disponibles en SC  }
```

```
1  procedure ProtocoloEntrada() ;
2     var esperar : boolean := true ;
3  begin
4     { mientras no haya plazas }
5     while esperar do begin
6        { entrar en excl. mutua }
7        while LeerAsignar(mutex) do
8           begin end
9        { si hay plazas, decrementar }
10       if plazas > 0 then begin
11          plazas := plazas - 1 ;
12          esperar := false; {no esperar mas}
13       end
14       { salir de excl. mutua }
15       mutex := false ;
16    end
17 end
```

```
1  procedure ProtocoloSalida() ;
2  begin
3     { entrar en excl. mutua }
4     while LeerAsignar(mutex) do
5        begin end
6     { incrementar plazas }
7     plazas := plazas + 1 ;
8     { salir de excl. mutua }
9     mutex := false ;
10 end
```

16

Para calcular el número combinatorio

$$\binom{n}{k} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{k!}$$

creamos un programa con dos procesos. El proceso P_1 calcula el numerador y deposita el resultado en una variable compartida, denominada x, mientras que P_2 calcula el factorial (el denominador) y deposita el valor en la variable y. Sincroniza los procesos P_1 y P_2, utilizando semáforos, para que el proceso P_2 realice correctamente la división x/y.

```
var n  : integer := .... ;
    k  : integer := .... ;
    x  : integer := 1    ;
```

```
process P1
begin
   for i := n-k+1 to n do
      x := x * i ;
end
```

```
process P2 ;
   var y : integer := 1 ;
begin
   for j := 2 to k do
      y := y * j ;
   print( x/y );
end
```

Respuesta

```
var n  : integer := .... ;
    k  : integer := .... ;
    x  : integer := 1    ;
    x_ya_calculado : semaphore := 0 ; { indica cuando ya se ha calculado 'x' }
```

```
process P1
begin
   for i = n-k+1 to n do
      x := x * i ;
   sem_signal( x_ya_calculado );
end
```

```
process P2
   var y : integer := 1 ;
begin
   for j := 2 to k do
      y := y * j ;
   sem_wait( x_ya_calculado );
   print( x/y );
end
```

17

Sean los procesos P_1, P_2 y P_3, cuyas secuencias de instrucciones son las que se muestran en el cuadro. Resuelva los siguientes problemas de sincronización (son independientes unos de otros):

a) P_2 podrá pasar a ejecutar e solo si P_1 ha ejecutado a o P_3 ha ejecutado g.

b) P_2 podrá pasar a ejecutar e solo si P_1 ha ejecutado a y P_3 ha ejecutado g.

c) Solo cuando P_1 haya ejecutado b, podrá pasar P_2 a ejecutar e y P_3 a ejecutar h.

d) Sincroniza los procesos de forma que las secuencias b en P_1, f en P_2, y h en P_3, sean ejecutadas como mucho por dos procesos simultáneamente.

```
{ variables globales }
```

```
process P1 ;
begin
   while true do begin
      a
      b
      c
   end
end
```

```
process P2 ;
begin
   while true do begin
      d
      e
      f
   end
end
```

```
process P3 ;
begin
   while true do begin
      g
      h
      i
   end
end
```

Respuesta

(a) P_2 podrá pasar a ejecutar e solo si P_1 ha ejecutado a o P_3 ha ejecutado g.

```
var S : semaphore := 0 ;
```

```
process P1 ;
begin
   while true do begin
      a
      sem_signal(S) ;
      b
      c
   end
end
```

```
process P2 ;
begin
   while true do begin
      d
      sem_wait(S);
      e
      f
   end
end
```

```
process P3 ;
begin
   while true do begin
      g
      sem_signal(S);
      h
      i
   end
end
```

(b) P_2 podrá pasar a ejecutar e solo si P_1 ha ejecutado a y P_3 ha ejecutado g.

```
var S1 : semaphore := 0 ;
    S3 : semaphore := 0 ;
```

```
process P1 ;
begin
   while true do begin
      a
      sem_signal(S1) ;
      b
      c
   end
end
```

```
process P2 ;
begin
   while true do begin
      d
      sem_wait(S1);
      sem_wait(S3);
      e
      f
   end
end
```

```
process P3 ;
begin
   while true do begin
      g
      sem_signal(S3);
      h
      i
   end
end
```

(c) Solo cuando P_1 haya ejecutado b, podrá pasar P_2 a ejecutar e y P_3 a ejecutar h

```
var S2 : semaphore := 0 ;
    S3 : semaphore := 0 ;
```

```
while true do
begin
   a
   b
   sem_signal(S2) ;
   sem_signal(S3) ;
   c
end
```

```
while true do
begin
   d
   sem_wait(S2);
   e
   f
end
```

```
while true do
begin
   g
   sem_wait(S3);
   h
   i
end
```

(d) Sincroniza los procesos de forma que las secuencias b en P_1, f en P_2, y h en P_3, sean ejecutadas como mucho por dos procesos simultáneamente.

```
var mutex : semaphore := 2 ;
```

```
while true do
begin
   a
   sem_wait(mutex) ;
   b
   sem_signal(mutex) ;
   c
end
```

```
while true do
begin
   d
   e
   sem_wait(mutex);
   f
   sem_signal(mutex);
end
```

```
while true do
begin
   g
   sem_wait(mutex);
   h
   sem_signal(mutex);
   i
end
```

18

Dos procesos P_1 y P_2 se pasan información a través de una estructura de datos, ED. Sea un entero, n, que indica en todo momento el número de elementos útiles en ED y cuyo valor inicial es 0. El proceso P_2 retira de ED en cada ejecución el último elemento depositado por P_1, y espera si no hay elementos a que P_2 ponga más. Supón que ED tiene un tamaño ilimitado, es decir, es lo suficientemente grande para que nunca se llene. Completa el código de la figura usando sincronización con semáforos de forma que el programa cumpla la función indicada.

```
var ED : array[ 0..∞ ] of integer ;
```

```
process P1 ;
   var dato : integer;
begin
   while true do begin
      dato := calcular();
      n := n+1 ;
      ED[n] := dato ;
   end
end
```

```
process P1 ;
   var dato : integer;
begin
   while true do begin
      dato := ED[n] ;
      n := n-1 ;
      usar( dato );
   end
end
```

Respuesta

```
var ED    : array[ 0..∞ ] of integer ;
    S     : semaphore := 0 ;
    mutex : semaphore := 1 ;
    n     : integer   := 0 ;
```

```
process P1 ;
   var dato : integer;
begin
   while true do begin
      dato := calcular() ;
      sem_wait(mutex);
         n := n+1;
         ED[n] := dato ;
      sem_signal(mutex);
      sem_signal(S);
   end
end
```

```
process P2 ;
var dato : integer;
begin
   while true do begin
      sem_wait(S);
      sem_wait(mutex);
         dato := ED[n] ;
         n := n-1 ;
      sem_signal(mutex);
      usar( dato );
   end
end
```

19

El cuadro que sigue nos muestra dos procesos concurrentes, P_1 y P_2, que comparten una variable global x (las restantes variables son locales a los procesos).

a) Sincronizar los procesos para que P_1 use todos los valores x suministrados por P_2.

b) Sincronizar los procesos para que P_1 utilice un valor sí y otro no de la variable x, es decir, utilice los valores primero, tercero, quinto, etc...

```
{ variables globales }
```

```
process P1 ;
   var m : integer ;
begin
   while true do begin
      m := 2*x-n ;
      print( m );
   end
end
```

```
process P2
   var d : integer ;
begin
   while true do begin
      d := leer_teclado();
      x := d-c*5 ;
   end
end
```

Respuesta

(a) Sincronizar los procesos para que P_1 use todos los valores x suministrados por P_2.

```
var x              : integer ;
    x_ya_calculado : semaphore := 0 ;
    x_ya_leido     : semaphore := 1 ;
```

```
process P1 ;
   var m : integer ;
begin
   while true do begin
      sem_wait( x_ya_calculado );
      m := 2*x-n ;
      sem_signal( x_ya_leido );
      print( m );
   end
end
```

```
process P2 ;
   var d : integer ;
begin
   while true do begin
      d := leer_teclado();
      sem_wait( x_ya_leido ) ;
      x := d-c*5 ;
      sem_signal( x_ya_calculado ) ;
   end
end
```

(b) Sincronizar los procesos para que P_1 utilice un valor sí y otro no de la variable x, es decir, utilice los valores primero, tercero, quinto, etc...

```
var x              : integer ;
    x_ya_calculado : semaforo := 0 ;
    x_ya_leido     : semaforo := 1 ;
```

```
process P1 ;
   var m : integer ;
begin
   while true do begin
      { consumir 1,3,5,.... }
      sem_wait( x_ya_calculado );
      m := 2*x-n ;
      sem_signal( x_ya_leido );
      print( m );
      { descartar 2,4,6, ... }
      sem_wait( x_ya_calculado );
      sem_signal( x_ya_leido );
   end
end
```

```
process P2 ;
   var d : integer ;
begin
   while true do begin
      d := leer_teclado();
      sem_wait( x_ya_leido ) ;
      x := d-c*5 ;
      sem_signal( x_ya_calculado ) ;
   end
end
```

20

En la fábrica de bicicletas `MountanenBike`, tenemos tres operarios que denominaremos OP_1, OP_2 y OP_3. OP_1 hace ruedas (procedimiento `h_rue`), OP_2 construye cuadros de bicicletas (`h_cua`), y OP_3, manillares (`h_mani`). Un cuarto operario, el `Montador`, se encarga de tomar dos ruedas (`c_rue`) , un cuadro (`c_cua`) y un manillar (`c_man`), y de hacer la bicicleta (`h_bic`). Sincroniza las acciones de los tres operarios y el montador en los siguientes casos:

a) Los operarios no tienen ningún espacio para almacenar los componentes producidos, y el Montador no podrá coger ninguna pieza si ésta no ha sido fabricada previamente por el correspondiente operario.

b) Los operarios OP_2 y OP_3 tienen espacio para almacenar 10 piezas de las que producen, por tanto, deben esperar si habiendo producido 10 piezas no es retirada ninguna por el Montador. El operador OP_1 tiene espacio para 20 piezas.

```
{ variables globales }
```

```
process OP1 ;
begin
 while true do
 begin
   h_rue();
 end
end
```

```
process OP2 ;
begin
 while true do
 begin
   h_cua();
 end
end
```

```
process OP3 ;
begin
 while true do
 begin
   h_man();
 end
end
```

```
process OP4 ;
begin
 while true do
 begin
   c_rue();
   c_rue();
   c_cua();
   c_man();
   m_bic();
 end
end
```

Respuesta

(a) Los operarios no tienen ningún espacio para almacenar los componentes producidos, y el Montador no podrá coger ninguna pieza si ésta no ha sido fabricada previamente por el correspondiente operario.

La frase del enunciado *Los operarios no tienen ningún espacio para almacenar los componentes producidos* puede interpretarse como que los operarios que fabrican las componentes únicamente tienen el espacio suficiente para fabricar un componente, que permanece en dicho espacio hasta que s entregada al montador.

Por tanto, los operarios no pueden comenzar a producir una pieza si habían producido antes otra que aún no ha sido retirada. Por otro lado, el montador no puede retirar una pieza hasta que esta haya sido producida.

Para solucionar el problema se usa, por cada operario que produce piezas, dos semáforos: uno que le permite indicar al montador cuando una pieza ha sido producida, y otro semáforo que el permite al montador indicar al operario cuando cada pieza ha sido ya retirada y se ha dejado el hueco.

```
var { semaforos que indican si hay espacio para comenzar pieza }
    erue, ecua, eman : semaphore := 1 ;
    { semaforos que indican si hay una pieza ya montada }
    hrue, hcua, hman : semaphore := 0 ;
```

```
process OP1 ;
begin
 while true do
 begin
   sem_wait(erue);
   h_rue();
   sem_signal(hrue);
 end
end
```

```
process OP2 ;
begin
 while true do
 begin
   sem_wait(ecua);
   h_cua();
   sem_signal(hcua);
 end
end
```

```
process OP3 ;
begin
 while true do
 begin
   sem_wait(eman);
   h_man();
   sem_signal(hman);
 end
end
```

```
process OP4 ;
begin
 while true do
 begin
   sem_wait(hrue);
     c_rue();
   sem_signal(erue);
   sem_wait(hrue);
     c_rue();
   sem_signal(erue);
   sem_wait(hcua);
     c_cua();
   sem_signal(ecua);
   sem_wait(hman);
     c_man();
   sem_signal(eman);
   m_bic();
 end
end
```

(b) Los operarios OP_2 y OP_3 tienen espacio para almacenar 10 piezas de las que producen, por tanto, deben esperar si habiendo producido 10 piezas no es retirada ninguna por el Montador. El operador OP_1 tiene espacio para 20 piezas.

En este caso el problema se puede resolver igual que antes, pero inicializando el semáforo `erue` a 20, y `ecua` y `eman` a 10 (si se considera el espacio que tienen los operarios para montar una pieza como un lugar adicional donde una pieza puede dejarse tras ser montada, habría que incrementar estos valores iniciales en una unidad).

21

Sentados en una mesa del Restaurante *Atreveteacomer*, están dos personas, cada una de las cuales esta tomando un plato de sopa, y entre las dos comparten una fuente de ensalada. Estas dos personas no tienen mucha confianza y no pueden comer las dos simultáneamente del plato de ensalada. Se pide:

a) Diseñar un programa que resuelva el problema.

b) Ahora, supón que existe un camarero encargado de retirar los platos de sopa cuando éstos están vacíos, es decir, cuando cada una de las personas ha tomado diez cucharadas. Soluciona esta variante teniendo en cuenta que el camarero debe cambiar los platos a las dos personas a la vez.

```
{ variables globales compartidas }
```

```
process Comensal_1 ;
begin
   while true do begin
      tomar_cucharada();
      comer_ensalada();
   end
end
```

```
process Comensal_2 ;
begin
   while true do begin
      tomar_cucharada();
      comer_ensalada();
   end
end
```

```
{ (apartado b) }
process Camarero ;
begin
   while true do begin
      quitar_sopa();
      poner_sopa();
   end
end
```

Respuesta (privada)

(a) Diseñar un programa que resuelva el problema.

```
var mutex_en : semaphore := 1 ; { exclusion mutua para la ensalada }
```

```
process Comensal_1 ;
begin
   while true do begin
      tomar_cucharada();
      sem_wait(mutex_en);
         comer_ensalada();
      sem_signal(mutex_en);
   end
end
```

```
process Comensal_2 ;
begin
   while true do begin
      tomar_cucharada();
      sem_wait(mutex_en);
         comer_ensalada();
      sem_signal(mutex_en):
   end
end
```

(b) Ahora, supón que existe un camarero encargado de retirar los platos de sopa cuando éstos están vacíos, es decir, cuando cada una de las personas ha tomado diez cucharadas. Soluciona esta variante teniendo en cuenta que el camarero debe cambiar los platos a las dos personas a la vez.

```
var mutex_en    : semaphore := 1 ; { exclusion mutua de la ensalada }
    esperando   : semaphore := 0 ; { senial al camarero de comensal esperando }
    cambiado1   : semaphore := 0 ; { senial a comensales de platos cambiados }
    cambiado2   : semaphore := 0 ;
```

```
process Comensal_1 ;
begin
while true do begin
 for i := 1 to 10 do begin
   tomar_cucharada();
   sem_wait(mutex_en);
      comer_ensalada();
   sem_signal(mutex_en);
 end
 sem_signal(esperando);
 sem_wait(cambiado1);
end
```

```
process Comensal_2 ;
begin
while true do begin
 for i := 1 to 10 do begin
   tomar_cucharada();
   sem_wait(mutex_en);
      comer_ensalada();
   sem_signal(mutex_en);
 end
 sem_signal(esperando);
 sem_wait(cambiado2);
end
```

```
process Camarero ;
begin
while true do begin
 sem_wait(esperando);
 sem_wait(esperando);
   quitar_sopa();
   poner_sopa();
 sem_signal(cambiado1);
 sem_signal(cambiado2);
end
```

22

Aunque un monitor garantiza la exclusión mutua, los procedimientos tienen que ser reentrantes. Explicar porqué.

Respuesta (privada)

Aunque se ejecute un mismo procedimiento en E.M., puede que un proceso abandone el control del monitor en un punto intermedio (después de invocar **wait**) y, en ese caso, otro proceso ejecutará el mismo código del procedimiento entrelazando su ejecución con el proceso inicial.

23

Se consideran dos recursos denominados r_1 y r_2. Del recurso r_1 existen N_1 ejemplares y del recurso r_2 existen N_2 ejemplares. Escribir un monitor que gestione la asignación de los recursos a los procesos de usuario, suponiendo que cada proceso puede pedir:

- Un ejemplar del recurso r_1.
- Un ejemplar del recurso r_2.
- Un ejemplar del recurso r_1 y otro del recurso r_2.

La solución deberá satisfacer estas dos condiciones:

- Un recurso no será asignado a un proceso que demande un ejemplar de r_1 o un ejemplar de r_2 hasta que al menos un ejemplar de dicho recurso quede libre.
- Se dará prioridad a los procesos que demanden un ejemplar de ambos recursos.
- Se asume semántica SU.

Respuesta (privada)

Variables del monitor: se usarán tres colas de espera, dos para los que solicitan cada uno de los dos recurso (**r1** y **r2**) y otra para los que solicitan ambos (**ambos**). Se usarán dos variables para contar cuantos ejemplares quedan de cada recurso (`n1` y `n2`).

```
Monitor Recurso;

var n1,n2         : integer;
    r1,r2,ambos   : condition;
```

Procedimiento que se debe invocar para pedir un recurso, el parámetro `num_recurso` debe ser `0` para pedir ambos, o `1` para pedir el recurso 1 y `2` para pedir el recurso 2:

```
procedure pedir_recurso( num_recurso : integer )
begin
   case num_recurso of
   0: begin
         if n1 == 0 or n2 == 0 ther
            ambos.wait();
         n1:= n1-1 ; n2 := n2-1 ;
      end;
   1: begin
         if n1 == 0 then
            r1.wait();
         n1:= n1-1 ;
      end;
   2: begin
         if n2 == 0 then
            r2.wait();
         n2 := n2-1;
      end;
  end { case }
end
```

Procedimiento para liberar uno de los dos recurso (el parámetro debe indicar que recurso se quiere liberar)

```
procedure liberar_recurso( num_recurso : integer )
begin
   case num_recurso of
   1: begin
         n1 := n1+1 ;
         if n2 > 0 and ambos.queue() then
            ambos.signal();
         else
            r1.signal();
      end
   2: begin
         n2 := n2+1 ;
         if n1 > 0 and ambos.queue() then
            ambos.signal();
         else
            r2.signal();
      end
end
```

Código de inicialización:

```
begin
   n1 := N1 ;
   n2 := N2 ;
end
```

24

Escribir una solución al problema de *lectores-escritores* con monitores:

a) Con prioridad a los lectores.

b) Con prioridad a los escritores.

c) Con prioridades iguales.

Respuesta (privada)

Suponemos que varias lecturas pueden ejecutarse en paralelo, pero si una escritura está en curso, no puede haber otras escrituras ni ninguna lectura.

Supondremos que los escritores llaman a **escritura_ini** y **escritura_fin** para comenzar y finalizar de escribir (respectivamente), mientras que los lectores hacen lo mismo con **lectura_ini** y **lectura_fin**

En general, para las tres soluciones, se usará una cola de espera para los escritores, y otra para los lectores. También se usará una variable para llevar la cuenta de cuantos lectores hay leyendo y otra variable (lógica) que indicará si hay algún escritor escribiendo.

(a) prioridad a los lectores

En esta solución, siempre que sea posible dar paso a un lector o a un escritor, se dará paso antes al lector. Esto ocurre en **escritura_fin**. Hay que tener en cuenta que en **lectura_fin** no puede haber ningún lector esperando, pues en ese procedimiento **nlectores** es mayor que cero y forzosamente **escribiendo** debe ser **false**.

```
Monitor LectoresEscritores_plec ;

var escribiendo           : boolean ;
    nlectores             : integer ;
    lectores, escritores  : condition ;

procedure escritura_ini() ;
begin
   if escribiendo or nlectores > 0 then { si no se puede escribir:       }
      escritores.wait() ;               {   esperar                      }
   escribiendo := true ;                { anotar que se esta escribiendo }
end

procedure escritura_fin() ;
begin
   escribiendo := false ;
   if lectores.queue() then { si hay lectores esperando:            }
      lectores.signal()     {   despertar uno                       }
   else                     { si no hay lectores esperando:         }
      escritores.signal()   {   despertar un escritor, si hay alguno }
end
```

```
procedure lectura_ini() ;
begin
   if escribiendo then          { si hay algun escritor escribiendo: }
      lectores.wait() ;         {    esperar                         }
   nlectores := nlectores+1 ;   { anotar un lector leyendo mas       }
   lectores.signal() ;          { permitir a otros lectores acceder  }
end

procedure lectura_fin() ;
begin
   nlectores := nlectores-1 ;   { anotar un lector menos             }
   if nlectores == 0 then       { si no hay lectores leyendo:        }
      escritores.signal() ;     {   despertar un escritor (si hay)   }
end

{ inicializacion }
begin
  nlectores := 0 ;          { no hay procesos leyendo       }
  escribiendo := false ;    { no hay un escritor escribiendo }
end
```

(b) prioridad a los escritores

En esta solución, siempre que sea posible dar paso a un lector o a un escritor, se dará paso antes al escritor (también en **escritura_fin**). La implementación es semejante a la anterior, excepto en:

- **escritura_fin**: se despierta antes a un escritor que un lector
- **lectura_ini**: el **signal** de los lectores no se hace si hay escritores esperando entrar (para impedir que una ráfaga de lectores deje esperando a los escritores).

```
Monitor LectoresEscritores_pesc ;

var escribiendo          : boolean ;
    nlectores            : integer ;
    lectores, escritores : condition ;

procedure escritura_ini() ;
begin
   if escribiendo or nlectores >  0 then
      escritores.wait() ;
   escribiendo := true ;
end

procedure escritura_fin() ;
begin
   escribiendo := false ;
   if escritores.queue() then { si hay escritores esperando:      }
      escritores.signal()     {   despertar un escritor           }
   else                       { si no hay escritores esperando:   }
```

```
      lectores.signal()          {   despertar un lector, si hay      }
end

procedure lectura_ini() ;
begin
   if escribiendo then
      lectores.wait() ;
   nlectores := nlectores+1 ;
   if not escritores.queue() then { si no hay escritores esperando }
      lectores.signal()           {   despertar un lector (si hay) }
end

procedure lectura_fin() ;
begin
   nlectores := nlectores-1 ; { anotar un lector menos          }
   if nlectores == 0 then     { si no hay lectores leyendo:     }
      escritores.signal() ;   {    despertar un escritor (si hay) }
end

{ inicializacion }
begin
  nlectores := 0 ;
  escribiendo := false ;
end
```

(b) sin prioridad

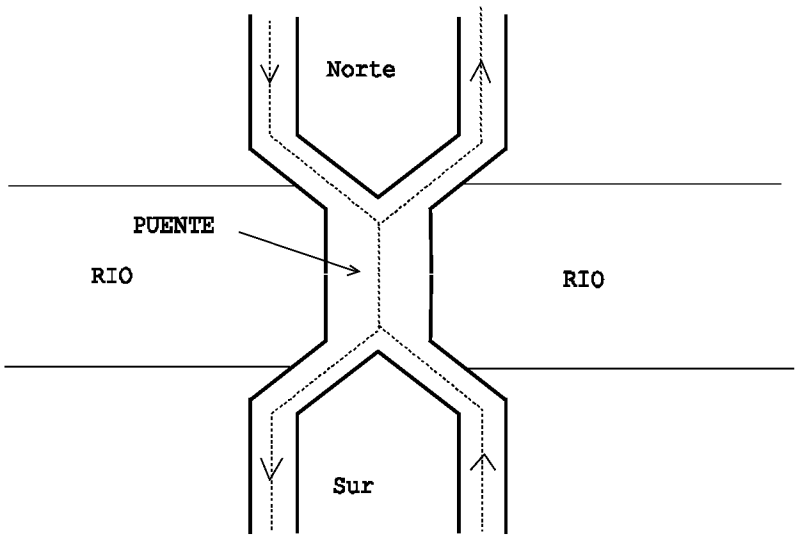
En esta solución no se pueden usar dos colas, en ese caso siempre habría que elegir una frente a otra para despertar un proceso. Por eso, todos los procesos esperan en una sola cola (llamada **cola**).

```
Monitor LectoresEscritores_noprio ;

var escribiendo          : boolean ;
    nlectores            : integer ;
    cola                 : condition ;

procedure escritura_ini() ;
begin
   if escribiendo or nlectores >  0 then { si no es posible escribir:    }
      cola.wait() ;                      {   esperar                    }
   escribiendo := true ;                 { anotar que se esta escribiendo }
end

procedure escritura_fin() ;
begin
   escribiendo := false ; { anotar que no se esta escribiendo   }
   cola.signal() ;        { dejar entrar a otro proceso (si hay) }
end

procedure lectura_ini() ;
begin
   if escribiendo then         { si hay un escritor:  }
```

```
      cola.wait() ;              { esperar          }
   nlectores := nlectores+1 ;  { anotar un lector mas }
end

procedure lectura_fin() ;
begin
   nlectores := nlectores-1 ;  { anotar un lector menos       }
   if nlectores == 0 then      { si no quedan lectores leyendo: }
      cola.signal() ;          {   despertar un proceso       }
end

{ inicializacion }
begin
  nlectores := 0 ;
  escribiendo := false ;
end
```

25

Varios coches que vienen del norte y del sur pretenden cruzar un puente sobre un río. Solo existe un carril sobre dicho puente. Por lo tanto, en un momento dado, el puente solo puede ser cruzado por uno o más coches en la misma dirección (pero no en direcciones opuestas).



a) Completar el código del siguiente monitor que resuelve el problema del acceso al puente suponiendo que llega un coche del norte (sur) y cruza el puente si no hay otro coche del sur (norte) cruzando el puente en ese momento.

```
Monitor Puente

var  ... ;
```

```
procedure EntrarCocheDelNorte()
begin
   ...
end
procedure SalirCocheDelNorte()
begin
   ....
end
procedure EntrarCocheDelSur()
begin
   ....
end
procedure SalirCocheDelSur()
begin
  ...
end

{ Inicializacion }
begin
   ....
end
```

b) Mejorar el monitor anterior, de forma que la dirección del trafico a través del puente cambie cada vez que lo hayan cruzado 10 coches en una dirección, mientras 1 ó más coches estuviesen esperando cruzar el puente en dirección opuesta.

Respuesta (privada)

Caso (a)

En el caso (a), usaremos dos colas, una para los coches del norte y otra para los del sur (**N** y **S**, respectivamente), y dos contadores (**N_cruzando** y **S_cruzando**) para saber cuantos coches están cruzando provenientes del norte y el sur, respectivamente.

```
Monitor Puente ;

var N_cruzando,S_cruzando : integer ;
    N,S                   : condition ;

procedure EntrarCocheDelNorte()
begin
   if S_cruzando >0 then
      N.wait() ;
   N_cruzando := N_cruzando+1 ;
   N.signal() ;
end

procedure SalirCocheDelNorte()
begin
   N_cruzando := N_cruzando-1 ;
   if N_cruzando == 0 then
```

```
      S.signal();
end

procedure EntrarCocheDelSur()
begin
   if N_cruzando > 0 then
      S.wait;
   S_cruzando := S_cruzando+1 ;
   S.signal();
end

procedure SalirCocheDelSur()
begin
   S_cruzando := S_cruzando-1 ;
   if S_cruzando ==0 then
      N.signal() ;
end

{ Inicializacion }
begin
   N_cruzando := 0 ;
   S_cruzando := 0 ;
end
```

Caso (b)

En este caso se usan las mismas variables y condiciones que en el anterior, solo que ahora añadimos dos nuevas variables enteras, `N_pueden` y `S_pueden`. La variable `N_pueden` indica cuantos coches del norte pueden todavía entrar al puente mientras haya coches del sur esperando (`S_pueden` es similar, pero referida a los coches del sur).

La condición asociada a la cola **`N`** es: **`S_cruzando`** `== 0 y N_pueden > 0`, cuando dicha condición no se da, los coches del norte esperan en **`N`**. Cuando en algún procedimiento (al final del mismo) que la condición es cierta, se debe hacer **`signal`** de la cola norte por si hubiese algún coche que ahora sí puede entrar. (el razonamiento es similar para la cola **`S`**).

```
Monitor Puente

var N_cruzando, S_cruzando,
    N_pueden,   S_pueden     : integer ;
    N,S                      : condition ;
```

```
procedure EntrarCocheDelNorte()
begin
   if S_cruzando > 0 or N_pueden == 0 then { si no se puede pasar }
      N.wait();   { esperar en la cola norte }

   { aqui se sabe con seguridad que se puede pasar, ya que se cumple: }
   { S_cruzando == 0 y N_pueden > 0 (==condicion de 'N')}

   N_cruzando := N_cruzando+1 ;  { hay uno mas del norte cruzando }
```

```
   if not S.empty() then { si hay coches del sur esperando al entrar este }
      N_pueden := N_pueden - 1 ;   { podra entrar uno menos }

   if N_pueden > 0 then { si aun puede pasar otro (se cumple: S_cruzando == 0)}
      N.signal(); { hacer entrar a uno justo tras este (si hay alguno) }
end

procedure  SalirCocheDelNorte
begin
   N_cruzando := N_cruzando-1 ; { uno menos del norte cruzando }

   if N_cruzando == 0 then begin { si el puente queda vacio }
      S_pueden := 10 ; { permitir a 10 coches del sur entrar }
      S.signal() ; { permite entrar al primero del sur que estuviese esperando, si hay }
   end
end
```

El código para los coches del sur es simétrico (se omiten los comentarios). Al final se incluye la inicialización.

```
procedure EntrarCocheDelSur
begin
   if N_cruzando > 0 or S_pueden == 0 then
      S.wait() ;
  S_cruzando := S_cruzando + 1 ;
  if not N.empty() then
     S_pueden := S_pueden - 1 ;
  if S_pueden > 0 then
     S.signal();
end
procedure  SalirCocheDelSur
begin
  S_cruzando := S_cruzando-1 ;
  if S_cruzando == 0 then begin
     N_pueden = 10 ;
     N.signal() ;
  end
end
{ Inicializacion }
begin
   N_cruzando :=  0  ; S_cruzando :=  0 ;
   N_pueden   := 10  ; S_pueden   := 10 ;
end
```

26

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.

Para solucionar la sincronizacion usamos un monitor llamado **Olla**, que se puede usar así:

```
monitor Olla ;
....
begin
   ....
end
```

```
process ProcSalvaje[ i:1..N ] ;
begin
   while true do begin
      Olla.Servirse_1_misionero();
      Comer(); { es un retraso aleatorio }
   end
end
```

```
process ProcCocinero ;
begin
   while true do begin
      Olla.Dormir();
      Olla.Rellenar_Olla();
   end
end
```

Diseña el código del monitor **Olla** para 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)

Se introducen dos variables de condición, para las esperas asociadas al cocinero y a los salvajes, respectivamente (**cocinero** y **salvajes**)

```
monitor Olla ;

var num_misioneros      : integer ; { numero de misioneros en la olla }
    cocinero, salvajes : condition ;

procedure Servirse_1_Misionero()
begin
   if num_misioneros == 0 then begin { si no hay comida:          }
      cocinero.signal();             {   despertar al cocinero    }
      salvajes.wait();               {   esperar a que haya comida }
   end

   num_misioneros := num_misioneros - 1 ; { coger un misionero }

   if num_misioneros > 0  then { si queda comida:                }
      salvajes.signal();       {   despertar a un salvaje (si hay) }
   else                        { si no queda comida:             }
      cocinero.signal();       {   despertar al cocinero (si duerme) }
end
```

```
procedure Dormir()
begin
   if num_misioneros > 0 then    { si ya hay comida:      }
      cocinero.wait() ;          {    esperar a que no haya }
end

Procedure Rellenar_Olla()
begin
   num_misioneros = M ;     { poner M misioneros en la olla  }
   salvajes.signal();       { despertar un salvaje (si hay)  }
end

{ Inicializacion  }
begin
   num_misioneros := M ;
end
```

27

Una cuenta de ahorros es compartida por varias personas (procesos). Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo.

a) Programar un monitor para resolver el problema, todo proceso puede retirar fondos mientras la cantidad solicitada c sea menor o igual que el saldo disponible en la cuenta en ese momento. Si un proceso intenta retirar una cantidad c mayor que el saldo, debe quedar bloqueado hasta que el saldo se incremente lo suficiente (como consecuencia de que otros procesos depositen fondos en la cuenta) para que se pueda atender la petición. El monitor debe tener 2 procedimientos: **depositar**(c) y **retirar**(c). Suponer que los argumentos de las 2 operaciones son siempre positivos. Hacer dos versiones: uno sin colas de prioridad y otra con colas de prioridad.

b) Modificar la respuesta del apartado anterior, de tal forma que el reintegro de fondos a los clientes sea servido según un orden FIFO. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está esperando un reintegro de 300 unidades. Si llega otro cliente debe esperarse, incluso si quiere retirar 200 unidades. Resolver el problema usando un monitor con colas de prioridad.

Respuesta (privada)

a) **Solución con colas sin prioridad:**

```
monitor CuentaCorriente;

   var saldo : integer ;
       cond  : condition ;
```

```
procedure Retirar( cantidad : integer )
begin
   while cantidad > saldo do { mientras no se pueda atender peticion }
   begin
      cond.signal() ; { permitir que otros comprueben si pueden sacar }
      cond.wait() ;   { esperar hasta volver a comprobar }
   end
   saldo = saldo - cantidad ;   { retirar cantidad }
   cond.signal(); { permitir a otros comprobar }
   { (se convierte en espera ocupada, se despiertan unos a otros
    cuando no hay saldo para ninguno) }
end
procedure Depositar( cantidad : integer )
begin
   saldo = saldo + cantidad;
   cond.signal();
end
{ inicializacion }
begin
   saldo = saldo_inicial ; { == constante predefinida }
end
```

Solución con colas con prioridad:

```
Monitor CuentaCorriente;

   var saldo : integer ;
       cond  : condition ;

procedure Retirar( cantidad : integer )
begin
  while cantidad > saldo do
     cond.wait( cantidad );
  saldo = saldo - cantidad;
  cond.signal() ;
end
procedure Depositar( cantidad : integer )
begin
  saldo = saldo + cantidad;
  cond.signal();
end
{ inicializacion }
begin
   saldo= saldo_inicial ;
end
```

b) Para el caso (b), se intenta da una solución aquí que usa colas de prioridad. En este caso la solución es parecida a las anteriores, con la diferencia de que la cola de clientes esperando en el banco es FIFO, ya que, en cualquier momento, de todos los clientes esperando el único que puede retirar dinero es el que más tiempo hace que llamó a **Retirar**.

Para lograr que la cola sea FIFO, los clientes que retiran hacen **wait** usando como prioridad un *número de ticket* que indica el número de orden en la cola de ese cliente. Para ello se usa una variable **ticket**, local al procedimiento **Retirar**. En el monitor se guarda una variable, llamada **contador**, que sirve para que cada cliente, cuando accede a retirar, pueda saber cual es su número de ticket.

Cuando un cliente entra al monitor para retirar y hay saldo suficiente para su cantidad, no podrá hacerlo si no es el más antiguo entre todos los procesos que están esperando. El más antiguo, lógicamente tendrá el número de ticket más bajo. Para saber si un cliente es el más antiguo, guardamos siempre el número de ticket más bajo en la cola, en una variable que se llama **ticket_minimo**.

Cuando un cliente *C* sale de la cola, su ticket es el minimo (ya que es fifo) y por tanto el número de dicho ticket minimo debe incrementarse en una unidad, pues corresponde al que llegó o llegará siguiente a *C*.

Al principio del monitor, el ticket minimo es el 0, que será el numero de ticket del primer cliente en llegar.

```
Monitor CuentaCorrienteFIFO ;

   var saldo, contador,
       ticket_minimo    : integer ;
       cola             : condition ;

procedure Retirar( cantidad : integer ) ;
var ticket : integer ;
begin
   ticket := contador ;                { leer numero de ticket propio          }
   contador := contador + 1 ;          { incrementar contador de tickets       }
   while cantidad > saldo or           { mientras el saldo no sea suficiente   }
         ticket > ticket_minimo do     { (o no sea el mas antiguo):            }
      wait(ticket) ;                   {   esperar                             }
   ticket_minimo := ticket_minimo+1 ;{ el ticket minimo es el del siguiente  }
   saldo := saldo - cantidad ;         { retirar la cantidad                   }
   cola.signal() ;                     { avisar al siguiente que llego, si hay }
end
procedure Depositar( cantidad : integer ) ;
begin
   saldo := saldo + cantidad ; { depositar }
   cola.signal() ;   { avisar al que mas tiempo lleve esperando, si hay alguno }
end
{ inicializacion }
begin
   saldo := 0 ;
   contador := 0 ;
   ticket_minimo := 0 ;
end
```

En este caso (b), también se puede escribir una solución algo más sencilla que no usa colas de prioridad. Esta solución se basa en tener dos variables condición:

- Una con la cola donde espera el cliente que llegó primero (decimos que es el ciente que *está en la*

ventanilla del banco, que es el único al que se puede dar dinero. A esa cola se le llama **ventanilla**. Esta cola no necesita prioridades, ya que tiene una hebra como mucho.

- El resto de clientes esperan en una cola distinta, a la que llamaremos **resto**, y que es una cola FIFO normal sin prioridad, ya que solo salen de ella una vez, cuando la ventanilla queda libre, y sale el que más tiempo lleva en **resto** (notese que si no hay ningún cliente en **ventanilla**, no puede haber ninguno en **resto**)

La solución podría quedar así:

```
Monitor CuentaCorrienteFIFO_noprio ;

   var saldo : integer ;
       ventanilla, resto  : condition ;

procedure Retirar( cantidad : integer ) ;
begin
   if ventanilla.queue() then  { si hay otro cliente en ventanilla      }
      resto.wait() ;           {    esperar junto con resto de clientes }
   while saldo < cantidad do   { mientras saldo no suficiente          }
      ventanilla.await() ;     {    esperar en ventanilla             }
   saldo := saldo-cantidad ;   { retirar cantidad del saldo           }
   resto.signal() ;            { hacer pasar otro a ventanilla, si hay }
end
procedure Depositar( cantidad : integer ) ;
begin
   saldo := saldo + cantidad ; { depositar }
   ventanilla.signal() ;   { avisar al de ventanilla, si hay }
end
{ inicializacion }
begin
   saldo := 0 ;
end
```

28

Los procesos P_1, P_2,...,P_n comparten un único recurso R, pero solo un proceso puede utilizarlo cada vez. Un proceso P_i puede comenzar a utilizar R si está libre; en caso contrario, el proceso debe esperar a que el recurso sea liberado por otro proceso. Si hay varios procesos esperando a que quede libre R, se concederá al proceso que tenga mayor prioridad. La regla de prioridad de los procesos es la siguiente: el proceso P_i tiene prioridad i, (con $1 \leq i \leq n$), donde los números menores implican mayor prioridad. Implementar un monitor que implemente los procedimientos **Pedir** y **Liberar**.

Respuesta (privada)

```
Monitor Recurso ;

var ocupado : boolean   ;
    recurso : condicion ;
```

```
procedure Pedir( i : integer )
begin
   if ocupado then
      recurso.wait(i);
   ocupado = true;
end
procedure Liberar()
begin
   ocupado = false;
   recurso.signal();
end
{ Inicializacion }
begin
   ocupado := false;
end
```

29

En un sistema hay dos tipos de procesos: A y B. Queremos implementar un esquema de sincronización en el que los procesos se sincronizan por bloques de 1 proceso del tipo A y 10 procesos del tipo B. De acuerdo con este esquema:

- Si un proceso de tipo A llama a la operación de sincronización, y no hay (al menos) 10 procesos de tipo B bloqueados en la operación de sincronización, entonces el proceso de tipo A se bloquea.
- Si un proceso de tipo B llama a la operación de sincronización, y no hay (al menos) 1 proceso del tipo A y 9 procesos del tipo B (aparte de él mismo) bloqueados en la operación de sincronización, entonces el proceso de tipo B se bloquea.
- Si un proceso de tipo A llama a la operación de sincronización y hay (al menos) 10 procesos bloqueados en dicha operación, entonces el proceso de tipo A no se bloquea y además deberán desbloquearse exactamente 10 procesos de tipo B. Si un proceso de tipo B llama a la operación de sincronización y hay (al menos) 1 proceso de tipo A y 9 procesos de tipo B bloqueados en dicha operación, entonces el proceso de tipo B no se bloquea y además deberán desbloquearse exactamente 1 proceso del tipo A y 9 procesos del tipo B.
- No se requiere que los procesos se desbloqueen en orden FIFO.

los procedimientos para `pedir` y `liberar` el recurso

Implementar un Monitor que implemente procedimientos para llevar a cabo la sincronización requerida entre los diferentes tipos de procesos (el monitor puede exportar una única operación de sincronización para todos los tipos de procesos o una operación espcífica para los de tipo A y otra para los de tipo B.

Respuesta (privada)

```
Monitor Sincronizacion ;
```

```
var  nA,nB            : integer ;    { numero de procesos de tipo A o B esperandp }
     condA, CondB     : condition ;  { colas para esperas de procesos tipo A o B  }

procedure SincA ()
begin
   nA := nA+1 ;                  { uno mas de tipo A.          }
   if nB < 10  then              { si aun no hay 10 de tipo B: }
      condA.wait() ;             {   esperar a que los haya    }
   else                          { si ya hay 10 de tipo B:     }
      for i := 1 to 10 do        {   para cada uno de ellos:   }
          condB.signal();        {     despertarlo             }
   nA := nA-1 ;                  { uno menos de tipo A         }
end

procedure SincB()
begin
   nB := nB+1 ;                  { uno mas de tipo B.                  }
   if nA < 1 or nB < 10 then     { si no esta el A o no estan 10 del B: }
      condB.wait() ;             {   esperar a que esten todos         }
   else begin                    { si ya esta el A y soy el ultimo B:  }
      condA.signal() ;           {   despertar al A                    }
      for i := 1 to 9 do         {   para cada uno de los otros 9 B:   }
         condB.signal() ;        {      despertarlo                    }
   end                           {                                     }
   nB := nB-1 ;                  { uno menos de tipo B                 }
end

{ Inicializacion }
begin
  nA := 0 ;
  nB := 0 ;
end
```

30

El siguiente monitor (**Barrera2**) proporciona un único procedimiento de nombre **entrada** que provoca que el primer proceso que lo llama sea suspendido y el segundo que lo llama despierte al primero que lo llamó, y así actúa cíclicamente. Obtener una implementación de este monitor usando semáforos.

```
Monitor Barrera2 ;

   var n : integer;       { num. de proc. que han llegado desde el signal }
       s : condicion ;    { cola donde espera el segundo                  }

procedure entrada() ;
begin
  n := n+1 ;          { ha llegado un proceso mas }
  if n<2 then         { si es el primero:         }
```

```
      s.wait()        { esperar al segundo     }
   else begin         { si es el segundo:      }
      n := 0;         { inicializa el contador }
      s.signal()      { despertar al primero   }
   end
end
{ Inicializacion }
begin
    n := 0 ;
end
```

Respuesta (privada)

```
{ variables compartidas }
var n     : integer  := 0 ;
    s     : semaphore := 0 ;
    mutex : semaphore := 1 ;

procedure entrada() ;
begin
   sem_wait(mutex);
   n := n+1 ;
   if n < 2 then begin { primero }
      sem_signal(mutex);
      sem_wait(s);
   end
   else begin { segundo }
      n := 0 ;
      sem_signal(s);
      sem_signal(mutex);
   end
end
```

31

Problema de los filósofos-comensales. Sentados a una mesa están cinco filósofos. La actividad de cada filósofo es un ciclo sin fin de las operaciones de pensar y comer. Entre cada dos filósofos hay un tenedor. Para comer, un filósofo necesita obligatoriamente dos tenedores: el de su derecha y el de su izquierda. Se han definido cinco procesos concurrentes, cada uno de ellos describe la actividad de un filósofo. Los procesos usan un monitor, llamado **MonFilo**.

Antes de comer cada filósofo debe disponer de su tenedor de la derecha y el de la izquierda, y cuando termina la actividad de comer, libera ambos tenedores. El filósofo i alude al tenedor de su derecha como el número i, y al de su izquierda como el número $i+1 \mod 5$.

El monitor **MonFilo** exportará dos procedimentos: **coge_tenedor**(num_tenedor,num_proceso) y **libera_tenedor**(num_tenedor,num_proceso), para indicar que un proceso filósofo desea coger un tenedor determinado.

El código del programa es el siguiente:

```
monitor MonFilo ;
   ....
begin
   ....
end

process Filosofo[ i: 0..4 ] ;
begin
   while true do begin
      MonFilo.coge_tenedor(i,i);              { argumento 1=codigo tenedor    }
      MonFilo.coge_tenedor(i+1 mod 5, i);     { argumento 2=numero de proceso }
      comer();
      MonFilo.libera_tenedor(i,i);
      MonFilo.libera_tenedor(i+1 mod 5,i);
      pensar();
   end
end
```

Diseña el monitor **MonFilo**, compartido por todos los procesos filósofos. Para evitar interbloqueo (que ocurre si todos los filósofos toman el tenedor a su izquierda antes de que ninguno de ellos pueda coger el tenedor de la derecha), la solución no debe permitir que haya más de cuatro filósofos simultáneamente intentado coger un tenedor

Respuesta (privada)

```
monitor MonFilo ;
var
   esperando     : condition ;                    { no mas de cuatro esperando primer tenedor }
   tenedor       : array[0..4] of condition ;     { i-esimo tenedor libre }
   num_filo_esp  : integer ;                      { numero de filosofos esperando }

procedure coge_tenedor( num_tenedor, num_proceso : integer );
begin
   if num_tenedor == num_proceso then begin { si es el primer tenedor de los dos: }
      num_filo_esp := num_filo_esp+1 ;       {  hay uno mas intentando coger su primer tenedor }
      if num_filo_esp > 3 then               {  si ya hay 4 esperando su primer tenedor }
         esperando.wait() ;                  {     esperar a que otros lo logren }
      tenedor[num_tenedor].wait() ;          {  esperar a que este libre el primer tenedor }
      num_filo_esp := num_filo_esp-1 ;       {  hay uno menos esperando }
      if num_filo_esp < 4 then               {  si ahora hay menos de 4 }
         esperando.signal() ;                {     dejar pasar a uno    }
   end
   else                                      { si es el segundo tenedor   }
      tenedor[num_tenedor].wait() ;          {   esperar que quede libre }
end
procedure libera_tenedor( tenedor, proceso : integer );
begin
   tenedor[num_tenedor].signal() ;
end
{ inicializacion  }
begin
```

```
   num_filo_esp := 0 ;
end
```

Chapter 3

Problemas resueltos: Sistemas basados en paso de mensajes.

32

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 es el siguiente, asumiendo que se usan operaciones síncronas.

```
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 not recibido[i] 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 siguiente:

```
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 }
```

33

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 ] ;
begin
   while true do begin
      Produce(&dato );
      send( &dato, Buffer );
   end
end
```

```
process Consumidor ;
begin
   while true do begin
      receive ( &dato, Buffer );
      Consume (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)

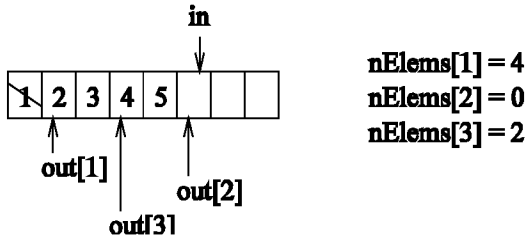
```
process Buffer ;

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
```

34

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.

```
process Buffer ;

var B       : integer  := ... ; { capacidad del buffer }
    in      : integer  := 0 ;
    dato    : integer;
```

```
    buf    : array [0..B-1] of integer ;
    nElems : array [0..3]   of integer := (0,0,0) ;
    out    : array [0..3]   of integer := (0,0,0) ;

begin
   while true do
      select
         for i := 0 to 2 when nElems[i]!=0 do
            send( &buf[out[i]], Consumidor(i) );
            out[i] := (out[i]+1) mod B;
            nElems[i] := nElems[i] - 1;
         when nElems[0]!=B and nElems[1]!=B and nElems[2]!=B receive(dato,Productor) do
            buf[in] := dato;
            in := (in+1) mod B;
            for j := 0 to 2 do
               nElems[j] := nElems[j]+1 ;
   end
end
```

35

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: }
      ......
      { rellenar 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.
- Los procesos usan operaciones de comunicación síncronas.

Respuesta (privada)

Vamos a asumir que se usa paso de mensajes síncrono.

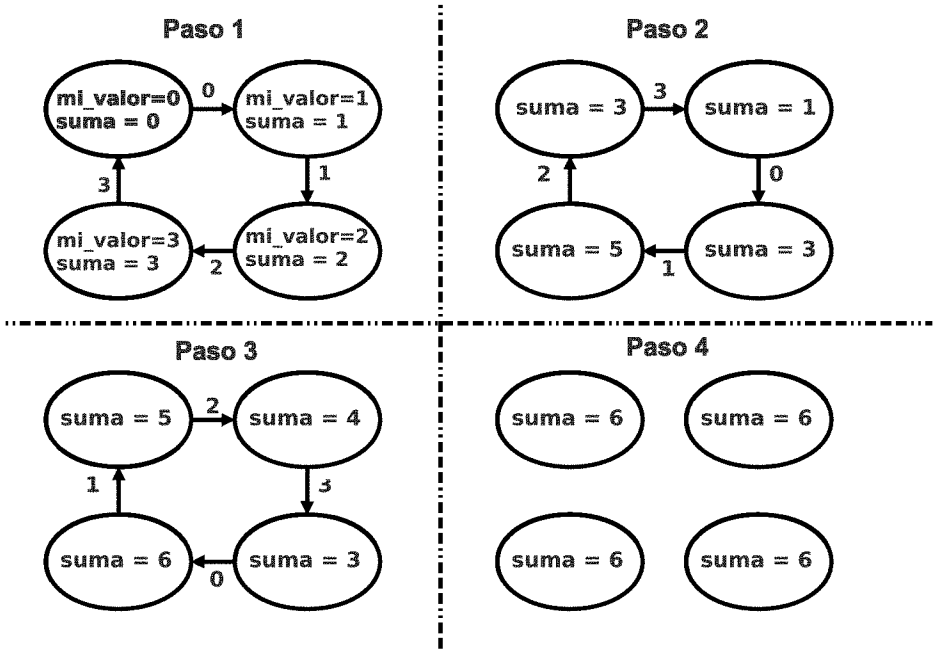
```
process Salvaje[ i : 0..2 ] ;
begin
   var peticion : integer := ... ;
begin
   while true do begin
      { esperar a servirse un misionero: }
      s_send( peticion, Olla );
      { comer: }
      Comer();
   end
end
```

```
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 := ...;
       llenar     : integer := ...;
       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);
            receive( esta_llena, Cocinero);
            contador := M;
      end { select }
end
```

36

Considerar un conjunto de N procesos, $P(i), i = 0, ..., N-1$ conectados 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 := ... ;
       suma     : integer ;
begin
   for j := 0 hasta N-1 do begin
      ...
   end
end
```

Respuesta (privada)

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

var mi_valor : integer := ... ;
    suma     : integer ;
    temp     : integer ;
begin
   for j := 0 hasta N-2 do
```

```
      if (i mod 2) = 0 then begin
         send( mi_valor, P[i+1 mod N] ) ;
         receive( mi_valor, P[i-1 mod N] );
         suma := suma + mi_valor ;
      end
      else begin
         temp := mi_valor;
         receive( mi_valor, P[i-1 mod N] );
         suma := suma + mi_valor ;
         send( temp, P[i+1 mod N] ) ;
      end
end
```

37

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
```

38

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 Cliente[ i : 0....n ] ;
var pet_usar: integer :=1;
    pet_liberar:integer:=2;
    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)

```
process Controlador ;
```

```
var
   permiso          : integer := ... ;
   peticion         : integer ;
   contador         : integer := 0  ; { numero de clientes usando el recurso }
   cli_esp_usar     : integer := -1 ; { cliente que espera usar (-1 si no hay)   }
   cli_esp_liberar  : integer := -1 ; { cliente que espera liberar (-1 si no hay) }
begin
   while true do begin
      select
      for i := 0 to n when receive( peticion, cliente[i] ) do
         if peticion = pet_usar then begin          { procesar peticion de uso: }
            if contador = 12  then begin            { si hay doce usando }
               if cli_esp_usar = -1  then           {  si no habia esperando usar: }
                  cli_esp_usar := i ;               {    ahora ya si hay }
               else begin                           {  habia uno esperando usar: }
                  contador := contador+2;           {    dos mas usando (pasa de 12 a 14) }
                  send( permiso, cliente[i] );      {    enviar permisos al que ha solicitado usar }
                  send( permiso, cli_esp_usar );    {    enviar permiso al que esperaba usar }
                  cli_esp_usar := -1 ;              {    ya no queda ninguno esperando usar }
               end
            end else if contador = 14 and cli_esp_liberar > -1 then begin
                                                    { si hay 14 y uno esperando liberar }
               send( permiso, cliente[i] );         {  enviar permiso al que ha solicitado usar }
               send( permiso, cli_esp_liberar );    {  enviar permiso al que esperaba liberar }
               cli_esp_lib := -1 ;                  {  ya no queda ninguno esperando liberar }
            end else begin                          { resto de casos: se puede permitir usar }
               contador := contador+1 ;             {  uno mas usando }
               send( permiso, cliente[i] );         {  enviar permiso para usar }
            end
         end else begin                             { procesar peticion de liberacion }
            if contador = 14 then begin             { si hay 14 usando }
               if cli_esp_liberar = -1 then         {  si ninguno esperaba liberar }
                  cli_esp_liberar := i;             {    ahora si espera uno liberar }
               else begin                           {  habia uno esperando liberar }
                  contador := contador-2 ;          {    dos menos usando (pasa de 14 a 12) }
                  send( permiso, cliente[i] );      {    enviar permiso al que ha solicitado liber. }
                  send( permiso, cli_esp_liberar);  {    enviar permiso al que esperaba liber. }
                  cliente_espera_liberar := -1 ;    {    no hay ninguno esperando liberar }
               end
            end else if contador = 12 and cli_esp_usar > -1 then begin
               send( permiso, cliente[i] );
               send( permiso, cli_esp_usar );
               cli_esp_usar := -1 ;
            end else begin
               contador := contador-1;
               send( permiso, cliente[i] );
            end
         end
      end { select }
   end { while true }
end { process }
```