Sistemas Concurrentes y Distribuidos:
Enunciados de Problemas.

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

Curso 2023-24

Universidad de Granada

Índice general

Tema 1

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
```

2

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

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

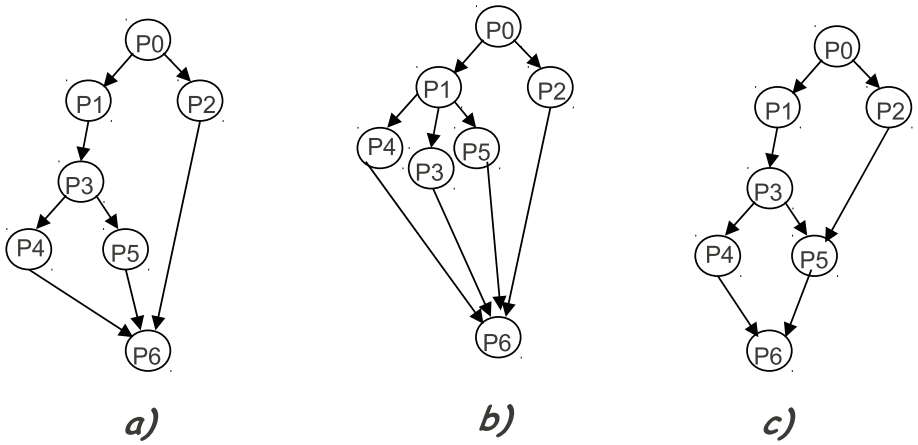
Para ilustrar como se accede a los archivos, aquí se encuentra una versión secuencial del código que copia **f** sobre **g**:

```
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
```

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 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).

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
```

```
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

$$T_m(p) \;=\; p$$

.

7

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

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

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

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

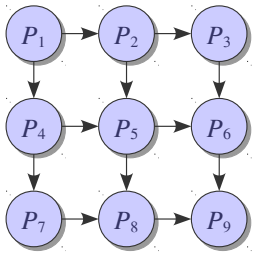
8

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

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 }
```

```
Process P0 ;
begin
   while true do begin
      { protocolo de entrada: }
      b0 := true ;          {indica quiere entrar}
      while b1 do begin     {si el otro también: }
         b0 := false ;      {cede temporalmente
         while b1 do begin end {espera     }
         b0 := true ;       {vuelve a cerrar paso}
      end
      { seccion critica  ....  }
      { protocolo de salida }
      b0 := false ;
      { resto sentencias  ....  }
   end
end
```

```
process P1 ;
begin
   while true do begin
      { protocolo de entrada: }
      b1 := true ;          {indica quiere entrar}
      while b0 do begin     {si el otro también: }
         b1 := false ;      {cede temporalmente }
         while b0 do begin end {espera     }
         b1 := true ;       {vuelve a cerrar paso}
      end
      { seccion critica  ....  }
      { protocolo de salida }
      b1 := false ;
      { resto sentencias  ....  }
   end
end
```

11

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 ;
```

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

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

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.

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
```

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.

```
{ 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
```

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:

```
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).

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 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
```

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).

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

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

```
{ variables globales }
```

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

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

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 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.

21

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

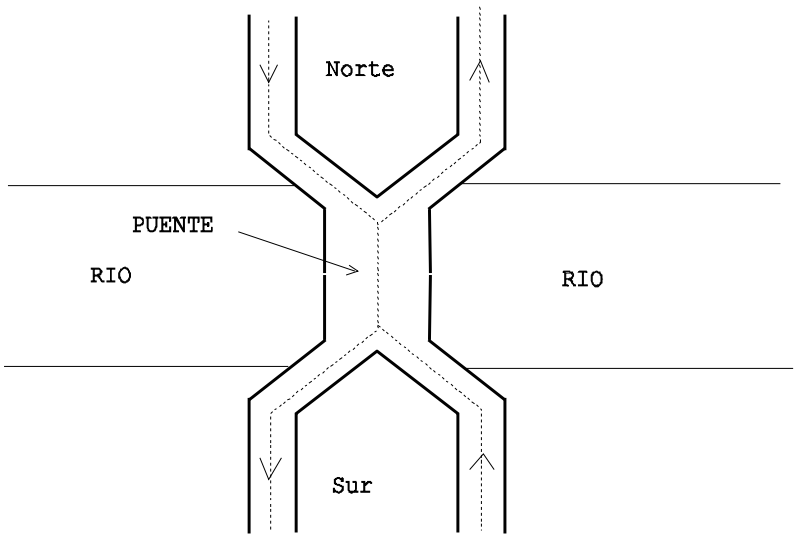
a) Con prioridad a los lectores. Quiere decir que, si en un momento puede acceder al recurso tanto un lector como un escritor, se da paso preferentemente al lector.

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.

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 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.

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 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.

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,...,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 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
```

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 $i+1 \mod 5$.

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

Sistemas basados en paso de mensajes.

29

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.

30

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** .

31

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`.

32

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
```

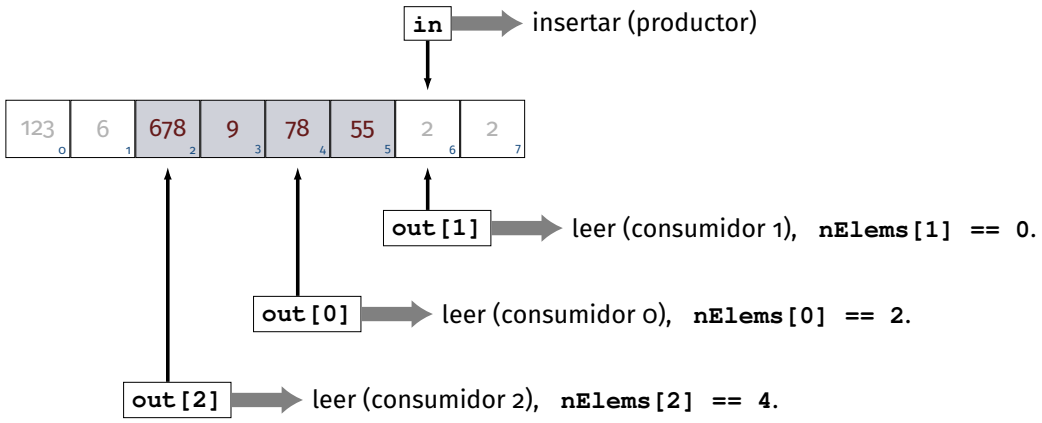
Describir en pseudocódigo el comportamiento del proceso Buffer, utilizando una orden de espera selectiva que permita implementar la sincronización requerida entre los procesos.

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

33

Suponer un proceso productor y 3 procesos consumidores que comparten un buffer acotado de tamaño **B**. Cada elemento depositado por el proceso productor debe ser retirado por todos los 3 procesos consumidores para ser eliminado del buffer. Cada consumidor retirará los datos del buffer en el mismo orden en el que son depositados, aunque los diferentes consumidores pueden ir retirando los elementos a ritmo diferente unos de otros. Por ejemplo, mientras un consumidor ha retirado los elementos 1, 2 y 3, otro consumidor puede haber retirado solamente el elemento 1. De esta forma, el consumidor más rápido podría retirar hasta **B** elementos más que el consumidor más lento.

Describir en pseudocódigo el comportamiento de un proceso que implemente el buffer de acuerdo con el esquema de interacción descrito usando una construcción de espera selectiva, así como el del proceso productor y de los procesos consumidores. Comenzar identificando qué información es necesario representar, para después resolver las cuestiones de sincronización. Una posible implementación del bufffer mantendría, para cada proceso consumidor, el puntero de salida (**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:



34

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
```

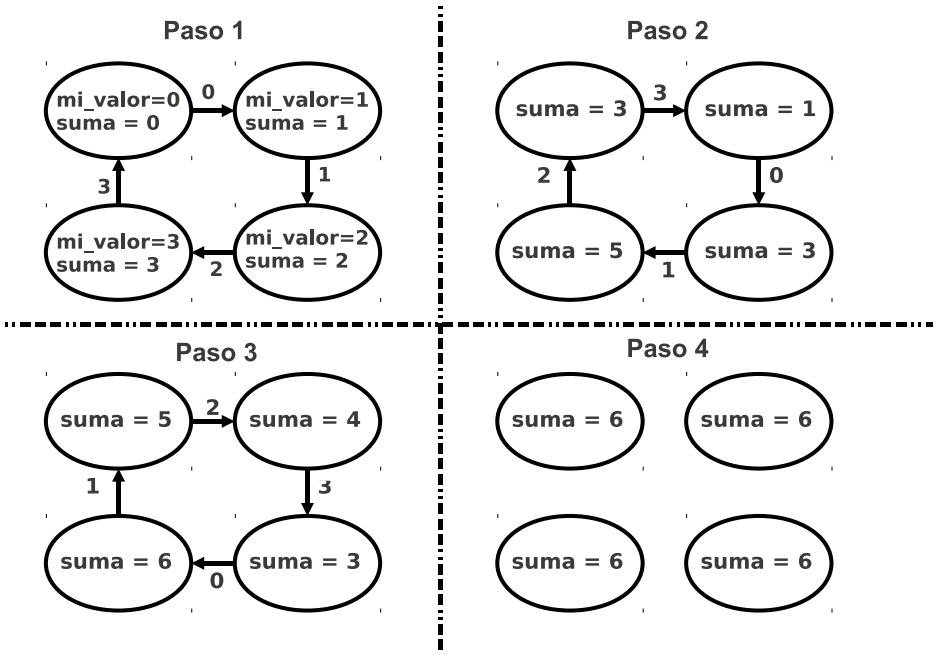
Implementar los procesos salvajes y cocinero usando paso de mensajes, usando un proceso olla que incluye una construcción de espera selectiva que sirve peticiones de los salvajes y el cocinero para mantener la sincronización requerida, teniendo en cuenta que:

- 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.

35

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
```

36

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
```

37

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`.

38

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
```

39

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.ż 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
```

40

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**

41

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`**

42

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).

43

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

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

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

44

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 podría estar consumiendo mucha memoria en el SPM por mensajes en tránsito pendientes cuya recepción se ve retrasada.

Escribe una solución en la cual cada proceso inicia sus envíos y recepciones y después espera a que se completen todas las recepciones antes de iniciar el primer consumo de un bloque recibido. De esta forma todos los mensajes pueden transferirse potencialmente de forma simultánea. Se debe intentar que la transimisión y las producción de bloques sean lo más simultáneas posible. 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.

45

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

46

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

47

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

48

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.

49

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

50

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

51

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