Sistemas Concurrentes y Distribuidos:
Problemas Resueltos.

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 2025-26

Universidad de Granada

Índice general

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

Dada la declaración de proceso secuencial (que forma parte de un programa concurrente) que aparece abajo, donde S es una sentencia que puede ser simple o compuesta, indica, razonando la respuesta, qué versiones de S son sentencias atómicas y cuáles no lo son. Para aquellas sentencias no atómicas, dar la traza detallada indicando aquel estado o estados que sean accesibles por otros procesos secuenciales.

```
{ Declaración variables globales }
var x : integer := 0;
    z : integer := 2;

{ Procesos }
process P;
   var i : integer := 0;
       j : integer := 1;
begin
   S;
end
```

(a) $S \equiv$ `z:=i+1;`

(b) $S \equiv$ `z:=2*x+j;`

(c) $S \equiv$ `i:=z+j;`

(d) $S \equiv$ `i:=i+j; x:=i+1;`

(e) $S \equiv$ `x:=x+j;`

(f) $S \equiv$ `j:=z+2; i:=i*5+j; j:=j+1;`

(g) $S \equiv$ `j:=i+2; i:=j+1; x:=j+i;`

(h) $S \equiv$ `x:=2*z+x;`

(i) $S \equiv$ `i:=2*i+j+3; j:=j+i*i;`

(j) $S \equiv$ `j:=z+2*x;`

Respuesta

Para las no atómicas, se dará la traza detallada abajo resaltando los estados intermedios accesibles.

Respecto a cuales son atómicas y cuales no, en estos casos ninguna instrucción tiene los ángulos < y >, por tanto, para decidir si son atómicas o no, basta con ver cuantos accesos hay a variables compartidas:

(a) $S \equiv$ `z:=i+1;`
Atómica, la sentencia únicamente realiza una acceso a variables compartidas (accede a la variable **z** para escribir). No hay estados intermedios accesibles a otros procesos.

(b) $S \equiv$ `z:=2*x+j;`
No atómica. La sentencia realiza dos accesos a variables compartidas. Primero lee **x** y luego escribe **z**. Entre ambos accesos hay estados intermedios accesibles a otros procesos.

(c) $S \equiv$ `i:=z+j;`
Atómica, la sentencia únicamente realiza una acceso a variables compartidas (accede a la variable **z** para leerla). No hay estados intermedios accesibles a otros procesos.

(d) $S \equiv$ `i:=i+j; x:=i+1;`
Atómica, unicamente realiza un acceso de escritura en las variables compartidas (escribe en **x** al final). No hay estados intermedios accesibles a otros procesos.

(e) $S \equiv$ `x:=x+j;`
No atómica. Realiza dos accesos a las variables compartidas, primero lee **x** y luego la escribe. Entre ambos accesos hay estados intermedios accesible a otros procesos.

(f) $S \equiv$ `j:=z+2; i:=i*5+j; j:=j+1;`
Atómica, unicamente realiza un acceso a las variables compartidas (lee de **z** al inicio). No hay estados intermedios accesibles a otros procesos.

(g) $S \equiv$ `j:=i+2; i:=j+1; x:=j+i;`
Atómica, unicamente realiza un acceso a las variables compartidas (escribe en **x** al final). No hay estados intermedios accesibles a otros procesos.

(h) $S \equiv$ `x:=2*z+x;`
No atómica. Realiza dos accesos a las variables compartidas, primero lee **x** y luego la escribe. Entre ambos accesos hay un estado intermedio accesible a otros procesos.

(i) $S \equiv$ `i:=2*i+j+3; j:=j+i*i;`
Atómica. No realiza accesos a variables compartidas. No hay estados intermedios accesibles a otros procesos.

(j) $S \equiv$ `j:=z+2*x;`
No atómica. Realiza dos accesos a las variables compartidas, primero lee **x** y luego lee **z**. Entre ambos accesos hay un estado intermedio accesible a otros procesos.

3

Dado el programa concurrente que se muestra abajo, responde a cada una de estas cuestiones:

(a) indica cómo se descomponen las sentencias de cada proceso en secuencias de sentencias que únicamente hacen operaciones aritméticas en los registros,

(b) indica como se descomponen las sentencias de cada proceso en un número mínimo de sentencias atómicas,

(c) proporciona todos los posibles valores que puede tomar la variable **x** al finalizar la ejecución del programa,

(d) proporciona una traza (en base a las sentencias del caso (a)) para cada valor diferente de **x** que se pueda obtener al finalizar la ejecución concurrente,

(e) indica cuantas interfoliaciones posibles hay de las sentencias del caso (a) y cuantas del caso (b).

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

```
process P0;
   var i0 : integer := 3 ;
begin
   x:= 2*i0+1;
end
```

```
process P1;
   var i1 : integer := 1 ;
begin
   x:= x+i1;
end
```

Respuesta

Apartado (a): Indica cómo se descomponen las sentencias de cada proceso en secuencias de sentencias que únicamente hacen operaciones aritméticas en los registros.

Para descomponer las sentencias, tal y como se pide en este apartado, suponemos que el registro **r0** estará asignado a **P0** mientras que los registros **r10** y **r11** serán asignados a **P1**.

La descomposición en sentencias (usando registros para las operaciones aritméticas) de **x**:=2*i0+1 es:

```
r0 := i0 ;
x  := 2*r0+1 ;
```

La descomposición de **x**:=**x**+**i1** es:

```
r10 := x ;
r11 := i1 ;
x   := r10+r11 ;
```

Apartado (b): Indica como se descomponen las sentencias de cada proceso en un número mínimo de sentencias atómicas.

La descomposición en un número mínimo de sentencias atómicas de la sentencia de **x**:=2*i0+1 es la propia sentencia, ya que esta sentencia es atómica de partida (únicamente hace un acceso a variables compartidas, la escritura en **x**):

```
x:= 2*i0+1;
```

La descomposición en un número mínimo de sentencias atómicas de la sentencia de **x**:=**x**+**i1** podría en principio ser la descomposición en tres sentencias que se ha dado en el apartado (a), pero en este caso, la segunda y tercera sentencias, ejecutadas en secuencia, forman una única sentencia atómica, ya que entre ambas únicamente producen un único acceso a **x**. Por tanto, la descomposición que se nos pide de **x**:=**x**+**i1** es:

```
r10 := x ;
x   := r10+i1 ;
```

Apartado (c): Proporciona todos los posibles valores que puede tomar la variable **x** al finalizar la ejecución del programa.

La variable x puede tener al final uno de estos valores: **8**, **7** y **1**.

Apartado (d): Proporciona una traza (en base a las sentencias del caso (a)) para cada valor diferente de **x** que se pueda obtener al finalizar la ejecución concurrente.

Una traza que da lugar al valor 8 es cualquiera en la que la escritura de x en P0 se haga antes de la lectura de x en P1. Un ejemplo es esta que aparece aquí, en ella P0 acaba antes de que empiece P1:

P0	P1	x	i0	i1	r0	r10	r11
		0	3	1	-	-	-
r0 := i0		0	3	1	**3**	-	-
x := 2*r0+1		**7**	3	1	3	-	-
	r10:= x	7	3	1	3	**7**	-
	r11:= i1	7	3	1	3	7	**1**
	x := r10+r11	**8**	3	1	3	7	1

Una traza que da lugar al valor 7 es cualquiera en la que el proceso P0 escriba en x después de que lo haga P1. Un ejemplo es esta de aquí (en ella P1 acaba antes de que empiece P0):

P0	P1	x	i0	i1	r0	r10	r11
		0	3	1	-	-	-
	r10:= x	0	3	1	-	**0**	-
	r11:= i1	0	3	1	-	0	**1**
	x := r10+r11	**1**	3	1	-	0	1
r0 := i0		1	3	1	**3**	0	1
x := 2*r0+1		**7**	3	1	3	0	1

Una traza que da lugar al valor 1 es cualquiera en la que la escritura de x en P0 ocurra entre los dos accesos a x que hace P1, luego la ejecución de las dos sentencias se solapa. Por ejemplo, sería esta:

P0	P1	x	i0	i1	r0	r10	r11
		0	3	1	-	-	-
	r10:= x	0	3	1	-	**0**	-
r0 := i0		0	3	1	**3**	0	-
x := 2*r0+1		**7**	3	1	3	0	-
	r11:= i1	7	3	1	3	0	**1**
	x := r10+r11	**1**	3	1	3	0	1

Apartado (d): Indica cuantas interfoliaciones posibles hay de las sentencias del caso (a) y cuantas del caso (b).

En el caso (a) una sentencia se descompone en otras 2, y la otra se descompone en otras 3, por tanto, el número interfoliaciones posibles es el dado por el coeficiente binomial, particularizando para $n_1 = 2$ y $n_2 = 3$, es decir, el número de interfoliaciones es

$$\frac{(n_1+n_2)!}{n_1!n_2!} = \frac{5!}{2!\cdot 3!} = \frac{5\cdot 4\cdot 3\cdot 2}{2\cdot(3\cdot 2)} = 10$$

En el caso (b) una sentencia no se descompone, y la otra se descompone en otras 2, por tanto ahora $n_1 = 1$ y $n_2 = 2$, es decir, ese número es:

$$\frac{(n_1+n_2)!}{n_1!n_2!} = \frac{3!}{2!\cdot 1!} = \frac{3\cdot 2}{2} = 3$$

Como vemos, si consideramos un número mínimo de sentencias atómicas, el numero de interfoliaciones se reduce bastante.

4

Consideramos un programa concurrente en el cual varios procesos comparten dos variables enteras **x** e **y**, y queremos que siempre se cumpla **x**+**y** == 10.

Apartado (a)

Dado un valor n cualquiera (puede ser una constante o una variable local), para asignarle n a la variable **x** (y actualizar adecuadamente **y**) todos los procesos usan **alguna** de estas tres sentencias alternativas (todos la misma):

```
{ Opción 1 }
begin
   x:= n;
   y:= 10-n;
end
```

```
{ Opción 2 }
begin
   <x:= n>;
   <y:= 10-n>;
end
```

```
{ Opción 3 }
begin
   < x:= n;
     y:= 10-n; >
end
```

Para cada opción: describe razonadamente si la opción es correcta. Se considera correcta una opción si, al ejecutarse la sentencia bloque en un estado previo en el cual se cumple **x**+**y**==10, se garantiza que siempre en el estado inmediatamente posterior se cumplirá **x**+**y**==10.

Apartado (b)

Para verificar que la suma de ambas variables siempre es 10, escribimos código que imprime dicha suma. Los procesos podrán ejecutar ese código en cualquier momento, de forma concurrente con los procesos que actualizan **x** e **y**. Para ello, todos los procesos usarán una variable local propia (**a**) y **alguna** de las siguientes sentencias alternativas (todos la misma):

```
{ Opción 1 }
begin
   a:= x+y;
   print(a);
end
```

```
{ Opción 2 }
begin
   < a:= x+y >;
   print(a);
end
```

```
{ Opción 3 }
begin
   < a:= x+y;
     print(a); >
end
```

Para cada opción: describe razonadamente si la opción es correcta. Se considera correcta una opción si, al ejecutarse la sentencia bloque en un estado previo con `x+y==10`, se garantiza que siempre se imprimirá el valor 10.

Además, responde a esta pregunta: ¿ es atómica la sentencia **`begin-end`** de la opción 2 ?

Respuesta

Apartado (a)

Opción 1

Es **incorrecta** ya que se hacen dos escrituras separadas en **x** y en **y**, la sentencia bloque no es atómica. Supongamos que un proceso **P** escribe 6 y 4 en **x** e **y** respectivamente, a la vez que otro proceso **Q** quiere escribir 7 y 3. Si las dos escrituras de **Q** ocurren ambas entre las dos escrituras de **P**, entonces al final los valores de **x** e **y** serán 7 y 4, así que la suma no sería 1.

Opción 2

Es **incorrecta** por el mismo motivo que la anterior, ya que hemos hecho atómicas las sentencias de escritura, pero esas sentencias ya eran atómicas antes, no hemos cambiado nada.

Opción 3

Es **correcta**, ya que hemos puesto los ángulos con las dos asignaciones dentro, por tanto hemos convertido en atómica la sentencia, y ningún proceso puede leer ni escribir ni **x** ni **y** cuando un proceso está ejecutando este **`begin-end`**.

Apartado (b)

Opción 1

Es **incorrecta** ya que se hacen dos lecturas separadas de **x** y de **y**, la sentencia bloque no es atómica. Supongamos que un proceso **P** ejecuta el **`begin-end`** en un estado en el cual los valores de **x** e **y** son 6 y 4, respectivamente, a la vez que otro proceso **Q** escribe 7 y 3. Si las dos escrituras de **Q** ocurren ambas entre las dos lecturas de **P**, entonces **P** leerá 6 y 3, y el valor impreso no sería 10.

Opción 2

Es **correcta**, ya que si un proceso **P** la ejecuta, hará las dos lecturas de **x** y de **y** sin que haya posibilidad de que ningún otro proceso acceda a las variables **x** e **y** entre las dos lecturas.

Se pregunta si la sentencia bloque **`begin-end`** es atómica, y la respuesta es que sí lo es. Aunque el **`print`** esté fuera de los ángulos, ese **`print`** únicamente accede a la variable local **a**. Así que el estado intermedio entre la lectura de **x** y la de **y** no es accesible por otros procesos.

Opción 3

Es **correcta**, ya que es una instrucción atómica por definición, y de hecho es equivalente a la anterior, ya que incluir el **print** dentro de los ángulos no cambia nada respecto a la atomicidad.

5

¿ 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.

Para ilustrar como se accede a los archivos, aquí se encuentra una versión secuencial del código que copia **f** sobre **g**:

```
process CopiaSecuencial ;
   var v  : T ;
begin
   v := leer(f) ;     { lectura adelantada }
   while not fin(f) do
   begin
      escribir(g,v); { leer de la variable v y escribir en el archivo g }
      v := leer(f);   { leer del archivo f y escribir variable v }
   end
end
```

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.

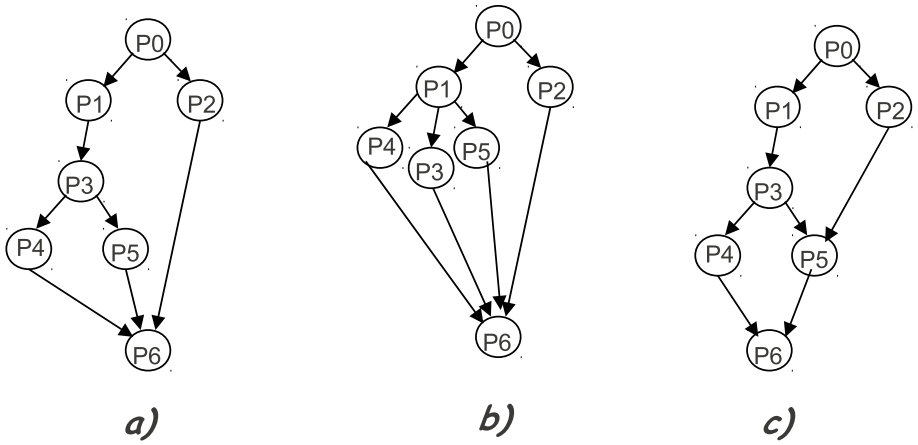
La solución más obvia sería usar una variable **v** (compartida entre la lectura y la escritura) para esto, es decir, usar en cada íteración la solución que aparece en la figura de la izquierda. El problema es que en esta solución la variable **v** puede ser accedida simultáneamente por la escritura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

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

```
process CopiaConcurrente ;
   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
```

6

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
```

7

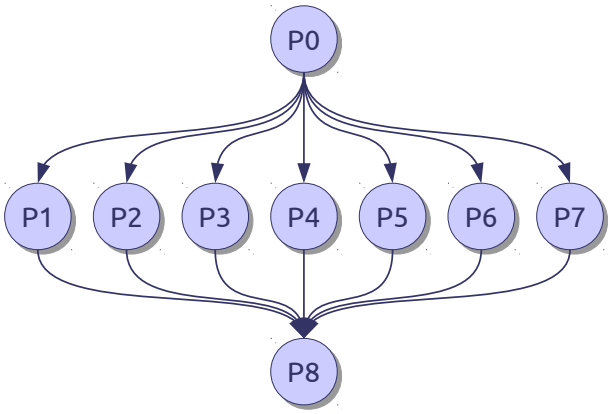
Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

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

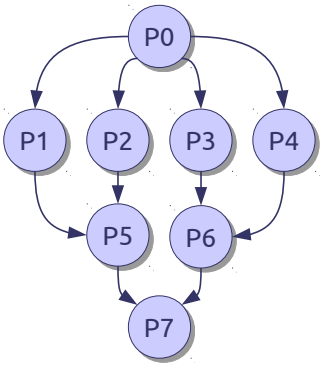
```
begin
   P0 ;
   cobegin
      begin
        cobegin
          P1;P2;
        coend
       P5;
      end
      begin
        cobegin
          P3;P4;
        coend
       P6;
      end
   coend
   P7 ;
end
```

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)

8

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 dispone de 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. El proceso acumulador puede invocar un procedimiento **Espera_impulso** para esperar a que llegue un impulso, y el proceso escritor puede llamar a **Espera_fin_hora** para esperar a que termine una hora.

El código de los procesos de este programa 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
```

En el programa 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.

Supongamos que en un instante dado el acumulador está esperando un impulso, el escritor está esperando el fin de una hora, y la variable **n** vale k. Después se produce de forma simultánea un nuevo impulso y el fin del periódo de una hora. Obtener las posibles secuencias de interfolicación de las instrucciones (1),(2), y (3) a partir de dicho instante, e indicar cuales de ellas son correctas y cuales incorrectas (las incorrectas son aquellas en las cuales el impulso no se contabiliza).

Respuesta

Supongamos que hay una variable entera (ficticia) llamada **OUT**, que se crea al terminar el **write** (sentencia (2)) y tiene el valor impreso (esto permite incluir en el estado del programa dicho valor impreso).

En el estado de partida, se cumple **n**==k, y a partir de ahí pueden ocurrir tres interfoliaciones posibles de las sentencias etiquetadas con los dígitos 1,2, y 3. Estas interfoliaciones son: **(a)** 1,2,3, **(b)** 2,1,3 y **(c)** 2,3,1.

Para cada interfoliación podemos considerar los valores de las variables en cada estado al final de cada sentencia, y podemos examinar el estado final, esto es, el valor con el que queda **n** y el valor impreso (el valor de **OUT**).

(a)

Instr.	n	OUT
	k	
n:= n+1	$k+1$	
write(n)	$k+1$	$k+1$
n:= 0	0	$k+1$

(b)

Instr.	n	OUT
	k	
write(n)	k	k
n:= n+1	$k+1$	k
n:= 0	0	k

(c)

Instr.	n	OUT
	k	
write(n)	k	k
n:= 0	0	k
n:= n+1	1	k

Son correctas únicamente las interfoliaciones en las cuales en el estado final se cumple:

$$\texttt{OUT} + \texttt{n} \;==\; k+1$$

es decir, el valor impreso más el valor de contador es igual al número total de impulsos producidos desde que comenzó la hora que acaba. Evidentemente, las interfoliaciones **(a)** y **(c)** son **correctas**, mientras que la **(b)** es **incorrecta**.

9

Supongamos un programa concurrente en el cual hay, en memoria compartida dos vectores `a` y `b` de enteros y con tamaño par, declarados como sigue:

```
var a,b : array[0..2*n-1] of integer ; { n es una constante predefinida (>2) }
```

Queremos escribir un programa para obtener en `b` una copia ordenada del contenido de `a` (nos da igual el estado en que queda `a` después de obtener `b`).

Para ello disponemos de la función `Sort` que ordena un tramo de `a` (entre las entradas `s`, incluida, y `t`, no incluida), usando el método de la burbuja. También disponemos la función `Copiar`, que copia un tramo de `a` (desde `s`, incluido, hasta `t`, sin incluir) sobre `b` (a partir de `o`).

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

```
procedure Copiar( o,s,t : integer );
   var d : integer ;
begin
   for d := 0 to t-s-1 do
      b[o+d] := a[s+d] ;
end
```

La función `swap` intercambia dos variables. El programa para ordenar se puede implementar de dos formas:

- Ordenar todo el vector `a`, de forma secuencial con la función `Sort`, y después copiar cada entrada de `a` en `b`, con la función `Copiar`.
- 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 vemos el código de ambas versiones:

```
procedure Secuencial() ;
   var i : integer ;
begin
   Sort( 0, 2*n );       { ordena a   }
   Copiar( 0, 0, 2*n ); { copia a en b }
end
```

```
procedure Concurrente() ;
begin
  cobegin
     Sort( 0, n   );
     Sort( n, 2*n );
  coend
  Merge( 0, n, 2*n );
end
```

El código de **Merge** se encarga de ir leyendo las dos mitades de **a**. En cada paso primero se selecciona el menor elemento de los dos siguientes por leer (uno en cada mitad), y después se escribe dicho menor elemento en la siguiente mitad del vector mezclado **b**. Al acabar este bucle, será necesario copiar el resto de elementos no leídos de una de las dos mitades. El código es el siguiente:

```
procedure Merge( inferior, medio, superior: integer ) ;
   var escribir : integer := 0 ;          { siguiente posicion a escribir en b      }
   var leer1    : integer := inferior ; { siguiente pos. a leer en primera mitad de a }
   var leer2    : integer := medio    ; { siguiente pos. a leer en segunda mitad de a }
begin
   { mientras no haya terminado con alguna mitad }
   while leer1 < medio and leer2 < superior do begin
      if a[leer1] < a[leer2] then begin { minimo en la primera mitad }
         b[escribir] := a[leer1] ;
         leer1 := leer1 + 1 ;
      end else begin { minimo en la segunda mitad }
         b[escribir] := a[leer2] ;
         leer2 := leer2 + 1 ;
      end
      escribir := escribir+1 ;
   end
   { se ha terminado de copiar una de las mitades, copiar lo que quede de la otra }
   if leer2 >= superior then Copiar( escribir, leer1, medio    ); { copiar primera }
                        else Copiar( escribir, leer2, superior ); { copiar segunda }
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 el tiempo de **Sort** ($T_s(2n)$) más el tiempo de **Copiar** (que es $2n$, pues copiar un elemento tarda una unidad de tiempo), luego

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

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

(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 $P_1 = n^2 + n$ con $S = 2n^2 + n$, 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 vemos 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.

10

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

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
```

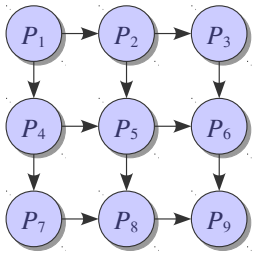
11

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):

Trozo de 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.

Respuesta

Una posible solución consiste en usar un vector de valores lógicos que indican si cada proceso ha terminado o no. Hay que tener en cuenta que, puesto que la ejecución concurrente de todas las rutinas está en un bucle, dicho vector debe reinicializarse entre una iteración del bucle y la siguiente. Para ello realizamos dicha reinicialización cuando el proceso 9 (el último) señale que ha acabado (en `Acabar`). La implementación queda como sigue:

```
{ compartido entre todas las tareas }
var finalizado : array [1..9] of boolean := (false,false,..., false) ;
```

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

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

12

En el problema anterior los procesos **P1**, **P2**, ..., **P9** se ponen en marcha usando **cobegin/coend**. Escribe un programa equivalente, que ponga en marcha todos los procesos, pero que use declaración estática de procesos, usando un vector de procesos **P**, con índices desde 1 hasta 9, ambos incluidos. El proceso **P[n]** contiene un bucle infinito, y en cada iteración se ejecuta P_n. Ejecutar P_n supone incluir las llamadas necesarias a **EsperarPor** (con la misma implementación que antes), luego una secuencia de instrucciones **Sn**, y finalmente las llamada a **Acabar**. Se incluye aquí un plantilla:

```
Process P[ n : 1..9 ]

begin
   while true do begin
      .....   { esperar (si es necesario) a los procesos que corresponda }
      Sn ;    { sentencias específicas de este proceso (desconocidas) }
      .....   { señalar que hemos terminado }
   end
end
```

Respuesta

Esta es una posible implementación, en la cual se busca la máxima simplicidad (menor número de instrucciones posibles).

El proceso de índice **n** tiene *arriba* al proceso **n**-3 (*arriba* según se ha escrito el pseudo-código en el enunciado del problema anterior), pero esto solo ocurre si el proceso no está en la primera fila, es decir, solo si **n** es mayor estricto que 3). Además, el proceso **n** tiene a su izquierda al proceso de índice **n**-1 (pero solo si **n** no está en la primera columna, es decir, solo si el valor **n**-1 no es múltiplo de 3, equivalente a que al reducir **n**-1 módulo 3, de un valor mayor que 0). Por tanto, el código es como sigue:

```
process P[ n : 1..9 ]
begin
   while true do begin
      { comprobamos qué procesos debemos esperar en función de 'n' }
      if n > 3          then EsperarPor( n-3 ); { esperamos proceso de arriba   }
      if n-1 mod 3 > 0  then EsperarPor( n-1 ); { esperamos proceso de la izqda. }

      Sn ;            { sentencias específicas de este proceso (desconocidas) }

      Acabar( n ); { señalar que hemos terminado }
   end
end
```

Sincronización en memoria compartida.

2.1. Soluciones software para exclusión mutua

13

¿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 b0 : boolean := false ,   { true si P0 quiere acceder o está en SC }
    b1 : boolean := false ;   { true si P1 quiere acceder o está en SC }
```

```
1  Process P0 ;
2  begin
3     while true do begin
4        { protocolo de entrada: }
5        b0 := true ;          {indica quiere entrar}
6        while b1 do begin     {si el otro también: }
7           b0 := false ;      {cede temporalmente
8           while b1 do begin end {espera     }
9           b0 := true ;       {vuelve a cerrar paso}
10       end
11       { seccion critica  ....  }
12       { protocolo de salida }
13       b0 := false ;
14       { resto sentencias  ....  }
15    end
16 end
```

```
1  process P1 ;
2  begin
3     while true do begin
4        { protocolo de entrada: }
5        b1 := true ;          {indica quiere entrar}
6        while b0 do begin     {si el otro también: }
7           b1 := false ;      {cede temporalmente }
8           while b0 do begin end {espera     }
9           b1 := true ;       {vuelve a cerrar paso}
10       end
11       { seccion critica  ....  }
12       { protocolo de salida }
13       b1 := false ;
14       { resto sentencias  ....  }
15    end
16 end
```

Respuesta

(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 `false` (en la línea 5). Sin pérdida de generalidad, asumiremos que el proceso `P0` realizó esa lectura antes que el `P1` (en caso contrario

se intercambian los papeles de los procesos, ya que son simétricos). Es decir, el proceso **P0** tuvo que leer **false** en **b1**, en un instante que llamaremos s, con $s < t$. A partir de s, la variable **b0** contiene el valor **true**, pues el proceso **P0** es el único que la escribe y entre s y t dicho proceso está en SC y no la modifica.

En s el proceso **P1** no podía estar en RS, ya que entonces no podría haber entrado a SC entre s y t (ya que **b0** es **true** siempre después s), luego concluimos que en s el proceso **P1** estaba en el PE. Más en concreto, el proceso **P1** estaba (en el instante s) forzosamente en el bucle de la línea 6, ya que en otro caso **b1** sería **true** 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 **b0** es **true**, entonces el proceso **P1** 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.

14

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

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.

15

Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (señalan el turno

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

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 interbloqueo (progreso), la ausencia de inanción (espera limitada) y la exclusión mutua.

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

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

```
process P1 ;
begin
 while true do begin
   n1 := 1 ;                          { E1.1 }
   n1 := n2+1 ;                       { L1.1 ; E2.1 }
   while n2 != 0 and                  { L2.1 }
         n2 < n1 do begin end;        { L3.1 }
   { seccion critica }                { SC.1 }
   n1 := 0 ;                          { E3.1 }
   { resto sentencias }               { RS.1 }
 end
end
```

```
process P2 ;
begin
 while true do begin
   n2 := 1 ;                          { E1.2 }
   n2 := n1+1 ;                       { L1.2 ; E2.2 }
   while n1 != 0  and                 { L2.2 }
         n1 <= n2 do begin end;       { L3.2 }
   { seccion critica }                { SC.2 }
   n2 := 0 ;                          { E3.2 }
   { resto sentencias }               { RS.2 }
 end
end
```

Respuesta

Apartado (a)

Demostraremos la ausencia de interbloqueo (progreso), la ausencia de inanición (espera limitada) y la exclusión mutua.

(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, haciendo las lecturas en L2 y L3 continuamente. 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  and  n2!=0  and  n2<n1  and  n1<=n2
```

tanto se cumple `n2<n1 and n1<=n2`, lo cual es imposible.

(a.2) ausencia de inanición

Supongamos que el proceso `P1` está en espera ocupada (en el bucle del PE) durante un intervalo T y comprobemos cuantas veces puede entrar `P2` a SC durante T (al razonamiento al contrario es similar).

En el intervalo T se cumple `n1>0`. El proceso `P2` puede entrar a SC una vez. Si `P2` intenta entrar a SC una segunda vez, durante T, antes de hacerlo tiene que ejecutar `n2:=n1+1` lo que forzosamente hace cierta la

condición `n1<n2`, y como se sigue cumpliendo `n1!=0`, vemos que el proceso `P2` no puede entrar de nuevo a SC. De hecho, en cuanto ejecuta `n2:=n1+1`, da paso a `P1` a SC, y luego queda a la espera.

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

(a.3) exclusión mutua

Para demostrar la EM, lo haremos por reducción al absurdo. Supongamos que en un instante t los dos procesos están en la sección crítica. Debe haber habido un instante previo s en el que se ejecutó la última escritura atómica en `n1` o `n2` (la escritura en E2.1 o en E2.2). En ese instante ambas variables tienen un valor distinto de 0. En el intervalo de tiempo I entre s y t, ningún proceso ha cambiado el valor de las variables compartidas.

Llamaremos A al proceso que realiza la escritura E2 primero, y B al proceso que la realiza después (justo en el instante s). Llamaremos `nA` a la variable `n1` (si A es `P1`) o a la variable `n2` (si A es `P2`). Igualmente haremos con `nB`, en función de que proceso sea B.

Vemos que si en s el proceso B escribe en `nB` un valor estictamente mayor que el que tiene en ese momento `nA`, entonces el proceso B no ha podido entrar después de s a la sección crítica, ya que cualquiera de los dos procesos queda en su bucle del PE cuando el valor de su variable es mayor que la del otro. Por tanto, deducimos que en s se cumple `nB<=nA`, ya que de otra forma llegaríamos a una contradicción con la hipótesis de partida (ambos procesos en SC). Como además en s ambos valores son distintos de cero, concluimos que `0<nB<=nA`.

Analizamos entonces las posibles combinaciones de valores escritos en `n1` y `n2` en las sentencias E2. Hay tres posibilidades, que se detallan aquí:

(a) El proceso A ha escrito un 2, y el B ha escrito un 3.

Ocurre cuando las dos variables se ponen a 1 (en E1, al inicio del PE), antes de las dos lecturas en L1, y después A a lee ese 1 y escribe un 2. A continuación B lee un 2 y escribe un 3. La traza de las operaciones de escritura es esta:

Proc.	Sent.	Lecturas o Escrituras	Valor `nA`	Valor `nB`	Estado resultante
			0	0	(1)
A/B	E1	`nA:=1` y `nB:=1`	1	1	(2)
A	L1	lee `1` en `nB`	1	1	(2)
A	E2	`nA :=2`	2	1	(3)
B	L1	lee 2 en `nA`	2	1	(3)
B	E2	`nB :=3`	2	3	(4)

Pero este caso (a) no ha podido ocurrir, ya que vemos que en el estado (4) no se cumple `nB<=nA`, y por tanto en ese estado B no puede entrar a SC.

(b) El proceso A ha escrito un 2, y el B ha escrito un 1.

Un primer proceso (en L1) ha leído un 0, estando el otro fuera del PE o SC. Al leer, ese primer proceso tiene su variable puesta a 1. Después el otro proceso ha accedido al PE y ha leido necesariamente un 1 (en L1).

Después, el orden de las escrituras en E2 puede ser arbitrario en principio. Sin embargo, vemos que el valor 1 no se ha podido escribir antes del valor 2, en ese caso el proceso B sería el que escribe el 2 (el mayor valor) y tras el instante s no se cumpliría `nB<=nA` (sabemos que no puede ser).

Asi que deducimos que A escribe un 2 primero, y después B escribe un 1, y ha ocurrido necesariamente esta traza de las operaciones de escritura:

Proc	Sent.	Lecturas o Escrituras	Valor `nA`	Valor `nB`	Estado resultante
			0	0	(1)
B	E1	`nB := 1`	0	1	(2)
B	L1	lee `0` en `nA`	0	1	(2)
A	E1	`nA := 1`	1	1	(3)
A	L1	lee `1` en `nB`	1	1	(3)
A	E2	`nA :=2`	2	1	(4)
B	E2	`nB :=1`	2	1	(5)

El proceso B ha entrado a SC en el estado (5), después de E2.B. El proceso A debe haber entrado a SC después de su escritura en E2.A, bien en el estado (4) o bien en el (5). Pero esto es imposible, ya que el estado (4) realmente es el mismo que el (5), y en ambos se cumple `0<nB<nA` lo cual deja con seguridad al proceso A en PE. Por tanto, esta opción (b) queda descartada, no ha podido ocurrir tampoco.

(c) Los procesos A y B escriben ambos un 2.

Ocurre cuando ambos procesos entran al PE de forma *sincrona*, de forma que ambos escriben un 1, después ambos leen un 1, y finalmente ambos escriben un 2.

El orden de las escrituras en E2 puede ser cualquiera en principio. Si P2 escribe después, al hacerlo hace cierto `0<n1==n2`. Pero en este estado, `P2` no puede entrar a SC, luego concluimos que necesariamente `P2` escribe antes y luego `P1`. La traza es esta:

Proc.	Sent.	Lecturas o Escrituras	Valor `n1`	Valor `n2`	Estado resultante
			0	0	(1)
P1/P2	E1	`n1:=1` y `n2:=1`	1	1	(2)
P1/P2	L1	leen `1` en `n1` y `n2`	1	1	(2)
P2	E2	`n2 := 2`	1	2	(3)
P1	E2	`n1 := 2`	2	2	(4)

El proceso `P1` puede entra en SC en el estado (4). Sin embargo, el proceso `P2` no puede entrar a SC en el estado (3) ni en el (4), ya que en ambos se cumple `0<n1<=n2`.

Luego este caso (c) tampoco ha podido ocurrir.

No hay ninguna posible combinación más de valores escritos en las variables `n1` y `n2`. Asi que hemos visto que no puede haber ninguna interfoliación que lleve a un estado en el cual ambos procesos están en SC. Luego se cumple EM.

Apartado (b)

Hay que demostrar que las asignaciones iniciales del PE son necesarias. Para resolver esto veremos que sin las asignaciones iniciales del PE, no se cumple exclusión mutua, encontrando una interfoliacion que deja ambos procesos en SC.

Supongamos que no están las asignaciones `n1:=1` ni `n2:=1`. No existe n E1.1 ni E1.2, el resto igual. 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. P1 lee un `0` en `n2` (en L1.1)
2. P2 lee un `0` en `n1` (en L1.2)
3. P2 escribe un `1` en `n2` (en E2.2)
4. P2 lee `0` en `n1` (en L2.2)
5. P2 ve que la condición `n1!=0` no se cumple y avanza hasta SC
6. P1 escribe `1` en `n1` (en E2.1) (en este momento, ambas variables están a `1`).
7. P1 hace las lecturas en L3.1, lee un 1 en ambas variables.
8. P1 que la condición `n2<n1` no se cumple, y avanza a la SC

16

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 ;
```

```
process P0 ;
begin
   while true do begin
      repeat
         c0 := 1-c1 ;
      until c1 != 0 ;
      { seccion critica }
      c0 := 1 ;
      { resto sentencias }
   end
end
```

```
process P1 ;
begin
   while true do begin
      repeat
         c1 := 1-c0 ;
      until c0 != 0 ;
      { seccion critica }
      c1 := 1 ;
      { resto sentencias }
   end
end
```

Respuesta

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.

2.2. Soluciones hardware para exclusión mutua

17

Diseñar una solución hardware basada en espera ocupada para el problema de la exclusión mutua utilizando la instrucción **`swap(x,y)`** (en lugar de usar **`LeerAsignar`**) cuyo efecto es intercambiar **de forma atómica** los dos valores lógicos almacenados en las posiciones de memoria `x` e `y`.

Respuesta

La forma de usar `swap` es como se indica aquí:

```
{ variables compartidas y valores iniciales }
var sc_libre : boolean := true ; { verdadero solo si la SC esta libre }

{ procesos }
process P[ i : 1 .. n ];
var { variable no compartida: true solo si este proceso ocupa la SC }
   sc_ocupada_proc : boolean := falso ;
begin
   while true do begin
      repeat
            swap( sc_libre, sc_ocupada_proc ) ;
      until sc_ocupada_proc ;

      { seccion critica }

      swap( sc_libre, sc_ocupada_proc ) ;
      { resto seccion }
   end
end
```

18

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

Usaremos una variable 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
```

2.3. Semáforos

19

El cuadro que sigue nos muestra dos procesos, uno productor **P1** y otro consumidor **P2** que se comunican a través de una variable compartida x, según el patrón sencillo de espera única. Las sentencias **a**:=**Producir**() y **Consumir**(**b**) son atómicas pues solo usan variables locales.

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

```
process P1;
   var a : integer ;
begin
   a := Producir();
   x := a ;
   sem_signal(s) ;
end
```

```
process P2 ;
   var b : integer ;
begin
   sem_wait(s) ;
   b := x ;
   Consumir(b) ;
end
```

Suponemos que `Producir` produce el entero 1 al ser invocado. Queremos analizar la traza del programa (interfoliación de sentencias atómicas) concurrente en dos casos:

(a) `P1` inicia `sem_signal` antes de que `P2` llegue a `sem_wait`, y

(b) `P2` inicia `sem_wait` antes de que `P1` llegue a `sem_signal`.

Escribe una traza para cada caso, completando la plantilla que se da en este enunciado. En las trazas debemos incluir una columna para el valor de `s` y otra para los procesos esperando en en el semáforo (además de las columnas con las sentencias de cada proceso y la columna de la variable compartida `x`).

Hay que tener en cuenta que `sem_signal` es siempre atómico, pero `sem_wait` lo es únicamente cuando el valor del semáforo es mayor que cero. Si un semáforo vale 0, entonces la llamada a `sem_wait` se descompone en dos sentencias atómicas: `ini sem_wait` y `fin sem_wait`, y además está última en realidad es simultánea a un `sem_signal` de otro proceso **en la misma fila** (ambas cosas forman una transición atómica de estado).

Plantilla:

P1	P2	x	V.s	Pr.s

Respuesta

Aquí vemos dos trazas posibles del programa concurrente. En el caso (b) hay varias opciones posibles, ya que las dos primeras sentencias de `P1` pueden hacerse antes o después del `sem_wait` de `P2`. Aquí hemos puesto una de las opciones posibles, en la que esas dos sentencias van después del `sem_wait`.

Caso (a):

P1	P2	x	V.s	P.s
		0	0	
a:= Producir()		0	0	
x:= a		1	0	
sem_signal(s)		1	1	
	sem_wait(s)	1	0	
	b:= x	1	0	
	Consumir(b)	1	0	

Caso (b):

P1	P2	x	V.s	P.s
		0	0	
	ini sem_wait(s)	0	0	P2
a:= Producir()		0	0	P2
x:= a		1	0	P2
sem_signal(s)	fin sem_wait(s)	1	0	
	b:= x	1	0	
	Consumir(b)	1	0	

20

Considera el siguiente programa concurrente:

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

```
process P1;
begin
   < x := x+1 > ;
   sem_signal(s) ;
   < print(x) >
end
```

```
process P2 ;
begin
   sem_wait(s) ;
   < x := x*2 > ;
end
```

Responde a estas cuestiones:

(a) ¿ Qué valores finales puede tener x al acabar el programa ?

(b) ¿ Qué valores puede imprimir el proceso P1 ?

(c) Para cada valor que pueda imprimir el proceso P1 escribe dos trazas (completas, es decir, hasta que acaba el programa) en las que se imprime dicho valor: una traza en la que P2 no se bloquea en el semáforo en ningún momento, y otra en la que sí se bloquea durante un intervalo de tiempo. Escribe las trazas en tablas con las mismas columnas que aparecen en la plantilla del enunciado del problema anterior, e incluyendo asimismo las sentencias atómicas derivadas de las llamadas a **sem_wait** y **sem_signal**.

Respuesta

(a) El único valor final de x posible es 2.

(b) Los valores posibles que puede imprimir P1 son 1 y 2.

(c) Aquí están las trazas:

Imprime 1, sin bloqueo.

P1	P2	x	V.s	P.s
		0	0	
<x:= x+1>		1	0	
sem_signal(s)		1	0	
<print(x)>		1	0	
	sem_wait(s)	1	0	
	<x:= x*2>	2	0	

Imprime 1, con bloqueo.

P1	P2	x	V.s	P.s
		0	0	
	ini sem_wait(s)	0	0	P2
<x:= x+1>		1	0	P2
sem_signal(s)	fin sem_wait(s)	1	0	
<print(x)>		1	0	
	<x:= x*2>	2	0	

Imprime 2, sin bloqueo.

P1	P2	x	V.s	P.s
		0	0	
<x:= x+1>		1	0	
sem_signal(s)		1	0	
	sem_wait(s)	1	0	
	<x:= x*2>	2	0	
<print(x)>		2	0	

Imprime 2, con bloqueo.

P1	P2	x	V.s	P.s
		0	0	
	ini sem_wait(s)	1	0	P2
<x:= x+1>		1	0	P2
sem_signal(s)	fin sem_wait(s)	1	0	
	<x:= x*2>	2	0	
<print(x)>		2	0	

21

Considera el siguiente programa concurrente de tres procesos, donde algunas sentencias atómicas se han etiquetado

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

```
process PA;
begin
   {S1} sem_signal(s) ;
   {A } < x := x+1 > ;
   {S2} sem_signal(s) ;
end
```

```
process PB ;
begin
   sem_wait(s) ;
   {B} < x := x*2 > ;
end
```

```
process PC ;
begin
   sem_wait(s) ;
   {C} < x := x*3 > ;
end
```

Para cada una de las secuencias de interfoliación siguientes, indica si dicha secuencia puede ocurrir o no puede ocurrir, en caso afirmativo, indica el valor final de **x** al acabar el programa, en caso negativo, explica muy brevemente el motivo.

Interfoliación	S/N	V.x	Motivo del no.
S1 - A - S2 - B - B			
A - S1 - S2 - B - C			
B - S1 - A - S2 - C			
C - S1 - B - A - S2			
S1 - A - B - C - S2			
S1 - C - A - S2 - B			
S1 - B - A - S2 - C			
S1 - A - S2 - B - C			
S1 - A - C - S2 - B			

Respuesta

Interfoliación	S/N	V.x	Motivo del no.
S1 - A - S2 - B - B	N		B nunca se ejecuta dos veces, y C siempre 1.
A - S1 - S2 - B - C	N		No respeta el orden secuencial de PA
B - S1 - A - S2 - C	N		B no puede ir antes de S1
C - S1 - B - A - S2	N		C no puede ir antes de S1
S1 - A - B - C - S2	N		B y C no pueden ir ambas antes de S2
S1 - C - A - S2 - B	S	2	
S1 - B - A - S2 - C	S	3	
S1 - A - S2 - B - C	S	6	
S1 - A - C - S2 - B	S	6	

22

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_
      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
```

23

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

(a) P_2 podrá ejecutar e una vez por cada vez que P_1 haya ejecutado a o P_3 haya ejecutado g.

(b) P_2 podrá ejecutar e una vez por cada vez que los procesos P_1 y P_3 hayan ejecutado una vez el par de sentencias a y g.

(c) Por cada vez que P_1 haya ejecutado b, podrá P_2 ejecutar una vez e y podrá P_3 ejecutar una vez 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 ejecutar e una vez por cada vez que P_1 haya ejecutado a o P_3 haya 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 ejecutar e una vez por cada vez que los procesos P_1 y P_3 hayan ejecutado una vez el par de sentencias a y 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) Por cada vez que P_1 haya ejecutado b, podrá P_2 ejecutar una vez e y podrá P_3 ejecutar una vez 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
```

24

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). Usando semáforos, queremos resolver los dos problemas de sincronización que se indican aquí:

(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 }
var x : integer ;
```

```
process P1 ;
   var m : integer ;
begin
   while true do begin
      m := 2*x-1 ;
      print( m );
   end
end
```

```
process P2
   var d : integer ;
begin
   while true do begin
      d := leer_teclado();
      x := 7*d+6 ;
   end
end
```

Respuesta

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

```
{ variables globales }
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-1 ;
      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 := 7*d+6 ;
      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
```

2.4. Monitores

25

Se consideran dos tipos de recursos accesibles por varios procesos concurrentes (denominamos a los recursos como recursos de tipo 1 y de tipo 2). Existen N_1 ejemplares de recursos de tipo 1 y N_2 ejemplares de recursos de tipo 2.

Para la gestión de estos ejemplares, queremos diseñar un monitor (con semántica SU) que exporta un procedimiento (**pedir_recurso**), para pedir un ejemplar de uno de los dos tipos de recursos. Este procedimiento incluye un parámetro entero (**tipo**), que valdrá 1 o 2 indica el tipo del ejemplar que se desea usar.

Asimismo, el monitor incorpora otro procedimiento (**liberar_recurso**) para indicar que se deja de usar un ejemplar de un recurso previamente solicitado (este procedimiento también ademite un entero que puede valer 1 o 2, según el tipo de ejemplar que se quiera liberar). En ningún momento puede haber un ejemplar de un tipo de recurso en uso por más de un proceso. En este contexto, responde a estas cuestiones:

(a) Implementa el monitor con los dos procedimientos citados, suponiendo que N_1 y N_2 son dos constantes arbitrarias, mayores que cero.

(b) El uso de este monitor puede dar lugar a interbloqueo. Esto ocurre cuando más de un proceso tiene algún punto en su código en el cual necesita usar dos ejemplares de distinto tipo a la vez. Describe la secuencia de peticiones que da lugar a interbloqueo.

(c) Una posible solución al problema anterior es obligar a que si un proceso necesita dos recursos de distinto tipo a la vez, deba de llamar a **pedir_recurso**, dando un parámetro con valor 0, para indicar que necesita los dos ejemplares. En esta solución, cuando un ejemplar quede libre, se dará prioridad a los poibles procesos esperando usar dos ejemplares, frente a los que esperan usar solo uno de ellos.

Respuesta

Cuestión (a):

Puesto que los procesos necesitan esperar en **pedir_recurso** cuando no hay ejemplares del tipo que quieren, necesitamos saber cuantos ejemplares quedan libres de cada tipo de recurso. Para eso usaremos un array con dos entradas indicando esos dos valores. Al array lo llamamos **libres**

En **pedir_recurso**, los procesos que piden un recurso de tipo 1 esperan la condición **libres**`[1]>0` Se usarán dos colas de espera, una por cada tipo de recurso, y dos variables enteras, con el número de ejemplares libres. Tanto las colas como las variables se ponen en dos arrays. El código puede ser como sigue:

```
Monitor DosRecursos_v1;

var libres : array[1..2] of integer;    { número de ejemplares libres (por cada tipo) }
    cola   : array[1..2] of condition;  { procesos esperando un ejemplar libre (por cada tipo) }

procedure pedir_recurso( tipo : integer )   { tipo == 1 ó 2 }
begin
   if libres[tipo] == 0 then          { si no quedan ejemplares:      }
      cola[tipo].wait();              {  esperar a que quede uno libre }
   libres[tipo] = libres[tipo]-1 ;    { queda un ejemplar menos libre  }
end
procedure liberar_recurso( tipo : integer )
begin
   libres[tipo] = libres[tipo]+1 ;    { uno más libre       }
   cola[tipo].signal() ;              { despertar uno (si hay) }
end
begin
```

```
   libres[1] := N1 ;
   libres[2] := N2 ;
end
```

Cuestión (b):

La situación de interbloqueo ocurre cuando dos procesos llaman cada uno dos veces seguidas a `pedir_recurso`, pidiendo los recursos en orden contrario, y quedando solo un ejemplar libre de cada uno de ellos. En ese caso, cada proceso supera la primera llamada, pero ambos quedan esperando en la segunda.

Cuestión (c):

Ahora se usarán tres colas de espera, dos para los que solicitan un único ejemplar de un recurso (igual que antes), y una nueva para los que solicitan ambos tipos de recursos. Estos esperan una nueva condición, en concreto esperan `libres[1]>0 and libres[2]>0`. Se mantienen las dos variables enteras para saber cuantos recursos libres quedan por cada tipo de recurso.

```
Monitor DosRecursos_v2 ;

var libres : array[1..2] of integer;    { numero de ejemplares libres (por cada tipo) }
    cola   : array[1..2] of condition;  { procesos esperando un ejemplar libre (por cada tipo) }
    ambos  : condition                  { procesos esperando dos ejemplares de los dos tipos }

procedure pedir_recurso( tipo : integer ) { tipo == 0,1 ó 2 }
begin
   if tipo == 0 then begin              { quiere dos ejemplares     }
      if libres[1]==0 or                { si no quedan de un tipo   }
         libres[2]==0 then              { o no quedan del otro:     }
         ambos.wait() ;                 {   esperar                 }
      libres[1] = libres[1] -1 ;        { un ejemplar menos de tipo 1 }
      libres[2] = libres[2] -1 ;        { un ejemplar menos de tipo 2 }
   end
   else begin                           { solo quiere 1 de un tipo }
      if libres[tipo] == 0 then         { si no quedan ejemplares: }
         cola[tipo].wait();             {  esperar                  }
      libres[tipo] = libres[tipo]-1 ;   { un ejemplar menos libre  }
   end
end
procedure liberar_recurso( tipo : integer )   { tipo == 1 o 2 }
   var otro_tipo : integer := 1+(tipo mod 2);    { el otro tipo }
begin
   libres[tipo] = libres[tipo]+1 ;      { uno más libre de este tipo   }
   if libres[otro_tipo] > 0             { si hay libres del otro y     }
     and ambos.queue() then             {  y esperan alguno por ambos: }
      ambos.signal();                   {   liberar a uno de 'ambos'   }
   else
      cola[tipo].signal();              { liberar a alguno de este     }
end
begin
   libres[1] := N1 ;
   libres[2] := N2 ;
end
```

26

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

a) Con prioridad a los lectores. Quiere decir que, si en un momento puede acceder al recurso tanto un lector como un escritor, se da paso preferentemente al lector.

b) Con prioridad a los escritores. Quiere decir que, si en un momento puede acceder tanto un lector como un escritor, se da paso preferentemente al escritor.

c) Con prioridades iguales. En este caso, los procesos acceden al recurso estrictamente en orden de llegada, lo cual implica, en paricular, que si hay lectores leyendo y un escritor esperando, los lectores que intenten acceder después del escritor no podrán hacerlo hasta que no lo haga dicho escritor.

Respuesta

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 una variable para llevar la cuenta de cuantos lectores hay leyendo (`nlectores`) y otra variable (lógica) (`escribiendo`) que indicará si hay algún escritor escribiendo. Estas variables son imprescindibles para poder implementar las esperas.

Respecto a las condiciones que espera cada tipo de proceso, los lectores esperan la condición `not escribiendo`, en la cola de nombre `lectores`. Los escritores esperan `not escribiendo and nlectores==0`, en la cola de nombre `escritores`.

(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`: hay que evitar ahora que una ráfaga de lectores deje esperando a los escritores. Para ello, se hacen dos modificaciones: por un lado, el `signal` de los lectores no se hace si hay escritores esperando entrar, y por otro lado ahora un lector espera no solo si hay un escritor escribiendo, sino también si hay escritores en su cola esperando a entrar.

```
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 or escritores.queue() 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
```

(c) sin prioridad

En esta solución no se pueden usar dos colas, una por tipo de proceso (una de lectores y otra de escritores), puesto que siempre habría que elegir una frente a otra para despertar un proceso, sin poder saber en absoluto cuál de las dos tiene el proceso que lleva más tiempo esperando.

Por eso, en principio, todos los procesos esperan en una sola cola (que es FIFO). Puesto que los procesos en esa cola esperan condiciones distintas, en el momento en que se haga un signal sobre ella, no podemos asegurar que se cumple la condición que espera el proceso que sale (el que más tiempo lleva esperando, ya que no sabemos si es lector o escritor). Por tanto, como siempre en estos casos, necesitamos incluir las llamadas a **`wait`** en un bucle while, de forma que al salir, si no se cumple su condición, volverán a la cola. Sin embargo, cuando un proceso sale del wait y comprueba que debe volver a esperar, entonces se pone el último en la cola (es FIFO), y por tanto no se cumple el requisito de que los procesos acceden al recurso en orden de llamada al monitor. Usar una sola cola es, por tanto, inviable.

Para solucionar el problema, en realidad podemos observar que en cualquier momento en el que haya uno o más procesos esperando aceder al recurso, sólo uno de ellos (el primero que invocó el procedimiento de acceso) debe realmente comprobar su condición de entrada. El resto de procesos (si hay alguno), están esperando que ese primer procedimiento pueda acceder, para ellos a su vez poder comprobar la condición (por orden de llegada).

Esta descripción sugiere el uso de dos colas: una con dicho **primer** proceso (`primero`), y otra con el **resto** de procesos (`resto`). Los procesos que esperan en la cola `resto` esperan que la cola `primero` quede vacía (es un ejemplo en el cual la condición lógica asociada a una variable condición involucra el estado de otra cola). Si un proceso espera en la cola `primero` , espera que se cumpla su condición de acceso al recurso.

En la cola `primero` hay un proceso como mucho. Si está vacía, también lo esta `resto`. Al entrar cualquier proceso, si en `primero` hay algún proceso, entonces espera en `resto`. Todos los que están en `resto` esperan lo mismo (que `primero` quede libre), y por tanto da igual el tipo de proceso (lector o escritor) que sean. Los procesos salen de `resto` en orden de entrada, y después comprueban su condición de acceso al recurso, si no pueden entrar, esperan en `primero`. Por tanto todos los procesos pasan a la cola `primero` en el orden en el que acceden al monitor.

Cuando cualquier proceso (ya sea lector o escritor) comprueba que ya puede acceder al recurso, antes de hacerlo debe hacer un `signal` de la cola `resto`. Esto se debe a que en esas circunstancias `primero` va a quedar vacía con seguridad, y por tanto se cumple la condición de espera en `resto`. Por tanto, la última sentencia de `escritura_ini` y `lectura_ini` debe ser `resto.signal()`.

Los lectores y escritores, al acabar de leer o escribir, modifican (actualizan) las variables permanentes del monitor (`nlectores` se decrementa o `escribiendo` se pone a `false`), y por tanto pueden haber hecho cierta la condición de espera del proceso en `primero`. Por tanto, al acabar de acceder al recurso, los procesos deben de hacer `signal` en `primero`.

Los procesos deben entrar a la cola `primero` si comprueban que no pueden acceder al recurso. En el caso de los lectores, al salir de esa cola su condición se cumple con seguridad. Esto se debe a que ese lector ha salido de `primero` por un signal de otro proceso señalador que ha terminado de acceder al recurso. Tanto si el señalador es un lector como un escritor, el valor de `escribiendo` ya es `false`, y por tanto el proceso (lector) en `primero` podrá entrar con seguridad al recurso.

Sin embargo, cuando un escritor sale de `primero`, es posible que el señalador sea un escritor que ha terminado de escribir, en ese caso puede acceder al recurso (ya que no hay lectores leyendo), o bien puede que el señalador sea un lector que ha terminado de leer, pero puede que todavía haya otros lectores leyendo, y en ese caso el escritor no puede acceder. Por tanto, para que el programa sea correcto, cuando un escritor sale de `primero` hacemos que vuelva a comprobar su condición, es decir, ponemos el `primero.wait()` de los escritores en un bucle `while`.

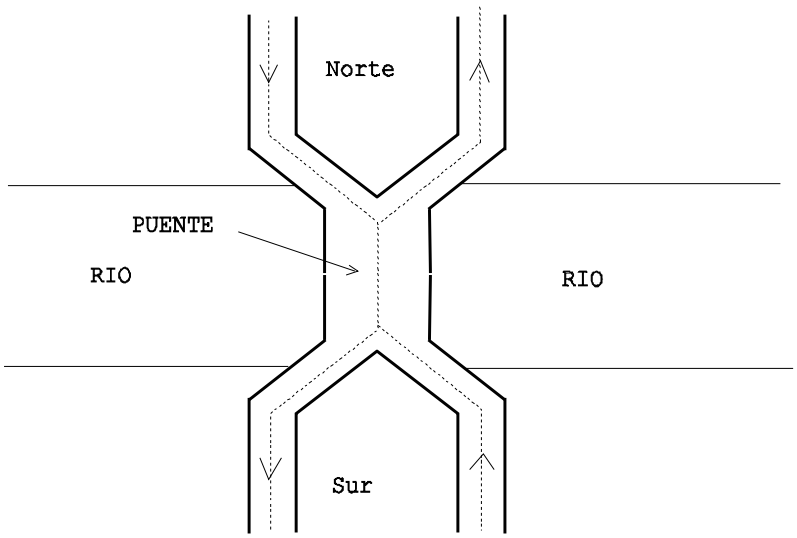
Con todo esto, el código fuente queda así:

```
Monitor LectoresEscritores_noprio ;

var escribiendo          : boolean := false ;
    nlectores            : integer := 0 ;
    primero              : condition ; { esperan aquí su condición de acceso (según tipo de proc.) }
    resto                : condition ; { esperan aquí hasta que "primero" queda vacía }  }

procedure escritura_ini() ;
begin
```

```
   if primero.queue() then                    { si otro tiene preferencia    }
      resto.wait() ;                          {   esperar                    }
   while escribiendo or nlectores >  0 do     { si no es posible escribir:   }
      primero.wait() ;                        {   esperar                    }
   escribiendo := true ;                      { anotar que se esta escribiendo }
   resto.signal() ;                           { pasar al siguiente a 'primero' }
end

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

procedure lectura_ini() ;
begin
   if primero.queue() then     { si otro tiene preferencia     }
      resto.wait() ;           {   esperar                     }
   if escribiendo do           { si hay un escritor escribiendo  }
      primero.wait() ;         {   esperar                     }
   nlectores := nlectores+1 ;  { anotar un lector mas          }
   resto.signal() ;            { pasar al siguiente a 'primero' (si hay alguno) }
end

procedure lectura_fin() ;
begin
   nlectores := nlectores-1 ;  { anotar un lector menos        }
   primero.signal() ;          { permitir comprobar a otro     }
end
```

27

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

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
```

28

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

Es evidente que necesitamos saber el estado de la olla, es decir, cuantos misioneros quedan disponibles para comer. Se usa una variable entera, incializada a M, llamada **num_misioneros**. Se introducen dos variables de condición, para las esperas asociadas al cocinero y a los salvajes, respectivamente (**cocinero** y **salvajes**).

La cola **salvajes** es la cola donde esperan los salvajes en el caso de que no haya comida, es decir donde los salvajes esperan hasta que haya al menos un misionero disponible, por tanto la condición es:

`0 < num_misioneros`

La variable condición **cocinero** es donde espera el cocinero hasta que es necesario rellenar la olla, por tanto la condición asociada es:

`0 == num_misioneros`

Por tanto, el código para impementar el monitor es el siguiente:

```
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 { si no hay comida:       }
      salvajes.wait();         {  esperar a que haya comida }

   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
```

29

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.

Queremos usar un monitor para resolver el problema. El monitor debe tener 2 procedimientos: **depositar**(c) y **retirar**(c). Suponer que los argumentos de las 2 operaciones son siempre positivos, e indican las cantidades a depositar o retirar. El monitor usará la semántica *señalar y espera urgente* (SU). Se deben de escribir varias versiones de la solución, según las variaciones de los requerimientos que se describen a continuación:

(a) 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. Hacer dos versiones:

(a.1) colas normales (FIFO), sin prioridad.

(a.2) con colas de prioridad.

(b) El reintegro de fondos a los clientes se hace únicamente según el orden de llegada, si hay más de un cliente esperando, solo el primero que llegó puede optar a retirar la cantidad que desea, mientras esto no sea posible, esperarán todos los clientes, independientemente de cuanto quieran retirar los demás. 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. De nuevo, resolverlo de dos formas:

(b.1) colas normales (FIFO), sin prioridad.

(b.2) con colas de prioridad.

Respuesta

(a.1) Puede retirar el primero que tenga saldo, colas sin prioridad

Se usa una cola (**cola**) para los clientes que esperan a retirar. La condición que esperan las hebras es que el saldo disponible sea igual o mayor a la cantidad a retirar. Si un cliente no puede retirar, antes de volver a la cola, debe de despertar a los otros clientes de la misma para que alguno de esos otros pueda retirar. También debe hacer lo mismo después de haber retirado (ya que puede ser que después de retirar aun quede saldo suficiente para otros). La solución queda así:

```
monitor CuentaCorriente;

   var saldo : integer ;
       cola  : condition ;

procedure Retirar( cantidad : integer )
```

```
begin
   while cantidad > saldo do begin { mientras no se pueda atender peticion: }
      cola.signal() ;              {  permitir que otros comprueben si pueden sacar }
      cola.wait() ;                {  esperar hasta volver a comprobar }
   end
   saldo = saldo - cantidad ;      { retirar cantidad }
   cola.signal();                  { permitir a otros comprobar }
end
procedure Depositar( cantidad : integer )
begin
   saldo = saldo + cantidad;
   cola.signal();
end
{  inicializacion  }
begin
   saldo = saldo_inicial ; { == constante predefinida }
end
```

Hay que tener en cuenta que en el caso de que en la cola haya varios clientes y se produzca un ingreso que no sea suficiente para que ninguno de ellos pueda sacar, entonces el primero en comprobar que no puede, al hacer **signal**, pasa a la cola de urgentes y saca al siguiente de la cola condición. Esto ocurre para todos los de la cola, en cadena, hasta el último, que hace **signal** de la cola vacía, no da paso a ningún otro proceso y por tanto puede ejecutar su **wait**, por lo cual a partir de entonces todos los procesos abandonan la cola de urgentes y van haciendo **wait**, entrando por consiguiente de nuevo en la cola condición.

(a.2) Puede retirar el primero que tenga saldo, colas con prioridad

Usando colas de prioridad, el problema se resuelve muy fácilmente (este es un ejemplo claro de la utilidad de las colas de prioridad). Para hacerlo, usamos como valor de prioridad para entrar en la cola la cantidad que el cliente quiere retirar. De forma que, si hay varios clientes esperando retirar, siempre tras un signal saldrá uno de los quieran retirar la cantidad mínima de entre todas las cantidades a retirar.

Si el cliente seleccionado no puede retirar, no puede hacerlo ningún otro, por lo cual no es necesario que antes de volver a la cola avise a los demás, y el algoritmo se simplifica bastante. Por otro lado, sigue siendo necesario que tras retirar un cliente despierte al siguiente, ya que puede que un ingreso permita más de una operación de retirada.

La solución queda así:

```
Monitor CuentaCorriente;

   var saldo : integer ;
       cola  : condition ;

procedure Retirar( cantidad : integer )
begin
  while cantidad > saldo do
     cond.wait( cantidad );
  saldo = saldo - cantidad;
  cola.signal() ;
end
procedure Depositar( cantidad : integer )
```

```
begin
  saldo = saldo + cantidad;
  cond.signal();
end
{ inicializacion }
begin
   saldo= saldo_inicial ;
end
```

(b.1) Puede retirar únicamente el primero que llegó al banco, colas sin prioridad

En este caso no se puede usar una única cola condición sin prioridad, ya que en ese caso cuando el proceso que más tiempo lleva en ella sale para comprobar el saldo y resulta insuficinte, dicho proceso vuelve a la cola y se pone el último, perdiendo la prioridad que debe tener según el enunciado, por ser el más antiguo. Sin embargo, se puede escribir una solución sencilla que se basa en tener dos variables condición (dos colas) para los clientes:

- Una 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**)

Cuando un cliente llega al banco, si la cola **ventanilla** está vacía, entonces pasa a esa cola, en otro caso (ya hay uno en **ventanilla**) el cliente espera en **resto**. La solución podría quedar así:

```
Monitor CuentaCorriente ;

   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
```

(b.2) Puede retirar únicamente el primero que llegó al banco, colas con 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 e inmediatamente verifica que hay saldo suficiente, no podrá hacerlo si ya había otros procesos en la cola, ya que eso significa que, aunque hay saldo para él, no es el cliente que puede retirar pues hay esperando al menos otro que llegó antes. Por tanto, si al entrar un cliente ve la cola con al menos otro cliente, el que entra debe ingresar en dicha cola.

```
Monitor CuentaCorriente ;

   var saldo, contador : 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    }
   if cola.queue() then                { si ya hay otros esperando:         }
      cola.wait(ticket) ;              {   ingresar en la cola              }
   while cantidad > saldo or           { mientras el saldo no sea suficiente }
      cola.wait(ticket) ;              {   esperar                          }
   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 ;
end
```

30

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 (es decir, si $i < j$, entonces P_i pasa por delante de P_j) Implementar un monitor que implemente los procedimientos **Pedir** y **Liberar**.

Respuesta

```
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
```

31

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.

Implementar un monitor (con semántica SU) 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 (con un parámetro) o una operación específica para los de tipo A y otra para los de tipo B.

Respuesta

En esta solución se exporta una procedimiento para los procesos de tipo a (llamada **SincA**), y otra para los de tipo B (**SincB**).

Parace basante evidente que todos los procesos necesitan saber cuantos procesos de cada tipo han llegado ya a la cita. Usaremos dos variables enteras para ello, llamadas **nA** y **nB**. Para implementar las esperas, cada proceso comprobará si es el último del grupo de 11 procesos en la cita. Si lo es, despierta a los otros 10. Si no lo es, espera hasta que otro proceso posterior lo despierte. Después, el proceso continua.

Cuando el último en llegar a la cita despierta a todos los demás, dicho último proceso permenece en la cola de urgentes hasta que todos los demás (que estaban esperando) abandonan todos el monitor, así que durante ese intervalo de tiempo se prohíbe la entrada de otros nuevos procesos a la operación del monitor (esperan en la cola del monitor).

```
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
```

32

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 (a continuación ambos continuan), 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

Necesitaremos un semáforo **s** para las esperas asociadas a la cita, y otro **mutex** que implementa la exclusión mutua en el acceso a las variables compartidas. Por supuesto, también se necesita una variable compartidad (**n**) para saber si otro proceso ha llegado antes a la cita o no.

El código quedaría como aparce aquí:

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

```
procedure entrada() ;
begin
   sem_wait( mutex );
   n := n+1 ;                { uno más ha llegado a la cita  }
   if n < 2 then begin       { si es el primero:             }
      sem_signal( mutex );   {  liberar exclución mutua      }
      sem_wait( s );         {  esperar al segundo           }
   end
   else begin                { si es el segundo:             }
      n := 0 ;               {  el siguiente será el primero }
      sem_signal( s );       {  despertar al primero         }
      sem_signal( mutex );   {  liberar exclusión mutua      }
   end
end
```

33

Este es un ejemplo clásico que ilustra el problema del *interbloqueo*, y aparece en la literatura con el nombre de **el problema de los filósofos-comensales**. Se puede enunciar como se indica a continuación:

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(n` para indicar que un proceso filósofo desea coger un tenedor determinado.

El código del programa (sin incluir la implementación del monitor) es el siguiente:

```
monitor MonFilo ;
   ....
   procedure coge_tenedor( num_ten, num_proc : integer );
      ....
   procedure libera_tenedor( num_ten : integer );
      ....
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);
      MonFilo.libera_tenedor(i+1 mod 5);
      pensar();
   end
end
```

Con este interfaz para el monitor, responde a las siguientes cuestiones:

(a) Diseña una solución para el monitor `MonFilo`

(b) Describe la situación de interbloqueo que puede ocurrir con la solución que has escrito antes.

(c) Diseña una nueva solución, en la cual se evie el interbloqueo descrito, para ello, esta solución no debe permitir que haya más de cuatro filósofos simultáneamente intentado coger su primer tenedor

Respuesta

Cuestión (a)

Respecto de las variables permanentes, los procesos necesitan saber si cada tenedor está libre u ocupado, lo cual se debe implementar obviamente con una variable lógica por cada tenedor. Dada la numeración de filósofos y tenedores, lo más sencillo es usar un array de 5 valores lógicas, con índices comenzando en cero. Lo llamamos `ten_ocup`. El valor de `ten_ocup[i]` es **`true`** si el i-ésimo tenedor está ocupado, **`false`** si está libre.

Respecto de las variables condición, vemos que un filósofo puede estar en un instante de tiempo esperando que se quede libre un tenedor concreto, por tanto necesitamos una cola de espera por cada tenedor, ya que la condición de espera de cada tenedor es distinta, en concreto, la condición del i-ésimo tenedor es **`not`** `ten_ocup[i]`. Cada una de estas colas tendrá un proceso como mucho. De nuevo, lo más fácil es disponerlas también en un array.

```
monitor MonFilo ;
var
 ten_ocup  : array[0..4] of boolean ;   { true <==> el i-ésimo tenedor está en uso }
 cola_ten  : array[0..4] of condition;  { colas de espera por cada tenedor       }

procedure coge_tenedor( num_ten, num_proc : integer );
begin
   if ten_ocup[num_ten] then           { si el tenedor esta ocupado: }
      cola_ten[num_ten].wait() ;       {  esperar a que este libre }
   ten_ocup[num_ten] := true ;         { marcar tenedor como ocupado }
end
procedure libera_tenedor( num_ten : integer );
begin
   ten_ocup[num_ten] := false ;        { el tenedor ya no esta ocupado }
   cola_ten[num_ten].signal() ;        { avisar a alguno que lo esperaba, si hay }
end
{  inicializacion  }
begin
   for i :=  0 to 4 do
```

```
      ten_ocup[i] := false ; { tenedores libres }
end
```

Cuestión (b)

El interbloqueo ocurre si todos los filósofos toman el tenedor a su izquierda antes de que ninguno de ellos pueda coger el tenedor de la derecha, en ese caso todos los filósofos quedan esperando en el **wait** de la segunda llamada al monitor para coger un tenedor, sin que ninguno de ellos pueda abandonar dicho **wait**, al no haber ninguna operación **signal**.

Cuestión (c)

Para esta nueva solución mantenemos las colas y variables permanentes que ya había en la anterior. Para solucionar el interbloqueo, un filósofo hara una espera previa, antes de intentar coger su primer tenedor, si ya hay 4 filósofos que han cogido su primer tenedor pero todavía no han cogido su segundo tenedor. Para implementarlo, añadimos una nueva variable permanente entera (**num_f12**), que nos indica cuantos filósofos hay entre su primer y segundo tenedor. Añadimos por tanto una cola de espera, llamada **previa**, para el caso de que un filósofo deba retrasar su intento de coger su primer tenedor. La condición que esperan los filósofos en la cola **previa** es **num_f12** < 4. De esta manera, evitamos que haya 5 filósofos esperando en las colas de tenedor, ya que como mucho habrá 1 esperando en **previa** y 4 en las colas de los tenedores (se garantiza que al menos uno de los 4 en las colas de tenedores podrá progresar, ya que al menos uno de ellos tendrá disponible su segundo tenedor con seguridad).

En la solución se debe distinguir si un filósofo está intentando coger su primer tenedor o el segundo. Esto es fácil hacerlo ya que en el caso de ser primer tenedor, el número de filósofo coincide con el del tenedor.

```
monitor MonFilo ;
var
 previa     : condition ;                 { no más de cuatro esperando primer tenedor }
 ten_ocup   : array[0..4] of boolean ;    { true <==> el i-ésimo tenedor está en uso }
 cola_ten   : array[0..4] of condition;   { colas de espera por cada tenedor }
 num_f12    : integer ;                   { número de filosofos con 1er y sin 2o ten. }

procedure coge_tenedor( num_ten, num_proc : integer );
begin
   { si es necesario, hacer espera previa }
   if num_ten == num_proc then begin { si es el primer tenedor de los dos: }
      if num_f12 == 4 then           {   si ya hay 4 con 1er y sin 2o }
         previa.wait() ;             {     esperar a que alguno logre su segundo }
   end

   { coger el tenedor, esperando si es necesario }
   if ten_ocup[num_ten] then         { si el tenedor está ocupado: }
      cola_ten[num_ten].wait() ;     {   esperar a que este libre }
   ten_ocup[num_ten] := true ;       { marcar tenedor como ocupado }

   { actualizar cuenta de procesos con 1o y sin 2o }
   if num_ten == num_proc then       { si ha cogido su primer tenedor: }
      num_f12 := num_f12+1 ;         {   hay uno más con 1o y sin 2o }
   else begin                        { si ha cogido su segundo tenedor: }
      num_f12 := num_f12-1 ;         {   hay uno menos con 1o y sin 2o }
      if num_f12 < 4 then            {   si ahora ya hay menos de 4 con 1o y sin 2o }
```

```
         previa.signal() ;              {    dejar uno coja primero, si hay }
   end
end
procedure libera_tenedor( num_ten : integer );
begin
   ten_ocup[num_ten] := false ;         { el tenedor ya no esta ocupado }
   cola_ten[num_ten].signal() ;         { avisar a alguno que lo esperaba, si hay }
end
{  inicializacion  }
begin
   num_f12 := 0 ;     { hay 0 filósofos con 1o y sn 2o }
   for i :=  0 to 4 do
      ten_ocup[i] := false ; { tenedores libres inicialmente }
end
```

Sistemas basados en paso de mensajes.

3.1. Funciones de envío y recepción.

34

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

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

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

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

Contesta a las siguientes cuestiones:

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

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

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

Respuesta

(a) A que todos los procesos empiezan con un **send** síncrono y ninguno hace el correspondiente **receive**, por lo que todos los procesos se quedan esperando al proceso receptor

(b) No, porque los mensajes se almacenarían en el buffer/memoria intermedia después de **send** y todos ejecutarían el **receive** (alguien podría poner que si el buffer intermedio está lleno el **send** asíncrono también bloquearía pero no sería un interbloqueo, porque se supone que cuando se liberase dicha memoria intermedia podrán salir del bloqueo del **send**).

(c) Bastaría con cambiar el orden de las operaciones **s_send** y **receive** en uno o dos procesos cualquiera.

35

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

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

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

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

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

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

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

Respuesta

(a) se transfiere 40, se puede imprimir 30 o 40.

(b) se puede transferir 20 o 40, se puede imprimir 30, 40 o 20.

(c) se transfiere 40, se imprime 40.

36

En un sistema distribuido se ejecutan dos procesos A y B. Para cada uno de los valores enteros 1,2,3 y 4, indica si el proceso A puede imprimir o no puede imprimir ese valor. En caso negativo, justifica tu respuesta, en caso afirmativo, indica la secuencia de accesos a variables que llevan a dicho resultado. Cada acceso puede ser uno de estos:

(a) una asignación atómica a una variable, por ejemplo *x:=34*,

(b) la lectura del valor de un variable por el SPM, por ejemplo *leer 35 en y*

(b) la escritura de un valor en una variable por el SPM, por ejemplo *escribir 46 en z*,

(d) imprimir un valor, por ejemplo *imprime 72* (lo que ocurra después da igual, así que esto es siempre lo último)

(suponemos que estos accesos son todos ellos atómicos)

```
process A ;
   var a : integer  ;
begin
   a := 1 ;
   i_send( a, B );
   a := 2 ;
   receive( a, B );
   print(a);
end
```

```
process B ;
   var b : integer  ;
begin
   b := 3 ;
   i_receive( b, A );
   b := 4 ;
   s_send( b, A );

end
```

Respuesta

Valor 1

1. `a:= 1`
2. leer 1 en `a`
3. `a:= 2`
4. `b:= 3`
5. `b:= 4`
6. escribir 1 en `b`
7. leer 1 en `b`
8. escribir 1 en `a`
9. imprime 1

Valor 2

1. `a:= 1`
2. `a:= 2`
3. leer 2 en `a`
4. `b:= 3`
5. `b:= 4`
6. escribir 2 en `b`
7. leer 2 en `b`
8. escribir 2 en `a`
9. imprime 2

Valor 4

1. `a:= 1`
2. leer 1 en `a`
3. `a:= 2`
4. `b:= 3`
5. escribir 1 en `b`
6. `b:= 4`
7. leer 4 en `b`
8. escribir 4 en `a`
9. imprime 4

Valor 3

Este valor no puede imprimirse nunca, ya que cuando B hace **`s_send`**, el valor leído en la variable `b` puede ser 4 (si no se ha recibido nada después de hacer `b:=4`), o bien 1 o 2 (si se ha recibido algún valor después de `b:=4`, ese valor puede ser 1 o 2)

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
```

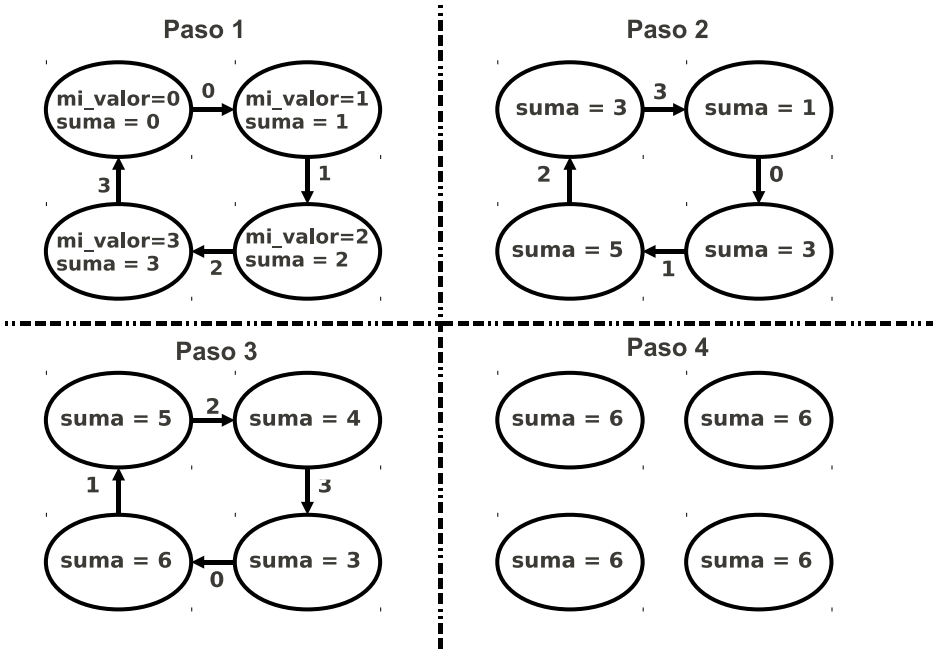
Respuesta

```
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

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



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

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

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

Respuesta

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

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

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

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

3.2. Espera selectiva

39

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

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

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

```
process Controlador ;
begin
   while true do begin

      ...

   end
end
```

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

Respuesta

A continuación se da una posible solución. Esta solución usa un vector de valores lógicos (`recibido`) que indica, para cada proceso cliente, si el controlador ha recibido o no ya la petición de dicho cliente. Usando

un contador, se determina cuando se han recibido 3 peticiones y por tanto cuando se puede dar paso a un grupo de tres procesos. En ese momento, el vector `recibido` almacena los procesos a los que hay que enviar respuesta.

```
process Controlador ;

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

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

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

40

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

de procesos:

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

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

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

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

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

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

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

Respuesta

Una posible respuesta sería esta:

```
process Buffer ;

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

begin
   while true do
      select
         for i := 0 to 2
         when contador < tam and ultimo!=i receive( dato, Productor[i] ) do
            ultimo := i ;
```

```
            buf[contador]:= dato;
            contador := contador + 1 ;
         when contador >= 2 do
            contador:=contador-1;
            send( buf[contador], Consumidor );
      end { select }
end
```

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

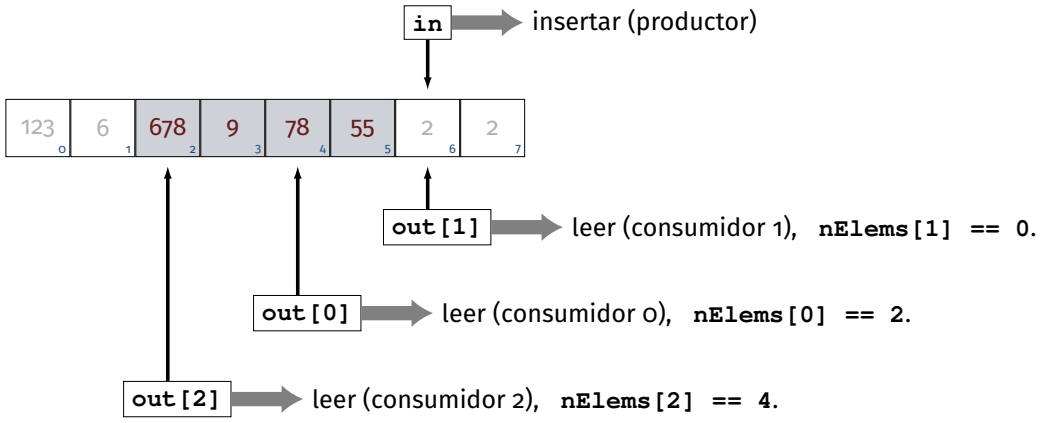
```
process Buffer ;
var tam      : integer :=  4 ; { capacidad del buffer }
    ultimo   : integer := -1 ; { indice del ultimo que escribio en buffer }
    contador : integer :=  0 ; { numero de entradas ocupadas }
    prim_ocu : integer :=  0 ; { primera entrada ocupada }
    prim_lib : integer :=  0 ; { primera entrada libre }
    dato     : integer ;
    buf      : array [0..tam-1] of integer ;
begin
   while true do
      select
         for i := 0 to 2
         when contador < tam and ultimo != i receive(&dato,Productor[i]) do
            ultimo := i ;
            buf[prim_lib]:= dato ;
            prim_lib := (prim_lib+1) mod tam ;
            contador := contador + 1 ;
         when contador >= 2 do
            prim_ocu := (prim_ocu+1) mod tam ;
            contador := contador - 1 ;
            send ( buf[prim_ocu], Consumidor);
      end { select }
end
```

41

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 pro-

ductor 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 (**out**) y el número de elementos que quedan en el buffer por consumir (**nElems**). En la figura se ve un esquema de un estado del buffer, a modo de ejemplo:



Respuesta

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

```
process Buffer ;

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

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

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

42

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

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

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

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

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

Respuesta

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

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

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

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

43

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

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

      Usar_recurso( );

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

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

       ...

   end
end
```

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

Respuesta

Una solución consiste en usar los valores (+1 y -1) asociados con la petición de uso y la petición de liberar, respectivamente. Asimismo, usamos un valor (**pendiente**) que indica si hay una petición de uso pendiente (vale +1), o bien si hay una petición de liberar pendiente (vale −1), o bien no hay ninguna petición pendiente (vale 0). Finalmente, usamos una variable con el identificador de proceso que está esperando (**cliente_e**).

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

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

         if contador + peticion + pendiente == 13 then  begin
            { no se puede atender (pendiente será  0 aquí con seguridad) }
            pendiente := peticion ;  { registrar que clase de petición queda pendiente }
            cliente_e := i ;  { registrar que cliente queda pendiente }
         end
```

```
            else begin
               { se puede atender, contador tomará un valor distinto de 13 }
               contador := contador + peticion + pendiente ;
               send( permiso, Cliente[i] );
               if pendiente != 0 then begin
                  send( permiso, Cliente[cliente_e])
                  pendiente := 0 ;
               end
            end
            else

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

44

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

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

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

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

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

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

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

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

Respuesta

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

45

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

Si en algún momento no hay ningún mensaje pendiente de recibir en el receptor, este proceso debe de imprimir ”no hay mensajes. duermo.z después bloquearse durante 10 segundos (con **`sleep_for`**`(10)`), antes de

volver a comprobar si hay mensajes (esto podría hacerse para ahorrar energía, ya que el procesamiento de mensajes se hace en ráfagas separadas por 10 segundos).

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

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

Respuesta

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

Sin embargo, la sentencia **select** sí permite incluir guardas sin sentencia de entrada, guardas que solo se considerarán para su ejecución en los casos en los que las guardas con sentencia de entrada ejecutables no tengan envíos pendientes que casen con ellas.

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

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

46

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

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

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

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

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

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

Respuesta

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

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

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

```
process Receptor ;
begin
```

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

3.3. Envios de muchos a muchos

47

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

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

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

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

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

Respuesta

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

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

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

```
   { hacer todas las recepciones }
   for j := 1 to N do
      if i != j then begin
         receive( c, P[j] ) ;
         print( c );
      end
end
```

48

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

Respuesta

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

El código queda así:

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

49

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

ben consumir con una llamada a `ConsumirBloque(b)`.

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

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

Respuesta

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

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

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

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

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

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

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

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

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

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

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

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

50

En los tres problemas anteriores, cada proceso va esperando a recibir un item de datos de cada uno de los otros procesos, consume dicho item, y después pasa recibir del siguiente emisor (en distintos órdenes). Esto implica que un envío ya iniciado, pero pendiente, no puede completarse hasta que el receptor no haya consumido los anteriores bloques, es decir, se introducen esperas que son innecesarias.

Escribe una solución en la cual cada proceso inicia sus envíos y recepciones y después espera a que se completen todas las recepciones antes de iniciar el primer consumo de un bloque recibido. De esta forma todos los mensajes pueden transferirse potencialmente de forma simultánea (hay más paralelismo potencial y por tanto el programa puede ejecutarse más rápido).

Suponer que cada proceso puede almacenar como mínimo 2N bloques en su memoria local, y que el orden de recepción o de consumo de los bloques es indiferente.

Respuesta

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

completado todos los envíos.

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

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

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

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

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

51

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

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

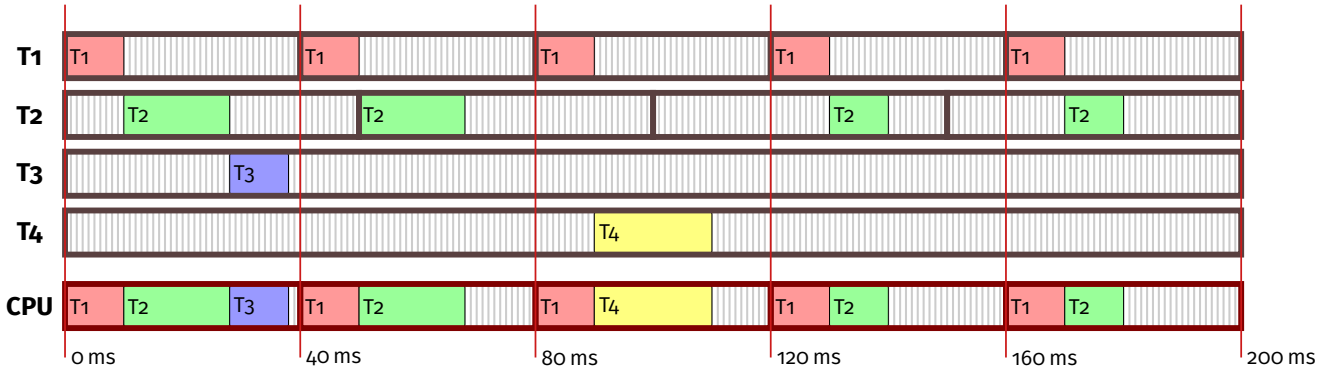
Respuesta

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

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

Como consecuencia, el marco secundario puede valer 20, 25 o 40. Para diseñar el ejecutivo cíclico tenemos que distribuir la ejecución de las distintas tareas entre los marcos secundarios que se han establecido dentro de un hiperperiodo, ya que luego el comportamiento se repite indefinidamente. Si suponemos que el marco secundario es $T_S = 40$, el cronograma podría ser:

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



52

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

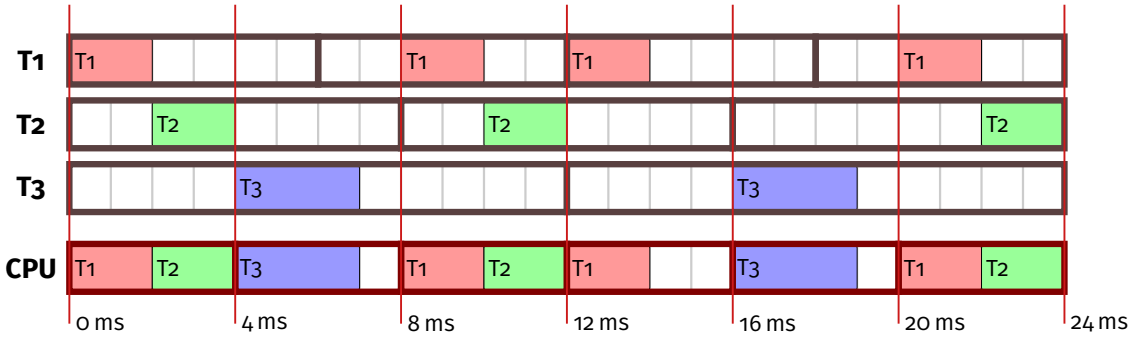
Tarea	C_i	T_i	D_i
T1	2	6	6
T2	2	8	8
T3	3	12	12

Respuesta

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

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

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



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

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

53

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

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

Respuesta

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

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

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

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

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

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

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

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

54

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

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

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

Respuesta

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

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

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

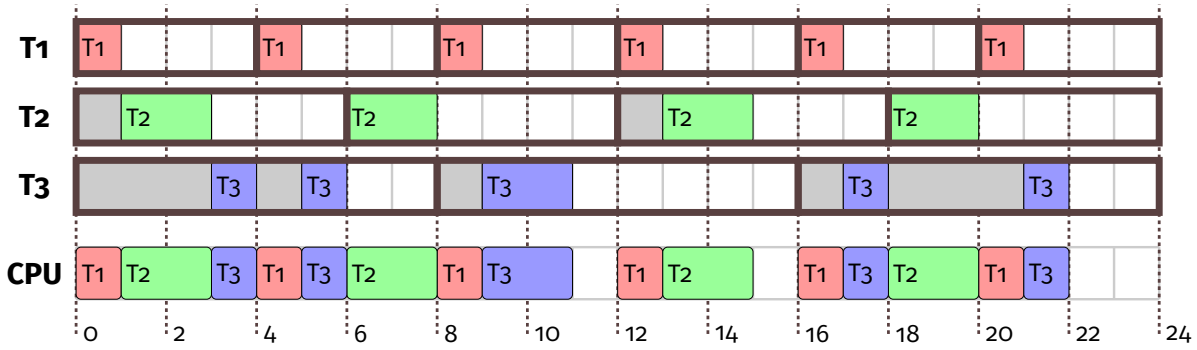
$$U \ = \ 0,8333 \ \not\leq \ 0,7779 \ = \ U_0(3)$$

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

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

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

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



55

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

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

Respuesta

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

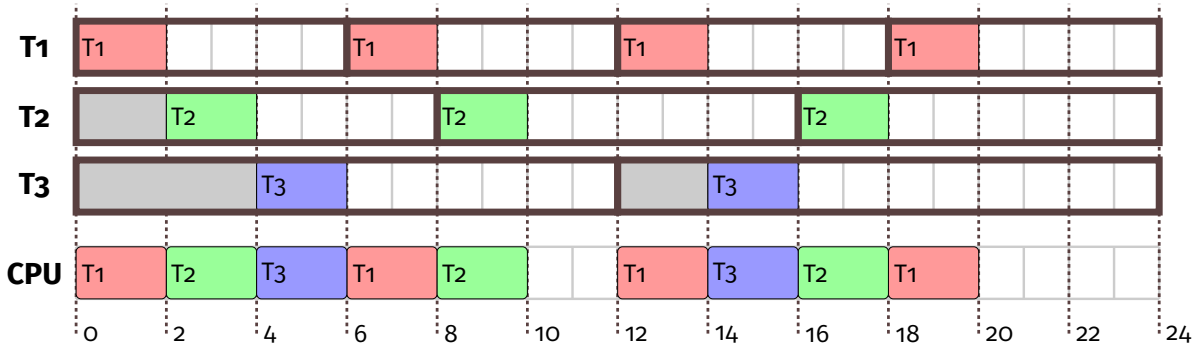
$$U = \frac{20}{60} + \frac{20}{80} + \frac{20}{120} = \frac{3}{4} = 0,75$$

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

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

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

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



56

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

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

Respuesta

En primer lugar calculamos el valor de U

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

el valor de T_M

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

y el valor de $U_0(5)$

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

(1) Prioridades estáticas RMS

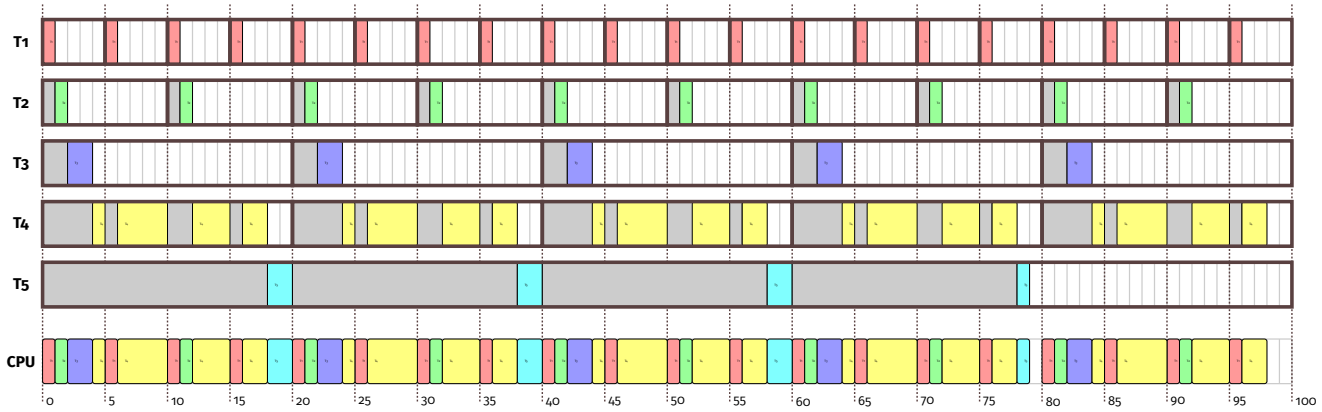
Pasamos el test de planificabilidad RMS basado en el factor de utilización:

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

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

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

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



(2) Prioridades dinámicas EDF

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

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

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

57

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

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