

# Sistemas Concurrentes y Distribuidos: **Enunciados de Problemas.**

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

Curso 2021-22



**Universidad de Granada**



# Índice general

<b>1. Introducción</b>	<b>5</b>
Problema 1. . . . .	5
Problema 2. . . . .	5
Problema 3. . . . .	6
Problema 4. . . . .	6
Problema 5. . . . .	7
Problema 6. . . . .	8
Problema 7. . . . .	10
Problema 8. . . . .	10
Problema 9. . . . .	11
<b>2. Sincronización en memoria compartida.</b>	<b>13</b>
Problema 10. . . . .	13
Problema 11. . . . .	13
Problema 12. . . . .	14
Problema 13. . . . .	15
Problema 14. . . . .	15
Problema 15. . . . .	15
Problema 16. . . . .	16
Problema 17. . . . .	16
Problema 18. . . . .	17
Problema 19. . . . .	17
Problema 20. . . . .	17
Problema 21. . . . .	18
Problema 22. . . . .	18
Problema 23. . . . .	20
Problema 24. . . . .	20
Problema 25. . . . .	21
Problema 26. . . . .	21
Problema 27. . . . .	22

---

Problema 28. . . . .	22
<b>3. Sistemas basados en paso de mensajes.</b>	<b>25</b>
Problema 29. . . . .	25
Problema 30. . . . .	25
Problema 31. . . . .	26
Problema 32. . . . .	27
Problema 33. . . . .	27
Problema 34. . . . .	28
Problema 35. . . . .	29
Problema 36. . . . .	30
Problema 37. . . . .	31
Problema 38. . . . .	31
Problema 39. . . . .	32
Problema 40. . . . .	32
Problema 41. . . . .	32
Problema 42. . . . .	33
<b>4. Sistemas de Tiempo Real.</b>	<b>35</b>
Problema 43. . . . .	35
Problema 44. . . . .	35
Problema 45. . . . .	35
Problema 46. . . . .	36
Problema 47. . . . .	36
Problema 48. . . . .	36
Problema 49. . . . .	37

## 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
```

## 2

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

- los archivos son secuencia de ítems 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 ;
```

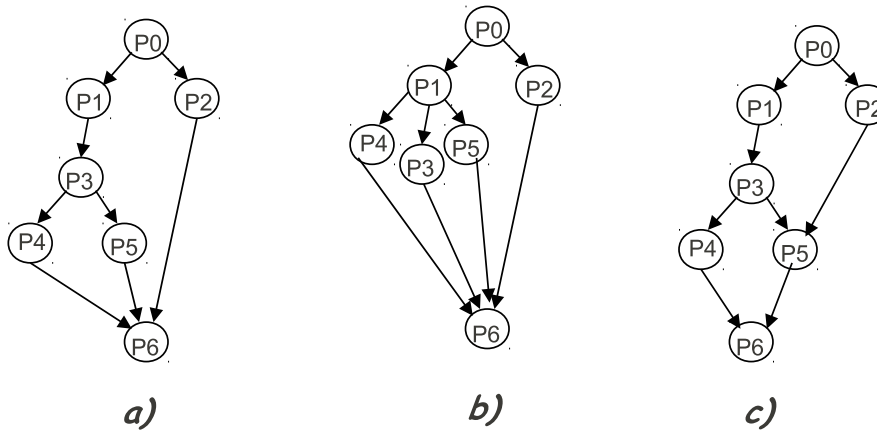
```

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
end

```

**3**

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


**4**

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

```

## 5

Suponer un sistema de tiempo real que dispone de un captador de impulsos conectado a un contador de energía eléctrica. La función del sistema consiste en contar el número de impulsos producidos en 1 hora (cada *Kwh* consumido se cuenta como un impulso) e imprimir este número en un dispositivo de salida.

Para ello se 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 período 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).

## 6

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
    end
  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 actúa 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** actúa 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

$$T_m(p) = p$$

## 7

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

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

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

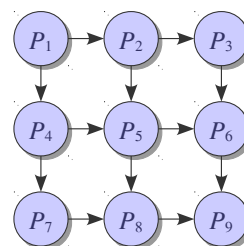
## 8

Un trozo de programa ejecuta nueve rutinas o actividades ( $P_1, P_2, \dots, P_9$ ), 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.

## 9

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 una secuencia de instrucciones desconocida, que llamamos **Sn**, y además debe incluir las llamadas necesarias a **Acabar** y **EsperarPor** (con la misma implementación que antes) para lograr la sincronización adecuada. Se incluye aquí un plantilla:

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

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



## Sincronización en memoria compartida.

10

¿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 }
```

<pre> 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</pre>	<pre> 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</pre>
---	---

11

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

<pre> { variables compartidas y valores iniciales } var c0 : integer := 1 ;     c1 : integer := 1 ;     turno : integer := 1 ; </pre>	
<pre> 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 </pre>	<pre> 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 </pre>

## 12

Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (señalan el turno en que ha de ser atendido el cliente), los números de los tickets se representan por dos variables **n1** y **n2** que valen inicialmente 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener 2 números iguales se procesa primero el proceso número 1.

- Demostrar que se verifica la ausencia de interbloqueo (progreso), la ausencia de inanición (espera limitada) y la exclusión mutua.
- Demostrar que las asignaciones **n1:=1** y **n2:=1** son ambas necesarias. Para ello

<pre> { variables compartidas y valores iniciales } var n1 : integer := 0 ;     n2 : integer := 0 ; </pre>	
<pre> 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 &lt; n1 do begin end; { L3.1 }     { seccion critica } { SC.1 }     n1 := 0 ; { E3.1 }     { resto sentencias } { RS.1 }   end end </pre>	<pre> 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 &lt;= n2 do begin end; { L3.2 }     { seccion critica } { SC.2 }     n2 := 0 ; { E3.2 }     { resto sentencias } { RS.2 }   end end </pre>

## 13

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.

<pre> { variables compartidas y valores iniciales } var c0 : integer := 1 ;     c1 : integer := 1 ; </pre>	
<pre> 1 process P0 ; 2 begin 3     while true do begin 4         repeat 5             c0 := 1-c1 ; 6             until c1 != 0 ; 7             { seccion critica } 8             c0 := 1 ; 9             { resto sentencias } 10        end 11    end </pre>	<pre> 1 process P1 ; 2 begin 3     while true do begin 4         repeat 5             c1 := 1-c0 ; 6             until c0 != 0 ; 7             { seccion critica } 8             c1 := 1 ; 9             { resto sentencias } 10        end 11    end </pre>

## 14

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

## 15

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

<pre>var x, y : integer ;</pre>		
<pre>{ accede a 'x' } process P1 ; begin   while true do begin     x := x+1 ;     { .... }   end end</pre>	<pre>{ accede a 'x' e 'y' } process P2 ; begin   while true do begin     x := x+1 ;     y := x ;     { .... }   end end</pre>	<pre>{ accede a 'y' } process P3 ; begin   while true do begin     y := y+1 ;     { .... }   end end</pre>

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).

## 16

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.

## 17

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

- a)  $P_2$  podrá pasar a ejecutar  $e$  solo si  $P_1$  ha ejecutado  $a$  o  $P_3$  ha ejecutado  $g$ .
- b)  $P_2$  podrá pasar a ejecutar  $e$  solo si  $P_1$  ha ejecutado  $a$  y  $P_3$  ha ejecutado  $g$ .
- c) Solo cuando  $P_1$  haya ejecutado  $b$ , podrá pasar  $P_2$  a ejecutar  $e$  y  $P_3$  a ejecutar  $h$ .
- d) Sincroniza los procesos de forma que las secuencias  $b$  en  $P_1$ ,  $f$  en  $P_2$ , y  $h$  en  $P_3$ , sean ejecutadas como mucho por dos procesos simultáneamente.



```
{ variables globales }
```

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

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

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

18

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).

- Sincronizar los procesos para que  $P_1$  use todos los valores  $x$  suministrados por  $P_2$ .
- Sincronizar los procesos para que  $P_1$  utilice un valor sí y otro no de la variable  $x$ , es decir, utilice los valores primero, tercero, quinto, etc...

```
{ variables globales }
```

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

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

19

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

20

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 admite 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 posibles procesos esperando usar dos ejemplares, frente a los que esperan usar solo uno de ellos.

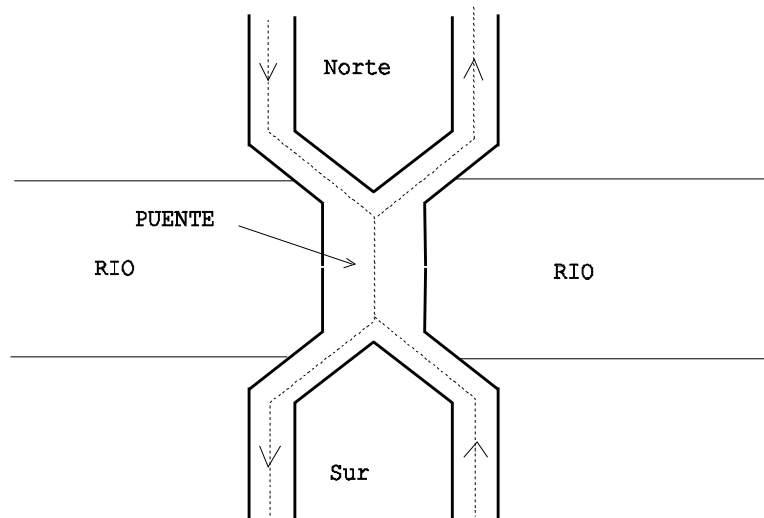
## 21

Escribir una solución al problema de *lectores-escriptores* 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 particular, 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.

## 22

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 tráfico 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.

## 23

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 sincronización 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.

## 24

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.

## 25

Los procesos  $P_1, P_2, \dots, P_n$  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**.

## 26

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.

## 27

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 continúan), 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
```

## 28

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  $i + 1 \bmod 5$ .

El monitor **MonFilo** exportará dos procedimientos: **coge\_tenedor**(num\_tenedor,num\_proceso) y **libera\_tenedor**(num\_tenedor,num\_proceso) para indicar que un proceso filósofo desea coger un tenedor determinado.

El código del programa (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 evite el interbloqueo descrito, para ello, esta solución no debe permitir que haya más de cuatro filósofos simultáneamente intentado coger su primer tenedor

