Sistemas Concurrentes y Distribuidos:
Enunciados de problemas del tema 1.

Curso 2011-12

Contents

1 Problemas de iniciación.

1

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

Suponemos que `seqA` y `seqB` no son variables compartidas. 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.

Declaración e inicialización de variables globales

```
var x=0: integer ;
```

Proceso P_1:

```
for seqA = 1 to 2 do
begin
   x := x+1 ;
end
```

Proceso P_2:

```
for seqB = 1 to 2 do
begin
   x := x+1 ;
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 iteración 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.

Incorrecto :

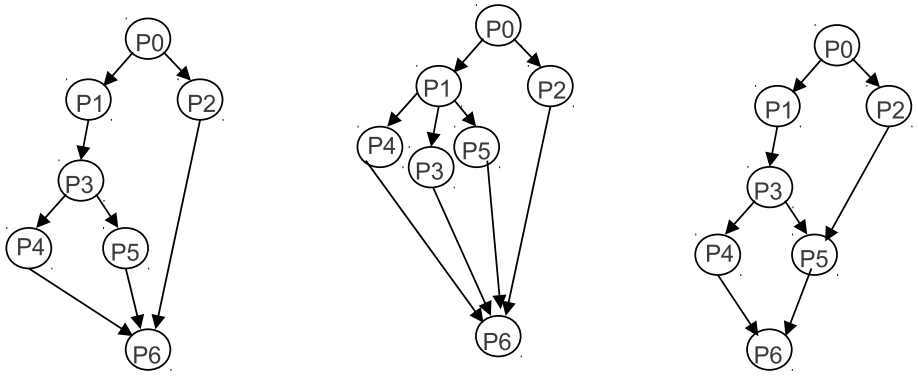
```
var v  : T ;
begin
   v := leer(f) ;
   while not fin(f) do
   cobegin
      escribir(g,v);
      v := leer(f) ;
   coend
end
```

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)

cobegin-coend :

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

fork-join :

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

(b)

cobegin-coend :

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

fork-join :

```
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).

cobegin-coend
(P4 espera a P2):

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

cobegin-coend
(P5 espera a P4):

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

fork-join:

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

4

Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

a) :

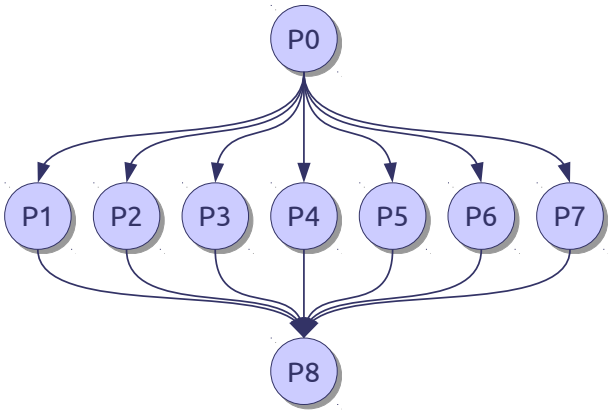
```
begin
   P0 ;
   cobegin
      P1 ;
      P2 ;
      cobegin
         P3 ; P4 ; P5 ; P6 ;
      coend
      P7 ;
   coend
   P8;
end
```

b) :

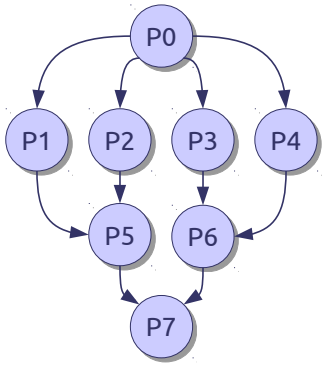
```
begin
   P0 ;
   cobegin
      begin
        cobegin
          P1;P2;
        coend
       P5;
      end
      begin
        cobegin
          P3;P4;
        coend
       P6;
      end
   coend
   P7 ;
end
```

Respuesta

En el caso a), anidar un bloque **cobegin**–**coend** dentro de otro, sin incluir ningún componente adicional en secuencia, tiene el mismo efecto que incluir directamente en el bloque externo las instrucciones del interno. Esta no es la situación en el el caso b), donde las construcciones **cobegin**–**coend** anidadas son necesarias para reflejar ciertas dependencias entre actividades.



(a)



(b)

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_periodo_hora` para esperar a que termine una hora.

Respuesta

El código de los procesos podría ser el siguiente:

Declaración e inicialización de variables globales

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

Proceso Acumulador :

```
while true do
 begin
   <Espera_Impulso()>;
   (1) <n:=n+1>;
 end
```

Proceso Escritor :

```
while true do
 begin
   <Espera_Fin_Periodo_hora()>;
   (2) write(n);
   (3) <n:=0>;

 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:

Declaración e inicialización de variables globales

```
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 }
```

ordenar un segmento de a :

```
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
```

ordenaciones sec. y conc. :

```
procedure secuencial ;
var i : integer ;
begin
   sort(1,1,2n) ; {ord. a}
   for i := 1 to 2*n do
      b[i] := a[i] ;
end
procedure concurrente ;
begin
  cobegin
     sort(1,1,n-1);
     sort(2,n,2n);
  coend
  merge(1,n+1,2n);
end
```

el código de **merge** puede quedar como sigue:

```
procedure merge ( inferior, medio, superior: integer)
var k,c1,c2,ind1,ind2:integer;
begin
   { k es la siguiente posición 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 terminación del while }
         end
      end
    end
end
```

Realiza las siguientes dos actividades:

a) Si el tiempo de ejecución secuencial del algoritmo es:

$$t(n) = \frac{n(n-1)}{2} \simeq n^2/2$$

¿Cuál sería el tiempo de ejecución del programa paralelo si la mezcla lleva en realizarse n operaciones?

b) Supongamos que queremos paralelizar las tres operaciones usando un procedimiento **concurrente2**, que tendría la siguiente estructura:

```
procedure concurrente2 ;
begin
  { inicialmente, no hay ninguna entrada completada: }
  uc[1] := 0 ;
  uc[2] := 0 ;
  { ordenación y mezcla concurrentes: }
  cobegin
     sort(1,1,n);
     sort(2,n+1,2n);
     merge2(1,n+1,2n);
  coend
end
```

Escribe el código de **merge2** adecuado para esto. Ten en cuenta que **merge2** es similar a **merge** solo que es necesario comprobar (antes de leer de a) qué parte de cada mitad de a ya ha sido ordenada y, por tanto, pueden ir siendo mezclada. Para ello, la función **merge2** debe incluir bucles de espera ocupada que leen los valores del vector uc. Como puedes observar (en **sort**), el valor uc[1] indica la última entrada escrita en la primera mitad de a, e igualmente el valor uc[2] indica la última escrita de la segunda mitad.

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 otro subprograma (**MultiplicacionConc**) con el mismo fin pero que use 3 procesos concurrentes (usando **cobegin**–**coend**). Suponer que los elementos de las matrices a y b se pueden leer simultáneamente, así como que elementos distintos de c pueden escribirse simultáneamente.

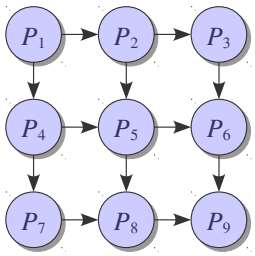
8

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

Programa: :

```
while true do
cobegin
    P1 ; P2 ; P3 ;
    P4 ; P5 ; P6 ;
    P7 ; P8 ; P9 ;
coend
```

Grafo de sincronización: :



Supón que queremos realizar la sincronización indicada en el grafo, usando para ello llamadas desde cada rutina a dos procedimientos (**EsperarPor** y **Acabar**). Se dan los siguientes hechos:

- El procedimiento **EsperarPor**`(i)` es llamado por una rutina cualquiera (la número `k`) para esperar a que termine la rutina número `i`, usando espera ocupada. Por tanto, se usa por la rutina `k` al inicio para esperar la terminación de las otras rutinas que corresponda según el grafo.
- El procedimiento **Acabar**`(i)` es llamado por la rutina número `i`, al final de la misma, para indicar que dicha rutina ya ha finalizado.
- Ambos procedimientos pueden acceder a variables globales en memoria compartida.
- Las rutinas se sincronizan única y exclusivamente mediante llamadas a estos procedimientos, siendo la implementación de los mismos completamente transparente para las rutinas.

Escribe una implementación de **EsperarPor** y **Acabar** (junto con la declaración e inicialización de las variables compartidas necesarias) que cumpla con los requisitos dados.

9

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

Declaración e inicialización de variables globales

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

Proceso 1:

```
while true do begin
   x := x+1 ;
   { .... }
end
```

Proceso 2:

```
while true do begin
   x := x+1 ;
   y := x ;
   { .... }
end
```

Proceso 3:

```
while true do begin
   y := y+1 ;
   { .... }
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:

Declaración e inicialización de variables globales

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

Proceso 1:

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

Proceso 2:

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

Proceso 3:

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

Declaración e inicialización de variables globales

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

Accede a x:

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

Accede a x e y:

```
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
```

Accede a y:

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

10

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.

2 Problemas a resolver usando semáforos.

11

Para calcular el número combinatorio

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

creamos un programa con dos procesos. El proceso $P1$ calcula el numerador y deposita el resultado en una variable compartida, denominada x, mientras que $P2$ calcula el factorial y deposita el valor en la variable y. Sincroniza los procesos $P1$ y $P2$, utilizando semáforos, para que el proceso $P2$ realice correctamente la división x/y.

Declaración e inicialización de variables globales

```
var n, k, x=1, i=1 : integer ;
var                : semaforo ;
```

Proceso P_1:

```


for i = n-k+1 to n do
begin

   x := x * i ;

end



.
```

Proceso P_2:

```

y := 1 ;

for j=2 to k do
begin

   y := y * j ;

end

escribir (x/y);

.
```

Nota: En los primeros problemas propuestos, vamos a dibujar una caja en la que aparece: (a) el recuadro superior contiene la declaración-inicialización de variables globales y semáforos necesarias para el ejercicio, (b) dos o más recuadros verticales que contienen fragmentos del código de los procesos. Se deja al lector la tarea de completar tanto las declaraciones-inicializaciones necesarias como el código necesario para resolver el ejercicio.

Respuesta

Declaración e inicialización de variables globales

```
var n, k, x=1, i=1       : integer ;
var x_ya_calculado = 0   : semaforo ;
```

Proceso P_1:

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

Proceso P_2:

```
y := 1 ;
for j=2 to k do
begin
   y := y * j ;
end
sem_wait(x_ya_calculado);
escribir (x/y);
```

12

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.

Declaración e inicialización de variables globales

```
.
```

Proceso P_1:

```
while true do
begin
   a
   b
   c
end
```

Proceso P_2:

```
while true do
begin
   d
   e
   f
end
```

Proceso P_3:

```
while true do
begin
   g
   h
   i
end
```

Respuesta

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

Declaración e inicialización de variables globales

```
var S := 0 : semaforo ;
```

Proceso P_1:

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

Proceso P_2:

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

Proceso P_3:

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

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

Declaración e inicialización de variables globales

```
var S1 := 0, S3 := 0 : semaforo ;
```

Proceso P_1:

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

Proceso P_2:

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

Proceso P_3:

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

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

Declaración e inicialización de variables globales

```
var S2, S3 := 0 : semaforo ;
```

Proceso P_1:

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

Proceso P_2:

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

Proceso P_3:

```
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.

Declaración e inicialización de variables globales

```
var mutex := 2 : semaforo ;
```

Proceso P_1:

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

Proceso P_2:

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

Proceso P_3:

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

13

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.

Declaración e inicialización de variables globales

```
.
```

Proceso P_1:

```
while true do begin
   dato_calculado:=
           calcular_dato();
   n:=n+1;
   ED[n]:=dato_calculado;
end
```

Proceso P_2:

```
while true do begin
   dato:=ED[n];
   n:=n-1
   operar_dato(dato);
end
```

Respuesta

Declaración e inicialización de variables globales

```
var S := 0, mutex := 1 : Semaforo ;
var n := 0 : Integer ;
```

Proceso P_1:

```
while true do begin
   dato_calculado:=
           calcular_dato();
   sem_wait(mutex);
      n := n+1;
      ED[n] := dato_calculado;
   sem_signal(mutex);
   sem_signal(S);
end
```

Proceso P_2:

```
while true do begin
   sem_wait(S);
   sem_wait(mutex);
      dato := ED[n] ;
      n := n-1 ;
   sem_signal(mutex);
   operar_dato(dato);
end
```

14

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...

Declaración e inicialización de variables globales

```
.
```

Proceso P_1:

```
while true do begin
   m := 2*x-n ;
   imprimir(m);
end
```

Proceso P_2:

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

Respuesta

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

Declaración e inicialización de variables globales

```
var x : Integer ;
var x_ya_calculado := 0, x_ya_leido := 1 : Semaforo ;
```

Proceso P_1:

```
while true do begin
   sem_wait( x_ya_calculado );
   m := 2*x-n ;
   sem_signal( x_ya_leido );
   imprimir(m);
end
```

Proceso P_2:

```
while true do begin
   leer_teclado(d);
   sem_wait( x_ya_leido ) ;
   x := d-c*5 ;
   sem_signal( x_ya_calculado ) ;
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...

Declaración e inicialización de variables globales

```
var x : Integer ;
var x_ya_calculado := 0, x_ya_leido := 1 : Semaforo ;
```

Proceso P_1:

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

Proceso P_2:

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

15

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.

Declaración e inicialización de variables globales

```
.
```

Proceso P_1:

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

Proceso P_2:

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

Proceso P_3:

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

Proceso P_4:

```
while true do
begin
  c_rue();
  c_rue();
  c_cua();
  c_man();
  m_bic();
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.

Declaración e inicialización de variables globales

```
{* sema. que indican si hay espacio para comenzar pieza *}
var erue, ecua, eman : semaforo := 1 ;
{* sem. que indican si hay una pieza ya montada *}
var hrue, hcua, hman : semaforo := 0 ;
```

Proceso P_1:

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

Proceso P_2:

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

Proceso P_3:

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

Proceso P_4:

```
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
```

(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).

16

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.

Declaración e inicialización de variables globales

```
.
```

Proceso P_1:

```
{* comensal 1 *}

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

Proceso P_2:

```
{* comensal 2 *}

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

Proceso P_3:

```
{* camarero
  (apartado b) *}
while true do begin
   quitar_sopa();
   poner_sopa();
end
```

17

Resuelva el problema del Productor-Consumidor con y sin buffer limitado utilizando semáforos binarios.

18

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. Necesitamos cinco procesos concurrentes, cada uno de ellos describe la actividad de un filósofo. Incluye las operaciones sobre semáforos necesarias para implementar el hecho de que antes de comer hay que disponer del tenedor de la derecha y el de la izquierda, y que cuando se termine la actividad de comer, hay que liberar 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. Diseña una solución a este problema.

19

Resuelva el problema de los Lectores-Escritores que se detalla a continuación utilizando semáforos. Suponemos un sistema de base de datos que es compartido por varios procesos. Estos procesos se dividen en dos clases distintas según el tipo de acceso a la base de datos: procesos lectores que leen información y procesos escritores que leen y modifican información de la base de datos común. Los objetivos propuestos son:

1) asegurar la consistencia de la base de datos y
2) procurar la máxima concurrencia en el acceso a la base de datos.

Y tenemos las siguientes situaciones:

a) varios lectores pueden acceder simultáneamente a la base de datos.
b) cuando un escritor está accediendo a la base de datos, ningún otro proceso puede acceder a la base de datos, ni lector ni escritor.

Diseña dos soluciones para este problema usando semáforos, cada una de ellas cumpliendo los requisitos que se describen en cada uno de estos dos puntos:

1. prioridad a los lectores: en esta solución, cuando un escritor accede para escribir en la BD esperará a que finalicen todos los lectores que haya leyendo (un lector que acceda cuando otros están lectores están leyendo bloqueará a los escritores en espera incluso aunque dichos escritores hubiesen llegado antes que el último lector). Decimos que los lectores pueden acceder en ráfagas.
2. prioridad a los escritores: en esta solución, cuando un escritor accede para escribir, únicamente tendrá que esperar al lector que haya leyendo, si hay alguno. Cuando un lector espera a que termine un escritor, el lector seguirá en espera hasta que no haya más escritores, incluyendo la espera a los escritores que pudiesen acceder después de la llegada del lector. En este caso, los escritores son los que pueden acceder en ráfagas.

20

La vida real esta llena de problemas de sincronización. Uno de estos casos se da en las barberías. Suponga una barbería con un barbero, una silla para cortar el pelo y n sillas para que los clientes esperen sentados a que les toque su turno. Si no hay clientes, el barbero, que es bastante vago, duerme en su silla. Cuando llega un cliente debe despertar al barbero. Si llegan más clientes mientras que el barbero esta pelando y hay sillas libres, se sientan y esperan su turno. Si no hay sillas libres se van. Programa una solución para este problema, denominado el problema del *Barbero y los Clientes.*

21

La exclusión mutua utilizando semáforos se logra inicializando un semáforo a 1 (llamémoslo **mutex**), comenzando todas las secciones críticas con una operación **sem_wait**(**mutex**) y finalizando todas las secciones críticas con una operación **sem_signal**(**mutex**) ¿Qué ocurre si un proceso que está dentro de una sección crítica realiza una operación **sem_wait**(**mutex**) ?

Respuesta

Al estar el semáforo a cero, ese proceso quedaría bloqueado, por tanto no llega a ejecutar el **sem_signal** que desbloquearía el semáforo y permanece bloqueado indefinidamente. Cualquier otro proceso que pretenda entrar en la exclusión mutua quedará asimismo bloqueado en el **sem_wait**. Por tanto hay interbloqueo que causa inanición.

22

Supongamos uno de los puntos de la vía ferroviaria donde dos vías (`via_1`, `via_2`) se unen en una sola (`via_3`). El problema que se plantea es controlar dicho punto para que cuando lleguen los trenes no se produzcan problemas. Para modelar dicho problema suponemos que tenemos los siguientes tipos de procesos: `tren_via_1`, `tren_via_2` (de cada uno de ellos puede haber un número arbitrario de instancias ejecutándose) y `controlador` (una única instancia). Considera la siguiente situación:

- `tren_via_1`, `tren_via_2`: representan a los trenes que llega por la `via_1`, y a los que llegan por la `via_2` respectivamente. Si llegan dos trenes a la vez (uno por la `via_1` y otro por la `via_2`) pasará a la `via_3` el primero que llegue, pero antes de pasar debe recibir el permiso del controlador. Una vez tiene el permiso, invocará la función `pasar()` para acceder a la `via_3`, y una vez ejecutada dicha función, el tren continua su viaje sin más restricciones (llama a `continuar()`).
- `controlador`: está en un ciclo infinito esperando que llegue un tren, cuando llega un tren realiza las acciones adecuadas (llamando a `acciones()`) y le da paso.

El esquema de los procesos (excluyendo la sincronización) sería este:

Declaración e inicialización de variables globales

```
...
```

tren vía 1:

```
...
pasar() ;
...
continuar() ;
...
```

tren vía 2:

```
...
pasar() ;
...
continuar() ;
...
```

Proceso 3:

```
while true do begin
   ...
   acciones();
   ...
end
```

A continuación se incluyen tres posibles soluciones, ninguna de las cuales es válida. Para cada una de ellas describe razonadamente porqué no es válida. Además, incluye una solución válida.

- **Solución 1.**

  Declaración e inicialización de variables globales

```
var paso : semaforo := 0 ;
var S    : semaforo := 1 ;
```

  tren vía 1:

```
sem_wait(paso) ;
pasar() ;
sem_signal(S);
continuar() ;
```

  tren vía 2:

```
sem_wait(paso) ;
pasar() ;
sem_signal(S);
continuar() ;
```

  Proceso 3:

```
while true do begin
   sem_wait(S);
   acciones();
   sem_signal(paso);
end
```

- **Solución 2.**

  Declaración e inicialización de variables globales

```
var tren : semaforo := 0 ;
var paso : semaforo := 0 ;
```

  tren vía 1:

```
sem_signal(tren);
sem_wait(paso) ;
pasar() ;
continuar() ;
```

  tren vía 2:

```
sem_signal(tren);
sem_wait(paso) ;
pasar() ;
continuar() ;
```

  Proceso 3:

```
while true do begin
   sem_wait(tren);
   acciones();
   sem_signal(paso);
end
```

- **Solución 3.**

  Declaración e inicialización de variables globales

```
var controlador : semaforo := 1 ;
var pasado1     : semaforo := 0 ;
var pasado2     : semaforo := 0 ;
```

  tren vía 1:

```
sem_wait(controlador) ;
pasar() ;
sem_signal(pasado1);
continuar() ;
```

  tren vía 2:

```
sem_wait(controlador) ;
pasar() ;
sem_signal(pasado2);
continuar() ;
```

  Proceso 3:

```
while true do begin
   sem_signal(controlador);
   acciones();
   sem_wait(pasado1);
   sem_wait(pasado2);
end
```

23

Dos procesos P_1 y P_2 se pasan información a través de una estructura de datos formada por un entero `n` y una matriz `ma` (de 10 elementos). El valor de `n` debe ser en cada momento el número de datos útiles en la matriz; su valor inicial es 0. El proceso P_1 coloca un dato en el elemento siguiente al `n`. El proceso P_2 retira cada vez el último elemento colocado por P_1. Para conseguir que P_2 no retire elementos si el número de elementos existentes es 0, se ha programado lo siguiente:

Declaración e inicialización de variables globales

```
var eltos : semaforo := 0 ;
var n     : integer  := 0 ;
var ma    : array [0..9] of integer ;
```

Proceso P_1:

```
while true do begin
   {a1} calcular dato
   {a2} n := n + 1 ;
   {a3} ma[n] := dato ;
   {a4} sem_signal(eltos) ;
end
```

Proceso P_2:

```
while true do begin
   {b0} if n = 0 then sem_wait(eltos);
   {b1} valor := ma[n];
   {b2} n := n-1;
   {b3} hacer calculos con  'valor'
end
```

¿Se consigue que P_2 espere antes de *{b1}* si el número de elementos existentes es 0 ? Si la respuesta es afirmativa, justifícalo y en caso contrario indica un posible intercalado de instrucciones que justifique tu respuesta

24

En un sistema electrónico de transferencia de fondos, existen miles de procesos idénticos que funcionan como sigue. Cada proceso lee en una línea la cantidad de dinero a transferir, el número de cuenta destinataria y el número de cuenta de origen de la transferencia. Entonces, bloquea las dos cuentas, transfiere el dinero, y libera las cuentas. Con muchos procesos así, existe un peligro real de interbloqueo (un proceso puede bloquear la cuenta x pero no puede bloquear la cuenta y porque otro proceso la tiene bloqueada y ese proceso esta directa o indirectamente esperando la cuenta x). Describa un esquema que evite el peligro de interbloqueo. No esta permitido desbloquear una cuenta una vez que se ha bloqueado hasta que la transacción se complete.

25

En 1910, tres fumadores empedernidos entraron en un estanco para comprar los materiales que necesitaban para hacerse un cigarro. Para fabricar y fumar un cigarro cada fumador necesitaba tres elementos: tabaco, papel y cerillas. El estanquero tenía existencias ilimitadas. El primer fumador, F_1, tenía su propio tabaco, el segundo, F_2, su propio papel, y el tercero, F_3, sus propias cerillas. La acción comienza cuando el estanquero colocó dos de los elementos necesarios en el mostrador con el fin de permitir que sólo uno de los fumadores liara el cigarro y fumara. Cuando el fumador terminaba, se lo notificaba al estanquero, quien colocaba otros dos elementos de forma aleatoria. Se propone la siguiente solución ¿es correcta? Razónelo.

Declaración e inicialización de variables globales

```
{* semaforos que indican: tabaco, papel, cerillas *}
var t,p,c    : semaforo := 0 ;
{* semaforo para indicar al estanquero que ponga productos *}
var trab  : semaforo := 1 ;
```

Proceso P_1:

```
{* estanquero *}

while true do
begin
  sem_wait(trabajo);
  n := random(3);
  case n of
  0: sem_signal(t);
     sem_signal(p);
  1: sem_signal(t);
     sem_signal(c);
  2: sem_signal(p);
     sem_signal(c);
  end
end
```

Proceso P_2:

```
{* F1 *}

while true do
begin
  sem_wait(p);
  sem_wait(c);
  fumar;
  sem_signal(trab);
end
```

Proceso P_3:

```
{* F2 *}

while true do
begin
  sem_wait(t);
  sem_wait(c);
  fumar;
  sem_signal(trab);
end
```

Proceso P_4:

```
{* F3 *}

while true do
begin
  sem_wait(t);
  sem_wait(p);
  fumar;
  sem_signal(trab);
end
```

26

Se pretende modelar parte del funcionamiento de una secretaría de un centro de estudiantes. En este sistema tenemos dos ventanillas para atender las peticiones de los estudiantes. Un estudiante solicita un certificado de notas y recibe un carta de pago. Una vez pagado el importe correspondiente, vuelve con la carta de pago sellada (resguardo), lo presenta en ventanilla y obtiene su certificado de notas. El encargado de la ventanilla debe saber si la petición del estudiante es:

a) la solicitud del certificado: en este caso le dará al estudiante la carta de pago.

b) la carta de pago sellada: en este caso le dará al estudiante el certificado de notas.

Aunque existen dos ventanillas (y dos encargados), sólo hay una cola de estudiantes que esperan ser atendidos.

Implemente los procesos `Estudiante` y `Encargado` utilizando semáforos y las variables que crea oportunas.

27

El código de la figura es una propuesta de solución al problema de lectores-escritores. Se asume que inicialmente todos los semáforos valen `1` y la variable `nlec` vale 0. Indicar, y justificar, si son verdaderas o falsas las siguientes afirmaciones relativas a la solución:

a) Garantiza la exclusión mutua en el acceso a la variable `nlec`

b) Existe prioridad para los escritores.

c) El objetivo de `sem3` es conseguir que no haya más de un lector leyendo simultáneamente.

d) En esta solución se da inanición para los lectores.

e) Permite la inanición de los escritores.

f) La solución no es correcta porque `sem2` debería valer 0 inicialmente.

Declaración e inicialización de variables globales

```
var sem1, sem2, sem3 : semaforo := 1 ;
var nlec             : integer  := 0 ;
```

Proceso P_1:

```
{* escritores *}

...
sem_wait(sem1);
sem_wait(sem2);
escribir;
sem_signal(sem2);
sem_signal(sem1);
...
```

Proceso P_2:

```
{* lectores *}
....
sem_wait(sem1);
sem_wait(sem3);
nlec := nlec+1;
if nlec = 1 then
   sem_wait(sem2);
sem_signal(sem3);
sem_signal(sem1)

leer;

sem_wait(sem3);
nlec := nlec-1;
if nlec = 0 then
   sem_signal(sem2);
sem_signal(sem3);
....
```

28

Suponga que tiene que implementar el siguiente sistema:

- Tenemos `n` procesos de tipo `Cliente` y tres procesos de tipo `Servidor`.
- Hay dos variables globales, `x` e **`y`**, que sirven de comunicación entre los procesos `Cliente` y los procesos `Servidor`.
- Cada proceso `Cliente` lee desde teclado una variable que deposita en la variable compartida `x`
- El valor almacenado en la variable `x` se utiliza por uno de los tres procesos `Servidores`, el primero que esté libre, para calcular su logaritmo.
- El `Servidor` calcula el logaritmo de `x` y lo deposita en la variable compartida **`y`** para que finalmente el `Cliente` visualice el resultado del cálculo.
- Se debe calcular el logaritmo (una única vez) de cada valor almacenado en `x` por cada uno de los procesos `Cliente`.
- Nótese que los procesos `Servidor` son ciclos infinitos y los procesos `Cliente` no.

Declaración e inicialización de variables globales

Proceso P_1:

```
{* clientes *}

x := leer() ;

mostrar(y) ;
```

Proceso P_2:

```
{* servidor 1 *}

while true do
begin

   y := log(x);

end
```

Proceso P_3:

```
{* servidor 2 *}

while true do
begin

   y := log(x);

end
```

Proceso P_4:

```
{* servidor 3 *}

while true do
begin

   y := log(x);

end
```

3 Problemas de verificación.

29

Dado el siguiente programa:

$$
\begin{array}{c}
int \quad x = 5, \quad y = 2; \\
\textbf{cobegin} \\
< \mathbf{x} = \mathbf{x} + \mathbf{y} > \; || \; < \mathbf{y} = \mathbf{x} * \mathbf{y} > \\
\textbf{coend}
\end{array}
$$

a) Cuáles son los posibles valores de x y de y suponiendo que los ángulos indican que la sentencia que encierran se ejecuta de forma atómica?

b) ¿Cuáles serían los posibles valores de x y de y en el caso de que se suprimieran los ángulos y cada orden de asignación fuera implementada usando tres sentencias atómicas (lectura de memoria a registro, suma o multiplicación y escritura de registro a memoria)?

Respuesta

a) - Si se ejecuta la secuencia $< \mathbf{x} = \mathbf{x} + \mathbf{y} >< \mathbf{y} = \mathbf{x} * \mathbf{y} >$, entonces $\{x == 7 \wedge y == 14\}$.
   - Si se ejecuta la secuencia $< \mathbf{y} = \mathbf{x} * \mathbf{y} >< \mathbf{x} = \mathbf{x} + \mathbf{y} >$, entonces $\{x == 15 \wedge y == 10\}$.

b) La expansión de las sentencias, asumiendo que A y B son registros quedaría:

Declaración e inicialización de variables globales

```
int x=5, y=2;
```

Proceso P_1:

```
(1) load x,A
(3) add A,y
(5) store A,x
```

Proceso P_2:

```
(2) load y,B
(4) mult B,x
(6) store B,y
```

Existen tres posibilidades generales:

1. Que (3) y (4) se ejecuten antes que (5) y (6), entonces $\{x == 7 \wedge y == 10\}$.
2. Que (5) se ejecute antes que (4), entonces $\{x == 7 \wedge y == 14\}$.
3. Que (6) se ejecute antes que (3), entonces $\{x == 15 \wedge y == 10\}$.

30

Dada la siguiente precondición y orden de asignación:

$$\{x \geq 2\} \quad < \mathbf{x} = \mathbf{x} - \mathbf{2} >$$

Para cada una de las siguientes ternas, comprobar si la orden anterior interfiere con dicha terna:

a) $\{x \geq 0\} \quad < \mathbf{x} = \mathbf{x} + \mathbf{3} > \quad \{x \geq 3\}$

b) $\{x \geq 0\} \quad < \mathbf{x} = \mathbf{x} + \mathbf{3} > \quad \{x \geq 0\}$

c) $\{x \geq 7\} \quad < \mathbf{x} = \mathbf{x} + \mathbf{3} > \quad \{x \geq 10\}$

e) $\{y \geq 0\} \quad < \mathbf{y} = \mathbf{y} + \mathbf{3} > \quad \{y \geq 3\}$

f) $\{x$ es impar$\} \quad < \mathbf{y} = \mathbf{x} + \mathbf{1} > \quad \{y$ es par$\}$

Respuesta

Hay que demostrar que la orden de asignación $< \mathbf{x} = \mathbf{x} - \mathbf{2} >$ asumiendo su precondición ($\{x \geq 2\}$) podría ejecutarse antes (resp. después) de cada una de las órdenes mostradas sin invalidar su aserto precondición (resp. poscondición). Eso implica que para cada terna $\{Pre(P)\} \quad P \quad \{Pos(P)\}$ hay que demostrar que las siguientes ternas son consistentes:

$$\{Pre(P) \wedge x \geq 2\} \quad < \mathbf{x = x - 2} > \quad \{Pre(P)\}$$
$$\{Pos(P) \wedge x \geq 2\} \quad < \mathbf{x = x - 2} > \quad \{Pos(P)\}$$

a) Sí interfiere ya que:
   a.1. $\{x \geq 0 \wedge x \geq 2\} \quad < \mathbf{x = x - 2} > \quad \{x \geq 0\}$ es consistente,
   a.2. $\{x \geq 3 \wedge x \geq 2\} \quad < \mathbf{x = x - 2} > \quad \{x \geq 3\}$ no lo es.

b) No interfieren ya que la precondición y poscondición coinciden y se cumple (a.1.).

c) Sí interfieren ya que:
   $\{x \geq 7 \wedge x \geq 2\} \quad < \mathbf{x = x - 2} > \quad \{x \geq 7\}$ no es consistente.

e) No interfiere ya que se referencia una variable diferente y en los asertos precondición y poscondición.

f) No interfiere ya que:
   $\{x$ es impar $\wedge\, x \geq 2\} \quad < \mathbf{x = x - 2} > \quad \{x$ es impar$\}$ es consistente.
   $\{y$ es par $\wedge\, x \geq 2\} \quad < \mathbf{x = x - 2} > \quad \{y$ es par$\}$ es consistente.

31

Se quiere estudiar cuáles son los valores finales de las variables x e y en el siguiente programa:

```
int   x = C1,   y = C2;
x = x + y;
y = x * y;
x = x - y;
```

Incrusta asertos después de cada sentencia para crear una traza de prueba que describa los efectos del programa.

32

Dada la siguiente construcción de composición concurrente P (en la que se asume que la variable x tiene valor 0 al comienzo de la misma):

cobegin
$< \mathbf{x = x + 1} > || < \mathbf{x = x + 2} > || < \mathbf{x = x + 4} >$
coend

Demostrar que:

$$\{x == 0\} \quad P \quad \{x == 7\}$$

33

Dado la siguiente construcción de composición concurrente P:

cobegin
$< \mathbf{x = x - 1} >$ $||$ $< \mathbf{y = y + 1} >$
$< \mathbf{x = x + 1} >$ $||$ $< \mathbf{y = y - 1} >$
coend

Demostrar que:

$$\{x == y\} \quad P \quad \{x == y\}$$

Sistemas Concurrentes y Distribuidos:
Problemas resueltos del tema 2

Curso 2011-12

Contents

1 Problemas sobre algoritmos de exclusión mutua.

1

¿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 }
variable entera c0, c1 ; { los valores iniciales no son relevantes }
```

```
1  { Proceso 0 }
2  mientras verdadero hacer
3    c0 := 0 ;
4    mientras c1 = 0 hacer
5        c0 := 1 ;
6        mientras  c1 = 0 hacer fin
7        c0 := 0 ;
8    fin
9    { sección crítica }
10   c0 := 1 ;
11   { resto sección }
12 fin
```

```
1  { Proceso 1 }
2  mientras verdadero hacer
3     c1 := 0 ;
4     mientras c0 = 0 hacer
5        c1 := 1 ;
6        mientras c0 = 0 hacer fin
7        c1 := 0 ;
8     fin
9     { sección crítica }
10    c1 := 1 ;
11    { resto sección }
12 fin
```

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 4). 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 6, 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 4 y 7, 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 6. Como ambas condiciones son forzosamente falsas, ambos pueden abandonarlo, ejecutando ambos la asignación de la línea 7 y la lectura de la línea 4 antes de que ninguno de ellos haga la asignación de la línea 5. Por tanto las dos condiciones de la línea 4 se cumplen cuando se comprueban y ambos vuelven a entrar en el bucle de la línea 6. A partir de aquí se repite la interfoliación descrita en este párrafo, lo cual puede ocurrir indefinidamente.

2

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 }
variable entera  c0 := 1, c1 := 1, turno := 1 ;
```

```
1  { Proceso 0 }
2  mientras verdadero hacer
3     c0 := 0 ;
4     mientras turno != 0 hacer
5        mientras c1 = 0 hacer fin
6        turno := 0 ;
7     fin
8     { sección crítica }
9     c0 := 1 ;
10    { resto sección }
11 fin
```

```
1  { Proceso 1 }
2  mientras verdadero hacer
3     c1 := 0 ;
4     mientras turno != 1 hacer
5        mientras c1 = 1 hacer fin
6        turno := 1 ;
7     fin
8     { sección crítica }
9     c1 := 1 ;
10    { resto sección }
11 fin
```

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

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

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 5 pero aún no ha realizado la asignación de la línea 6. 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 cualquier 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 6 y la lectura de la línea 4, 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 5. 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.

3

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` 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 }
variable entera n1:=0,n2:=0;
```

```
{ Proceso 1 }
mientras verdadero hacer
   n1 := 1 ;                      { 1.0 }
   n1 := n2+1 ;                   { 1.1 }
   mientras n2 != 0  y            { 1.2 }
            n2 <  n1 hacer fin;   { 1.3 }
   { sección crítica }
   n1 := 0 ;                      { 1.4 }
   { resto sección }
fin mientras
```

```
{ Proceso 2 }
mientras verdadero hacer
   n2 := 1 ;                      { 2.0 }
   n2 := n1+1 ;                   { 2.1 }
   mientras n1 != 0 y             { 2.2 }
            n1 <= n2 hacer fin;   { 2.3 }
   { sección crítica }
   n2 := 0 ;                      { 2.4 }
   { resto sección }
fin mientras
```

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 n_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

4

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 }
variable entera c0:=1,c1:=1;
```

```
1  { Proceso 0 }
2  mientras verdadero hacer
3     repite
4        c0 := 1-c1 ;
5     hasta c1 != 0 ;
6     { sección crítica }
7     c0 := 1 ;
8     { resto sección }
9  fin mientras
```

```
1  { Proceso 1 }
2  mientras verdadero hacer
3     repite
4        c1 := 1-c0 ;
5     hasta c0 != 0 ;
6     { sección crítica }
7     c1 := 1 ;
8     { resto sección }
9  fin mientras
```

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 4, y las lecturas de la 5 (ambos procesos escriben y después leen el valor 0), antes de que ninguno de los dos repita las asignaciones de la línea 4.
2. Se repiten las asignaciones de la línea 4 y las lecturas de la 5 (ambos procesos escriben y después leen el valor 1) antes de que ningún proceso alcance la línea 7.

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

5

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 }
variable vector[0..n-1] de (pasivo,solicitando,enSC) c := [pasivo,...,pasivo];
variable entera turno := 0 ;
```

```
1  { Proceso número i }
2  mientras verdadero hacer
3     repetir
4        c[i] := solicitando ;
5        j := turno;
6        mientras j <> i hacer
7           si c[j] <> pasivo entonces
8              j := turno ;
9           sino
10             j := (j-1) mod n ;
11       fin
12       c[i] := enSC ;
13       k := 0;
14       mientras  k<=n y ( k=i o c[k]<>enSC ) hacer
15          k := k+1;
16       fin
17    hasta k > n ;
18    turno := i ;
19    { sección crítica }
20    turno := (i-1) mod n ;
21    c[i] := pasivo ;
22    { resto sección }
23 fin mientras
```

Respuesta (privada)

En el bucle de las líneas 6 a 11, 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 **mientras** 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 **`repetir...`** inicial, y vuelven a poner su estado a `intentando`.

2 Problemas de programación de monitores.

6

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.

7

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.

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;
var r1,r2,ambos : condicion;
```

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:

```
procedimiento Pedir_recurso( num_recurso : integer )
begin
   case num_recurso of
   0: begin
         if n1 == 0 or n2 == 0 then
            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)

```
procedimiento Liberar_recurso( num_recurso : integer )
begin
   case num_recurso of
   1: begin
         n1 := n1+1 ;
         if n2 > 0 and not ambos.empty() then
            ambos.signal();
         else
            r1.signal();
      end
   2: begin
         n2 := n2+1 ;
         if n1 > 0 and not ambos.empty() then
            ambos.signal();
         else
            r2.signal();
      end
end
```

Código de inicialización:

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

8

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_PL ;

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

procedure escritura_ini() ;
begin
   if escribiendo or nlectores > 0 then { si no se puede escribir:       }
      escritores.wait() ;               {   esperar                      }
   escribiendo := true ;                { anotar que se está 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 algún escritor escribiendo: }
      lectores.wait() ;        {    esperar                         }
   nlectores := nlectores+1 ; { anotar un lector leyendo más       }
   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

{ inicialización }
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_PL ;

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

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

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

(b) sin prioridad

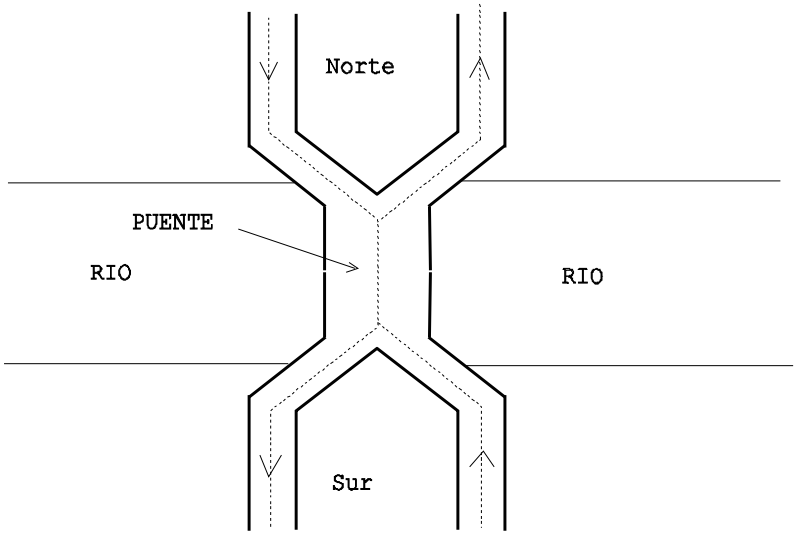
En esta solución no se pueden usar dos colas, en ese caso siempre habría que elegir una frente a otra para despertar un proceso. Por eso, todos los procesos esperan en una sola cola (llamada `cola`).

```
Monitor LectoresEscritores_PL ;

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

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

```
procedure escritura_fin() ;
begin
   escribiendo := false ; { anotar que no se está 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 más  }
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

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

9

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  ... ;

Procedimiento EntrarCocheDelNorte()
begin
   ...
end
Procedimiento SalirCocheDelNorte()
begin
   ....
end
Procedimiento EntrarCocheDelSur()
begin
   ....
end
Procedimiento SalirCocheDelSur()
begin
  ...
end

{ Inicialización }
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 ;
var N,S                   : condicion ;

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

Procedimiento SalirCocheDelNorte()
begin
   N_cruzando := N_cruzando-1 ;
   if N_cruzando == 0 then
      S.signal();
end

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

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

{ Inicialización }
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_delante, S_delante    : integer ;
var N,S                     : condicion ;
```

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

   { aquí 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 más del norte cruzando }

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

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

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

   if N_cruzando == 0 then begin { si el puente queda vacío }
      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.

```
Procedimiento 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
Procedimiento  SalirCocheDelSur
begin
  S_cruzando := S_cruzando-1 ;
  if S_cruzando == 0 then begin
     N_pueden = 10 ;
     N.signal() ;
  end
end
{ Inicialización }
begin
   N_cruzando :=  0  ; S_cruzando :=  0 ;
   N_pueden   := 10  ; S_pueden   := 10 ;
end
```

10

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.

Proceso Salvaje :

```
while true do
  begin
    Servirse_1_misionero();
    Comer();
  end
```

Proceso Cocinero :

```
while true do
  begin
    Dormir();
    Rellenar_Olla();
  end
```

Implementar un monitor 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)

Suponemos que `Servirse_1_misionero`, `Dormir` y `Rellenar_Olla` son procedimientos del monitor, mientras que `Comer` es un procedimiento con un retardo arbitrario que no pertenece al monitor (no se incluye su código).

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 : condicion ;

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
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

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

11

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.

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. Suponer que existe una función denominada `cantidad(cond)` que devuelve el valor de la cantidad (parámetro `c` de los procedimientos `retirar` y `depositar`) que espera retirar el primer proceso que se bloqueó (tras ejecutar **`wait`** en la señal `cond`).

Respuesta (privada)

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

```
Monitor CuentaCorriente;
int saldo; condition cond;

Retirar(int cantidad)
 {  while (cantidad>saldo)
     {cond.signal; cond.wait;}
      saldo-=cantidad;
      if (cond.queue()) cond.signal;
 }

Depositar (int cantidad)
 { saldo+=cantidad;
   if (cond.queue()) cond.signal;
}

{saldo=? };
```

Solución con colas con prioridad:

```
Monitor CuentaCorriente;
int saldo; condition cond;

Retirar(int cantidad)
{ while (cantidad>saldo) cond.wait (cantidad);
  saldo-=cantidad;  if (cond.queue()) cond.signal;
}

Depositar(int cantidad)
{ saldo+=cantidad;
if (cond.queue()) cond.signal;
```

```
}

{saldo=? };
```

b)

```
Monitor CuentaCorriente;
int saldo; condition cond;

Retirar(int cantidad)
{ if (cantidad>saldo || cond.queue( )) cond.wait;
  saldo-=cantidad;
  if ( cantidad(cond) <=saldo && cond.queue( ) )
      cond.signal;
}

Depositar(int cantidad)
  { saldo+=cantidad;
    if (cond.queue() && cantidad(cond) <=saldo )
     cond.signal;
  }

{saldo=? };
```

caso (b) con colas de prioridad

para el caso (b), se intenta da una solución aquí que usa colas de prioridad. Esto permite una solución sencilla sin necesidad de hacer mención a la función que se indica en el enunciado original. Para ello, los clientes que retiran hacen **wait** usando como prioridad un contador que indica el número de orden en la cola.

```
Monitor cuenta ;

var saldo, contador : integer ;
    cola            : condicion ;


procedure retirar( cantidad : integer ) ;
var numero : integer ;
begin
   numero := contador ;         { guardar numero propio                }
   contador := contador + 1 ;   { incrementar contador                 }
   while cantidad > saldo do    { mientras el saldo no sea suficiente  }
      wait(numero) ;            {   esperar                            }
   saldo := saldo - cantidad ;  { retirar la cantidad                  }
   cola.signal() ;              { avisar al siguiente que llegó, si hay }
end

procedure depositar( cantidad : integer ) ;
begin
   saldo := saldo + cantidad ; { depositar }
```

```
   cola.signal() ;  { avisar al que más tiempo lleve esperando, si hay alguno }
end

{ inicialización }
begin
   saldo := 0 ;
   contador := 0 ;
end
```

12

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, $(1 \leq i \leq n)$, donde los números menores implican mayor prioridad. Implementar un monitor que implemente los procedimientos para `pedir` y `liberar` el recurso

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

{ Inicialización }
begin
   ocupado := false;
end
```

13

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.

Respuesta (privada)

```
Monitor Sincronizacion ;

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

procedure SincA ()
begin
   nA := nA+1 ;               { uno más de tipo A.          }
   if nB < 10  then           { si aún 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 más de tipo B.                   }
   if nA < 1 or nB < 10 then  { si no está el A o no están 10 del B: }
      condB.wait() ;          {    esperar a que estén todos         }
```

```
   else begin                    { si ya está el A y soy el último 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

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

14

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                  }

procedimiento entrada;
begin
  n := n+1 ;       { ha llegado un proceso más }
  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

{ Inicialización }
begin
   n := 0;
end
```

Respuesta (privada)

la solución puede estructurarse como un procedimiento más unas variables y semáforos compartidos:

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

```
procedimiento barrera2()
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
```