2.ª Edición corregida

# PROGRAMACIÓN CONCURRENTE

Jorge E. Pérez Martínez

*Editorial Rueda*
PORTO CRISTO, 13 - ALCORCON (Madrid)
APARTADO 43.001 - TELEFONO 619 27 79

*«El segundo, dividir cada una de las dificultades que examinare, en cuantas partes fuere posible y en cuantas requiriese su mejor solución.»*

«Discurso del Método»

René Descartes

# *Prólogo a la segunda edición*

*Este texto es una introducción a la programación concurrente y su problemática asociada, sin prestar especial atención a ningún lenguaje de programación en particular.*

*El libro está centrado en la descripción de los principios y metodologías de la programación concurrente, en los problemas que genera la ejecución en paralelo de procesos y en las técnicas y herramientas existentes para afrontar tales problemas.*

*Es cierto que los algoritmos están escritos en un seudopascal y que las herramientas utilizadas son un conjunto particular de las descritas por Brinch Hansen [BriH73]. Sin embargo, tal lenguaje y tales herramientas son recursos hacia la generalidad. Por una parte, el lenguaje Pascal es un exponente de la programación estructurada, y por otro lado, los constructores descritos por Brinch Hansen pueden utilizarse para implementar prácticamente todos los mecanismos de concurrencia y sincronización que aparecen en los lenguajes concurrentes de alto nivel.*

*El capítulo 0 está orientado a introducir el concepto de concurrencia y de ejecución no determinista. De ahí surge la necesidad de la verificación formal de los programas, así como de obtener sistemas funcionales.*

*En el capítulo 1 se describe el problema de la exclusión mutua y varios intentos de solución, utilizando variables compartidas. Se termina con el algoritmo de Dekker.*

*El capítulo 2 está dedicado íntegramente a la descripción sintáctica y semántica de las herramientas definidas por Brinch Hansen para gestionar la concurrencia. Se habla de la región crítica como mecanismo para conseguir la exclusión mutua; de los semáforos como elementos de sincronización; de las regiones críticas condicionales para manejar la espera condicional; de los buzones para gestionar la comunicación entre procesos; de los sucesos como herramienta que permite manejarla puesta encola de procesos, y, por último, de los monitores, como exponente de constructor de alto nivel, que permite tanto la exclusión mutua como la sincronización entre procesos. Al final se comenta un mecanismo de sincronización/comunicación presente en algunos lenguajes actuales de programación concurrente, como CSP o Ada, y que es el rendez-vous.*

*El capítulo 3 está dedicado al tema de los interbloqueos, verdadera «bestia negra» de la programación concurrente. Se analizan sus causas, así como métodos y estrategias para evitarlos o, en todo caso, detectarlos y eliminarlos.*

*En el capítulo 4 se da una perspectiva de los lenguajes de programación concurrente actuales. Se hace un repaso de las herramientas de las que disponen estos lenguajes para gestionar la concurrencia.*

*En el capítulo 5 se presentan resueltos y discutidos seis problemas de interés, algunos de ellos muy típicos en programación concurrente.*

*La sección 6 presenta enunciados de problemas a resolver. Los problemas planteados corresponden a ejercicios propuestos en la asignatura de Sistemas Operativos. Cada problema está caracterizado con un número entre corchetes que indican su nivel aproximado de dificultad, según la siguiente escala.*

*[1] Un problema que se puede resolver en menos de 20 minutos.*
*[2] Un problema que se puede resolver entre 20 y 40 minutos.*
*[3] Un problema que se puede resolver entre 40 y 60 minutos.*
*[4] Un problema que se puede resolver entre 60 y 120 minutos.*

*Estas cifras de tiempo son orientativas y pueden servir como marco de referencia a la hora de medir la complejidad de un problema.*

*A lo largo de todo el texto, también están planteados problemas tipo en relación con la materia tratada. Creemos que de esta manera se ameniza la lectura y se presenta rápidamente una aplicación sobre la cuestión en estudio.*

# *Agradecimientos*

*Este libro constituye la recopilación de las experiencias del autor en la docencia de sistemas operativos y más concretamente en programación concurrente.*

*La finalidad al escribirlo es la de proporcionar al alumno un soporte escrito en relación con las enseñanzas teóricas y prácticas sobre programación concurrente.*

*Quiero agradecer la colaboración prestada por Luis Cearra Zabala, que ha ideado una parte importante de los enunciados de problemas aquí propuestos. Asimismo, quiero agradecer su colaboración y ayuda a Eduardo Pérez Pérez. Miguel Angel de Miguel Cabello, Norberto Cañas de Paz, y a Isabel Muñoz Fernández, así como a todos los alumnos que con sus comentarios ayudaron a confeccionar este libro. No quiero dejar de mencionar a Luis Redondo López, que ha elaborado las secciones relacionadas con el lenguaje Ada.*

*El autor*

# Índice

*PROGRAMACION CONCURRENTE*

# *0. Introducción*

## 0.1. Definición de concurrencia

Los ordenadores actuales de tamaño medio y grande (y algunos micros) llevan sistemas operativos multitarea, que permiten la ejecución concurrente de varios procesos. Pero, ¿qué entendemos por concurrencia? Intuitivamente todos tenemos en la mente alguna definición del tipo: «simultaneidad de hechos». En la vida diaria podemos ver multitud de acciones realizándose concurrentemente: coches que circulan por una calle, gente paseando, semáforos que cambian de color, etc. Y todo ello ocurre al mismo tiempo.

Más formalmente podemos decir que dos procesos, Pl y P2, se ejecutan concurrentemente si la primera instrucción de PI se ejecuta entre la primera y la última instrucción de P2. La ejecución concurrente puede ser real o simulada. En concurrencia real, las instrucciones de ambos procesos se solapan en el tiempo (Fig. la); mientras que en concurrencia simulada (o seudoparalelismo), las instrucciones de P1 y P2 se intercalan en el tiempo (Fig. lb).

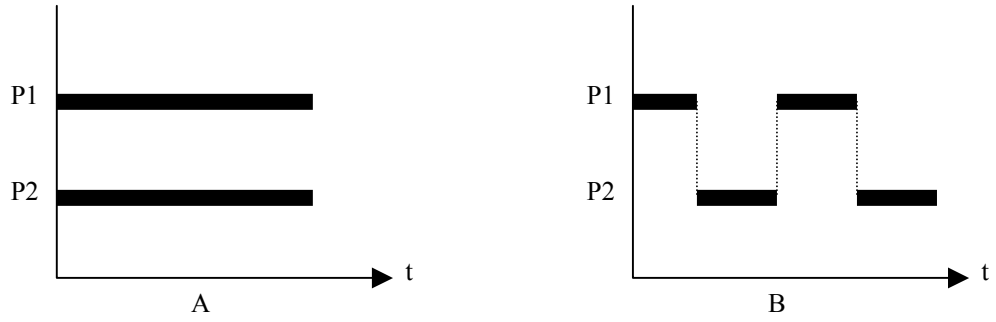



**FIGURA 1**

(pg14)Para darse la primera es necesario que existan tantos procesadores físicos como procesos. En seudoparalelismo, el número de procesos es mayor que el de procesadores, por lo que estos últimos se deben multiplexar en el tiempo entre los procesos.

Conviene ahora preguntarse por qué un sistema operativo implementa mecanismos para permitir la ejecución concurrente de procesos (aunque ello suponga un incremento en la complejidad del mismo). Existen varias razones para ello, aunque quizá la más significativa sea la económica. Es más barato que varios usuarios utilicen al mismo tiempo una instalación que lo hagan de uno en uno. De hecho, un solo usuario suele infrautilizar la instalación. Por ello, durante estos tiempos en los que el procesador no hace nada útil, es aconsejable que existan otros procesos que ejecutar. Evidentemente, esto repercute en un ahorro de tiempo a la hora de ejecutar varios trabajos. Un buen ejemplo de ello es la ejecución concurrente de un proceso que imprime un fichero y de otro que resuelve un sistema de ecuaciones. Otra razón para desear la programación concurrente es la claridad conceptual. Si existen dos actividades que se desarrollan en paralelo, su programación es más clara y sencilla como dos programas independientes.

Como ejemplo de concurrencia en la vida cotidiana en el que se produce ahorro de tiempo, considere el siguiente programa para hacer una tortilla francesa:

**PROGRAM** tortillaFrancesa;

```
BEGIN
  pon aceite en la sartén;          «a»
  prepara dos huevos;               «b»
  saca un plato;                    «c»
  enciende el fogón;                «d»
  COBEGIN
      calientaAceite;               «e»
      bateHuevos                    «f»
  COEND;
  pon batido en sartén;             «g»
  espera y mueve;                   «h»
  saca tortilla;                    «i»
END.
```

Los procesos calientaAceite y bateHuevos se desarrollan concurrentemente, lo que evidencia un ahorro de tiempo. Para expresar dónde comienza y dónde termina una ejecución concurrente, se utilizarán las palabras **COBEGIN y COEND.** Todos los procesos entre estas palabras se ejecutarán concurrentemente. La siguiente sentencia a **COEND** (en el caso anterior, la (pg15) etiqueta con «g») tan solo se ejecutará cuando hayan terminado todos y cada uno de los procesos entre **COBEGIN** y **COEND**.

Otro ejemplo que ilustra el ahorro de tiempo por ejecución concurrente es el siguiente algoritmo paralelo para ordenar un vector de números de menor a mayor. Supongamos que el vector a ordenar es el siguiente:

7 3 5

El algoritmo tiene las siguientes fases:

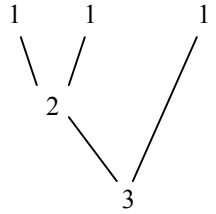
Fase 1: Se comparan todos los números entre sí. Si existen N números, necesitamos $N^2$ procesadores, puesto que cada procesador realiza una comparación. El resultado de la misma se anota en un enrejado como el mostrado en la Fig. 2.

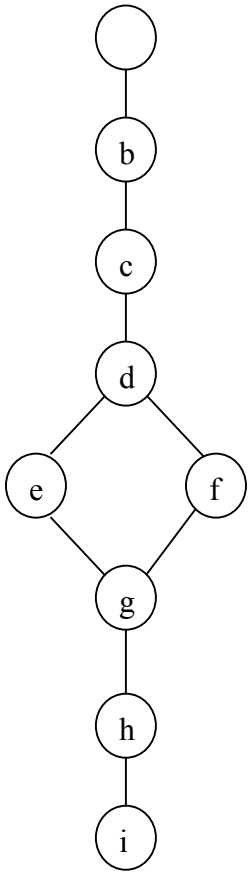
| | **7** | **3** | **5** |
|---|---|---|---|
| **7** | **1** | **1** | **1** |
| **3** | **0** | **1** | **0** |
| **5** | **0** | **1** | **1** |

FIGURA 2

Si al comparar un número de una fila con uno de una columna, el primero es mayor o igual que el segundo, anotamos un 1 en su casilla correspondiente y en caso contrario, anotamos un 0. El tiempo total empleado es independiente de N y es igual al tiempo que necesita un procesador para realizar una comparación y anotar el resultado de la misma.

Fase 2: Se suman los valores de cada fila. La suma de los valores de una Fila se puede realizar concurrentemente como se indica a continuación:






(pg16)

FIGURA 3

Puesto que existen N filas (cuya suma discurrirá en paralelo) y en cada fila se precisan N/2 procesadores, necesitamos $N^2/2$ procesadores para esta fase (número que es menor que los necesarios para la primera). El tiempo total empleado en esta fase es proporcional al $\log_2 N$.

Fase 3: Los valores obtenidos en la fase anterior indican la posición de los números en el vector ordenado. Ahora sólo resta llevarlos a su posición correcta. Con N procesadores se pueden llevar todos a la vez y el tiempo total empleado para esta fase es el que emplea un procesador en llevar un número de una posición de memoria a otra (tiempo que es independiente de N).

Con este algoritmo se consigue un ahorro significativo de tiempo en comparación con los algoritmos secuenciales de ordenación a costa de incrementar el número de procesadores.

### 0.2. Diagramas de precedencia

Volvamos ahora al ejemplo anterior de la tortilla francesa. Podemos representar mediante un grafo el orden de ejecución de las sentencias tal y como se muestra en la Fig. 3.

(pg17)

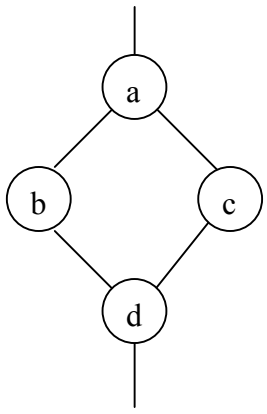



FIGURA 4

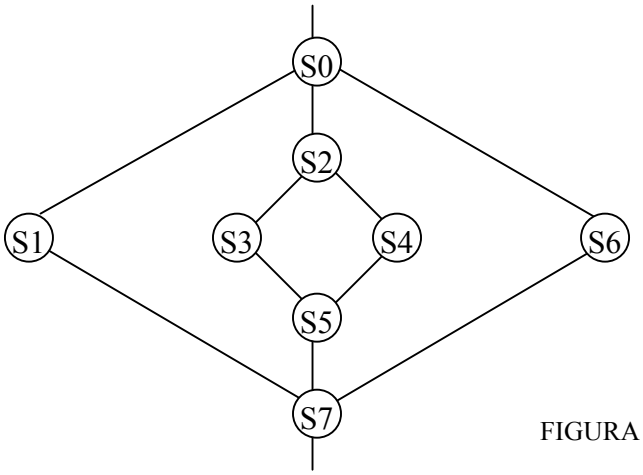
Esta representación se **conoce como diagrama o grafo de precedencia.** En él se observa la secuencialidad de las sentencias «a» a «d» y la concurrencia en «e» y «f». También se aprecia claramente que la sentencia «g» sólo se ejecutará cuando hayan terminado «e» y «f».

El diagrama de precedencia del siguiente programa se ilustra en la Fig. 4.

```
PROGRAM Ejemplo;
   VAR a, b, c, d, e, f.     INTEGER
   BEGIN
     Readln (a, b, c, d);        «a»
```

```
    COBEGIN
        e:= a + b;          «b»
        f:=c-d              «c»
    COEND;
    Writeln (e * f)         «d»
END.
```

Operando a la inversa, podemos obtener la estructura del programa que corresponde a un diagrama de procedencia dado. El programa que corresponde al diagrama de la Fig. 5 es:




(pg18) FIGURA 5

```
SO;
COBEGIN
  S1;
  BEGIN
    S2;
    COBEGIN
          S3; S4
    COEND;
    S5
  END;
  S6
COEND;
S7
```

Observe que si se suprimiese el BEGIN-END, S5 se ejecutaría concurrentemente con S3 y S4, y eso no es lo que se indica en el diagrama.

## 0.3. Indeterminismo

A diferencia de la programación secuencial en la que al finalizar un conjunto de operaciones podemos predecir el resultado, en programación concurrente esto no es cierto. El resultado dependerá de la velocidad relativa de los procesos. Vea el siguiente ejemplo:

```
a:= 0;
FOR i:= 1   TO    10 DO a:= a + 1;
```

Se sabe que al terminar el bucle FOR, el valor de «a» es 10.

Ahora considere el siguiente código:

```
a:= 0;
COBEGIN
    FOR i:= 1   TO    5   DO    a:= a + 1;
    FOR j:= 1   TO    5   DO    a:= a + 1
COEND;
```

¿Qué valor tendrá la variable «a» al terminar la ejecución? No está definido. Puede valer 5, 6, 7, 8, 9 ó 10. Si pensamos que existen dos procesadores físicos (y razonamiento similar se aplica si existe uno solo, que está multiplexado en el tiempo entre los dos procesos), puede ocurrir el siguiente escenario:

1) Los dos acceden al mismo tiempo al valor de a (0).(pg19)
2) Los dos realizan la suma de «a + 1» y dejan el resultado en su acumulador.
3) Los dos llevan al mismo tiempo ese resultado a la posición de memoria ocupada por «a».

Se ha obtenido 1 cuando el resultado debería de ser 2. Además puede ocurrir que ejecutamos el programa 1.000 veces y se obtenga siempre 10 y a la ejecución 1.001 se obtenga el valor 8.

Esta característica de **entorno no reproducible** hace muy difícil encontrar errores en los programas concurrentes. En el ejemplo presentado, el error se encuentra en un acceso no controlado a una variable («a») compartida por dos procesos. En capítulos sucesivos estudiaremos diferentes formas de resolver este problema.

## 0.4. Aserciones

La verificación de la corrección de un programa secuencial es tarea ardua. No basta con someter al programa a una adecuada batería de pruebas (esto sólo probará que existen errores, pero no su ausencia). Y si para los programas secuenciales es una tarea difícil, qué decir acerca de los concurrentes. De momento, nos introduciremos en la comprobación de la corrección de un programa secuencial. Más adelante aplicaremos esta técnica a programas concurrentes.

La técnica consiste en establecer aserciones antes y después de una operación. Dichas aserciones definen el estado del cálculo en ese momento. Como ejemplo considere la operación «sort» que ordena los N elementos de un array A de enteros en una secuencia creciente:

«A: array [l..n] OF INTEGER»
  sort (A)
«Para todo i, j: 1..n → [i ≤ j => A(i) ≤ A(j)]

Otro ejemplo lo podemos obtener de la multiplicación de dos números «x» e «y» por sumas sucesivas.

```
a:= x; b:= y; c:= 0;
{a*b+c=x*y & b # 0}
WHILE (b # 0) DO BEGIN
    {a*b+c=x*y}
    { b#0}
    b:= b - l;
    c:=c+a
END;
{a*b+c=x*y & b=0 → c=x*y}(pg20)
```

## 0.5. Sistemas funcionales

Se dice que un sistema es funcional si exhibe un comportamiento independiente respecto del tiempo. Es decir, no se hace suposición alguna sobre las velocidades de progreso de los procesos que lo componen.

Como ejemplo de un sistema cuyo comportamiento es independiente del tiempo, considere el siguiente programa que copia una secuencia de entrada (f) en una secuencia de salida (g).

```
PROCEDURE copy (VAR f, g: SEQUENCE OF T);
    VAR s,t: T;
          acabada: BOOLEAN;
    BEGIN
      IF NOT vacia (f)
      THEN BEGIN
              acabada:= FALSE;
              toma (s, f);
              REPEAT
                t:= s;
                COBEGIN
                   pon (g, t);
                   IF vacia (f)
                   THEN acabada:= TRUE
                   ELSE toma (s, f)
```

```
        COEND
      UNTIL acabada
      END
END.
```

El diagrama de precedencia de este programa es el mostrado en la Fig. 6. En el algoritmo anterior, inicialmente «f» está llena y «g» vacía:

f = (1, 2, ..., m)  g = ( )

Cuando se ha completado la copia, «g» contiene todos los registros en el orden original, mientras que «f» está vacía: f=( )  g=(1,2,...,m)

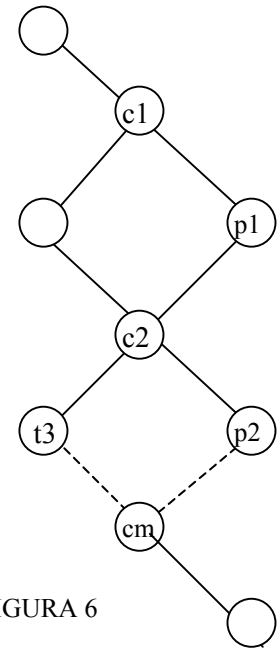



FIGURA 6

(pg21)

El núcleo del algoritmo es la repetición de los procesos de copia, pon y toma:

```
REPEAT
   copia;
   COBEGIN
      pon;
      toma
   COEND
UNTIL acabada
```

Este algoritmo es correcto. Pero supongamos que por equivocación hemos escrito:

```
REPEAT
   COBEGIN
      copia;
      pon;
      toma
   COEND
UNTIL acabada
```

Si se realiza primero la copia y luego la e/s, el proceso es correcto. Si se realiza la salida antes de que se inicie la copia, duplicaremos un registro. Y si se realiza la entrada antes de que se inicie la copia y la copia se completa antes de que se inicie la salida, perderemos un registro. Es decir, si suponemos (pg22) que el 1.° registro se copia correctamente de « f» a «g», el cálculo estará en el siguiente estado.

s = 2;  f = (3, 4, ..., m)
t = 1;  g = (1)

Y la secuencia descrita anteriormente dará:

COPIA, PON, TOMA → G = (1, 2)
COPIA, TOMA, PON → G = (l, 2)
PON, COPIA, TOMA → G = (1, 1)

PON, TOMA, COPIA → G = (1, 1)
TOMA, COPIA, PON → G = (l, 3)
TOMA, PON, COPIA → G = (1, 1)

Vemos, pues, que el sistema no es funcional en cuanto que su salida, para una misma entrada, depende del tiempo (de la velocidad relativa de los procesos).

Si volvemos al algoritmo correcto de copiado, vemos que éste sí exhibe comportamiento funcional. Y ello se debe a que los procesos concurrentes que en él intervienen son **procesos disjuntos** o no interactivos. Dos procesos son disjuntos si operan sobre conjuntos disjuntos de variables. El proceso «pon» opera sobre (t, g), mientras que el proceso «toma» lo hace sobre (s, f, acabada); siendo $(t,g) \cap (s,f,acabada) = (\emptyset)$. En la versión errónea de este algoritmo los procesos no son disjuntos.

copia (s, t)
pon: (t, g)
toma: (s, f, acabada)

El proceso de salida hace referencia a la variable «t» cambiada por el proceso de copia y el proceso de copia hace referencia a la variable «s» cambiada por el proceso de entrada. Cuando un proceso hace referencia a una variable cambiada por otro proceso, es inevitable que el resultado del primer proceso dependa del momento en que el segundo proceso hace una asignación a esa variable.

Por ello, la disyunción es una condición suficiente para el comportamiento independiente del tiempo de los procesos concurrentes. Poniendo ciertas restricciones a un lenguaje de programación, el compilador será capaz de comprobar si los procesos concurrentes son disjuntos o no.

Ahora bien, en algunos casos los procesos concurrentes deben de ser capaces de acceder y cambiar variables comunes. La condición suficiente que debe cumplir un sistema para exhibir un comportamiento funcional es la condición de consistencia: «si a un programa se le suministran 2 entradas(pg23) diferentes X y X', tal que X está contenida en X', entonces se obtiene la misma relación entre las salidas Y e Y'».

$X \leq X' \rightarrow Y \leq Y'$

Si se ejecuta un programa consistente dos veces con la misma entrada, da la misma salida en ambos casos.

$$X=X' \rightarrow X \leq X' \;\&\; X' \leq X \rightarrow Y \leq Y' \;\&\; Y' \leq Y \rightarrow Y = Y'$$

No nos extenderemos más en este punto. A los lectores interesados se les remite a la bibliografía [BriH73].

# *Capítulo 1*

# *Exclusión mutua*

## 1.1. Definición del problema

Sean dos procesos que comparten un recurso. El acceso a dicho recurso debe ser exclusivo. Es decir, cuando un proceso lo esté utilizando el otro proceso no puede acceder a él. Considere, por ejemplo, que el recurso es un armario de cinta y que dos procesos, Pl y P2, intentan acceder simultáneamente a la cinta. De forma esquemática el ciclo de vida de los procesos es:

```
«Pl»                    «P2»
  REPEAT                  REPEAT
    usa cinta;              usa cinta;
    otras cosas             otras cosas
  FOREVER                 FOREVER
```

Debemos arbitrar algún mecanismo para conseguir la exclusión mutua de los procesos respecto a la cinta.

En lo que sigue utilizaremos las aproximaciones de Ben-Ari [BenA82] para resolver el problema de la exclusión mutua.

(pg26)

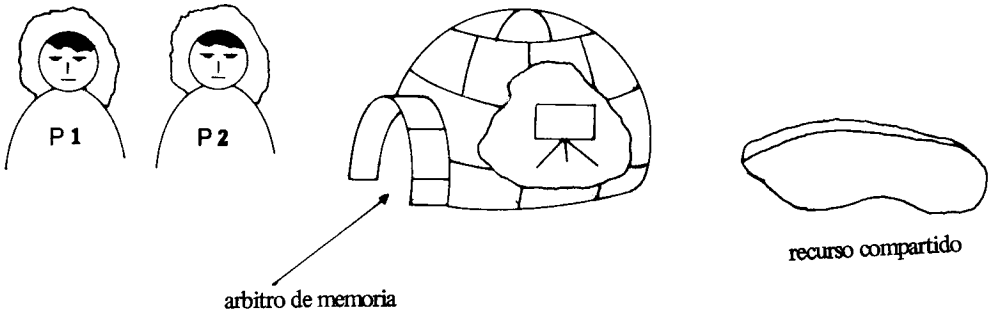



FIGURA 7

## 1.2. 1.º intento: «alternancia»

En una primera aproximación al problema considere que los dos procesos están representados por dos esquimales. Existe un iglú, dentro del cual hay una pizarra. La entrada al iglú es tan estrecha que tan sólo cabe un esquimal por ella. De la misma manera, cuando un esquimal esté dentro del iglú, el otro no puede entrar. El recurso compartido por los dos esquimales es un agujero en el hielo para pescar. El agujero está a medio día de marcha del iglú (Fig. 7). Cuando el esquimal 1 quiere utilizar este agujero, va primero al iglú y mira qué hay anotado en la pizarra. Si hay un 2, significa que el otro esquimal está utilizando el agujero. Por ello empieza a pasearse y de vez en cuando entra al iglú a comprobar el valor de la pizarra. Cuando se encuentre que en ella hay un 1, significa que ya puede pescar, porque el otro esquimal ha dejado de hacerlo. Por tanto, se va a pescar; está un rato y cuando termina, va al iglú y pone en la pizarra un 2, indicando que el otro esquimal ya puede pescar.

La anterior situación se puede programar como:

```
PROGRAM PrimerIntento;
   VAR turno: [1..2];
   PROCEDURE P1;
     BEGIN
       REPEAT
          WHILE turno = 2 DO;        (* pasea *)
          ****** usa el agujero de pesca ******
          turno:= 2;
          otras cosas
       FOREVER
     END;
(pg27)
   PROCEDURE P2;
     BEGIN
       REPEAT
         WHILE turno = 1 DO;      (* pasea *)
         ****** usa el agujero de pesca ******
         turno:= 1;
         otras cosas
       FOREVER
     END;

BEGIN
   turno:= 1;
   COBEGIN
     P1; P2
   COEND
END.
```

En el paradigma anterior los esquimales son los procesos, el agujero de pesca el recurso compartido y la pizarra una variable que los procesos utilizan para saber si pueden o no utilizar el recurso. El iglú es una manera de indicar que existe un árbitro de memoria para controlar los accesos simultáneos a la variable. En el caso de que los dos procesos intentasen acceder simultáneamente a la variable, la entrada al iglú permitirá que pase uno u otro, pero no los dos.

Con la solución propuesta se consigue la exclusión mutua. Es fácil comprobar que cuando turno=l el proceso 2 no puede utilizar el recurso y viceversa.

A pesar de que la solución satisface la exclusión mutua tiene el problema de que los accesos al recurso se deben de producir de una manera estrictamente alternada: PI, P2, P1, P2, etc. Piense en lo que ocurriría si al esquimal 1, mientras da una vuelta esperando utilizar el agujero de pesca, se lo come un oso polar. Ello significaría que el esquimal 2, después de haber pescado, ya nunca lo podrá hacer de nuevo. De la misma forma, si un proceso debe acceder al recurso más frecuentemente que el otro se encontrará que no puede hacerlo.

## 1.3. 2.º intento: «falta de exclusión»

El problema de la alternancia se ha producido por utilizar una única variable para sincronizar ambos procesos. Para evitarlo utilizaremos dos variables. (pg28) Para ello, y siguiendo con el paradigma anterior, utilizaremos dos iglús con una pizarra en cada uno de ellos (que llamaremos pizarral y pizarra2) (Fig. 8). En las pizarras figurará o bien la palabra `pescando' o bien `NOpescando' indicándose con ello que el esquimal dueño del iglú está utilizando o no el agujero. Cuando el esquimal 1 quiera pescar va al iglú del esquimal 2, mira la pizarra y si está escrito `pescando' significa que el esquimal 2 está pescando y que por lo tanto debe esperar (dándose un paseo). Cada cierto tiempo entrará de nuevo al iglú del esquimal 2 para comprobar la pizarra. Cuando se encuentre en ella la palabra `NOpescando' significa que ya puede pescar. Por ello va a su iglú y pone en la pizarra la palabra `pescando' notificando con ello al esquimal 2 que él está pescando. Cuando acabe de hacerlo volverá a su iglú y escribirá NOpescando' en la pizarra.

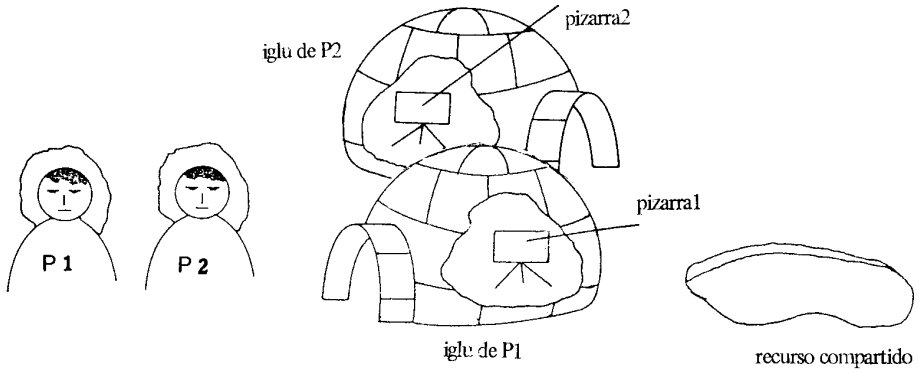



FIGURA 8

El programa que implementa este algoritmo es el siguiente:

```
PROGRAM SegundoIntento;
   VAR pizarral, pizarra2: (pescando, NOpescando);
   PROCEDURE P1;
      BEGIN
         REPEAT
            WHILE pizarra2 = pescando DO;      (* pasea *)
            pizarral:= pescando;
            ****** usa el agujero de pesca ******
            pizarral:= NOpescando;
            otras cosas
         FOREVER
      END;
(pg29)
   PROCEDURE P2;
      BEGIN
         REPEAT
            WHILE pizarral = pescando DO;     (* pasea *)
            pizarra2:= pescando;
            ****** usa el agujero de pesca ******
            pizarra2:= NOpescando;
            otras cosas
         FOREVER
      END;
BEGIN
   pizarral:= NOpescando;
   pizarra2:= NOpescando;
   COBEGIN
     P1; P2
   COEND
END.
```

El protocolo esbozado en el programa anterior parece correcto. Cuando un proceso quiere utilizar el recurso, primero se asegura que el otro proceso no lo está haciendo (consultando la pizarra del vecino). Una vez que ha utilizado el recurso, vuelve a su iglú e indica que ya ha terminado de hacerlo.

Sin embargo, este programa puede conducir a una falta de exclusión mutua. Considere el siguiente escenario:

1. el proceso Pl comprueba el valor de la variable pizarra2 (NOpescando) y sale del lazo while;
2. el proceso P2 comprueba el valor de la variable pizarral (NOpescando) y sale del lazo while;
3. el proceso Pl pone pizarral a pescando;
4. el proceso P2 pone pizarra2 a pescando;
5. el proceso PI accede al agujero de pesca;
6. el proceso P2 accede al agujero de pesca;

En esta situación se ha producido una falta de exclusión mutua. Puesto que puede producirse, la solución no es correcta.

## 1.4. 3.º intento: «interbIoqueo (espera infinita)»

La situación anterior se ha producido porque primero se mira en el iglú del vecino y luego se anota en el propio la situación. Para evitarlo, actuaremos(pg30) a la inversa; es decir, cuando un proceso quiera utilizar el recurso, primero indica que quiere hacerlo y luego se esperará hasta que quede libre. El programa que satisface este algoritmo es el siguiente:

```
PROGRAM TercerIntento;
   VAR pizarral, pizarra2: (pescando, NOpescando);
   PROCEDURE P1;
      BEGIN
         REPEAT
            pizarral:= pescando;
            WHILE pizarra2 = pescando DO; (* pasea *)
            ****** usa el agujero de pesca ******
            pizarral:= NOpescando;
            otras cosas
         FOREVER
      END;
   PROCEDURE P2;
      BEGIN
         REPEAT
            pizarra2:= pescando;
            WHILE pizarral = pescando DO;  (* pasea *)
            ****** usa el agujero de pesca ******
            pizarra2:= NOpescando;
            otras cosas
         FOREVER
      END;
BEGIN
   pizarral:= NOpescando;
   pizarra2:= NOpescando;
   COBEGIN
     P1; P2
   COEND
END.
(pg31)
```

Sin embargo, esta solución presenta un problema. Suponga el siguiente escenario. Inicialmente las dos pizarras contienen `NOpescando'.

1. P1 pone pizarral a `pescando';
2. P2 pone pizarra2 a `pescando';
3. Pl comprueba que la condición de su while es cierta y se queda ejecutando el bucle;
4. P2 comprueba que la condición de su while es cierta y se queda ejecutando el bucle.

Estamos por tanto en una situación no deseable. Cada proceso espera que el otro cambie el valor de su pizarra y ambos se quedan en una espera infinita. Ninguno de los dos puede continuar.

## 1.5. 4. intento: «espera indefinida»

Para solucionar este problema de espera infinita introduciremos un tratamiento de cortesía. Cuando un proceso ve que el otro quiere utilizar el recurso, le cede el turno cortésmente. El programa que implementa este algoritmo es el siguiente:

```
PROGRAM CuartoIntento;
   VAR pizarral, pizarra2: (pescando, NOpescando);
   PROCEDURE P1;
      BEGIN
         REPEAT
            pizarral:= pescando;
            WHILE pizarra2 = pescando DO
            BEGIN
               (* tratamiento de cortesía *)
               pizarral:= NOpescando;
               (* date una vuelta *)
               pizarral:= pescando
            END;
            ****** usa el agujero de pesca ******
            pizarral := NOpescando;
            otras cosas
         FOREVER
      END;
```

(pg32)

```
   PROCEDURE P2;
      BEGIN
         REPEAT
            pizarra2:= pescando;
            WHILE pizarral = pescando DO
            BEGIN
               (* tratamiento de cortesía *)
               pizarra2:= NOpescando;
               (* date una vuelta *)
               pizarra2:= pescando
            END;
            ****** usa el agujero de pesca ******
            pizarra2:= NOpescando;
            otras cosas
         FOREVER
      END;
   BEGIN
     pizarral:= NOpescando;
     pizarra2:= NOpescando;
     COBEGIN
          P1; P2
     COEND
   END.
```

En esta solución cuando el proceso i intenta acceder al recurso y comprueba que el otro también lo desea, cede su turno (pone pizarra-i a NOpescando), espera un tiempo, y luego reclama su derecho a utilizar el recurso.

Sin embargo, este tratamiento de cortesía puede conducir a que los procesos se queden de manera indefinida cediéndose mutuamente el paso. Si bien es difícil que se produzca indefinidamente, esta solución no asegura que se acceda al recurso en un tiempo finito.

## 1.6. Algoritmo de Dekker

Para evitar esta situación se utilizará a parte de las dos pizarras otra más que indicará, en caso de conflicto, a quién se le concede el acceso al recurso (Fig. 9). Llamaremos a esta nueva pizarra turno. La siguiente solución, debida a Dekker, es una combinación de los programas PrimerIntento y CuartoIntento anteriormente escritos.

(pg33)

El algoritmo de Dekker es el siguiente:

```
PROGRAM AlgoritmoDeDekker;
    VAR  pizarral, pizarra2: (pescando, NOpescando);
         turno: [1..2];
    PROCEDURE P1;
       BEGIN .
          REPEAT
             pizarral:= pescando;
             WHILE pizarra2 = pescando DO
                IF turno = 2
                THEN BEGIN
                   (* tratamiento de cortesía *)
                   pizarral:= NOpescando;
                   WHILE turno = 2 DO; (* date una vuelta *)
                   pizarral:= pescando
                   END;
                ****** usa el agujero de pesca ******
                turno:= 2;
                pizarral:= NOpescando;
                otras cosas
          FOREVER
       END;
    PROCEDURE P2;
       BEGIN
          REPEAT
             pizarra2:= pescando;
             WHILE pizarral = pescando DO
                IF turno = 1
                THEN BEGIN
                   (* tratamiento de cortesía *)
                   pizarra2:= NOpescando;
                   WHILE turno = 1 DO; (* date una vuelta *)
                   pizarra2:= pescando
                   END;
                ****** usa el agujero de pesca ******
                turno:= 1;
                pizarra2:= NOpescando;
                otras cosas
          FOREVER
       END;
(pg34)
    BEGIN
       pizarral:= NOpescando;
       pizarra2:=
    NOpescando;
       turno:= 1;
       COBEGIN
          P1; P2
       COEND
    END.
```

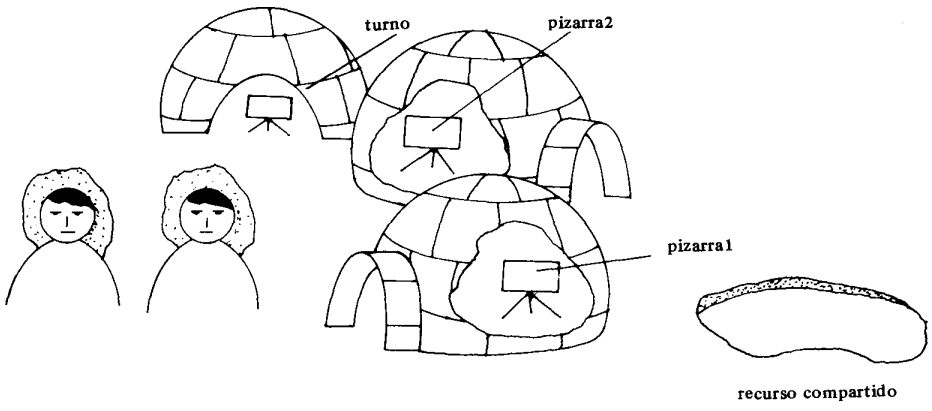



FIGURA 9

Es fácil comprobar que esta solución asegura la exclusión mutua y que está libre de interbloqueos y lockout.

Es importante recordar que este algoritmo sirve para conseguir la exclusión mutua entre dos procesos. El algoritmo utilizado para N procesos tan sólo tiene un interés teórico ya que consume demasiado tiempo para conseguir la exclusión mutua.

Por último indicar que muchos ordenadores proporcionan instrucciones que permiten comprobar y modificar el contenido de una palabra de memoria, o intercambiar el contenido de dos palabras en un solo ciclo de memoria (instrucciones atómicas o indivisibles). Se deja al lector que implemente la exclusión mutua utilizando estas instrucciones.

# *Capitulo 2*

# *Herramientas para manejar la concurrencia*

En este capítulo describiremos las herramientas propuestas por Brinch Hansen, en su libro «Operating System Principles», para controlar la ejecución concurrente de varios procesos.

Evidentemente no son las únicas que existen para tal fin. Es más, comercialmente no se utilizan. Existen lenguajes tales como el Pascal Concurrente, CCNPascal, Ada, Modula-2 y otros que tienen incorporadas herramientas de más alto nivel (monitores, clases, etc.) que permiten al programador manejar los procesos y su interacción.

La razón para describir los constructores de Hansen estriba, por un lado, en la claridad conceptual de los mismos y, por otro, en que existe en el Centro de Cálculo de la E.U.I. un compilador-intérprete (PSR3) que soporta estas herramientas. Esto permite que se puedan programar y ejecutar procesos concurrentes en una instalación real.

## 2.1. Región critica

Es frecuente en programación concurrente que varios procesos compartan una misma variable. Se debe evitar, por consistencia, que mientras un (pg36) proceso esté accediendo a una variable otro la esté modificando al mismo tiempo. Es decir, debemos conseguir la exclusión mutua de los procesos respecto a la variable compartida.

Los criterios de corrección para tal exclusión son los siguientes:

1. Exclusión mutua con respecto al recurso.
2. Cuando existan N procesos que quieran acceder al recurso, se le concede el acceso a uno de ellos en un tiempo finito.
3. Se libera el recurso en un tiempo finito.
4. No existe espera activa.

En el capítulo 1 vimos que un posible método para conseguir la exclusión mutua era utilizar el algoritmo de Dekker. Sin embargo, este algoritmo conlleva una espera activa de los procesos. Es decir, cuando un proceso está in tentando acceder a un recurso que ya ha sido asignado a otro proceso continúa consumiendo tiempo de procesador en su intento para conseguir el recurso. Esto provoca pérdidas de eficiencia en el procesador (incluso puede provocar interbloqueos en ciertas situaciones y con algoritmos de planificación por prioridades).

Brinch Hansen propuso un constructor, la **región crítica,** para conseguir la exclusión mutua. Su sintaxis es la mostrada en la Fig. 10:

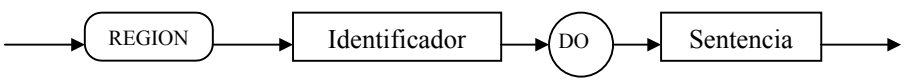



FIGURA 10

donde identificador es el nombre de una variable que debe haberse declarado como compartida tal y como se indica en la Fig. 11:

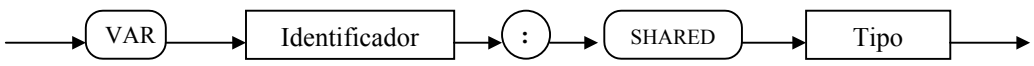



FIGURA 11

(pg37)

Por ejemplo:

```
.....
VAR v: SHARED INTEGER;
.....
PROCEDURE P1;
    BEGIN
        .....
        REGION v DO v:= v + 2;
        .....
    END;
.....
```

En el ejemplo anterior se declara una variable v como compartida. El acceso a dicha variable debe efectuarse siempre dentro de la región crítica (RC en adelante) asociada a v; de lo contrario, el compilador dará un mensaje de error.

La semántica de la RC establece que:

1. Los procesos concurrentes sólo pueden acceder a las variables compartidas dentro de sus correspondientes RC.
2. Un proceso que quiera entrar a una RC lo hará en un tiempo finito.
3. En un instante t de tiempo sólo un proceso puede estar dentro de una RC determinada. Esto no impide que, por ejemplo, un proceso Pl esté dentro de la RC asociada con la variable v y un proceso P2 dentro de la RC asociada con la variable w. Es decir, las RC que hacen referencia a variables distintas pueden ejecutarse concurrentemente.
4. Un proceso está dentro de una RC un tiempo finito, al cabo del cual la abandona.

Esto significa que:

1. Si el número de procesos dentro de una RC es igual a 0, un proceso que lo desee puede entrar a dicha RC.
2. Si el número de procesos dentro de una RC es igual a 1 y N procesos quiere entrar, esos N procesos deben esperar.
3. Cuando un proceso sale de una RC se permite que entre uno de los procesos que esperan.
4. Las decisiones de quien entra y cuando se abandona una RC se toman en un tiempo finito.
5. Se supone que la puesta en cola de espera es justa.

Queremos hacer notar que la **espera** que realizan los procesos es **pasiva**. Con esto se quiere decir que cuando un proceso intenta acceder a una RC y está ocupada, abandona el procesador en favor de otro proceso. Con esto se evita que un proceso ocupe el procesador en un trabajo inútil. Por otra parte (pg38) se supone que la puesta en cola es justa; es decir, no se retrasa a un proceso indefinidamente a la entrada a una RC.

Veamos a continuación algunos ejemplos de aplicación:

Ejemplo N.° 1: Considere el siguiente programa en el que dos procesos Pl y P2 comparten la variable x

```
PROGRAM Ejemplol; (* versión 1 *)
    VAR x: INTEGER;
    PROCEDURE P1;
        BEGIN
            x:=x+10
        END;
    PROCEDURE P2;
    BEGIN
        IF x > 100
        THEN Writeln (x)
        ELSE Writeln (x-50)
    END;
BEGIN
```

```
    x:= 100;
    COBEGIN
        P1; P2
    COEND
END.
```

¿Cuál es el valor de x después de ejecutar el programa anterior? Si nos fijamos en el texto del programa veremos que el único proceso que modifica el valor de x es el P1. Por tanto, al final x siempre valdrá 110. Ahora bien, ¿qué valor se imprime? La respuesta a esta pregunta es que el resultado está indeterminado y depende de las velocidades relativas de los procesos Pl y P2. La programación es no determinista. Si Pl se ejecuta totalmente antes de que lo haga P2 pondrá la variable x a 110, por lo que luego P2 escribirá 110. Si por el contrario P2 se ejecuta antes que P1, comprobará que x=100 y, por tanto, escribirá 50. Por otro lado, si Pl y P2 se ejecutan solapadamente (se dispone de 2 procesadores) el resultado dependerá de si Pl actualiza el valor de x antes o después de que P2 lo consulte. Puede incluso ocurrir la siguiente secuencia de operaciones:

1. Pl trae el valor de x (100) a su acumulador.
2. P1 suma 10 al acumulador dejándolo en 110.
3. P2 comprueba que x vale 100 y toma la rama ELSE del IF.
4. P1 almacena el valor del acumulador en x (x=110).
5. P2 escribe el valor de x-50 que es ¡60!

(pg39)

Las combinaciones de ejecución son muy variadas, pero en cualquier caso existe un error potencial. No es incorrecto que el valor a escribir no esté determinado (de hecho la programación concurrente no es determinista). Lo que es incorrecto es que obtengamos un resultado imprevisto (60). El programa especifica que se escriba 110 ó 50, pero no otra cosa. El error se ha producido por una falta de exclusión mutua sobre la variable x. Lo solucionaremos utilizando una RC y obligando a que los accesos a x se efectúen dentro de dicha RC. El programa resultante es el siguiente:

```
PROGRAM Ejemplol; (* versión 2 *)
    VAR x: SHARED INTEGER;
    PROCEDURE P1;
        BEGIN
            REGION x DO x:= x + 10
        END;
    PROCEDURE P2;
        BEGIN
            REGION x DO
                IF x > 100
                THEN Writeln (x)
                ELSE Writeln (x-50)
        END;
    BEGIN
      x:= 100;
      COBEGIN
            P1; P2
      COEND
    END.
```

Como se puede observar en el programa anterior, el resultado de la impresión sigue estando indeterminado, pero lo que no ocurrirá es que se imprima un valor diferente de 50 ó 110. De hecho ahora la ejecución es secuencial en el sentido de que un proceso se ejecutará antes o después del otro, pero no se pueden solapar ni intercalar en el tiempo. La razón de ello es que el cuerpo de ambos procesos está totalmente incluido dentro de la misma RC. También se puede observar que en el cuerpo principal del programa se accede a x (x:=100) fuera de la RC. Esto está permitido ya que el cuerpo del programa principal se ejecuta secuencialmente. Son los procesos que se ejecutan concurrentemente los que no pueden acceder a variables compartidas fuera de su RC.

La ejecución secuencial mencionada anteriormente tiene su moraleja: «Las regiones críticas deben ser tan pequeñas como sea posible», ya que con (pg40) esto se consigue una mayor concurrencia.

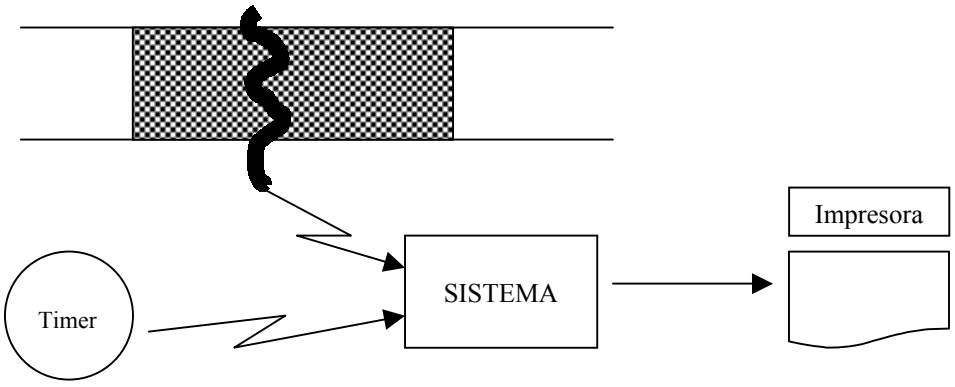



FIGURA 12

El caso opuesto es el mostrado en el programa anterior en el que la RC abarcaba a todo el cuerpo de los procesos obteniéndose un procesamiento secuencial. Para evitar esto bastaría con declarar una variable local en P2 (xLocal) y llevar el valor de x a ella. La pregunta sobre el valor de x se transforma en preguntar el valor de xLocal que se puede realizar fuera de la RC. Lo importante aquí es que el proceso de impresión, que consume bastante tiempo si lo comparamos con la ejecución de instrucciones, se realiza fuera de la RC. El programa quedaría como:

```
PROGRAM Ejemplol; (* versión 3 *)
   VAR x: SHARED INTEGER;
   PROCEDURE P1;
     BEGIN
          REGION x DO x:= x + 10
     END;
   PROCEDURE P2;
     VAR xLocal: INTEGER;
     BEGIN
          REGION x DO xLocal:= x;
          IF xLocal > 100
          THEN Writeln (xLocal)
          ELSE Writeln (xLocal-50)
     END;
   BEGIN
       x:= 100;
       COBEGIN
           P1; P2
       COEND
   END.
```

(pg41)

Ahora la condición de carrera que provoca la indeterminación en el resultado se halla en si P2 realiza la asignación xLocal:=x antes o después de que Pl actualice x.

**Ejemplo N.º 2:** Considere que nos encargan programar un sistema que cuente el número de vehículos que pasan por un determinado punto cada hora. Para ello disponemos un sensor que cruza la carretera en el punto que nos interesa y cada vez que pasa un coche produce una señal. Al cabo de la hora imprime la cuenta y la pone a 0 de nuevo. Ver Fig. 12.

Para solucionar este problema crearemos dos procesos: uno que se encargue de recibir las señales de paso de un coche y otro que cada hora imprima el valor del contador. Podemos obtener el siguiente programa:

```
PROGRAM Ejemplo2; (* versión 1 *)
   VAR contador: INTEGER;
   PROCEDURE Cuenta;
     BEGIN
        REPEAT
             espera a que pase un coche;
```

```
            contador:= contador + 1
        FOREVER
    END;
  PROCEDURE Imprime;
    BEGIN
        REPEAT
            espera una hora;
            Writeln (`Han pasado', contador, `coches');
            contador:= 0
        FOREVER
    END;
  BEGIN
      contador:=0;
      COBEGIN
          Cuenta;
          Imprime
      COEND
  END.
```

En el programa esbozado anteriormente figuran dos procesos, Cuenta e Imprime, que acceden a la variable `contador' que es compartida. La ejecución de este programa puede originar los siguientes escenarios erróneos:

0. En un instante t dado, el contador vale 19.
1. Pl detecta que pasa un coche y pone el valor del contador a 20. (pg42)
2. P2 detecta que pasó la hora y comienza a escribir el valor del contador (20).
3. P1 detecta que pasó otro coche y pone contador a 21.
4. P2 termina de imprimir el valor 20 y pone contador a 0.

El resultado de este escenario es que se ha perdido la cuenta de un coche.

El siguiente escenario está basado en que las instrucciones «contador:= contador + 1» y «contador:= 0» no son indivisibles en el tiempo. Consideraremos que para efectuar:

contador:= contador + 1

hay que realizar la siguiente secuencia de instrucciones:

```
LDA contador; carga el valor de contador en el acumulador.
ADD 1   ; suma 1 al acumulador.
STA contador; almacena el valor del acumulador en contador.
```

El escenario es el siguiente:

0. En un instante t dado, el contador vale 19.
1. P1 detecta que pasa un coche y ejecuta LDA contador. En ese momento contador valía 19.
2. P2 detecta que pasó un hora y escribe el valor de contador, que es 19, poniéndolo más tarde a 0.
3. P1 suma 1 a su acumulador y almacena su valor en contador (con lo que lo pone a 20).

El resultado de este escenario es que se han contabilizado dos veces los mismos coches.

El problema proviene por la falta de exclusión mutua respecto de la variable contador. Para solucionarlo declaramos esta variable como compartida, obteniéndose el siguiente programa:

```
PROGRAM Ejemplo2; (* versión 2 *)
  VAR contador: SHARED INTEGER;
  PROCEDURE Cuenta;
    BEGIN
      REPEAT
        espera a que pase un coche;
        REGION contador DO contador:= contador + 1
      FOREVER
```

```
    END;
(pg43)
  PROCEDURE Imprime;
    BEGIN
      REPEAT
        espera una hora;
        REGION contador DO
        BEGIN
          Writeln (`Han pasado', contador, `coches');
          contador:= 0
        END
      FOREVER
    END;
  BEGIN
    contador:=0;
    COBEGIN
      Cuenta; Imprime
    COEND
  END.
```

Con este programa se evitan los problemas que se han mencionado anteriormente. Se deja al lector que compruebe la corrección de este algoritmo. ¿Qué ocurriría si mientras P2 está en su RC imprimiendo pasa un coche?

Con estos ejemplos ha quedado claro que la RC es un instrumento conceptualmente sencillo y queda probada su utilidad para resolver la exclusión mutua.

Una posibilidad que no se ha mencionado hasta ahora es que las RC se pueden anidar. Considere el siguiente trozo de texto:

```
«P1»                              «P2»
  …..                               …..
  REGION v DO BEGIN                 REGION x DO BEGIN
    …..                               …..
    REGION x DO .....;                REGION v DO .....;
    …..                               …..
  END (* región v *)                END (* región x *)
  .....                             .....
```

En el programa anterior el proceso P1 tiene anidadas las regiones v y x al igual que el proceso P2 que las tiene anidadas en orden inverso. Cuando tenga que utilizar anidamiento de RC cuide que la situación antes descrita nunca se produzca. Imagine que el proceso P1 comienza a ejecutarse y entra en su RC v al mismo tiempo que el proceso P2 entra en su RC x. Hasta ahora todo es correcto. Siguiendo la ejecución de P1 cuando éste llegue a la RC x se detendrá porque esta RC está ocupada por P2. De igual manera, cuando P2 (pg44) llegue a la RC v se detendrá porque esta RC está ocupada por P1. Ninguno de los dos procesos puede avanzar en su ejecución. Estamos ante un caso de interbloqueo. Los dos procesos estarán esperando a que el otro libere la RC que ocupa, cuando ninguno de ellos podrá hacerlo. Aquí se está violando una de las reglas de las RC, a saber: «el tiempo que un proceso está dentro de una RC es finito».

En el ejemplo de las RC anidadas, conteste a las siguientes preguntas:
1) ¿Cuántos procesos como máximo pueden estar encolados simultáneamente a la entrada de la RC x?
2) ¿Cuántos del tipo «P1»?
3) ¿Cuántos del tipo «P2»?

Con esto acabamos la descripción de las regiones críticas. En sucesivos temas las iremos utilizando y veremos su uso.

## 2.2. Semáforos

Tenemos cuatro procesos A, B, C y D. Los vamos a ejecutar concurrentemente, tal que satisfagan las siguientes restricciones:

- el proceso A debe ejecutarse antes que el B;

- el proceso C debe ejecutarse antes que el proceso D.

El grafo de precedencia asociado a este enunciado es el mostrado en la Fig. 13: (pg45) y el programa que corresponde al grafo anterior es el siguiente (no nos interesa lo que hacen los cuatro procesos):

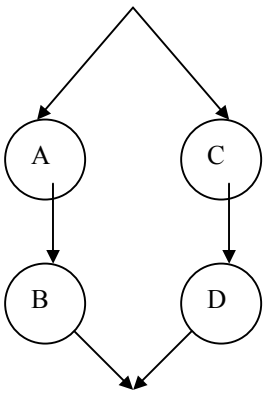



FIGURA 13

```
PROGRAMA Ejemplo;
   PROCEDURE A;
      BEGIN
         …..
         …..
      END;
   PROCEDURE B;
      BEGIN
         …..
         …..
      END;
   PROCEDURE C;
      BEGIN
         …..
         …..
      END;
   PROCEDURE D;
      BEGIN
         …..
         …..
      END;
   BEGIN
     COBEGIN
         BEGIN
            A; B
         END;
         BEGIN
            C; D
         END
      COEND
   END.
```

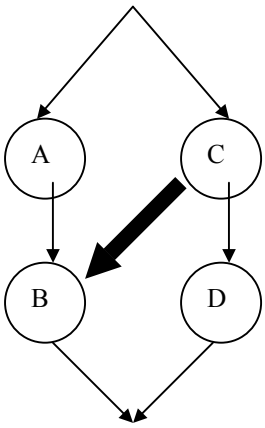



FIGURA 14

Ahora añadimos a las anteriores restricciones la condición de que C debe ejecutarse antes que B. El nuevo diagrama de precedencia es el mostrado en la Fig. 14.

Con las herramientas de que disponemos hasta ahora no es posible realizar un programa que se restringa únicamente a las condiciones dadas. Necesitamos, pues, una nueva herramienta, tal que nos permita expresar que B espere a que se haya terminado de ejecutar C. Este nuevo constructor es el **semáforo**. (pg46)

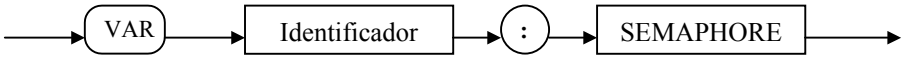



FIGURA 15

Un semáforo es un tipo abstracto de datos. Con esto queremos decir que consiste en unas estructuras de datos y en un conjunto de operaciones asociadas a tales estructuras. Un semáforo está caracterizado por:

a) Estructuras de datos:

- un contador entero positivo en el rango 0...infinito (teóricamente, ya que un ordenador tiene una capacidad limitada para representar números enteros);

- una cola de procesos esperando por ese semáforo.

b) Operaciones. Si s es una variable de tipo semáforo, podemos realizar las siguientes operaciones:

- WAIT (s)

- SIGNAL (s)

- INIT (s, valor)

Las operaciones WAIT y SIGNAL se excluyen mutuamente en el tiempo. La operación INIT tan sólo está permitida en el cuerpo principal del programa (la parte que no es concurrente). Por el contrario, las otras dos operaciones sólo se permiten en procesos concurrentes.

En la Fig. 15 se expresa el diagrama sintáctico de la declaración de una variable de tipo semáforo.

Las operaciones sobre los semáforos se comportan de la siguiente manera:

**WAIT (s)**

- Si el contador del semáforo s es igual a 0, se lleva el proceso que hizo la operación a la cola asociada con el semáforo s. Esto significa que se suspende su ejecución, abandonando el procesadora favor de otro proceso.

- Si el contador del semáforo s es mayor que 0, se decrementa en 1 dicho contador y el proceso que hizo la llamada continúa ejecutándose.

(Observe que estos semáforos, por definición, no pueden tener valores negativos. Existen otro tipo de semáforos que sí lo permiten, pero que no trataremos aquí.)

**SIGNAL (s)**

- Si el contador del semáforo s es mayor que 0, significa que no hay ningún proceso en la cola del semáforo s, y por tanto, incrementa en 1 dicho contador y el proceso que ejecutó esta operación continua. (pg47)

- Si el contador del semáforo s es igual a 0 y hay procesos esperando en la cola del semáforo, se toma a uno de ellos y se le pone en un estado de preparado para ejecutarse. El proceso que ejecutó la operación continúa ejecutándose.

- Si el contador del semáforo s es igual a 0 y no hay procesos esperando en la cola del semáforo, el resultado de esta operación es incrementar en 1 dicho contador. El proceso que ejecutó la operación continúa ejecutándose.

**INIT (s, valorInicial)**

Esta operación pone el contador del semáforo s al valor indicado por el parámetro «valorInicial».

Para que tenga una idea más clara de qué es un semáforo, considere el siguiente ejemplo, debido a Tanenbaum [Tane84]. En un pabellón de deportes existen 10 pistas para jugar al baloncesto. Para jugar un encuentro se precisa de un balón. Existe una caja donde están todos los balones (el contador del semáforo). A la hora convenida, el delegado del campo pone 8 balones en dicha caja (operación INIT, que pone el valor del contador a 8). Cuando dos equipos quieran jugar un partido mandan a un representante a la caja a por un balón (operación WAIT). Si en la caja hay algún balón (contador del semáforo mayor que 0), el jugador lo coge (decrementa en uno el contador del semáforo) y puede desarrollarse el partido (el proceso que hizo el WATT continúa). Si por el contrario no hay ningún balón en la caja (contador del semáforo igual a 0), el partido debe esperar hasta que alguien deje un balón. Cuando se ha terminado un encuentro, uno de los jugadores debe devolver el balón (operación SIGNAL). Si cuando va a la caja se encuentra con algún jugador que espera un balón (hay procesos en la cola del semáforo), simplemente se lo da y se vuelve con sus compañeros. Ahora el jugador que esperaba el balón lo toma y puede jugar su partido. Por el contrario, si cuando se va a dejar un balón en la cesta (operación SIGNAL) no hay nadie esperando un balón, simplemente lo deposita en la cesta y continúa.

El escenario anterior podría programarse de la siguiente manera:

```
PROGRAM Baloncesto;
   VAR cesta: SEMAPHORE;
   PROCEDURE Partido (i: INTEGER);
      BEGIN
         WAIT (cesta);
         juega el partido;
```

```
            SIGNAL (cesta)
        END;  (pg48)
    BEGIN
        INIT (cesta, 8);
        COBEGIN
            Partido (1); Partido (2); Partido (3);
            Partido (4); .....; Partido (10)
        COEND
    END.
```

De la misma manera, el diagrama de precedencia esbozado en la Fig. 14 se puede plasmar en el siguiente programa.

```
PROGRAM EjemploBien;
    VAR s: SEMAPHORE;
    PROCEDURE A;
        BEGIN
            …..
            …..
        END;
    PROCEDURE B;
        BEGIN
            …..
            …..
        END;
    PROCEDURE C;
        BEGIN
            …..
            …..
        END;
    PROCEDURE D;
        BEGIN
            …..
            …..
        END;
    BEGIN
        INIT (s, 0);
        COBEGIN
            BEGIN
                A;
                WAIT (s);
                B
            END;  (pg49)
            BEGIN
                C;
                SIGNAL (s);
                D
            END
        COEND
    END.
```

Como habrá podido observar, los semáforos son señales de sincronización. Se les puede considerar como mensajes vacíos. Los semáforos descritos aquí tienen el siguiente invariante (condiciones que en todo momento deben cumplir):

Sea e(v) el número de señales enviadas a un semáforo. y r(v) el número de señales recibidas de un semáforo.

El invariante dice que:

$0 \leq r(v) \leq e(v) \leq r(v) +$ máximo entero

si a esto le añadimos que un semáforo puede tener un valor inicial que denotamos con i(v), obtenemos:

$0 \leq r(v) \leq e(v) + i(v) \leq r(v) + \text{máximo}$

En definitiva, el invariante de los semáforos expresa que:

1. No se pueden recibir señales más rápidamente de lo que se envían.

2. El número de señales enviadas a un semáforo y no recibidas no puede exceder de la capacidad del semáforo.

Con este invariante las operaciones WAIT y SIGNAL se pueden expresar de la siguiente forma:

**WAIT**

si r(v) < e(v) + i(v), entonces r(v) := r(v) + 1 y el proceso continua
si r(v) = e(v) + i(v), entonces el proceso se espera en la cola

**SIGNAL**

```
e(v) := e(v) + 1;
IF not empty (cola)
THEN BEGIN
        selecciona un proceso de la cola;
        r(v) := r(v) + 1
   END
```

Los semáforos se pueden utilizar para implementar las RC de la siguiente forma: (pg50)

```
INIT (s, 1);
.....
.....
WAIT (s);   (* entrada a la región *)
.....
.....       (* cuerpo de la RC *)
.....
SIGNAL (s)(* salida de la región *)
```

Es fácil comprobar que con un semáforo inicializado a 1 en el cuerpo de la RC sólo puede haber un proceso al mismo tiempo.

No obstante, no es conveniente utilizar los semáforos para este fin. De hecho, para eso está la RC. Es fácil que cuando intentamos hacer una RC con semáforos, nos confundamos en algún sitio y programemos cosas como:

```
INIT (s, 1);
.....
.....
SIGNAL (s);   (* entrada a la región *)
.....
.....   (* cuerpo de la RC *)
WAIT (s)   (* salida de la región *)
```

con lo que, evidentemente, no se consigue la exclusión mutua. El lector puede intentar obtener diferentes combinaciones erróneas sobre esta cuestión. En cualquier caso existen razones para no implementar RC con semáforos, entre ellas:

1. El compilador no conoce qué variables protege un semáforo, con lo cual no nos ayudaría si estamos utilizando una variable compartida fuera de su RC (esto tendría que controlarlo también el usuario).

2. El compilador no podrá distinguir procesos disjuntos y, por tanto, debe permitir el acceso a cualquier variable. Si realmente los procesos son disjuntos, el compilador no nos ayuda.

También hay que tener sumo cuidado cuando se combinan los semáforos y las RC. Tenga en cuenta que si un proceso que está dentro de una RC realiza una operación WAIT sobre un semáforo cuyo contador vale 0, este proceso se parará dentro de la RC, bloqueándola en consecuencia.

Para terminar, veamos un último ejemplo. Suponga que existen 4 montones de papeles y que hay que coger uno de cada montón y grapar los cuatro juntos. El proceso debe repetirse hasta que se acaben los montones (que contienen el mismo número de papeles). Este problema los podemos programar con dos procesos: uno que se encargue de formar los grupos de 4 papeles (pg51) y otro que tome estos grupos y los vaya grapando. Evidentemente, el proceso que grapa no puede hacerlo si no tiene nada que grapar. Tendrá que esperar a que exista algún montón de 4 papeles. Este es el punto de sincronización de los dos procesos y los implementaremos con un semáforo. El proceso que hace montones ejecutará un SIGNAL cada vez que haya hecho uno y el que grapa ejecutará un WAIT cada vez que quiera grapar uno. En este caso el semáforo está actuando como contador del número de montones de 4 papeles que quedan por grapar (número de señales enviadas al semáforo y no recibidas). El problema se podría programar como sigue:

```
PROGRAM graparHojas;
   VAR s: SEMAPHORE;
   PROCEDURE amontonar;
      BEGIN
         REPEAT
            coge una hoja de cada monton;
            deja el grupo de 4 hojas en la mesa;
            SIGNAL (s)
         UNTIL se acaben las hojas
      END;
   PROCEDURE grapar;
      BEGIN
         REPEAT
            WAIT (s);
            toma un grupo de la mesa y grapalo;
         UNTIL no queden montones que grapar
      END;
   BEGIN
      INIT (s, 0);
      COBEGIN
         amontonar;
         grapar
      COEND
   END.
```

Para completar el programa anterior, habría que poner una RC cuando los procesos acceden a la mesa.

Observe que las condiciones de finalización de los bucles no están refinadas. Esta es una cuestión que por ahora no nos interesa.

Una posible alternativa de solución al problema anterior es considerar (pg52) que existen 4 procesos, cada uno de los cuales contribuye con una hoja (de uno de los cuatro montones y siempre del mismo) a formar un grupo de cuatro hojas. Otro proceso sería el encargado de graparlas (Fig. 16).

Ahora cada proceso P(i) tomará una hoja de su montón y la dejará en la mesa. Para dejar la siguiente debe esperar a que los otros procesos P(i) hayan hecho lo mismo. Cuando haya un grupo de 4 hojas sobre la mesa, el proceso que grapa (Pg) lo tomará y lo grapará. Ahora caben dos aproximaciones:

a) que en la mesa pueda existir como mucho un grupo de 4 hojas;

b) que los procesos P(i) puedan formar sobre la mesa más de un montón.

Veamos a continuación la primera aproximación. Los procesos P(i) tomarán una hoja del montón i y esperarán a que una vez formado el grupo de 4 hojas, el proceso Pg lo tome y les avise que ya pueden continuar. Por su parte, el proceso Pg esperará a que haya un grupo de 4 hojas, lo tomará y avisará a los procesos P(i) que ya pueden colocar el siguiente. Una aproximación a ambos tipos de procesos podría ser la siguiente:

```
  PROCEDURE P (i: INTEGER);
      BEGIN
```

```
        REPEAT
            toma una hoja del monton i;
            dejala sobre la mesa;
            espera a que te avise Pg de que puedes continuar
        UNTIL se acaben las hojas del monton i
    END;
PROCEDURE Pg;
    BEGIN
        REPEAT
            espera a que haya un grupo de 4 hojas en la mesa;
            tomalo;
            indica a los P(i) que pueden continuar;
            grapa
        UNTIL no queden montones que grapar
    END;
```

De momento no nos ocuparemos de las condiciones de terminación de los bucles. El único problema radica en cómo sabe el proceso Pg que ya hay un grupo de 4 hojas sobre la mesa. O lo que es lo mismo, cómo sabe un pro ceso P(i) que su hoja es la última para completar el grupo de 4 y que, por tanto, debe avisar a Pg (observe que este aviso no figura en el código anterior). (pg53)

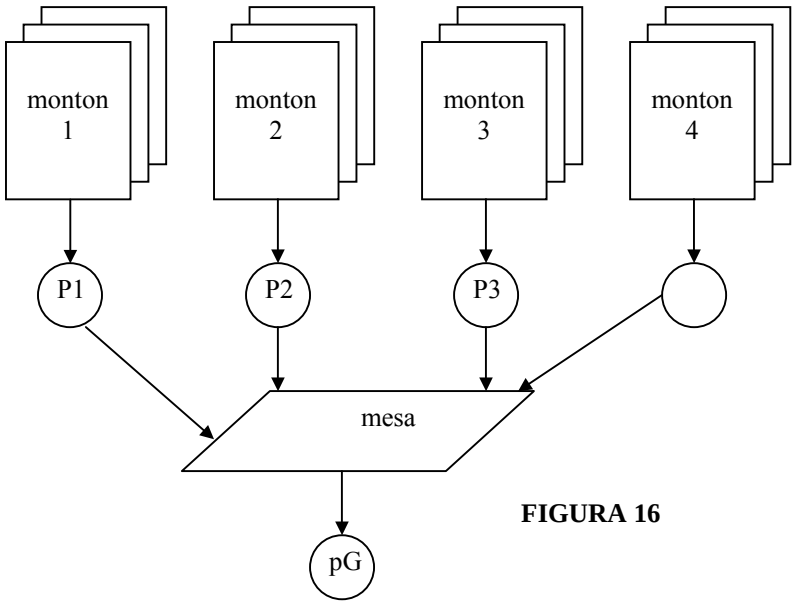



**FIGURA 16**

Si respondemos a la primera pregunta, podemos declarar 1 semáforo sobre el que cada P(i) hará un SIGNAL después de haber dejado una hoja y a continuación se bloqueará en un segundo semáforo, esperando que Pg le avise. Por su parte, Pg realizará 4 WAIT sobre el primer semáforo y un SIGNAL sobre el otro.

El programa sería el siguiente:

```
PROGRAM Grapar;
    VAR s, s1: SEMAPHORE;
    PROCEDURE P (i:INTEGER);
        BEGIN
            REPEAT
                toma una hoja del monton i;
                SIGNAL (s); (* déjala sobre la mesa *)
                WAIT (s1); (* espera a que te avise Pg *)
            UNTIL se acaben las hojas del monton i
        END; (pg54)
    PROCEDURE Pg;
        BEGIN
            REPEAT
```

```
            WAIT (s); WAIT (s); WAIT (s); WAIT (s);
            (* montón cogido *)
            SIGNAL (sl); SIGNAL (sl); SIGNAL (sl); SIGNAL (sl);
            grapa
        UNTIL se acabe el proceso de grapar
    END;
BEGIN (* main *)
    INIT (s, 0); INIT (sl, 0);
    COBEGIN
        P(l); P(2); P(3); P(4);
        Pg
    COEND
END.
```

La solución anterior no satisface el enunciado. Para ello, vea el siguiente escenario:

- P(1) pone su hoja y queda en WAIT (sl).
- P(2), P(3) y P(4) ponen sus hojas y quedan también en WAIT (sl).
- Pg pasa los 4 WAIT (s) y da el primer SIGNAL (sl).
- P1 pasa el WAIT (sl) y pone la siguiente hoja, quedándose de nuevo en WAIT (sl).
- Pg da el segundo SIGNAL (sl).
- P(1) pasa rápidamente su WAIT (sl) y coloca otra hoja.
- .....

Este escenario muestra que se pueden grapar 4 hojas del mismo montón y eso es incorrecto. Para corregirlo, haremos que cada proceso P(i) espere la señal del proceso Pg en un semáforo diferente. Es decir:

```
PROGRAM Grapar;
    VAR  s: SEMAPHORE;
         sl: ARRAY [1..4] OF SEMAPHORE;
         i : INTEGER;
(pg55)
    PROCEDURE P (i:INTEGER);
        BEGIN
            REPEAT
                toma una hoja del monton i;
                SIGNAL (s); (* déjala sobre la mesa *)
                WAIT (sl[i]); (* espera a que te avise Pg *)
                UNTIL se acaben las hojas del monton i
            END;
    PROCEDURE Pg;
        VAR i: INTEGER;
        BEGIN
            REPEAT
                WAIT (s); WAIT (s); WAIT (s); WAIT (s);
                (* montón cogido *)
                FOR i:=1 TO 4 DO SIGNAL (sl[i]);
                grapa
            UNTIL se acabe el proceso de grapar
        END;
    BEGIN (* main *)
        INIT (s, 0);
        FOR is=1 TO 4 DO INIT (sl[i]), 0);
        COBEGIN
            P(1); P(2); P(3); P(4);
            Pg
        COEND
    END.
```

Si respondemos ahora a la segunda pregunta, tal que sea el último proceso que colocó la hoja el que avise a Pg, la solución podría ser la siguiente:

```
PROGRAM Grapar;
   VAR  s: SEMAPHORE;
        s1: ARRAY [0..3J OF SEMAPHORE;
        mesa: SHARED ARRAY [0..3] OF BOOLEAN; is INTEGER;
(pg56)
   PROCEDURE P (i:INTEGER);
      BEGIN
         REPEAT
            toma una hoja del monton i;
            REGION mesa DO BEGIN
               mesa[i]:= TRUE;
               IF mesa[(i+1) mod 4] & mesa [(i+2) mod 4] & mesa[(i+3) mod 4]
               THEN SIGNAL (s)
            END;
            WAIT (sl[i]);
         UNTIL se acaben las hojas del monton i
      END;
   PROCEDURE Pg;
      VAR i: INTEGER;
      BEGIN
         REPEAT
            WAIT (s);
            REGION mesa DO
               FOR i:=0 TO 3 DO mesa[i]:= FALSE;
               FOR i:=0 TO 3 DO SIGNAL (S1[i]);
               grapa
         UNTIL se acabe el proceso de grapar
      END;
   BEGIN (* main *)
      INIT (s, 0);
      FOR i:=0 TO 3 DO INIT (sl[i], 0);
      FOR i:=0 TO 3 DO mesa[i]:= FALSE;
      COBEGIN
           P(0); P(l); P(2); P(3);
           Pg
      COEND
   END.
```

Por último, abordaremos el problema considerando que pueden existir varios montones sobre la mesa. En este caso haremos que uno de los procesos P(i) (digamos el P(1)) sea el que controle si ya hay un montón en la mesa. La solución a este problema podría ser la siguiente: (pg57)

```
PROGRAM Grapar;
   VAR  s, aviso: SEMAPHORE;
        sigue: ARRAY [2..4] OF SEMAPHORE;
        i: INTEGER;
   PROCEDURE P1;
      VAR i: INTEGER;
      BEGIN
         REPEAT
            toma una hoja del monton 1;
            dejala en la mesa;
            FOR i:=2 TO 4 DO WAIT (aviso);
            SIGNAL (s);
            FOR i:=2 TO 4 DO SIGNAL (sigue[i])
         UNTIL se acaben las hojas del monton 1
      END;
   PROCEDURE P (i: INTEGER);
      BEGIN
         REPEAT
```

```
            toma una hoja del monton i;
            dejala en la mesa;
            SIGNAL (aviso);
            WAIT (sigue[i])
        UNTIL se acaben las hojas del monton i
    END;
PROCEDURE Pg;
    BEGIN
        REPEAT
            WAIT (s);
            grapa
        UNTIL se acabe el proceso de grapar
    END;
BEGIN (* main *)
    INIT (s, 0); INIT (aviso, 0);
    FOR i:=2 TO 4 DO INIT (sigue[i], 0);
    COBEGIN
            P(1); P(2); P(3); P(4);
            Pg
    COEND
END.
```

(pg58)

Para hacer simétrico el código de los procesos Pi, podemos crear un nuevo proceso, controlador, que realice la sincronización que antes efectuaba P1. El programa resultante es el siguiente:

```
PROGRAM Grapar;
VAR s, aviso: SEMAPHORE;
    sigue: ARRAY [1..4] OF SEMAPHORE;
    i: INTEGER;
PROCEDURE P (i: INTEGER);
    BEGIN
        REPEAT
            toma 1 hoja del monton i;
            dejala en la mesa;
            SIGNAL (aviso);
            WAIT (sigue[i])
        UNTIL se acaben las hojas del monton i
    END;
PROCEDURE controlador;
    VAR i: INTEGER;
    BEGIN
        REPEAT
            FOR i:=1 TO 4 DO WAIT (aviso);
            SIGNAL (s);
            FOR i:=1 TO 4 DO SIGNAL (sigue[i])
        UNTIL se acaben los montones
    END;
PROCEDURE Pg;
    BEGIN
        REPEAT
            WAIT (s);
            grapa
        UNTIL se acabe el proceso de grapar
    END;
BEGIN (* main *)
    INIT (s, 0); INIT (aviso, 0);
    FOR i:=1 TO 4 DO INIT (sigue[i], 0);
    COBEGIN
        P(1); P(2); P(3); P(4);
```

```
        controlador;
        Pg
    COEND
END. (* main *)    (pg59)
```

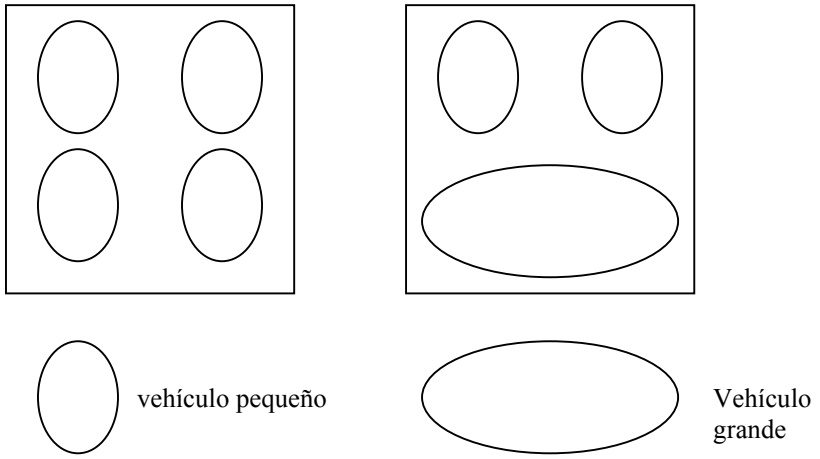



**FIGURA 17**

## 2.3. Región crítica condicional

«En un hotel hay 10 vehículos automáticos pequeños y otros 10 grandes. Todos ellos están controlados por un programa con procesos concurrentes (uno por vehículo). Estamos interesados en la parte del programa que con trola la entrada en un montacargas, en el que caben hasta 4 vehículos pequeños ó 2 vehículos pequeños y 1 grande (Fig. 17).»

La solución al enunciado propuesto pasa por lanzar concurrentemente 20 procesos que representan a cada uno de los vehículos.

```
COBEGIN
   Vp(1); Vp(2); ...; Vp(10);
   Vg(1); Vg(2); ...; Vg(10);
COEND
```

donde Vg(i) representa al vehículo grande i y Vp(i) al vehículo pequeño i (para $1 \leq i \leq 10$).

La definición del montacargas simplemente será la cantidad de vehículos de cada clase que están dentro de él. Es decir: (pg60)

```
VAR montacargas: RECORD
            (* número de vehículos grandes que están dentro del montacargas *) nmrVg: 0..1;
            (* número de vehículos pequeños que están dentro del montacargas *) nmrVp: 0..4
                END;
```

Observe de la definición anterior, que a lo sumo cabe un vehículo grande.

Con la definición anterior podemos escribir la primera aproximación de los procedimientos que representan a los vehículos:

```
PROCEDURE Vg (i: INTEGER);
   VAR dentro: BOOLEAN; (* indica si el vehículo está dentro del montacargas *)
   BEGIN
     ..... (* otras cosas *)
     dentro:= FALSE;
     REPEAT
        IF cabVg        (* si cabe en el montacargas *)
        THEN BEGIN
                entrar al montacargas;
                dentro:= TRUE
                END;
     UNTIL dentro;
     (* aquí ya está dentro del montacargas *)
```

```
      ..... (* otras cosas *)
   END;
PROCEDURE Vp (i: INTEGER);
   VAR dentro: BOOLEAN;
   BEGIN
      ..... (* otras cosas *)
      dentro:= FALSE;
      REPEAT
        IF cabVp
        THEN BEGIN
                entrar al montacargas;
                dentro:= TRUE
                END;
      UNTIL dentro;
      (* aquí ya está dentro del montacargas *)
      ..... (* otras cosas *)
   END;     (pg61)
```

Las condiciones de cabeVg y cabeVp se indican a continuación:

```
cabeVg = WITH montacargas DO
            (nmrVp < 2) & (nmrVg = 0)
```

un vehículo grande puede entrar al montacargas si no existe ninguno de su clase y si el número de vehículos pequeños es menor o igual que 2.

```
cabeVp = WITH montacargas DO
                 ( (nmrVp < 4) & (nmrVg = 0)) OR
                 ( (nmrVp < 1) & (nmrVg = 1) )
```

un vehículo pequeño puede entrar al montacargas cuando no habiendo ninguno grande, el número de pequeños es menor que 4; o bien, si hay uno grande, que el número de pequeños sea menor que 2.

Por último, la acción de entrar al montacargas se reduce a aumentar en 1 el número de vehículos que hay de esa clase. Con esto podemos escribir el programa como sigue:

```
PROGRAM Montacargas; (* algoritmo (2.3.1): 1.ª aproximación *)
   VAR montacargas: RECORD
                       nmrVg: 0..1;
                       nmrVp: 0..4
                    END;
   PROCEDURE Vg (i: INTEGER);
      VAR dentro: BOOLEAN;
      BEGIN
         ..... (* otras cosas *)
         dentro:= FALSE;
         REPEAT
            IF (montacargas.nmrVp < 2) & (montacargas. nmrVg = 0)
            THEN BEGIN
                    montacargas. nmrVg:= montacargas. nmrVg + 1;
                    dentro:= TRUE
                 END;
         UNTIL dentro;
         (* aquí ya está dentro del montacargas *)
         ..... (* otras cosas *)
      END;     (pg62)
PROCEDURE Vp (i: INTEGER);
   VAR dentro: BOOLEAN;
   BEGIN
      ..... (* otras cosas *)
      dentro:= FALSE;
      REPEAT
```

```
        IF ( (montacargas.nmrVp < 4) & (montacargas.nmrVg = 0 ) OR
           ( (montacargas.nmrVp < 1) & (montacargas.nmrVg = 1) )
        THEN BEGIN
                montacargas.nmrVp:= montacargas.nmrVp + 1;
                dentro:= TRUE
             END;
      UNTIL dentro;
      (* aquí ya está dentro del montacargas *) ..... (* otras cosas *)
   END;
BEGIN
   WITH montacargas DO BEGIN
      nmrVp:= 0;
      nmrVg:= 0
   END;
   COBEGIN
      Vp(1); Vp(2); ...; Vp(10);
      Vg(1); Vg(2); ...; Vg(1o)
   COEND
END.
```

En el algoritmo anterior observamos que todos los vehículos realizan un lazo cuando intentan entrar al montacargas. Comprueban la condición de entrada y si no se cumple, siguen intentándolo (hasta que la variable dentro sea TRUE).

La primera objeción que hay que poner a este programa es que no funciona según indica el enunciado. Considere la situación en la que inicialmente no hay ningún vehículo en el montacargas. En ese momento, dos vehículos grandes pueden evaluar la condición concurrentemente y descubrir que ésta es cierta, con lo cual los dos se introducen en el montacargas (y no caben). O bien si existen 3 pequeños en el montacargas, puede ocurrir que dos vehículos pequeños evalúen al mismo tiempo la condición de entrada y entrar los dos con lo que en el montacargas estarán 5 vehículos pequeños (que según el enunciado no caben). 0 bien si existen 2 pequeños en (pg63) el montacargas, un vehículo grande y 1 pequeño pueden evaluar concurrentemente sus respectivas condiciones de entrada a TRUE con lo cual en el montacargas habrá 3 pequeños y 1 grande (que no caben).

Todos estos escenarios erróneos (y otros que pueda imaginar) se han producido por una misma razón: la variable montacargas es compartida (tanto actualizada como consultada) por varios procesos concurrentes y no hemos arbitrado ningún mecanismo para conseguir el acceso exclusivo a esa variable. Debemos conseguir que la consulta (reflejada en la condición de entrada) y la actualización (reflejada en el proceso de entrada al montacargas) constituyan una unidad indivisible. Para ello, y como vimos en la sección 2.1, utilizaremos la región crítica. Con esto el programa anterior se puede escribir como:

```
PROGRAM Montacargas; (* algoritmo (2.3.2): 2.ª aproximación *)
   VAR montacargas: SHARED RECORD
                       nmrVg: 0..1;
                       nmrVp: 0..4
                    END;
   PROCEDURE Vg (i: INTEGER);
      VAR dentro: BOOLEAN;
      BEGIN
         ..... (* otras cosas *)
         dentro:= FALSE;
         REPEAT
            REGION montacargas DO
               IF (nmrVp < 2) & (nmrVg = 0)
               THEN BEGIN
                       nmrVg:= nmrVg + 1;
                       dentro:= TRUE
                    END;
         UNTIL dentro;
         (* aquí ya está dentro del montacargas *)
```

```
        ..... (* otras cosas *)
      END;    (pg64)
   PROCEDURE Vp (i: INTEGER);
      VAR dentro: BOOLEAN;
      BEGIN
         ..... (* otras cosas *)
         dentro:= FALSE;
         REPEAT
            REGION montacargas DO
            IF ( (nmrVp < 4) & (nmrVg = 0)) OR
               ( (nmrVp < 1) & (nmrVg = 1) )
            THEN BEGIN
                     nmrVp:= nmrVp + 1;
                     dentro:= TRUE
                 END;
         UNTIL dentro;
         (* aquí ya está dentro del montacargas *)
         ..... (* otras cosas *)
      END;
BEGIN
  WITH montacargas DO BEGIN
      nmrVp:= 0;
      nmrVg:= 0
   END;
   COBEGIN
      Vp(1), Vp(2); ...;Vp(10);
      Vg(1); Vg(2); ...; Vg(10)
   COEND
END.
```

Es interesante observar que la RC está dentro del lazo REPEAT y no al revés. Es decir, la siguiente solución sería incorrecta:

```
REGION montacargas DO
   REPEAT
     IF (nmrVp < 2) & (nmrVg = 0)
     THEN BEGIN
               nmrVg:= nmrVg + 1;
               dentro:= TRUE
          END
     UNTIL dentro;
     .....
     .....          (pg65)
```

¿Por qué? La razón es bien simple. Cuando un vehículo acceda a la región crítica y no pueda entrar, monopolizará dicha región. Puesto que los vehículos saldrán alguna vez y puesto que esta salida debe quedar reflejada en el número de vehículos que quedan en el montacargas, significa que el vehículo que salga, debe entrar a la región crítica montacargas para actualizar su estado. Puesto que esta región está ocupada permanentemente por un vehículo que no puede entrar, se produce un interbloqueo. El vehículo que permitirá entrar a otro (al salir él), no puede hacerlo, porque el que quiere entrar no le deja salir. Es el caso de un ascensor donde una persona no puede entrar porque la que tiene que salir tiene obstruido el paso. Moraleja: «CUIDADO CON LOS LAZOS DENTRO DE LAS REGIONES CRITICAS».

La solución anterior (algoritmo (2.3.2)) es correcta desde el punto de vista del enunciado. Sin embargo, los procesos realizan **espera activa**. Cuando un vehículo quiere entrar y no puede, continúa ejecutándose en un bucle hasta que lo consiga o pierda el procesador. Este tiempo de proceso es inútil y supone un bajo rendimiento de la CPU.

Para evitar este gasto inútil deberíamos conseguir que el vehículo que, una vez comprobada la condición, descubra que no puede entrar, no lo intentase de nuevo hasta que haya variado alguna

condición. Para satisfacer este protocolo utilizaremos una nueva herramienta: la REGION CRITICA CONDICIONAL.

Al igual que las RC y los semáforos proporcionaban exclusión mutua y sincronización entre procesos respectivamente, la región crítica condicional permite que un proceso espere hasta que los componentes de una variable compartida v satisfagan una condición B.

La programación del caso de los vehículos grandes utilizando región crítica condicional (RCC) es la siguiente:

```
PROCEDURE Vg (i: INTEGER);
   BEGIN
      ..... (* otras cosas *)
      REGION montacargas DO
         AWAIT (nmrVp < 2) & (nmrVg = 0);
         nmrVg:= nmrVg + 1
      END;
      (* aquí ya está dentro del montacargas *)
      ..... (* otras cosas *)
   END;      (pg66)
```

La sintaxis de las RCC es la siguiente:

```
VAR v: SHARED T;
REGION v DO BEGIN
   sentencia 1; (* puede no existir o existir varias *)
   AWAIT condiciónBoolean
   sentencia 2 (* puede no existir o existir varias *)
END;
```

Como puede apreciar una RCC sólo se diferencia de una RC en que dentro de la RCC existe la sentencia AWAIT. Dicha primitiva sólo puede estar dentro de una RC. Si existen varias RC anidadas, AWAIT se asocia con la más próxima. Esta sentencia produce una ESPERA PASIVA. Su funcionamiento es el siguiente:

- Si la condiciónBoolean es TRUE el proceso continua por la siguiente sentencia al AWAIT.

- Si la condiciónBoolean es FALSE, el proceso detiene su ejecución, abandona la RC para permitir a otros procesos entrar en ella y pasa a una cola de espera, Qs asociada con la RC.

Cuando un proceso, que había evaluado la condiciónBoolean a FALSE, vuelve a entrar en su RC lo hace ejecutando de nuevo la sentencia AWAIT, repitiéndose el comportamiento ya descrito de la misma.

Un proceso que haya evaluado a FALSE la condición del AWAIT no vuelve a entrara su RC hasta que otro proceso abandone ésta. Esto significa que un proceso espera para que se cumpla una condición de manera pasiva (sin ejecutarse). Se le vuelve a ejecutar cuando ¡posiblemente! alguien haya modificado dicha condición. Para alterar el valor de la condición es necesario que un proceso entre a la RC. Cuando salga de ella puede que haya cambiado alguna variable que interviene en la condición. Es en este momento cuando a los procesos que estaban esperando en la cola Qs de la RC se les da oportunidad de ejecutarse. Evidentemente, puede que la condición no haya variado, por lo que dichos procesos se suspenderán de nuevo cuando ejecuten la sentencia AWAIT. Es más, si estaban esperando N procesos, uno conseguirá el acceso a la RC y puede que después de evaluar a TRUE la condición del AWAIT cambie dicha condición haciéndola FALSE de nuevo. En estas circunstancias los N-1 procesos restantes se dormirán de nuevo después de ejecutar la sentencia AWAIT.

En la Fig. 18 se muestra las colas que tiene asociadas una RCC y la transición entre dichas colas.

En esa figura se observa que una RCC tiene asociada dos colas: (pg67)

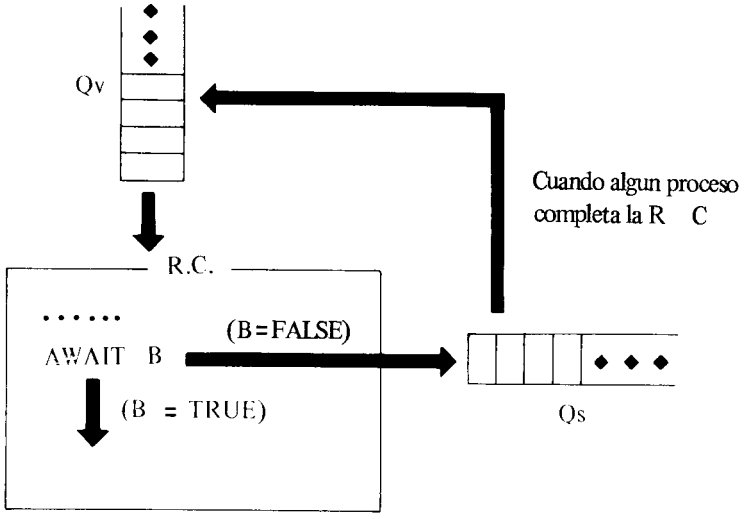



FIGURA 18

- Qv que es donde se espera un proceso cuando quiere entrar a una RC que está ocupada. Constituye la cola de entrada a la RC.

- Qs que es donde se esperan los procesos que evaluaron a FALSE la condición del AWAIT.

La transición de Qs a Qv se produce cuando un proceso abandona la RC. La razón para tener la cola Qs estriba en que los procesos que evaluaron a FALSE la condición del AWAIT no puedan volver a entrar hasta que ésta haya sido modificada (posiblemente). Si los pasaremos directamente a Qv, podrían desde el AWAIT intentar entrar cuando la condición no ha variado todavía, con la consiguiente perdida de tiempo del procesador. Teniendo dos colas separamos a los que están esperando por una condición de los que van a entrar a la RC por primera vez.

Con esta nueva herramienta podemos programar el problema de los vehículos como muestra el algoritmo (2.3.3)

```
PROGRAM Montacargas; (* algoritmo (2.3.3): 3.ª aproximación *)
  VAR montacargas: SHARED RECORD
                           nmrVg: 0..1;
                           nmrVg: 0..4
                         END;
  PROCEDURE Vg (i: INTEGER);
    BEGIN
    REGION montacargas DO BEGIN
      AWAIT (nmrVg < 2) & (nmr Vg = 0);
      nmrVg:= nmrVg + 1
    END;        (pg68)
    (* en el montacargas *)
    REGION montacargas DO (* salida del montacargas *)
      nmrVg:= nmrVg - 1;
    END;
  PROCEDURE Vp (i: INTEGER);
    BEGIN
      REGION montacargas DO
        AWAIT ( (nmrVp < 4) & (nmrVg = 0)) OR
                ( (nmrVp < 1) & (nmrVg = 1) );
        nmrVp:= nmrVp + 1;
      END;
      (* en el montacargas *)
      REGION montacargas DO
        nmrVp:= nmrVp - 1
    END;
  BEGIN
```

```
    WITH montacargas DO BEGIN
       nmrVp:= 0;
       nmrVg:= 0
    END;
    COBEGIN
       Vp(1); Vp(2); ...; Vp(10);
       Vg(1); Vg(2); ...; Vg(10)
    COEND
END.
```

En este programa también hemos incluido la salida de los vehículos del montacargas que se reduce a decrementar en 1, el número de vehículos que hay en el montacargas de la clase del que sale.

Para dar mayor claridad al funcionamiento de las RCC vamos a realizar un seguimiento del programa anterior en un número determinado de escenarios.

0: En un instante t dado, en el montacargas está el vehículo Vpl.

| en el montacargas | en Qs | en Qv | en la RC |
|---|---|---|---|
| $V_{p}l$ | ---- | ---- | ---- |

1: Vgl llega a la RC y se queda evaluando el AWAIT.

| en el montacargas | en Qs | en Qv | en la RC |
|---|---|---|---|
| Vpl | ---- | ---- | Vgl |

(pg69)

2: intentan entrar a la RC Vg2, Vg3, Vp2, Vp3, Vp4 y pasan a Qv.

| en el montacargas | en Qs | en Qv | en la RC |
|---|---|---|---|
| Vpl | ---- | Vg2, Vg3 | Vgl |
| | | Vp2, Vp3 | |
| | | Vp4 | |

3: sale Vgl de la RC después de evaluar a TRUE la condición.

| en el montacargas | en Qs | en Qv | en la RC |
|---|---|---|---|
| Vpl, Vgl | ---- | Vg2, Vg3 | ---- |
| | | Vp2, Vp3 | |
| | | Vp4 | |

4: entra Vg2 a la RC, evalúa a FALSE la condición y pasa a Qs.

| en el montacargas | en Qs | en Qv | en la RC |
|---|---|---|---|
| Vpl, Vgl | Vg2 | Vg3, Vp2 | ---- |
| | | Vp3,Vp4 | |

5. lo mismo para vg3.

| en el montacargas | en Qs | en Qv | en la RC |
|---|---|---|---|
| Vpl, Vgl | Vg2, Vg3 | Vp2, Vp3 | ---- |
| | | Vp4 | |

6: entra a la RC Vp2, evalúa a TRUE la condición y está a punto de salir de la RC.

| en el montacargas | en Qs | en Qv | en la RC |
|---|---|---|---|
| Vpl, Vgl | Vg2, Vg3 | Vp3, Vp4 | Vp2 |

7: sale Vp2 de la RC y termina llevando a Qv los procesos que están en Qs.

| en el montacargas | en Qs | en Qv | en la RC |
|---|---|---|---|
| Vpl, Vgl, Vp2 | ---- | Vp3, Vp4 | ---- |

Vg2, Vg3

Las salidas del montacargas se producen entrando a la RC y saliendo incondicionalmente de ésta después de haber variado alguno de los campos componentes de la variable montacargas.

Podemos utilizar el constructor de RCC para programar el problema de grapar hojas en su versión de permitir que existan varios montones sobre la mesa. En una primera aproximación podemos escribir el siguiente algoritmo: (pg70)

```
PROGRAM Grapar;
   VAR monton: SHARED RECORD
                       hojas: ARRAY [1..4] OF BOOLEAN;
                       montonCompleto: BOOLEAN
                     END;
       s: SEMAPHORE;
       i: INTEGER
   PROCEDURE P (i: INTEGER);
       BEGIN
          REPEAT
             toma 1 hojas del monton i;
             REGION monton DO BEGIN
                hojas [i]:= TRUE;
                AWAIT montonCompleto;
                montonCompleto:= FALSE
             END
          UNTIL se acaben las hojas del montón i
       END;
   PROCEDURE gestor;
       BEGIN
          REPEAT
             REGION monton DO BEGIN
                AWAIT hojas[I] & hojas[2] & hojas[3] & hojas[4];
                hojas[1]:= FALSE; hojas[2]:= FALSE;
                hojas[3]:= FALSE; hojas[4]:= FALSE;
                montonCompleto:= TRUE
             END;
             SIGNAL (s)
          UNTIL se acaben los montones
       END;
   PROCEDURE Pg;
       BEGIN
          REPEAT
             WAIT (s);
             grapa
          UNTIL se acabe el proceso de grapar
       END;                                                    (pg71)
BEGIN (* main *)
   INIT (s, 0);
   FOR i:=1 TO 4 DO monton.hojas[i]:= FALSE;
   monton.montonCompleto:= FALSE;
   COBEGIN
        P(1); P(2); P(3); P(4);
        gestor; Pg
   COEND
END. (* main *)
```

Las objecciones que pongo al programa anterior son dos:

1. Inicialmente montónCompleto y hojas[i] es false. Si en estas condiciones se ejecuta el gestor, se bloqueará en el Await. Ahora los procesos Pi ponen hojas[i] a true y se quedan esperando en el Await. Ningún proceso puede avanzar. INTERBLOQUEO.

2. Si, por el contrario, primero se ejecutan los procesos Pi, éstos pondrán hojas[i] a true y se quedarán en el Await. Cuando se ejecute el gestor, pondrá hojas[i] a false y montónCompleto a true. Sale de la RC (con lo cual despierta a los procesos Pi) y da el signal (s). En este momento cualquiera de los processo Pi puede entrar a la RC y comprobar la condición del Await. Si entra primero Pl pone montónCompleto a false y abandona la RC. Ahora pueden entrar secuencialmente P2 a P4 que se quedarán de nuevo dormidos en el Await. Por su parte, el proceso Pl puede ahora entrar de nuevo a la RC, poner hojas[1] a true y dormirse en el await. El proceso gestor después de dar el signal(s) entrará a la RC y también se quedará dormido en el await. Por su parte, el proceso que grapa después de pasar el primer wait(s) grapará un montón y se quedará dormido en el wait(s). INTERBLOQUEO.

Para resolver este problema, proponemos el siguiente algoritmo:

```
PROGRAM Grapar;
   VAR  hojas: SHARED ARRAY [1..4] OF BOOLEAN;
        puede: ARRAY [1..4] OF SEMAPHORE;
        s: SEMAPHORE;
        i: INTEGER
   PROCEDURE P (i: INTEGER);
      BEGIN
         REPEAT
            toma 1 hoja del monton i;
            REGION hojas DO hojas [i]:= TRUE;
            WAIT (puede[i]);
         UNTIL se acaben las hojas del monton i
      END;                                                    (pg72)
   PROCEDURE gestor;
      VAR i: INTEGER;
      BEGIN
         REPEAT
            REGION hojas DO BEGIN
               AWAIT hojas[1] & hojas[2] & hojas[3] & hojas[4];
               hojas[1]:= FALSE; hojas[2]:= FALSE;
               hojas[3]:= FALSE; hojas[4]:= FALSE;
            END;
            SIGNAL (s);
            FOR i:=1 TO 4 DO SIGNAL (puede[i]);
         UNTIL se acaben los montones
      END;
   PROCEDURE Pg;
      BEGIN
         REPEAT
            WAIT (s);
            grapa
         UNTIL se acabe el proceso de grapar
      END;
BEGIN (* main *)
   INIT (s, 0);
   FOR i:=1 TO 4 DO BEGIN
      hojas[i]:= FALSE;
      INIT (puede[i], 0)
   END;
   COBEGIN
      P(I); P(2); P(3); P(4);
      gestor; Pg
   COEND
```

END. (* main *)

Por último, y para demostrar la potencia de las RCC, intentaremos resolver el problema del montacargas utilizando semáforos. El algoritmo se basa en inicializar un semáforo a 9 (¿número mágico?) y hacer que los coches pequeños ejecuten dos wait sobre el semáforo antes de entrar al montacargas y dos signal al salir. Los coches grandes ejecutarán 5 wait y 5 signal, respectivamente. Es decir: (pg73)

```
INIT (s, 9);
COBEGIN
   Vp(1); Vp(2); ...; Vp(10);
   Vg(1); Vg(2); ...; Vg(10)
COEND

"Vp"
      .....
      (* entrada *)
      WAIT (s); WAIT (s);
      (* dentro del montacargas *)
      SIGNAL (s); SIGNAL (S);
      (* fuera del montacargas *)

"Vg"
      .....
      (* entrada *)
      WAIT (s); WAIT (s); WAIT (s); WAIT (s); WAIT (s);
      (* dentro del montacargas *)
      SIGNAL (s); SIGNAL (s); SIGNAL (s); SIGNAL (s); SIGNAL (s);
      (* fuera del montacargas *)
```

Esta solución puede llegar a provocar un interbloqueo si 9 vehículos pequeños ejecutan el primer WAIT(s) de entrada. Solución: poner una RC:

```
"Vp"
      REGION v DO BEGIN
         WAIT (s); WAIT (s)
      END;
      (* dentro del montacargas *)
      REGION v DO BEGIN
         SIGNAL (s); SIGNAL (s)
      END;
      (* fuera del montacargas *)

"Vg"
      .....
      (* entrada *)
      REGION v DO BEGIN
         WAIT (s); WAIT (s); WAIT (s); WAIT (s); WAIT (s);
      END;     (pg74)
      (* dentro del montacargas *)
      REGION v DO BEGIN
         SIGNAL (s); SIGNAL (s); SIGNAL (s); SIGNAL (s); SIGNAL (s);
      END
      (* fuera del montacargas *)
```

Supongamos que tres vehículos pequeños están dentro. Esto significa que el contador del semáforo está a 3. Si intenta entrar 1 grande se quedará bloqueado en el cuarto WAIT, bloqueando con ello la RC y produciéndose un interbloqueo. De hecho, se ha violado uno de los principios de las RC: el tiempo que un proceso permanece dentro de una RC es finito.

Si se quitasen las RC de salida del montacargas se obviaría el problema. Pero un coche pequeño sólo ha salido realmente después de ejecutar 2 WAIT. Ahora dos pequeños concurrentemente ejecutan el primer WAIT de salida → ninguno de los ha salido realmente y ya puede entrar otro pequeño.

Queda, pues, claro que la RCC cumple con la labor de espera condicional.

Ahora intente responder a las siguientes cuestiones:

1.ª ¿Cree posible que en la condición del Await tan sólo intervengan variables no compartidas? ¿Por qué?

2.° Cuando un proceso evalúa a FALSE la condición del Await, ¿se queda dentro de la RC o la abandona?

3.ª Cuando hay procesos esperando en Qv y Qs y algún proceso abandona la RC, ¿quién entra primero, los de Qv o los de Qs?

## 2.4. Buzones

«Queremos escribir los números primos menores que 1.000 con numeración romana. Para hacerlo, utilizaremos dos procesos. Pl calculará los números primos y P2 los imprimirá en notación romana.»

El enunciado anterior es un típico problema de comunicación entre procesos. Pl debe mandarle los números calculados a P2, mientras que éste debe recibirlos de algún sitio. El esquema de comunicación sería el mostrado en la Fig. 19. (pg75)

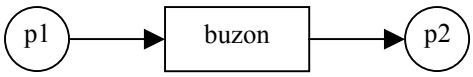



FIGURA19

Como se aprecia en la figura anterior, se ha utilizado un buzón para comunicar los dos procesos. Esta es una nueva herramienta cuya sintaxis es la mostrada en la Fig. 20:

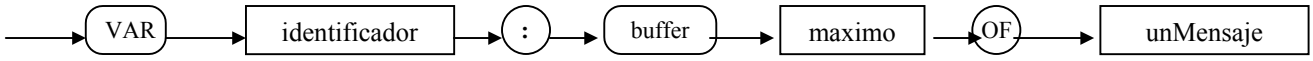



FIGURA 20

El *identificador* es el nombre de la variable buzón. La constante máximo indica la capacidad del buzón. Es decir, el máximo número de mensajes que puede albergar el buzón. El tipo *unMensaje* indica el tipo de mensajes que puede almacenar el buzón.

El buzón es un tipo abstracto de datos compuesto por una estructura de datos y un conjunto de operaciones asociadas a tal estructura. Las operaciones que se pueden realizar sobre una variable de tipo buzón son:

**SEND (mensaje, B)**

- Si el buzón B no está lleno, deja el mensaje y continúa.

- Si el buzón B está lleno, espera a que haya un hueco en el buzón donde depositarlo. Esto significa que el proceso pierde el procesador hasta que se dé la condición de buzón no lleno.

**RECEIVE (mensaje, B)**

- Si el buzón b no está vacío, recoge el mensaje y continúa.

- Si el buzón B está vacío, espera a que haya un mensaje. El proceso deja de ejecutarse hasta que se dé la condición de buzón no vacío.

Las operaciones SEND y RECEIVE sobre un mismo buzón se excluyen en el tiempo: son regiones críticas respecto del buzón. La semántica de los buzones implica las siguientes reglas de sincronización:

1. El productor no puede superar la capacidad del buzón.
2. El consumidor no puede consumir mensajes más rápidamente de lo que se producen.
3. El orden en el que se consumen los mensajes es el mismo que en el que se producen.

(pg76)

Si r representa el número de mensajes recibidos y e el número de mensajes enviados, las 3 reglas anteriores las podemos expresar como:

1. $0 \leq e\text{-}r \leq máx$
2. $2.0 \leq r \leq e$
3. for i: 1..r = > R(i) = E(i)

Con esto podemos obtener el siguiente **invariante** para los buzones:

$0 \leq r \leq e \leq r+max$
for i: l..r = > R(i) = E(i)

Con este invariante es fácil ver que un proceso productor y uno consumidor conectados a través de un buzón no pueden interbloquearse (si el buzón tiene una capacidad mayor que 0). De hecho:

Productor pasivo si e = r + max

Consumidor pasivo si e = r

Para que el productor y el consumidor estén parados al mismo tiempo, es necesario que max = 0.

No se puede preguntar sobre si un buzón está lleno o vacío. Las únicas operaciones permitidas son SEND y RECEIVE.

Con las reglas de sincronización definidas es posible conectar varios productores y consumidores a un solo buzón (Fig. 21).

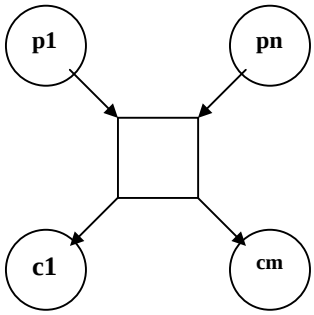



FIGURA 21

Cuando se hace esto, la entrada al buzón es una mezcla dependiente del tiempo de los mensajes de n procesos distintos, y la salida es una división dependiente del tiempo de mensajes de m receptores distintos. En este caso no hay relación funcional entre la salida de cada productor y la entrada de cada consumidor. (pg77)

Otro aspecto de interés es que las primitivas SEND y RECEIVE transmiten mensajes por valor. Cuando éstos son muy grandes interesa hacerlo por referencia (mandando y recibiendo la dirección del mensaje en cuestión).

Vamos a resolver ahora el problema de escribir los números primos en romano. Para comunicar a los procesos Pl y P2 vamos a declarar una variable de tipo buzón. El proceso productor (P1) después de calcular un número primo lo meterá en el buzón. El proceso consumidor (P2) cogerá un número del buzón y lo imprimirá. El programa es el algoritmo (2.4.1)

```
PROGRAM Primos; (* algoritmo (2.4.1) *)
   CONST tamaño = 10; (* máxima capacidad del buzón *)
   TYPE mensaje = INTEGER; (* tipo de mensaje que almacenará el buzón)
   VAR b: buffer tamaño OF mensaje;
   PROCEDURE buscaPrimos;
      VAR i: INTEGER;
      BEGIN
         FOR i:=1 TO 1000 DO
            IF esPrimo (i)
            THEN SEND (i, b);
         SEND (0, b)    (* centinela para indicar que ya no va a enviar más *)
      END;
   PROCEDURE escribePrimos;
      VAR is INTEGER;
      BEGIN
```

```
        RECEIVE (i, b);
        WHILE i # 0 DO BEGIN
            escribeRomano (i);
            RECEIVE (i, b)
        END
    END;
BEGIN (* Primos *)
    COBEGIN
        buscaPrimos
        escribePrimos
    COEND
END.
```
(pg78)

En el programa anterior ha sido necesario poner un centinela para decirle al proceso escribePrimos cuándo debe terminar. Si no fuera así, aunque el proceso que calcula los primos termine, el proceso que los imprime se que daría esperando en el RECEIVE. Puesto que ya no se van a enviar más, el proceso escribePrimos se quedaría esperando indefinidamente un mensaje.

Los buzones claramente sirven para comunicar procesos. Además actúan de amortiguadores de las velocidades relativas de cada uno de ellos. Evidentemente, si el proceso productor fuese más rápido que el consumidor, siempre se encontraría el buzón lleno y debería esperar. Si, por el contrario, el consumidor fuese más rápido, es éste quien tendría que esperar a que el buzón contuviese algún mensaje, ya que a lo sumo habría uno. En cualquier caso, uno de los principios en la programación concurrente es que no se puede aseverar nada sobre las velocidades relativas de los procesos. En este problema, el buzón permite que durante una cierta cantidad de tiempo el productor adelante al consumidor.

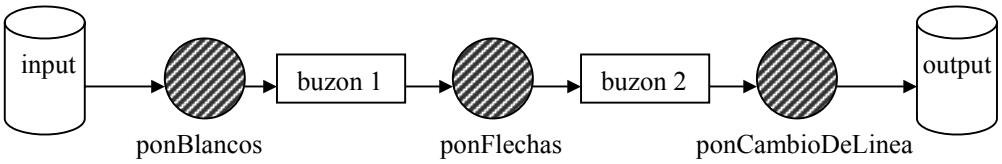
Veamos otro ejemplo: «Tenemos como entrada un fichero de caracteres que no contiene ningún caracter **!** Se trata de realizar las operaciones siguientes, una a continuación de otra:

1. Cambiar 1 caracter de cada 11 por un blanco.
2. Cambiar dos asteriscos seguidos por un ^. (Tres asteriscos darán ^*).
3. Escribir en un fichero los caracteres resultantes y un cambio de línea cada 13 caracteres.

El problema se presenta gráficamente en la Fig. 22. Existen 3 procesos, uno para cada una de las actividades indicadas anteriormente, comunicados mediante buzones.

Vamos a declarar dos variables de tipo buzón con una capacidad máxima de 3 caracteres y que almacenen mensajes de tipo CHAR. Las declaraciones son:

```
CONST capacidad = 3;
TYPE mensaje = CHAR;
VAR bl, b2: buffer capacidad OF mensaje;
```




(pg79)

Además vamos a utilizar el caracter **!** como indicador de fin de comunicación (ya que este caracter no aparece en el texto de entrada). Para ello declaramos

```
CONST caracterFin ='!';
```

Los procesos se van a ejecutar concurrentemente, con lo que podemos tener la primera aproximación a la solución del problema (algoritmo 2.4.2).

```
PROGRAM TransformaTexto; (* algoritmo (2.4.2): l.ª aproxi. *)
    CONST capacidad = 3; caracterFin = '!';
    TYPE mensaje = CHAR;
    VAR bl, b2: buffer capacidad OF mensaje;
    PROCEDURE ponBlancos;
        …..
        BEGIN
```

```
        …..
        …..
    END;
PROCEDURE ponFlechas;
    …..
    BEGIN
        …..
        …..
    END;
PROCEDURE ponCambioDeLinea;
    …..
    BEGIN
        …..
        …..
    END;
BEGIN (* TransformaTexto *)
    COBEGIN
        ponBlancos;
        ponFlechas;
        ponCambioDeLinea
    COEND
END.
```

(pg80)

Veamos ahora el proceso que se encarga de cambiar 1 caracter de cada 11 por un blanco (ponBlancos). Lo único que tiene que hacer es leer del fichero de entrada y llevar la cuenta del orden del caracter leído. Si hace el 11, cambiarlo por un blanco. A continuación, manda el caracter al buzón bl. Este proceso se repite hasta que detecte un fin de fichero en cuyo caso manda el caracter !, indicando que ya no mandará nada más. El procedimiento descrito se implementa como:

```
PROCEDURE ponBlancos;
    VAR  i: INTEGER;
         c: CHAR;
    BEGIN
        i:= 0;
        Read(c);
        WHILE NOT EOF DO BEGIN
            i:= i + 1;
            IF (i mod 11) = 0
            THEN c:=' ' ;
            SEND (c, bl);
            Read(c)
        END;
        SEND (caracterFin, bl)
    END;
```

Para implementar el proceso que convierte asteriscos por flechas conviene describir el proceso con el autómata de la fig. 23.

| | c = '*' | c # '*' |
|---|---|---|
| pendiente | SEND('^', b2)<br>pendiente:= FALSE | SEND('*', b2)<br>SEND(c, b2)<br>pendiente:= FALSE |
| No pendiente | pendiente:= TRUE | SEND(c, b2) |

FIGURA 23

(pg81)

Ahora, escribir el procedimiento ponFlechas es trivial si seguimos el autómata anterior.

```
PROCEDURE ponFlechas;
   VAR  pendiente: BOOLEAN;
        c: CHAR;
   BEGIN
      pendiente:= FALSE;
      REPEAT
         RECEIVE (c, bl);
         IF pendiente
         THEN BEGIN
            IF c = '*'
            THEN SEND ('^', b2)
            ELSE BEGIN
                    SEND ('*', b2);
                    SEND (c, b2)
                 END;
            pendiente:= FALSE
            END
         ELSE IF c ='*'
            THEN pendiente:= TRUE
            ELSE SEND (c, b2)
      UNTIL c = caracterFin
   END;
```

Por último, el proceso que escribe cambios de líneas (ponCambioDeLínea) se detalla a continuación:

```
PROCEDURE ponCambioDeLinea;
   VAR  i: INTEGER;
        c: CHAR;
   BEGIN
      i:= 0;
      RECEIVE (c, b2);
      WHILE c # caracterFin DO BEGIN
         Write (c);
         i:= i + 1;
         IF (i mod 13) = 0 THEN Writeln;
         RECEIVE (c, b2)
      END
   END;      (pg82)
```

El programa resultante se muestra en la segunda aproximación del algoritmo (2.4.2).

```
PROGRAM TransformaTexto; (* algoritmo (2.4.2): 2.ª aproxi. *)
   CONST  capacidad = 3;
          caracterFin = '!';
   TYPE mensaje = CHAR;
   VAR bl, b2: buffer capacidad OF mensaje;

   PROCEDURE ponBlancos;
      VAR  i: INTEGER;
           c: CHAR;
      BEGIN
         i:= 0;
         Read (c);
         WHILE NOT EOF DO BEGIN
            i:= i + 1;
```

```
            IF (i mod 11) = 0
            THEN c:= ' ';
            SEND (c, bl);
            Read(c)
        END;
        SEND (caracterFin, bl)
    END;
  PROCEDURE ponFlechas;
    VAR  pendiente: BOOLEAN;
         c: CHAR;
    BEGIN
      pendiente:= FALSE;
      REPEAT
      RECEIVE (c, bl);
      IF pendiente
      THEN BEGIN
            IF c ='*'
            THEN SEND ('-', b2)
            ELSE BEGIN
                  SEND ('*', b2);
                  SEND (c, b2)
            END;
            pendiente:= FALSE
      END          (pg83)
      ELSE IF c ='*'
      THEN pendiente:= TRUE
      ELSE SEND (c, b2)
      UNTIL c = caracterFin
    END;
  PROCEDURE ponCambioDeLinea;
    VAR  i: INTEGER;
         c: CHAR;
    BEGIN
    i:= 0;
    RECEIVE (c, b2)
    WHILE c # caracterFin DO BEGIN
      Write (c);
      i:= i + 1;
      IF (i mod 13) = 0 THEN Writeln;
      RECEIVE (c, b2)
    END
  END;
BEGIN ( TransformaTexto *)
  COBEGIN
    ponBlancos;
    ponFlechas;
    ponCambioDeLinea
  COEND
END.
```

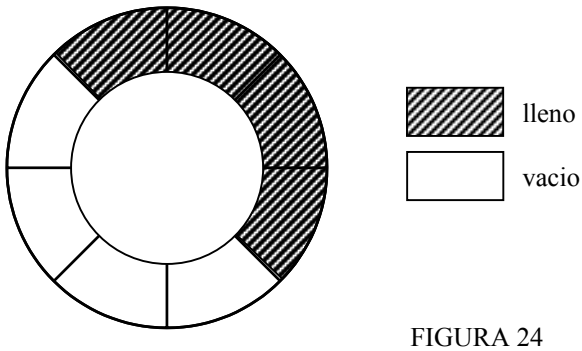



FIGURA 24

Mostraremos a continuación cómo se puede implementar un buzón de mensajes y las primitivas SEND y RECEIVE en función de las otras primitivas vistas hasta ahora (semáforos, RC y RCC).

Consideraremos que un buzón consiste de un número finito de elementos idénticos puestos en círculo. El círculo consiste en una secuencia de elementos vacíos que pueden ser llenados por un productor y una secuencia de elementos llenos que pueden ser vaciados por un consumidor (Fig.24).(pg84)

El productor y el consumidor hacen referencia a los elementos vacíos y llenos a través de dos punteros, p y c, que se mueven en el sentido de las agujas del reloj (sin adelantarse uno a otro).

Podemos declarar:

```
buffer: ARRAY [O..max-1] OF T;
p, c: O..max-1;
```

y para lograr la semántica de las operaciones SEND y RECEIVE podemos escribir la siguiente implementación utilizando RCC.

```
type B = SHARED RECORD
            buffer: ARRAY [O..max-1] OF T;
            p, c: O..max-1;
            lleno: 0...max; (* número de elementos del buzón que están llenos llenos *)
         END;
(* inicialmente p = c = lleno = 0 *)
PROCEDURE SEND (mensaje:T; VAR b: B);
   BEGIN
      REGION b DO BEGIN
         AWAIT lleno < max;
         buffer [p]:= mensaje;
         p:= (p+l) mod max;
         lleno:= lleno + 1
      END
   END;
PROCEDURE RECEIVE (VAR mensaje: T; VAR b: B);
   BEGIN
      REGION b DO BEGIN
         AWAIT lleno > 0;
         mensaje:= buffer [c];
         c:= (c+l) mod max;
         lleno:= lleno - 1
      END
   END;
```

Observe que el cuerpo de ambos procedimientos está por completo dentro de una RC. Las operaciones SEND y RECEIVE son RC respecto del buzón: se excluyen mutuamente en el tiempo.

Observe también que lleno inicialmente es 0. Aumenta en una unidad (pg85) cada envía y disminuye en una unidad cada recibe. Por tanto, lleno = e - r. La recepción sólo se produce cuando lleno > 0 o e > r; el envío sólo se realiza cuando lleno < max o e-r < max. Luego, la solución descrita es correcta.

Es interesante especular sobre lo que sucedería si en cualquiera de los dos procedimientos anteriores separamos en dos regiones críticas, por una parte, el await, y, por otra, el resto del procedimiento. Podría ocurrir que dos procesos realicen el SEND al mismo tiempo, quedando sólo un hueco en el buzón. En estas circunstancias los dos pasarían la primera RC (la que contiene el await) y los dos dejarían el mensaje. Obviamente, el último sobrescribe al primer mensaje que tuviese que recibir el consumidor. La moraleja es que es peligroso separar una condición de sincronización de la RC que le sigue.

Resolveremos ahora el mismo problema, pero utilizando RC y semáforos. Declaramos un semáforo «lleno» (inicialmente a 0) y otro semáforo «vacío» (inicialmente al valor max). Antes de recibir, se ejecutará un wait (lleno) y después de enviar un signal (lleno). De la misma forma, antes de enviar se ejecutará un wait (vacío) y después de recibir un signal (vacío). La solución completa es la siguiente:

```
type B: RECORD
            v: SHARED RECORD
                  buffer: ARRAY [O..max-1] OF T;
                  p, c: O..max-1
               END;
               vacío, lleno: SEMAPHORE
               END;
(* inicialmente p = c = lleno = 0 & vacío = max *)
PROCEDURE SEND (mensaje:T; VAR b: B);
```

```
   BEGIN
      WITH b DO BEGIN
         WAIT (vacío);
         REGION v DO BEGIN
            buffer [p]:= mensaje;
            p:= (p+1) mod max
         END;
         SIGNAL (lleno)
      END
   END;        (pg86)
PROCEDURE RECEIVE (VAR mensaje: T; VAR b: B);
   BEGIN
      WITH b DO BEGIN
         WAIT (lleno);
         REGION v DO BEGIN
            mensaje:= buffer [c];
            c:= (c+l) mod max;
         END;
         SIGNAL (vacío)
      END
   END;
```

Observe que los buzones definidos e implementados en esta sección tienen un comportamiento «bloqueante». Es decir, si un proceso ejecuta una operación SEND sobre un buzón lleno o una operación RECEIVE sobre un buzón vacío, dicho proceso se bloqueará.

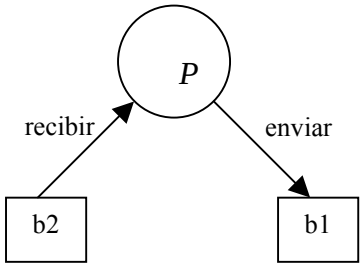
Hay situaciones en las que este comportamiento no es deseable. Imagínese el caso en el que un proceso hace pooling sobre varios buzones, de tal manera que si uno de ellos no tiene mensajes, debería comprobar los otros por si tienen alguno. Con los buzones definidos hasta ahora este comportamiento no se puede implementar.

Lo que necesitamos son buzones en los que un proceso no se bloquee en el caso de que no pueda completar la operación requerida. La implementación de este «buzón no bloqueante» y de las primitivas ENVIAR y RECIBIR podría ser la siguiente:

```
TYPE buzonNoBloqueante =   SHARED RECORD
            buffer: ARRAY [O..max-1] OF T;
            p,c : O..max-1;
            lleno: O..max
                                 END;
(* inicialmente p=c=lleno=0 *)
PROCEDURE ENVIAR (mensaje: T; VAR b: buzonNoBloqueante; VAR hecho: BOOLEAN);
   BEGIN
      REGION b DO
         IF lleno = max
         THEN hecho:= FALSE
         ELSE BEGIN        (pg87)
               buffer [p]:= mensaje,
               p:= (p+1) mod max;
               lleno:= lleno + 1;
               hecho:= TRUE
               END
   END;
PROCEDURE RECIBIR (VAR mensaje: T;VAR b: buzónNoBloqueante; VAR hecho: BOOLEAN);
   BEGIN
      REGION b DO
         IF lleno = 0
         THEN hecho:= FALSE
         ELSE BEGIN
               mensaje:= buffer[c];
```

```
            c:= (c + 1) mod max;
            lleno := lleno -1;
            hecho:= TRUE
            END
    END;
```

Como puede apreciarse, ahora las primitivas ENVIAR y RECIBIR llevan un parámetro adicional que indica si la operación se pudo realizar o no. Es responsabilidad del usuario del buzón decidir qué hacer si la operación no se pudo realizar. Para mostrar su uso, veamos el ejemplo de la Fig. 25:




(pg88)

FIGURA 25

```
VAR  bl, b2: buzonNoBloqueante;
       ml, m2: unMensaje;
       recibido, enviado: BOOLEAN;
REPEAT
    enviado:= FALSE; recibido:= FALSE;
    REPEAT
       ENVIAR (ml, bl, enviado);
       IF NOT enviado
       THEN RECIBIR (m2, b2, recibido)
    UNTIL enviado OR recibido;
    IF enviado
    THEN (* producir otro *)
    ELSE (* consumir lo recibido *);
FOREVER
```

Para terminar con el tema de buzones, comentar que en ciertas aplicaciones se emplean buzones cuyo comportamiento es una combinación de bloqueante y no bloqueante. Son los buzones con timeout. En este tipo de buzones, cuando se especifica una operación send o receive, también se indica el tiempo máximo que el proceso puede estar bloqueado por dicha operación. Si el proceso se bloquea y el tiempo especificado expira, el proceso abandona el intento de operar con el buzón, siendo responsabilidad del programador decidir qué hacer en este caso.

## 2.5. Sucesos

Cuando vimos las RCC estudiamos un problema relacionado con un montacargas. En la solución dada a dicho problema con RCC se observaba que cuando un proceso abandonada la RC, permitía a los que estaban esperando en el AWAIT evaluar de nuevo su condición. Esto nos llevaba a que si en el montacargas había 4 vehículos pequeños y estaban esperando entrar 2 grandes y 2 pequeños, cuando salía uno pequeño se despertaba a los 4 procesos, independientemente de que los grandes no tengan oportunidad de entrar. Esto, evidentemente, supone una pérdida de eficacia en el procesador, que pasa a ejecutar procesos que inmediatamente se dormirán de nuevo.

Lo que deberíamos hacer es despertar únicamente a los pequeños, ya que los grandes no tienen oportunidad de entrar. Para resolver esta cuestión, introduciremos una nueva herramienta: **los sucesos.** Con esta herramienta (pg89) pretendemos conseguir una gestión explícita de la cola de entrada a la RC, diciendo quién puede entrar y quién no.

Un suceso siempre debe ir asociado a una RC. La sintaxis de la declaración de un suceso es la mostrada en la Fig. 26:

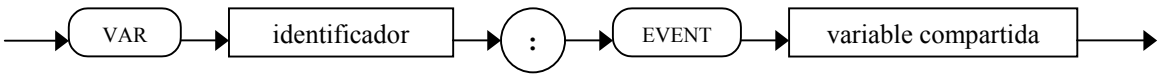



FIGURA 26

Por ejemplo, podemos declarar un suceso *ev* asociado a la variable compartida v como:

```
VAR v: SHARED RECORD
          .....
          ev: EVENT v
       END;
```

El suceso es un tipo abstracto de datos compuesto por una estructura de datos (una cola) y que soporta las siguientes operaciones:

**AWAIT (ev)**

El proceso que ejecuta esta operación abandona la RC donde se ejecutó, pasa a la cola del suceso *ev* asociada con dicha RC y se suspende, cediendo la CPU a otro proceso.

**CAUSE (ev)**

Los procesos encolados en este suceso pasan a la cola principal de la RC y entrarán a la misma cuando ésta quede libre. Si no hay nadie en la cola del suceso, ésta operación no tiene efecto. En cualquier caso, el proceso que ejecutó el Cause continúa.

Las dos operaciones anteriores se excluyen mutuamente en el tiempo y tan sólo se pueden ejecutar dentro de una RC. Puede haber varios sucesos asociados a una misma variable compartida. Si ha N sucesos, existen N+1 colas (las N de los sucesos y la cola de entrada a la RC). Por ejemplo, podríamos declarar: (pg90)

```
VAR v: SHARED RECORD
          .....
          ev0: EVENT v;
          evl: EVENT v;
          ev2: EVENT v;
       END;
```

Para ver un ejemplo de cómo funcionan los sucesos, considere la sección de código siguiente ejecutada por procesos Ai y Bi:

```
«Ai»                          «Bi»
  .....                         .....
  REGION v DO BEGIN               REGION v DO BEGIN
    .....                           .....
    AWAIT (ev);                     CAUSE (ev);
    .....                           .....
  END;                            END;
  .....                           .....
```

Supongamos que se ejecutan sucesivamente Al y A2. Ambos caerán en el AWAIT, por lo que pasarán a la cola asociada al evento ev, abandonan la RC y se suspenden. Si a continuación se ejecuta el Bl, cuando ejecute la sentencia CAUSE, pasará a los procesos Al y A2 a la cola de entrada a la RC. Cualquiera de estos procesos podrá entrar a la RC v cuando el proceso B1 la abandone y ejecutar la siguiente sentencia a AWAIT (ev).

Para demostrar la eficacia de los sucesos en lo que se refiere a la gestión explícita de colas, vamos a resolver un problema utilizando primero RCC y luego sucesos. El problema es el siguiente: «cierto número de procesos comparten un fondo de recursos equivalentes. Cuando hay recursos disponibles, un proceso puede adquirir uno inmediatamente; en caso contrario, debe enviar una petición y esperar a que se le conceda un recurso».

La solución con RCC es la siguiente:

```
TYPE  P = 1..nmrProcesos;
      R = 1..nmrRecursos;
VAR v: SHARED RECORD
          disponibles: SEQUENCE OF R;
          peticiones: QUEUE OF P;
          turno: ARRAY P OF BOOLEAN          (pg91)
```

```
        END;
PROCEDURE reserva (proceso: P; VAR recurso: R);
   BEGIN
      REGION v DO BEGIN
         WHILE empty (disponibles) DO BEGIN
            mete (proceso, peticiones);
            turno [proceso]:= FALSE;
            AWAIT turno[proceso]
         END;
         toma (recurso, disponibles)
      END
   END;
PROCEDURE libera (recurso: R);
   BEGIN
      REGION v DO BEGIN
         devuelve (recurso, disponibles);
         IF NOT empty (peticiones)
         THEN BEGIN
                  saca (proceso, peticiones);
                  turno [proceso]:= TRUE
               END
      END
   END;
```

La primera objeción a esta solución es que el programador no controla el orden en que los procesos entran a sus RC a partir de la cola principal. Es posible que un proceso Pl haga una petición cuando no hay recursos disponibles. Por ello introduce su identificación en peticiones, pone su turno a false y espera en el await. Si ahora un proceso P2 libera un recurso, sacará la identificación de Pl de la cola de peticiones, pondrá su turno (el de PI) a true y abandonará la RC v. En este momento se despierta a PI, que podrá entrar a la RC a evaluar de nuevo la condición del await (que ahora es true). Pero antes de que lo haga, un proceso P3 puede realizar una petición entrando a la RC a través de la cola principal Qv y tomar el recurso que en principio fue concedido a PI.

La solución a este problema sería establecer que los procesos que entran a la RC a través de la cola de suceso Qs tengan prioridad sobre los procesos que intentan entrar a la RC directamente a través de la cola principal Qv.

La segunda objeción a este algoritmo es que es ineficaz. Si hay 7 procesos esperando un recurso y se libera uno, la semántica de la RCC implica despertar a los 7. Tan sólo uno evaluará a true la condición del await, por lo que los (pg92) otros 6 se volverán a dormir. Esto es inevitable porque sólo hay una cola de suceso asociada con la RC.

Para controlar la gestión explícita de la cola, debemos ser capaces de asociar un número arbitrario de colas de sucesos con una variable compartida y controlar la transferencia de procesos a y desde ellas. Como vimos al principio de esta sección, la cuestión queda resuelta utilizando sucesos.

Un algoritmo que resuelve el problema propuesto, utilizando sucesos, es el siguiente:

```
TYPE P = 1..nmrProcesos;
     R = 1..nmrRecursos;
VAR v: SHARED RECORD
            disponibles: SEQUENCE OF R;
            peticiones: QUEUE OF P;
            turno: ARRAY P OF EVENT v
         END;
PROCEDURE reserva (proceso: P; VAR recurso : R);
   BEGIN
      REGION v DO BEGIN
         WHILE empty (disponibles) DO BEGIN
            mete (proceso, peticiones);
```

```
            AWAIT (turno[proceso])
        END;
        toma (recurso, disponibles)
    END
  END;
PROCEDURE libera (recurso: R);
  BEGIN
    REGION v DO BEGIN
      devuelve (recurso, disponibles);
      IF NOT empty (peticiones);
      THEN BEGIN
            saca (proceso, peticiones);
            CAUSE (turno [proceso])
            END
    END
  END;
```

Como puede comprobar, este algoritmo no obvia la primera objeción citada anteriormente: un recurso concedido a un proceso puede ser «robado» (pg93) por otro. Sólo si se da mayor prioridad a los procesos que entran a la RC a partir de una cola de suceso que a los que lo hacen a través de la cola principal Qv de la RC, se obviaría el problema. Por otra parte, esto significaría hacer explícita para el programador la política de gestión de colas.

Por el contrario, si resuelve la segunda objeción. Ahora sólo se despierta a un proceso y no a todos los que estaban esperando. Somos, pues, capaces de indicar a qué proceso va destinado el recurso, algo que era totalmente imposible utilizando RCC.

Veamos otro problema de aplicación: «A una ventanilla llegan alumnos para matricularse. Cuando a uno le toca el turno, le dan un papel para que ponga una póliza verde. Sale de la cola y espera al ordenanza que vende pólizas verdes. Una vez conseguida, vuelve a la cola de la ventanilla y cuando le toca el turno, le dan otro papel para que ponga una póliza azul. Sale de la cola y espera al ordenanza que vende pólizas azules. Cuando la ha conseguido, va de nuevo a la cola de la ventanilla y cuando le toca el turno, por fin se matricula y se va».

En este enunciado es fácil reconocer que la ventanilla constituye una RC en cuanto que en ella sólo puede estar un estudiante al mismo tiempo. El hecho de tener que esperar por pólizas es comparable a tener que esperar a que se produzca un evento. En realidad, y puesto que existen dos eventos (pólizas azules y verdes), habrá dos colas de espera. De hecho, cuando llegue el ordenanza que vende las pólizas verdes no hay por qué despertar a los alumnos que están esperando por pólizas azules. Para resolver el problema declararemos, por tanto, dos eventos asociados a una variable compartida. El programa que resuelve el anterior enunciado es el algoritmo 2.5.1.

```
PROGRAM Matricula; (* algortimo (2.5.1)*)
  VAR ventanilla: SHARED RECORD
                        polizaVerde, polizaAzul : EVENT ventanilla
                  END;
PROCEDURE Alumno (i: INTEGER);
  BEGIN
    llega a la ventanilla
    REGION ventanilla DO BEGIN
      …..
      AWAIT (polizaVerde);
      …..
      AWAIT (polizaAzul)
      …..
    END
  END;        (pg94)
PROCEDURE Ordenanza1;
  BEGIN
    REPEAT
```

```
            ve a por polizas verdes
            REGION ventanilla DO
                CAUSE (polizaVerde);
        FOREVER
    END;
PROCEDURE Ordenanza2;
    BEGIN
        REPEAT
            ve a por polizas azules
            REGION ventanilla DO
                CAUSE (polizaAzul);
            FOREVER
    END;
BEGIN (* Matrícula *)
    COBEGIN
        Alumno (1); Alumno (2); ..., Alumno (2000);
        Ordenanzal; Ordenanza2
    COEND
END.
```

Observe el procedimiento Alumno. Cuando llega a la ventanilla (a la RC), espera en la cola a que sea su turno (cola de entrada a la RC. Podrá entrar cuando la RC esté vacía y sea el primero en la cola). El alumno habla con el secretario (.... ) y abandona la cola a la espera de que el ordenanza que vende las pólizas verdes le proporcione una (ejecuta el AWAIT, que hace que el proceso abandone la RC, permitiendo entrar a otro y pasando a una cola de espera). Cuando tiene la póliza verde (el ordenanza1 ejecuta CAUSE(pólizaVerde)) se pone de nuevo en la cola de la ventanilla (de la RC) y cuando sea su turno entrará. Una vez dentro, habla de nuevo con el secretario, que le pide una póliza azul. Como no la tiene, se repite el comportamiento anterior, esperando esta vez al ordenanza que vende las pólizas azules. Una vez conseguida, va a la ventanilla, espera su turno y se matricula. Observe que los procesos Ordenanza-i para ejecutar el CAUSE deben entrar a la RC, lo que significa que los ordenanzas se confunden con los alumnos en la cola, y que también esperan llegar a la ventanilla. Imagine que los ordenanzas van al estanco a comprar pólizas, se ponen en la cola y cuando llega su turno, se las dan al secretario. A continuación avisan por el megáfono que todos los alumnos que esperan la póliza azul (o verde), ya pueden comprarla en la ventanilla. A continuación, el Ordenanza-i se va de nuevo al (pg95) estanco. El justificante de que antes ya se había estado en la cola es el papel en el cual estampar la póliza (el contador de programa). Es importante fijarse en que un Cause (ev) despierta a todos los procesos dormidos en un Await (ev).

Veamos ahora cómo se puede solucionar el problema del montacargas utilizando sucesos (algoritmo (2.5.2)).

```
PROGRAM Vehículos; (* algoritmo (2.5.2) *)
    VAR montacargas: SHARED RECORD
                                nmrVg: 0..1;
                                nmrVp: 0..4;
                                puedeGrande, puedePequeño: EVENT montacargas
                            END;
PROCEDURE Vg (i: INTEGER);
    BEGIN
        REGION montacargas DO
            IF ( (nmrVp <- 2) & (nmrVg = 0)
            THEN nmrVg:= nmrVg + 1
            ELSE AWAIT (puedeGrande);
        (* en el montacargas *)
        REGION montacargas DO BEGIN
            nmrVg:= nmrVg - 1;
            CAUSE (puedeGrande);
            CAUSE (puedePequeño)
        END
```

```
    END;
PROCEDURE Vp (i: INTEGER);
    BEGIN
        REGION montacargas DO
            IF ( (nmrVp < 4) & (nmrVg = 0) ) OR ( (nmrVp <_ 1) & (nmrVg = 1) )
            THEN nmrVp:= nmrVp + 1
            ELSE AWAIT (puedePequeño);
        (* en el montacargas *)
        REGION montacargas DO BEGIN
            nmrVp:= nmrVp - 1);
            CAUSE (puedePequeño);
            IF (nmrVg = 0) & (nmrVp < 2)
            THEN CAUSE (puedeGrande)
        END
    END;                (pg96)
BEGIN (* Vehículos*)
    nmrVg:= 0;
    nmrVp:= 0;
    COBEGIN
        Vp (1); Vp (2); ...; Vp (10);
        Vg (1); Vg (2); ...; Vg (10)
    COEND
END.
```

La objeción que hay que hacer al programa anterior es que no funciona según las especificaciones. Imagine que en el montacargas hay un vehículo grande. En esta situación dos vehículos grandes intentan entrar y quedan parados en el AWAIT. Cuando salga el vehículo grande despertará a los que estaban dormidos. Ahora el montacargas está vacío. Los dos grandes continúan por la sentencia posterior al AWAIT. Es decir, entran los dos al montacargas y esto es erróneo (aparte del hecho de que no incrementan nmrVg). Esta situación se puede arreglar si cuando se despierta a un proceso, se le hace que compruebe de nuevo la condición de entrada. Para ello basta sustituir las sentencias IF que dan el acceso al montacargas por sentencias WHILE:

```
WHILE NOT ((nmrVg=O) & (nmrVp<2)) DO AWAIT (puedeGrande);
nmrVg:= nmrVg + 1;
```

y para el caso de los vehículos pequeños:

```
WHILE NOT [ (nmrVp < 4 & nmrVg=O) OR (nmrVp < 1 & nmrVg=1) ] DO AWAIT (puede Pequeño);
nmrVp:= nmrVp + 1;
```

Observe que cuando un vehículo grande abandona el montacargas se despierta tanto a los pequeños como a los grandes que estaban esperando entrar. Esto es similar al comportamiento de las RRC. Evidentemente, el hueco que deja un vehículo grande puede ser cubierto por otro grande o por dos pequeños. Sin embargo, cuando un vehículo pequeño abandona el montacargas, despertará a los de su mismo tamaño y a los grandes sólo si hay espacio para que entre uno de ellos. Este comportamiento es diferente del de las RCC, en las que cuando un vehículo pequeño abandona una RC, despierta incondicionalmente a todos. Aquí el uso de sucesos nos ha servido para gestionar de una manera explícita a qué procesos despertar. (pg97)

Para terminar, veamos cómo se puede implementar una RCC utilizando sucesos.

El constructor:

```
REGION v DO BEGIN
    Sl;
    AWAIT B;
    S2
END;
```

se puede sustituir por:

```
REGION v DO BEGIN
    S1;
```

```
        WHILE NOT B DO AWAIT ($evl);
        S2
        CAUSE ($evl)
    END;
```

Mientras que B no sea cierto el proceso se encola en un evento ($ev 1) declarado por el compilador y asociado con la variable compartida v. Además y puesto que la semántica de la RCC impone que cuando un proceso complete su RC, debe despertar a todos los que estaban esperando en el await, debemos añadir una operación CAUSE ($evl) como última sentencia en la RC. Por supuesto, la operación cause hay que ponerla al final de todas las RCC contengan o no la sentencia await.

## 2.6. Monitores

Las anteriores herramientas descritas para manejar la ejecución concurrente de procesos nos proporcionan un enfoque conceptual de los problemas inherentes a la programación concurrente a la vez que nos dotan de mecanismos para evitar problemas tales como el interbloqueo, inanición, etc., haciendo la programación más fiable. De las herramientas definidas, pocas de ellas están incorporadas como constructores en lenguajes de programación. Quizás las más extendidas sean el semáforo y el buzón, aunque la sintaxis y la semántica varíen dependiendo del lenguaje que las soporte.

Ahora vamos a describir otro constructor más para controlar la interacción entre procesos concurrentes: el **monitor**. Esta herramienta está incorporada en lenguajes tales como el Concurrent Pascal, Concurrent Euclid, CCNPascal, CSP/K, etc. Fue propuesta por Brinch Hansen y por Hoare, aunque de manera independiente. (pg98)

Un monitor es un mecanismo que permite compartir de una manera fiable y efectiva tipos abstractos de datos entre procesos concurrentes. Así pues, un monitor proporciona:

a) abstracción de datos.

b) exclusión mutua y mecanismos de sincronización entre procesos. A continuación describiremos estos dos aspectos con más detalle.

### Abstracción de datos: Clase

Un tipo abstracto de datos está caracterizado por un conjunto de declaraciones de variables cuyos valores definen el estado de una instancia de ese tipo, y por un conjunto de operaciones que actúan sobre dichas variables.

Gráficamente un tipo abstracto de datos se puede representar como se indica en la Fig. 27.

Esta abstracción de datos ha sido implementada de diferentes formas según el lenguaje que la soporta.

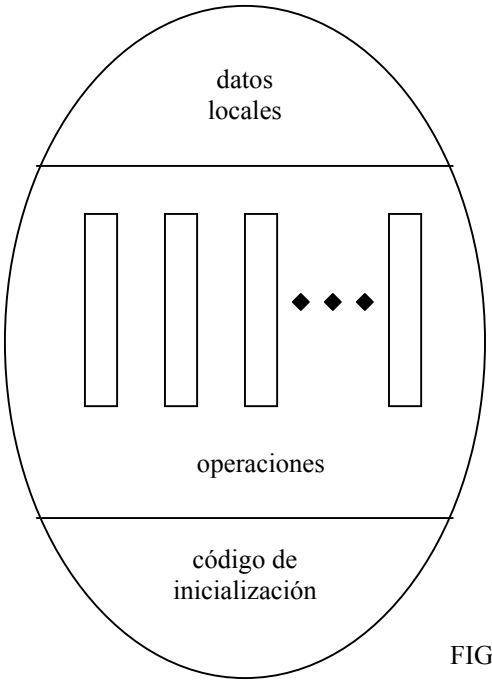



FIGURA 27

(pg99)

Así en Simula-67, un módulo que proporcione estas características se denomina **clase**, al igual que en el Concurrent Pascal de Brinch Hansen. En lenguajes tales como el Modula-2 dicha abstracción se consigue a través de los tipos opacos.

La estructura de una clase es la siguiente:

```
type nombreClase = class (parametros)
        declaracion de variables;
        procedure P1 (...);
            begin ... end;
        procedure P2 (...);
            begin ... end;
            …
            …
        procedure Pn (...);
            begin ... end;
        begin
            codigo de inicializacion
        end;
```

Las variables declaradas dentro de la clase sólo pueden ser accedidas mediante los procedimientos declarados en la misma. Un usuario de la clase sólo puede manipular su estructura de datos (de la clase) a través de los procedimientos que ofrece. Es más, puede que alguno de dichos procedimientos sea interno a la propia clase, lo que significa que un usuario de la clase no puede invocarlo. Para diferenciar los procedimientos privados de la clase de aquellos que pueden ser invocados desde fuera, a los segundos se les añade la palabra reservada **entry**. Por ejemplo:

procedure **entry** P1 (...);

Para ilustrar este concepto supongamos que queremos definir una clase para distribuir memoria en tamaños fijos (páginas, bloques de disco, etc.). La información acerca de la memoria libre se mantiene en un vector de bits.

```
TYPE gestorMemoria = class (numeroElementos:INTEGER)
   VAR    libres: ARRAY [1..numeroElementos] OF BOOLEAN;
                  i: INTEGER;
   PROCEDURE ENTRY adquiere (VAR indice: INTEGER);
      BEGIN
         indice:= 1;
         WHILE (indice < = numeroElementos) & (NOT libres [indice]) DO indice:= indice + 1; (pg100)
         IF indice < = numeroElementos
         THEN libres [indice]:= FALSE
         ELSE indice:= -1  (* no hay memoria disponible *)
      END;
   PROCEDURE ENTRY libera (indice: INTEGER);
      BEGIN
         libres [indice]:= TRUE
      END;
   BEGIN
      FOR i:=1 TO numeroElementos DO libres [i]:= TRUE
   END;
```

Las clases se inicializan una sola vez a través de la sentencia **init** y sus variables locales permanecen durante todo el programa. Así, en el ejemplo anterior, si un proceso quisiera utilizar la clase gestorMemoria para gestionar la memoria de disco podríamos declarar:

```
CONST nmrBloques = 3000;
```

```
VAR disco: gestorMemoria;
BEGIN (* del proceso *)
    init disco (nmrBloques);
    ......
END;
```

La sentencia init crea el espacio para las variables de la clase y además permite que se ejecute el código de inicialización de la misma. Si ese proceso necesita adquirir un bloque de disco ejecutará:

```
disco.adquiere (bloque)
```

y para liberarlo efectuará la llamada:

```
disco.libera (bloque)
```

**Exclusión mutua y sincronización**

Un monitor es muy similar a una clase en el sentido de que esconde la representación interna de sus variables y proporciona al exterior sólo el comportamiento funcional definido por los procedimientos exportados (entry). Pero un monitor proporciona más: garantiza que el número de procesos que (pg101)

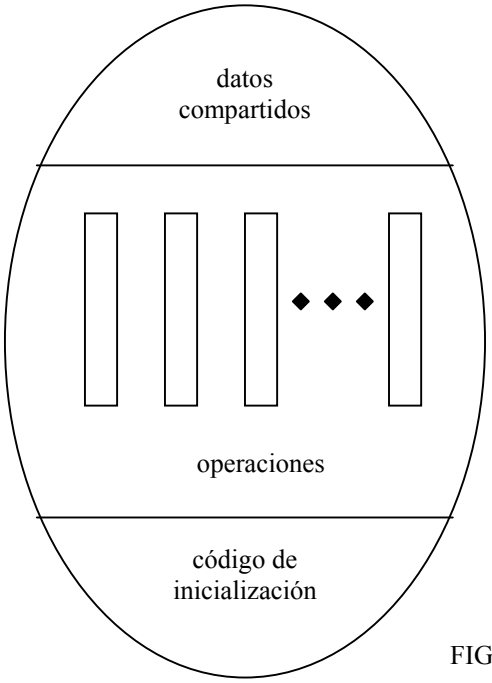



FIGURA 28

en un instante de tiempo están ejecutando código del monitor es como máximo de 1. Esta propiedad, exclusión mutua, asegura la consistencia de los datos del monitor (ver Fig. 28). Además el monitor proporciona un mecanismo de sincronización entre procesos: el constructor **queue**. Un procedimiento monitor puede retrasar a un proceso durante una cantidad arbitraria de tiempo ejecutando una operación **delay** sobre una variable de tipo queue. En un instante t de tiempo sólo un proceso puede estar esperando en una queue. (En algunas implementaciones se permite que el número de procesos encolados sea mayor que 1.) Cuando un proceso es retrasado por la operación delay, pierde el acceso al monitor (lo abandona) y se suspende hasta que otro proceso le reanude con una operación **continue** sobre la variable queue en la que se retrasó el primero. Si se permite que existan varios procesos encolados en una queue la operación continue despertará a uno de ellos en función de la política de gestión de la cola. Lo más frecuente es que ésta sea FIFO (evita el lockout), aunque si los procesos tienen un atributo de prioridad se desencolará al más prioritario, para lo cual la operación delay necesita un parámetro adicional que es la prioridad del proceso (aunque esto puede provocar lockout).

En cualquier caso, cuando se ejecuta una operación continue corremos el peligro de que en el monitor se estén ejecutando concurrentemente dos procesos: el que ejecutó la operación continue y el que estaba esperando y ahora es reanudado por la siguiente sentencia a delay. Para evitar esta situación (pg102) Hoare propuso que el proceso que ejecutó continue abandone el monitor y se suspenda en una cola de alta prioridad del mismo. Ahora el proceso recién despertado entra de nuevo al monitor y terminará el procedimiento que estaba ejecutando. Cuando el monitor quede libre se permite que el

proceso que estaba en la cola de alta prioridad reanude su ejecución dentro del monitor. Brinch Hansen también propuso que el proceso que ejecutó la operación continue abandone inmediatamente el monitor. Pero en lugar de encolarse, Hansen obligó a que la operación continue fuese la última de un procedimiento monitor. Esto tiene el problema de que no puede ejecutarse más de una operación continue dentro de un procedimiento monitor, pero tiene la ventaja de que la implementación de esta política es muy sencilla.

En cualquiera de los dos métodos, la filosofía es la misma: se reanuda inmediatamente el proceso suspendido, con lo cual se asegura que el estado de las variables del monitor no se ha modificado en el intervalo desde que se ejecutó la operación continue y el proceso despertado ha reanudado su ejecución.

Quizás el ejemplo más claro para poner de relieve el uso de monitores sea el problema del productor-consumidor, que ahora podemos implementarlo como:

```
PROGRAM productorConsumidor;
   TYPE buffer = monitor (capacidad: INTEGER); (* implementa un buffer de caracteres *)
   VAR contenido: ARRAY [O..capacidad-1] OF CHAR;
          in, out: O..capacidad-1;
          cuantosHay: O..capacidad;
          pro, con: queue;
   PROCEDURE lleno (): BOOLEAN;
      BEGIN
         RETURN cuantosHay = capacidad
      END;
   PROCEDURE vacio (): BOOLEAN;
      BEGIN
         RETURN cuantosHay = 0
      END;      (pg103)
   PROCEDURE entry manda (mensaje: CHAR);
      BEGIN
         IF lleno THEN delay (pro);
         contenido [in]:= mensaje;
         in:= (in + 1) mod capacidad;
         cuantosHay:= cuantosHay + l;
         IF cuantosHay = 1 THEN continue (con)
      END;
   PROCEDURE entry recibe (VAR mensaje: CHAR);
      BEGIN
         IF vacio THEN delay (con);
         mensaje:= contenido [out];
         out:= (out + 1) mod capacidad;
         cuantosHay:= cuantosHay - 1;
         IF cuantosHay = (capacidad - 1) THEN continue (pro)
      END;
   BEGIN
      in:= 0; out:= 0; cuantosHay:= 0
   END; (* del monitor *)

   VAR nuestroBuzon: buffer;
   PROCEDURE productor; VAR m:CHAR;
      BEGIN
         REPEAT
            elabora mensaje (m);
            nuestroBuzon.manda (m)
         UNTIL te canses
      END;
   PROCEDURE consumidor;
      VAR m: CHAR;
```

```
    BEGIN
      REPEAT
        nuestroBuzon.recibe (m);
        trabajaElMensaje (m)
      UNTIL te canses
    END;      (pg104)
  BEGIN (* productorConsumidor *)
    init nuestroBuzon (20);
    COBEGIN
      productor;
      consumidor;
    COEND
  END.
```

Observe en el ejemplo que los monitores, al igual que las clases, también se inicializan con la sentencia init.

Otro ejemplo típico del uso de monitores es la gestión de recursos de un sistema. Considere que quiere gestionar el acceso de procesos concurrentes a un disco. Supondremos que no se puede denegar ninguna petición y que por lo tanto en un tiempo finito se habrá atendido cada una de ellas. Como cualquier proceso en el sistema puede hacer una petición de disco, el monitor de disco debe estar preparado para retrasar a cualquier proceso si su petición llega cuando se está atendiendo a otra. Esto significa que el monitor debe declarar tantas variables queue como procesos existan en el sistema. Cuando el disco es liberado por un proceso se usa el algoritmo del ascensor (con el fin de minimizar el tiempo de acceso disminuyendo los movimientos del brazo) para seleccionar la siguiente petición a atender. Lo siguiente es una implementación de este monitor (ver [Mat84]).

```
TYPE monitorDisco = monitor
  TYPE procesos = 1..maxProcesos;
  VAR colaProcesos = ARRAY procesos OF RECORD
                                        dirDis: unaDireccionDeDisco;
                                        Q: queue
                                            END;
    up, (* dirección del movimiento del brazo *)
    ocupado (*disco trabajando *): BOOLEAN;
    posicionCabeza: unaDireccionDeDisco;
  PROCEDURE entry petición (dir: unaDirecciónDeDisco);
    BEGIN
      (* la función IdPro devuelve la identificación del proceso que realiza la llamada *)
      IF ocupado
      THEN WITH colaProcesos [idPro] DO BEGIN
             dirDis:= dir;
             delay (Q)
           END;      (pg105)
      posicionCabeza:= dir; ocupado:= TRUE;
      realizaTransferencia
    END;
  PROCEDURE entry libera;
    VAR p: procesos;
        encontrado: BOOLEAN;
    PROCEDURE siguientePeticion (VAR p: procesos);
      BEGIN
        (* busca la petición pendiente más próxima a la posición actual de la cabeza y en el sentido
        del movimiento del brazo. encontrado = TRUE si se encontró dicha petición. encontrado
        = FALSE si no se encontró *)
      END;
    BEGIN
      ocupado:= FALSE;
      siguientePeticion (p);
```

```
        IF NOT encontrado
        THEN BEGIN
                up:= NOT up;
                siguientePeticion (p)
             END;
        IF encontrado
        THEN continue (colaProcesos[p].Q)
    END;
BEGIN (* del monitor *)
    up:= TRUE; ocupado:= FALSE; posiciónCabeza:= 0
END; (* del monitor *)
```

## 2.7. Sincronización por rendez-vous

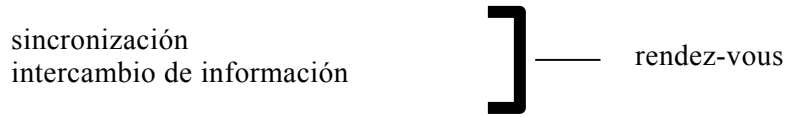
Como ya hemos podido observar a lo largo de este capítulo, existen diversas formas para comunicar procesos entre sí. Ahora vamos a estudiar el mecanismo mediante el cual interactúan los procesos en Ada.

El lenguaje de Programación Ada utiliza el mecanismo de rendez-vous para la comunicación entre procesos, el cual se basa en el «Communicating Sequential Process» (C.S.P.) de Hoare que fue influenciado por «Distributed Processes» de Brinch Hansen. Que la concurrencia sea real o simulada no afectará al resultado de la ejecución, ya que el programa es independiente de la configuración hardware. (pg106)

Los procesos en Ada se conocen como tareas. Dichas tareas pueden contener unas entradas, que podrán ser llamadas desde otras tareas para establecer el rendez-vous. Las tareas se estarán ejecutando paralelamente con otras tareas. Cada tarea puede ser ejecutada por un procesador lógico, con lo cual diferentes tareas se ejecutarán independientemente, excepto en los puntos donde se sincronizan e intercambian información.

**Rendez-vous**

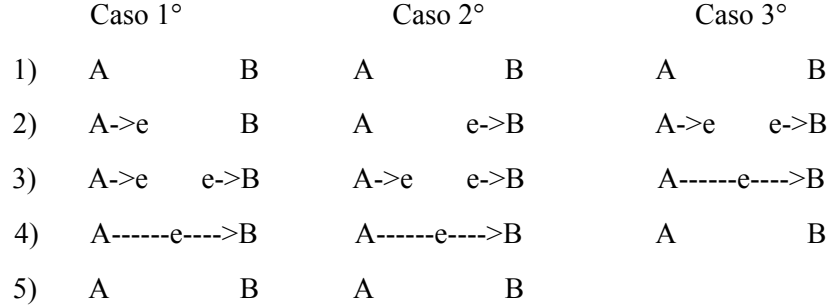
Rendez-vous significa sincronización e intercambio de información entre dos tareas dadas. Estos dos hechos siempre serán dos actividades inseparables. El hecho de establecerse rendez-vous se conoce como la cita de las tareas.




Dos tareas interactuarán, primero mediante la sincronización, luego mediante el intercambio de información y, por último, continuarán sus actividades por separado. El intercambio de la información entre las tareas no es unidireccional como en el C.S.P., sino que puede ser bidireccional.

Una tarea contiene puntos de sincronización llamados puntos de entrada. Una llamada a un punto de entrada es sintácticamente muy similar a una llamada a un procedimiento, aunque semánticamente son muy diferen tes. La sincronización entre dos tareas ocurre cuando una tarea llama a un punto de entrada y la otra tarea acepta esa llamada estableciendo rendezvous.

Vamos a representar gráficamente (Fig. 29) tres posibles casos que se pueden dar cuando los procesos A y B se quieren comunicar mediante rendez-vous:




(pg107)

FIGURA 29

Donde,

| | |
|---|---|
| A | el proceso A se está ejecutando |
| B | el proceso B se está ejecutando |
| A- > e | el proceso A llama al punto de entrada «e» y espera a que sea aceptada su petición. |
| e-> B | el proceso B puede aceptar inmediatamente una llamada al punto de entrada «e» y espera a que algún proceso llame a ese punto de entrada. |
| A----e--- > B | los procesos A y B se sincronizan y comunican. |

A continuación vamos a pasar a la explicación de los tres casos, viendo qué es lo que ocurre en cada uno de los momentos antes de establecerse rendez-vous entre ambas tareas.

*Caso 1.°:*

1) Las tareas A y B se están ejecutando independientemente.

2) La tarea A desea establecer rendez-vous con la tarea B, llamando al punto de entrada «e» de la tarea B. Por tanto, la tarea A se queda esperando a que la tarea B acepte la llamada a su punto de entrada. Cabe señalar que la espera realizada por la tarea A es pasiva.

3) La tarea B puede aceptar, inmediatamente, una llamada del punto de entrada «e».

4) Debido a que la tarea A ha llamado al punto de entrada «e» y la tarea B puede aceptar la llamada, entonces se da la cita de las tareas porque están sincronizadas. Puede haber un paso de información de la tarea A a la tarea B. La tarea A se queda esperando a que la tarea B ejecute las sentencias de la cita y en el momento en el que la tarea B termine puede haber un nuevo paso de información de B hacia A.

5) Ahora ya se ha terminado la cita y, por tanto, cada una de las tareas sigue ejecutándose independientemente.

*Caso 2.°:*

1) Las tareas A y B se están ejecutando independientemente.

2) La tarea B desea establecer rendez-vous con otra tarea, ya que puede aceptar la llamada al punto de entrada «e». Por tanto, la tarea B se queda esperando a que alguna tarea realice una llamada a su punto de entrada «e». Cabe señalar que la espera que realiza la tarea B es también pasiva.

3) La tarea A llama al punto de entrada «e» de la tarea B.

4) En este momento se da la cita de las tareas y es idéntico al primer caso. (pg108)

5) Ahora ya se ha terminado la cita y, por tanto, cada una de las tareas sigue ejecutándose independientemente.

*Caso 3.°:*

1) Las tareas A y B se están ejecutando independientemente.

2) La tarea A desea establecer rendez-vous con la tarea B llamando al punto de entrada «e» de dicha tarea B y, en ese mismo instante, la tarea B puede aceptar la llamada a su punto de entrada «e».

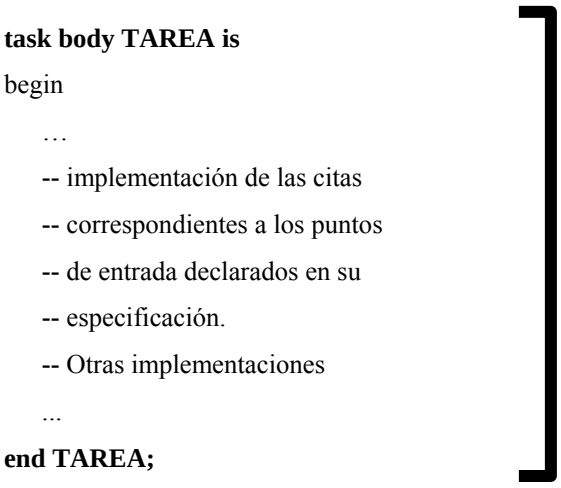
3) A partir de este momento se da la cita de las tareas y es idéntico al primer caso.

4) Ahora ya se ha terminado la cita y, por tanto, cada una de las tareas sigue ejecutándose independientemente.

A continuación veremos cómo se puede implementar en Ada este mecanismo de rendez-vous que ha sido explicado teóricamente. Se ha elegido Ada porque es el único lenguaje de programación de alto nivel que utiliza el mecanismo de rendez-vous. Por esta razón está asociado directamente rendez-vous con Ada. Debemos comenzar mostrando cuál es la estructura de una tarea Ada.

```
Task TAREA is
-- declaración de puntos de entrada
-- otras declaraciones
end TAREA;
```

**especificación**
(interfaz)




Por tanto, una tarea estará formada por:

* una *parte de especificación* de la tarea: contendrá las entradas de la tarea, es decir, las declaraciones de los puntos de entrada mediante la palabra reservada **entry**. Será la parte visible a otras tareas que deseen llamar a estos puntos de entrada. (pg109)

* una *parte de implementación* de la tarea: contendrá lo que se debe ejecutar cuando se da la cita de las tareas, es decir, contendrá entre otras, sentencias compuestas **accept**.

Cuando una tarea no tiene puntos de entrada, su parte de especificación se reduce a:

task SIN PUNTOS__ DE ENTRADA;

Una tarea sin puntos de entrada significa que es una tarea padre, ya que utilizará los servicios de otras tareas y ella misma no será utilizada por ninguna otra.

Debido a que hemos hecho mención de dos palabras reservadas: accept y entry, es necesario describir su sintaxis y semántica.

### Declaración de puntos de entrada

El formato para la declaración de puntos de entrada será de la siguiente forma:

**entry** NOMBRE PUNTO DE ENTRADA(parámetros formales);

Los parámetros formales son el medio para el intercambio de información entre las tareas que se den en la cita. Si en la declaración de un punto de entrada no hay parámetros formales significa que estamos utilizando el punto de entrada únicamente para sincronización entre tareas.

Cada uno de los puntos de entrada lleva asociada una cola de procesos, cuya gestión es mediante el criterio FIFO. Todas las tareas que llamen a un mismo punto de entrada se irán disponiendo en la cola de ese punto de entrada y la tarea propietaria de tal punto irá aceptando las llamadas una a una.

A continuación damos la parte de especificación de una tarea que contiene dos puntos de entrada: uno de ellos sin parámetros formales y el otro con ellos.

```
task EJEMPLO is
   entry SIN PARAMETROS;
   entry CON PARAMETROS(A:in INTEGER; Bout FLOAT; C:in out BOOLEAN);
end EJEMPLO;
```

### Sentencia accept

Las sentencias que se ejecutarán durante la cita se describen mediante (pg110) sentencias compuestas accept, las cuales se encuentran en el cuerpo de la tarea que contiene la declaración del punto de entrada.

Utilizando la parte de especificación de la tarea EJEMPLO podríamos tener el siguiente cuerpo:

```
task body EJEMPLO is
begin
...
accept SIN PARÁMETROS do
- - conjunto de sentencias que se ejecutarán en la cita
- - correspondiente al punto de entrada SIN PARÁMETROS
end SIN__ PARÁMETROS;
...
accept CON PARAMETROS(A:in INTEGER;B:out FLOAT; C:in out BOOLEAN) do
-- conjunto de sentencias que se ejecutarán en la cita
-- correspondiente al punto de entrada CON PARAMETROS
end CON PARÁMETROS;
...
end EJEMPLO;
```

Puede haber varias sentencias accept correspondientes a un mismo punto de entrada dentro del cuerpo de una tarea. Cuanto menor sea el número de sentencias a ejecutar dentro de una sentencia accept mayor será el nivel de concurrencia.

### Llamada a un punto de entrada

Para realizar una llamada a un punto de entrada utilizaremos la notación con punto para especificar explícitamente el nombre de la tarea que contiene ese punto de entrada. Siguiendo con la tarea EJEMPLO tendremos que la forma de llamar al punto de entrada SIN PARAMETROS es:

```
EJEMPLO.SIN PARÁMETROS;
```

Siempre tenemos que utilizar la notación con punto, esto da lugar a la existencia del **nombramiento asimétrico,** ya que la tarea que llama al punto de entrada tiene que dar el nombre de la tarea donde está declarado el entry. Por el contrario, la tarea que contiene el accept no puede saber quién ha llamado a su punto de entrada. Por esta razón no podemos restringir a ciertas tareas la llamada a ciertos puntos de entrada.

Cuando una tarea llama a un punto de entrada, esta tarea se pone en la (pg111) cola asociada a ese entry. Cuando la tarea que contiene ese punto de entrada llega al correspondiente accept, la primera tarea que se encuentre en la cola se saca de ella para poder establecer rendez-vous. De igual forma, si varias tareas llaman a un mismo punto de entrada, éstas se encolan y la tarea receptora las irá aceptando según el criterio FIFO.

### La cita entre tareas

Una vez vista la sintaxis que utiliza Ada, vamos a describir nuevamente rendez-vous, pero esta vez haciendo uso de la sintaxis Ada.

La cita se produce como consecuencia de la llamada de una tarea a un punto de entrada declarado en otra tarea.

Una tarea se comunica con otra invocando un punto de entrada de esta última. Entonces la tarea que contenga «entry» aceptará una llamada de una de sus entradas, ejecutando la correspondiente sentencia **accept**.

La sincronización se realiza mediante rendez-vous entre una tarea que llama a un «entry» y la tarea que acepta la llamada mediante «accept». Estas entradas pueden tener parámetros formales para el intercambio de informa ción, donde los modos de estos parámetros pueden ser de entrada (in), de salida (out) y de entrada/salida (in out).

Las sentencias entry y accept serán las que utilicemos para sincronizar y comunicar tareas entre sí.

La sentencia accept la ejecuta la tarea propietaria del punto de entrada. No se puede ejecutar el conjunto de sentencias incluidas en la sentencia compuesta accept hasta que una tarea llame al punto de entrada y la tarea propietaria del punto de entrada llegue a ese accept. Uno de estos dos sucesos ocurrirá uno antes del otro y, por tanto, se suspenderá una de las dos tareas hasta que llegue la otra a la cita. En el momento que comienza el rendez-vous, la tarea propietaria del punto de entrada ejecuta las sentencias que hay dentro de la sentencia compuesta accept y ta tarea que llamó al punto de entrada

quedará suspendida. Cuando llegue al end del accept ambas tareas continuarán ejecutándose independientemente.

*Caso particular:* rendez-vous anidado
En algunos casos podemos llegar a realizar un rendez-vous anidado, el cual se puede plantear de la siguiente forma:

A=> B = > C siendo A, B y C tareas

A llama al punto de entrada de B y dentro del accept de B se llama al punto de entrada de C. (pg112)

El rendez-vous de A y B no terminará hasta que termine el rendez-vous de ByC.

Para acabar de entender el concepto de rendez-vous nos vamos a plantear el siguiente ejercicio:

*Ejercicio 1*

*«Un cliente que ha hecho una consumición en una cafetería desea pagarla, para lo cual llama a un camarero, le da el dinero y éste le devuelve el cambio.»*

Tanto el cliente como el camarero son procesos independientes, ya que en un principio el cliente está tomándose su consumición, mientras que el camarero está poniendo comidas a otros clientes, por ejemplo. Cuando el cliente llama al camarero significa que el cliente está realizando una llamada a un punto de entrada del proceso camarero. Tal petición no podrá ser atendida hasta que el camarero acabe con lo que está haciendo. En el momento que el camarero acaba con lo que está haciendo va a la mesa del cliente, es decir, los procesos camarero y cliente están sincronizados. Entonces, a partir de este momento puede comenzar el intercambio de información, de hecho el cliente le da al camarero dinero para pagar su consumición y el camarero le da las vueltas al cliente. Una vez hecho esto se va del restaurante y el camarero sigue atendiendo a otros clientes. Por tanto, volvemos a obtener procesos independientes.

Esto que se ha explicado informalmente quedaría implementado de la siguiente forma:

```
- - Declaración de dependencias con otros módulos
with TEXT_IO,IIO;
use TEXT_IO,IIO;
- - Importamos de TEXT_IO:     PUT,PUT LINE
- - Importamos de IIO:         PUT;

- - programa principal
   procedure CLIENTES is
   - - declaración de las tareas CLIENTE y CAMARERO
      task CLIENTE;
      task CAMARERO is
         entry DEVOLVER CAMBIO(DINERO CLIENTE:in out INTEGER);
      end CAMARERO;
   - - implementación de las tareas CLIENTE y CAMARERO
      task body CLIENTE is     (pg113)
      MONEDERO:INTEGER:=100; -- pesetas
   begin
      PUT("CLIENTE: Tengo ");
      PUT(MONEDERO);
      PUT LINE(" pesetas y estoy tomando mi consumición");
         CAMARERO.DEVOLVER_CAMBIO(MONEDERO);
               - - llamada a un punto de entrada
      PUT("CLIENTE: Ahora me quedan ");
      PUT(MONEDERO);
      PUT LINE(" pesetas ");
      PUT LINE("CLIENTE: Me voy del restaurante, habiendo pagado");
   end CLIENTE;
   task body CAMARERO is
      TODAS LAS CONSUMICIONES:INTEGER:=50; --pesetas
```

```
    begin
        PUT LINE(" CAMARERO: Estoy atendiendo a clientes");
        accept DEVOLVER_CAMBIO(DINERO_CLIENTE: in out INTEGER) do
            DINERO_CLIENTE:=DINERO_CLIENTE_TODAS_LAS_CONSUMICIONES;
        end DEVOLVER. CAMBIO;
        PUT LINE(" CAMARERO: Un cliente me acaba de pagar y " & («me voy a hacer otras cosas");
    end CAMARERO;
begin
    null;
end CLIENTES;
```

Hay que decir que en este programa hay declaradas tres tareas, CLIENTE, CAMARERO y el programa principal. Todo el contenido del programa principal se tomará como una tarea hipotética. En el ejemplo anterior no hemos querido hacer uso de esto, por lo que le hemos puesto la sentencia null, indicando que está vacío. Es más, la tarea del programa principal será la progenitora del resto de las tareas que se declaren.

La ejecución de este pequeño, aunque representativo, programa nos da el siguiente listado:

```
    CAMARERO: Estoy atendiendo a clientes
CLIENTE: Tengo      100 pesetas y estoy tomando mi consumición    (pg114)
    CAMARERO: Un cliente me acaba de pagar y me voy a hacer otras cosas
CLIENTE: Ahora me quedan        50 pesetas
CLIENTE: Me voy del restaurante, habiendo pagado.
```

## 2.8. Equivalencia de herramientas

Todas las herramientas descritas para controlar la ejecución concurrente de procesos son lógicamente equivalentes en el sentido de que se pueden implementar unas con otras. Ya hemos visto cómo implementar buzones con RCC y con RC y semáforos. También hemos implementado RCC con sucesos. A continuación detallamos unas posibles implementaciones de algunas herramientas utilizando otras.

### *2.8.1. Implementar un semáforo con RC*

```
VAR sem: SHARED RECORD
                contador: 0..maxInt;
                esperando: QUEUE OF P
            END;
WAIT: REGION sem DO
            IF contador > 0
            THEN contador:= contador - 1
            ELSE BEGIN
                    mete (proceso, esperando);
                    suspende proceso;
                    selecciona otro proceso para ejecutar
                    END;
SIGNAL: REGION sem DO
            IF NOT empty (esperando)
            THEN BEGIN
                    saca (proceso, esperando);
                    pon a ese proceso como ejecutable
                    END
            ELSE contador:= contador + 1;          (pg115)
```

### *2.8.2. Implementar un semáforo con RCC*

```
    VAR sem: SHARED 0..maxInt;
```

```
WAIT: REGION sem DO BEGIN          SIGNAL: REGION sem DO
        AWAIT sem > 0;                        sem:= sem + 1;
        sem:= sem – 1
        END
```

Moraleja: Las RCC permiten escribir programas más cortos.

### *2.8.3. Implementar un semáforo con buzones*

```
WAIT: RECEIVE (m, b);        SIGNAL: SEND (m, b);
```

(los mensajes son vacíos).

A pesar de esta equivalencia debe quedar claro que cada una de ellas tiene una semántica específica, por lo cual es más apropiada que las otras herramientas para resolver ciertos tipos de problemas. En la siguiente relación se indica la labor específica para la que fueron diseñadas:

| | |
|---|---|
| RC ..... | exclusión mutua |
| Semáforo..... | intercambio de señales de sincronización |
| RCC ..... | espera condicional |
| Buzón ..... | comunicación |
| Suceso ..... | gestión explícita de colas |
| Monitor ..... | abstracción de datos y exclusión mutua y sincronización |
| Rendez-vous ..... | sincronización de alto nivel |

La equivalencia de los monitores con algunas de las otras herramientas definidas aquí pueden estudiarse en [Tane87].

# *Capítulo 3*

# *Interbloqueos*

En un entorno de multiprogramación varios procesos pueden competir por un número finito de recursos (procesadores, memoria, cintas magnéticas, impresoras, etc.). Si un proceso pide un recurso que en ese momento no está disponible entra en un estado de espera (se bloquea). Suponga que un ordenador tiene una cinta magnética y una impresora y que existen dos procesos A y B que han solicitado y obtenido la cinta y la impresora respectivamente. Si ahora el proceso A pide la impresora y el proceso B pide la cinta, ninguna de las peticiones puede ser satisfecha y ambos procesos se bloquean. Esta situación se denomina **interbloqueo**.

Un término muy relacionado con los interbloqueos es **«starvation»** (inanición) que denota un estado en el que uno o varios procesos son retrasados indefinidamente en el acceso a un recurso en favor de otros procesos.

Se ha argumentado que los interbloqueos son una cuestión económica. La experiencia de sala indica que se producen pocas veces al año y que se solucionan con técnicas ad-hoc. Como ya veremos, la prevención, detección y recuperación de los interbloqueos supone un tiempo de ejecución significativo. En cualquier caso, y como no se puede predecir la frecuencia con que se producirán los interbloqueos, lo más honesto que un sistema operativo puede hacer es utilizar los mecanismos adecuados para evitarlos o reducir sus consecuencias al mínimo. (pg118)

## 3.1. Ejemplos de interbloqueos con las herramientas definidas

En adelante supondremos:

```
COBEGIN
    P1; P2
COEND;
```

### *3.1.1. Con regiones críticas*

```
    VAR r, s: SHARED ...
«P1»                                    «P2»
    …..                                     …..
    …..                                     …..
    REGION r DO BEGIN                       REGION s DO BEGIN
        …..                                     …..
        REGION s DO BEGIN                       REGION r DO BEGIN
        …..                                     …..
        END; (* s *)                            END; (* r *)
        …..                                     …..
    END; (* r *)                            END; (* s *)
```

### *3.1.2. Con un semáforo dentro de una región crítica*

```
    VAR  r: SHARED ....;
         s: SEMAPHORE;  (* INIT (s, 0) *)
«Pl»                                    «P2»
```

```
.....                                .....
.....                                .....
REGION r DO BEGIN                    REGION r DO BEGIN
    .....                                .....
    WAIT (s);                            SIGNAL (s); (* único *)
    .....                                .....
END;                                 END;
```

(pg119)

*3.1.3. Con regiones críticas condicionales*

```
VAR  r: SHARED RECORD
            cl, c2: BOOLEAN (* init a FALSE *)
        END;
```

«P1»

```
.....
.....
REGION r DO BEGIN
    .....
    c1:= TRUE;
    AWAIT c2;
END;
```

«P2»

```
.....
.....
REGION r DO BEGIN
    .....
    AWAIT c1;
    c2:= TRUE;
END;
```

*3.1.4. Con semáforos*

```
VAR  s: SEMAPHORE; (* INIT (s, 0) *)
```

«Pl»

```
.....
WAIT (s);
.....
SIGNAL (s);
.....
```

«P2»

```
.....
WAIT (s);
.....
SIGNAL (s);
.....
```

*3.1.5. Con buzones*

```
VAR bl, b2: buffer 3 OF INTEGER;
```

«Pl»

```
.....
FOR i:= 1 TO 4 DO
    SEND (m, bl);
.....
FOR i:= 1 TO 4 DO
    SEND (m, b2);
.....
```

«P2»

```
.....
FOR j:= 1 TO 4 DO
    RECEIVE (m, b2);
.....
FOR j:= 1 TO 4 DO
    RECEIVE (m, bl);
.....
```

(pg120)

*3.1.6. Con sucesos*

```
VAR r: SHARED RECORD
            .....
            sc: EVENT r
        END;
```

«P1»

```
.....
.....
REGION r DO BEGIN
    .....
    CAUSE (sc);
    .....
```

«P2»

```
.....
.....
REGION r DO BEGIN
    .....
    AWAIT (sc);
    .....
```

```
        AWAIT (sc);                          CAUSE (sc);
        …..                                  …..
    END;                                 END;
```

## 3.2. Recursos

Distinguiremos recursos permanentes y temporales. Un recurso permanente puede ser utilizado repetidamente por muchos procesos. Un recurso temporal es producido por un proceso y consumido por otro. Como ejemplos de recursos permanentes tenemos los dispositivos físicos y como ejemplos de recursos temporales los mensajes.

Estudiaremos en primer lugar un sistema con un número fijo de recursos permanentes de varios tipos y dejaremos para más adelante el estudio de los interbloqueos cuando se trabaja con mensajes.

Un sistema consiste en un número de recursos que pueden ser distribuidos entre procesos que compiten por ellos. Los recursos pueden ser de varios tipos y dentro de cada tipo existir varias instancias. Por ejemplo, puede haber 2 discos, 3 impresoras y 8 pantallas. Para simplificar consideraremos que desde el punto de vista de los procesos, los recursos que sean instancias de un mismo tipo son equivalentes en el sentido de que su petición puede ser servida con cualquiera de ellos indistintamente.

La secuencia de operaciones necesarias para usar un recurso son:

1. Pide el recurso. Si el recurso no está disponible el proceso se bloquea.
2. Usa el recurso. (pg121)
3. Libera el recurso.

Tanto la petición como la liberación del recurso son llamadas al sistema operativo, que es el que mantiene el estado de cada uno de ellos. El uso del recurso puede también implicar llamadas al S.O. como read, write, etc., para acceder a ficheros o a dispositivos de e/s.

## 3.3. Definición y caracterización de los interbloqueos

Un conjunto de procesos está interbloqueado si cada proceso en el conjunto está esperando por un suceso que solamente puede producir otro proceso del conjunto. Puesto que todos los procesos están esperando, ninguno de ellos podrá producir el suceso que despierte a cualquiera de los otros miembros del conjunto, y todos los procesos continuarán esperando siempre.

Antes de discutir los diferentes métodos para tratar con los interbloqueos describiremos algunas características de los mismos.

### *3.3.1. Condiciones necesarias*

Coffman (1971) ha señalado que las siguientes condiciones son necesarias para que se produzca un interbloqueo, respecto a los recursos permanentes:

1) Exclusión mutua. Cada recurso o bien está disponible o bien asignado a un único proceso.
2) Asignación parcial. Un proceso puede adquirir sus recursos por partes.
3) Programación no expulsora. Un recurso sólo puede ser liberado por el proceso que lo ha adquirido.
4) Espera circular. Debe existir un conjunto de procesos esperando (p0, pl, ..., pn) tal que p0 está esperando por un recurso que tiene p1; pl está esperando por un recurso que tiene p2; ... p(n-1) está esperando por un recurso que tiene pn y pn está esperando por un recurso que tiene p0.

Se previenen interbloqueos asegurando que no se cumplen nunca una o más de las condiciones anteriores.

### *3.3.2. Grafos de asignación de recursos*

Los interbloqueos se pueden describir de una manera más precisa utilizando un grafo dirigido conocido como grafo de asignación de recursos del (pg122) sistema. Este grafo consiste, por una parte, de un conjunto de nodos compuestos por los procesos y recursos que hay en el sistema y, por otra, por un conjunto de arcos dirigidos que unen procesos y recursos. A fines de presentación representaremos los procesos con círculos y los recursos con áreas rectangulares. Un arco dirigido de un proceso Pi a un recurso Ri indica

que Pi está bloqueado esperando que se le conceda el recurso Ri (Fig. 30a). Si el arco está dirigido de Ri a Pi significa que el proceso Pi tiene asignado el recurso Ri (Fig. 30b).

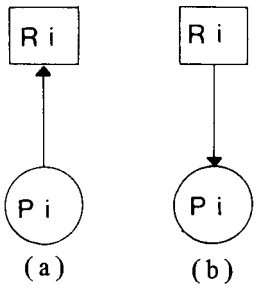



FIGURA 30

Dado el grafo de asignación de recursos de un sistema es fácil comprobar que, si el grafo no contiene ciclos, entonces no existen interbloqueos. Si, de otra manera, el grafo contiene un ciclo es posible que exista un interbloqueo.

Si cada tipo de recurso tiene una única instancia, entonces la existencia de un ciclo en el grafo es condición necesaria y suficiente para la existencia de un interbloqueo. Si por el contrario cada tipo de recurso tiene varias instancias, entonces un ciclo no implica necesariamente la existencia de un interbloqueo. En este caso, un ciclo es condición necesaria pero no suficiente para la existencia de un interbloqueo. Para ilustrar mejor estos conceptos veamos los siguientes ejemplos:

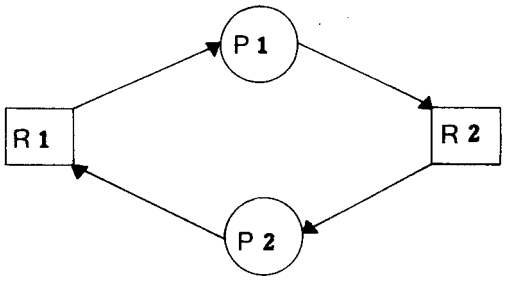



FIGURA 31 (pg123)

En el grafo de la Fig. 31 el proceso Pl tiene asignado el recurso R1 y pide R2. El proceso P2 tiene asignado R2 y pide R1. Puesto que existe una sola instancia de cada recurso y un ciclo en el grafo, se ve que existe un interbloqueo.

(En los ejemplos siguientes representaremos cada instancia de un recurso como un punto dentro de la caja que representa su tipo.) Considere ahora el grafo de la Fig. 32:

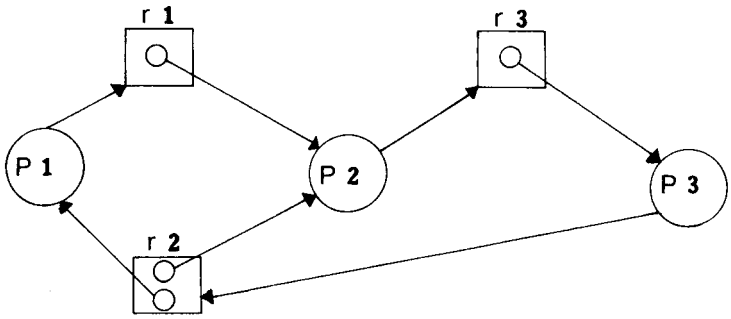



FIGURA 32

En este grafo existen 2 ciclos:

p1 => r1 => p2 => r3 => p3 => r2 => p1

p2 = > r3 = > p3 = > r2 = > p2

Los procesos p1, p2 y p3 están interbloqueados. El proceso p2 está esperando por r3 que posee p3. El proceso p3 está esperando a que r2 lo liberen o bien p1 o bien p2. Además p1 está esperando a p2.

Ahora considere el grafo de la Fig. 33:

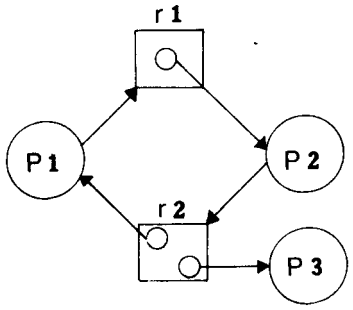



FIGURA 33 (pg124)

Aquí existe un ciclo:

P1 => r1 => p2 => r2 => p1.

Sin embargo, no existe interbloqueo. El proceso p3 con el tiempo liberará una instancia de r2 que puede asignarse a p2 rompiéndose con ello el ciclo.

En resumen, si en un grato de asignación de recursos no existe un ciclo, entonces el sistema no está en un estado de interbloqueo. Por el contrario, si existe un ciclo, el sistema puede estar o no en un estado de interbloqueo.

## 3.4. Estrategias para tratar los interbloqueos

Hablando en términos generales podemos distinguir tres estrategias para tratar con los interbloqueos:

a) Ignorar el problema.
b) Asegurar que el sistema nunca entra en un estado de interbloqueo.
c) Permitir que el sistema entre en un interbloqueo y entonces intentar romperlo de alguna manera.

La última estrategia es compleja y costosa en tiempo de ejecución como ya veremos.

### *3.4.1. Ignorar el problema*

La aproximación más simple para tratar los interbloqueos es el algoritmo del avestruz: esconder la cabeza en la arena y pretender que el problema no existe. Se puede defender esta estrategia diciendo que si la probabilidad de producirse un interbloqueo es menor que la que existe de producirse un fallo hardware, es más económico enfrentarse con el problema únicamente cuando se produzca.

Por ejemplo, en Unix existen interbloqueos potenciales debido a causas como que se agoten las entradas en la tabla de procesos (creados por fork) o en la tabla de i-nodes que mantiene los ficheros abiertos actualmente (y lo mismo se puede aplicar a otros recursos del sistema de los cuales existe un número finito). Unix supone que los usuarios preferirán enfrentarse a un interbloqueo ocasional a que se les restrinja el número de procesos que pueden crear o el número de ficheros que pueden tener simultáneamente abiertos (a pesar de que a nivel práctico existen estas restricciones). Por ello Unix ignora el problema. (pg125)

Todos los S.O. deben comprometerse o bien a tratar los interbloqueos, con el gasto que ello supone (sobrecarga del procesador y la memoria para ejecutar los algoritmo adecuados), o a ignorarlos y utilizar técnicas ad-hoc cuando se produzcan.

### *3.4.2. Asegurar que el sistema nunca entra en un estado de interbloqueo*

Esta estrategia consiste en imponer ciertas restricciones a los procesos de manera tal que los interbloqueos no puedan producirse. Existen dos métodos:

- Prevenir los interbloqueos.
- Evitar los interbloqueos.

Puesto que la diferencia semántica de estos dos métodos puede ser confusa, observe la siguiente analogía: «Asociamos la existencia de un interbloqueo con «mojarse por la lluvia». Entonces:

- prevenir los interbloqueos es equivalente a asegurarse antes de salir de casa que no va a llover.

- evitar los interbloqueos es equivalente a llevarse un paraguas.

*3.4.2.1. Técnicas para prevenir los interbloqueos*

Estas técnicas están basadas en lograr que una o más de las condiciones enunciadas por Coffman como necesarias para que se produzca un interbloqueo nunca lleguen a producirse. Examinaremos una por una las cuatro condiciones.

- *Exclusión mutua*

La exclusión mutua debe mantenerse para tipos de recursos no compartibles. Por ejemplo, no se debe permitir que dos procesos escriban simultáneamente en la misma impresora. Por otro lado, los recursos compartibles no necesitan exclusión mutua y por ello no pueden verse envueltos en un interbloqueo. Por ejemplo, si varios procesos intentan acceder simultáneamente a un fichero de sólo lectura, el acceso puede concederse a todos ellos. Un proceso nunca esperará por recursos compartibles.

En general, no es posible eliminar la exclusión mutua. Incluso en el caso de la impresora, en el que puede recurrirse al spooling (con lo cual varios procesos escriben su salida simultáneamente en disco y la impresora siempre está asignada al «demonio» de impresión), existe competencia por espacio en disco y ello puede conducir a un interbloqueo. Puede ocurrir que dos (pg126) procesos A y B llenasen cada uno la mitad del espacio de disco destinado al spooling y ninguno de ellos hubiese acabado de generar su salida. Si el «demonio» estuviese programado para comenzar a imprimir antes de que toda la salida generada por un proceso estuviese en el disco, podría ocurrir que un proceso decidiera esperar durante horas antes de continuar con lo cual la impresora quedaría ociosa. Por ello los «spooler» generalmente esperan a que los ficheros de spooling estén completos antes de comenzar a imprimirlos. El interbloqueo se produce.

- *Asignación parcial*

Para eliminar esta condición debemos asegurarnos que siempre que un proceso solicite un recurso no tenga asignado ninguno en ese momento.

Para conseguirlo podemos seguir cualquiera de los dos siguientes protocolos:

a) Asignación total: antes de que comience a ejecutarse un proceso, éste debe solicitar todos los recursos que necesite. Si todos ellos están disponibles se le asignan a este proceso y se ejecuta hasta terminarse (momento en el cual libera todos los recursos). Si alguno de los recursos pedidos no está disponible, el proceso debe esperar hasta que se le concedan todos. El primer problema con esta aproximación es que muchos procesos no conocen «a priori» cuantos recursos necesitan. Otro problema es que los recursos no se utilizarán de forma óptima. Considere un proceso que lee datos de una cinta, calcula durante una hora y escribe el resultado en una cinta y en un plotter. Si todos los recursos se deben asignar al principio, el proceso retendrá el plotter durante más de una hora.

b) Una aproximación diferente es que un proceso puede solicitar recursos solamente cuando no tenga otros asignados. Antes de que pueda solicitar cualquier recurso debe liberar todos los que tenía asignados.

Ambas soluciones padecen de un posible «starvation»: un proceso que necesite recursos muy utilizados (populares) por otros procesos puede que tenga que esperar indefinidamente por algún recurso que esté siempre asignado a otro proceso.

- *Programación no expulsora*

Eliminar esta condición significa obligar a los procesos a que liberen sus recursos temporalmente a favor de otros procesos.

Para lograr este objetivo se puede seguir el siguiente protocolo: si un proceso que tiene asignado un número determinado de recursos pide otro(s) recurso(s) que no se le puede(n) asignar inmediatamente (es decir, que el pro ceso debe esperar) entonces libera todos los recursos que actualmente tiene (pg127) asignados. El proceso sólo será arrancado cuando pueda acceder a sus viejos recursos y a los nuevos que solicitaba.

Otro método consiste en que si un proceso pide algún recurso, primero se comprueba si dicho recurso está disponible. Si es así se le asigna y el proceso continúa. Si tal recurso no está disponible se comprueba si está asig nado a algún otro proceso que está esperando por otro(s) recurso(s). Si es así expulsamos el recurso deseado del proceso que espera y se lo asignamos al proceso que lo pedía. Si el recurso solicitado no está disponible ni lo tiene asignado ningún proceso bloqueado, el proceso que lo solicitaba se suspende. Mientras está en este estado otro proceso le puede quitar algún recurso. Un proceso puede reanudarse

solamente cuando se le asignan los nuevos recursos que pedía y cualquier otro que le hubiese sido quitado mientras estaba bloqueado.

Este protocolo frecuentemente se aplica sobre aquellos recursos cuyo estado pueda ser fácilmente salvado y restaurado tales como los registros de la CPU y la memoria.

La expulsión no es práctica en los dispositivos que necesitan intervención del operador como cintas y discos magnéticos.

- *Espera circular*

Una cuarta posibilidad es prevenir los interbloqueos evitando la espera circular. Esta espera se puede eliminar imponiendo una ordenación lineal de todos los tipos de recursos, como por ejemplo:

1. Lectora de tarjetas. 2. Impresora.
3. Plotter.
4. Cinta magnética.
5. Perforadora de tarjetas.

Ahora la regla es: un proceso puede pedir recursos siempre que lo desee, pero todas las peticiones deben realizarse en orden numérico creciente. Un proceso puede solicitar primero una impresora y luego una cinta magnética, pero no puede pedir primero un plotter y luego una impresora. Esto es lo que se denomina asignación jerárquica de recursos. En términos generales, los recursos se distribuyen en una jerarquía formada por N niveles. Cada nivel a su vez consiste en un número finito de tipos de recursos. Los recursos que un proceso necesita de un nivel dado deben ser adquiridos en una única petición.

Cuando un proceso ha adquirido recursos a un nivel Lj, sólo puede pedir recursos a un nivel más alto Lk, donde k > j. (pg128)

Si un proceso tiene asignados recursos del nivel Lk y quiere recursos del nivel Lj (siendo k > j) debe primero liberar los que tiene del nivel Lk.

Suponemos que un proceso liberará los recursos adquiridos a un cierto nivel Lj en un tiempo finito a menos que sea retrasado indefinidamente por peticiones de recursos de un nivel más alto Lk, donde k > j.

Con estas suposiciones no puede ocurrir un interbloqueo en un sistema con asignación jerárquica de recursos. Una petición de recursos al nivel más alto Lmax no puede ser retrasada indefinidamente por peticiones a otro ni vel superior (puesto que no existe). Solamente se puede retrasar a un proceso porque algún otro proceso tenga esos recursos. Como este último proceso los liberará en un tiempo finito, una petición de recursos del nivel más alto no puede producir un interbloqueo. Similarmente la liberación de recursos del nivel L(max-1) sólo puede ser retrasada indefinidamente por peticiones al nivel Lmax. Hemos visto que estas peticiones se resuelven en un tiempo finito y, por tanto, la liberación de recursos al nivel L(max-1) también se hará en un tiempo finito.

Aunque la asignación jerárquica de recursos elimina el problema del interbloqueo, puede ser imposible encontrar una ordenación que satisfaga a todos los usuarios.

Las diferentes aproximaciones para prevenir los interbloqueos se resumen en la siguiente tabla:

| *condición* | *aproximación* |
|---|---|
| exclusión mutua | spooling |
| asignación parcial | asignación total |
| no expulsión | tomar recursos de donde los haya |
| espera circular | asignación jerárquica |

### *3.4.2.2. Técnicas para evitar los interbloqueos*

En sistema en los que no se puede eliminar ninguna de las condiciones enunciadas por Coffman, para evitar los interbloqueos debemos poseer alguna información adicional sobre los recursos que utilizará un proceso. El modelo más sencillo y utilizado es exigir que cada proceso diga, antes de ser arrancado, cuantos recursos necesitará a lo largo de su ejecución.

A continuación describiremos un algoritmo, frecuentemente utilizado por los sistemas operativos, que evita los interbloqueos: «el algoritmo del banquero». En la terminología de Dijkstra (que fue quien primero lo propuso y resolvió) el problema lo podemos enunciar como:

«Un banquero quiere administrar un capital fijo en florines (recursos de un solo tipo) entre un número fijo de clientes (procesos). (La solución con (pg129) varios tipos de monedas la veremos más tarde.) Cada cliente especifica por adelantado su necesidad máxima de florines (el total de recursos que necesita). El banquero aceptará a su cliente si su necesidad máxima no excede del capital que posee (número de recursos que tiene la instalación).

Es posible que un cliente tenga que esperar antes de que se le conceda prestado el siguiente florín. Pero el banquero asegura que este tiempo de espera es finito.

Si el banquero es capaz de satisfacer la necesidad máxima de un cliente, el cliente garantiza que devolverá el préstamo dentro de un tiempo finito».

La situación actual es «segura» (no puede producirse un interbloqueo) si es posible para el banquero satisfacer a todos sus clientes en un tiempo finito; en caso contrario es «insegura». El banquero adopta un punto de vista pesimista y siempre evita las situaciones inseguras.

La situación de un cliente está caracterizada por su préstamo actual (recursos asignados) y su petición futura (recursos que va a pedir), siendo:

peticiónActual = necesidadTotal - préstamoActual

La situación del banquero se caracteriza por su capital inicial y la cantidad actual en caja (recursos disponibles), siendo:

caja = capitalInicial - préstamosActuales

El banquero sólo concederá un préstamo si esto conduce a una situación segura. Para verificar esto, simula lo que ocurriría si realizase dicho préstamo: se va hacia una situación segura o insegura.

No vamos a programar el algoritmo, limitándonos a poner un ejemplo de la mecánica de dicho algoritmo. El lector interesado puede recurrir a la bibliografía ([BriH73] o [Tane87]).

La Fig. 34 muestra una situación en la que tres clientes P, Q y R comparten un capital de 10 florines. Su necesidad total es de 20 florines: P=8, Q=3, R=9. En la situación actual (Fig. 34a) el cliente Q tiene un préstamo de 2 florines y una petición de 1 florín [2(1)]. Para P y R la situación es 4(4) y 2(7) respectivamente. Lo que en este momento hay en caja es:

10 – 4 – 2 – 2 = 2.

Si ahora Q pide un florín, el banquero antes de prestárselo simula lo que ocurriría de hacerlo. En la Fig. 34b se muestra la situación después de prestarle un florín a Q, el cual acabará y devolverá su préstamo. Ahora se puede satisfacer a P, cuya terminación nos llevará a la Fig. 34c. En esta situación el cliente R puede terminar, recuperando el banquero su capital inicial. (pg130)

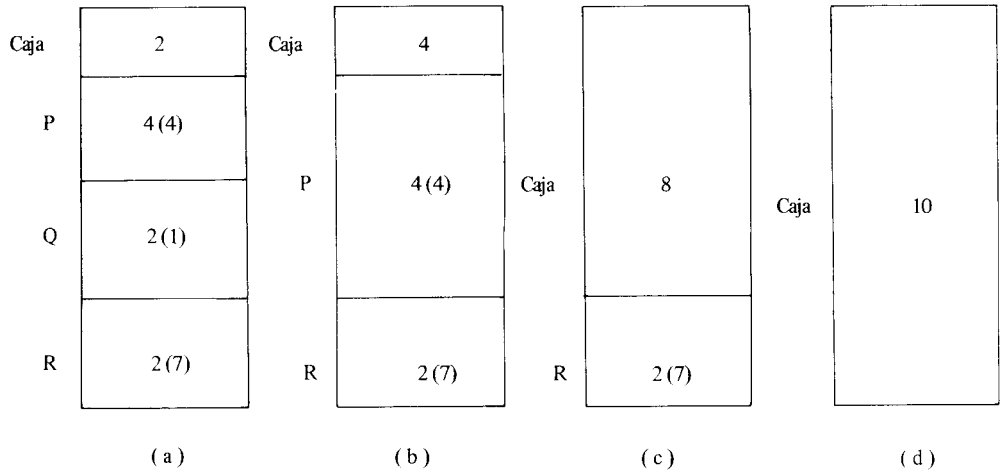



FIGURA 34

Consecuentemente, la petición de Q puede ser atendida ya que nos conduce a un estado seguro. Ahora bien, de la situación original se puede pasar a un estado inseguro de la siguiente manera: el banquero presta un florín al cliente R y la situación actual es la mostrada en la Fig. 35a.

En esta situación Q puede ser satisfecho llegándose a la situación mostrada en la Fig. 35b. Ahora ni P ni R pueden progresar ya que el banquero no puede satisfacer ninguna de sus peticiones: situación insegura. Por tanto, la petición inicial de R será retrasada.

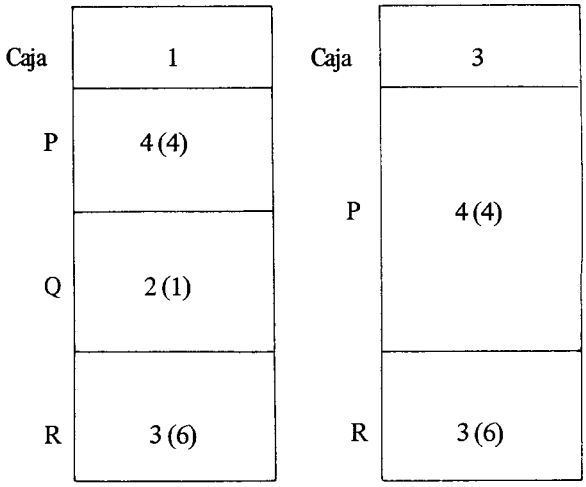



FIGURA 35

(pg131)

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| A | 3 | 0 | 1 | 1 |
| B | 0 | 1 | 0 | 0 |
| C | 1 | 1 | 1 | 0 |
| D | 1 | 1 | 0 | 1 |
| E | 0 | 0 | 0 | 0 |

recursos asignados

| | R1 | R2 | R3 | R4 |
|---|---|---|---|---|
| A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 1 | 2 |
| C | 3 | 1 | 0 | 0 |
| D | 0 | 0 | 1 | 0 |
| E | 2 | 1 | 1 | 0 |

necesidades de recursos

recursos existentes E = (6 3 4 2)
recursos asignados A = (5 3 2 2)
recursos disponibles D = (1 0 2 0)

FIGURA 36

En el algoritmo anterior hemos tratado con un solo tipo de recurso. El algoritmo del banquero se puede generalizar a un número arbitrario de tipos de recursos, cada uno con varias instancias. Para ello, y al igual que en el caso de un solo tipo de recurso, es necesario que los procesos indiquen sus necesidades máximas por adelantado. Con esta información y sabiendo el sistema operativo cuantos recursos de cada tipo tiene asignado cada proceso, se construyen dos tablas como las mostradas en la Fig. 36.

Dada una petición, el algoritmo simula concederla y a continuación comprueba si esto conduce a una situación segura o no, de la siguiente manera:

1. Busca en la tabla de la derecha, una fila R, cuyas necesidades de recursos son todas más pequeñas que el vector D. Si no se encuentra tal fila, el sistema está interbloqueado.

2. Encontrada la fila R, se supone que el proceso de esa fila pide todos los recursos, termina y los libera sumándose dichos recursos al vector D.

3. Se repiten los pasos 1 y 2 hasta que todos los procesos terminan en cuyo caso la situación es segura, o hasta que ocurre un interbloqueo en cuyo caso la situación es insegura.

Volviendo a la Fig. 36, si B pide un recurso R3, se le puede conceder, ya que esto nos lleva a una situación segura (puede terminar D, luego A o E y (pg132) luego el resto). Suponga que a continuación de conceder un recurso R3 a B, E pide otro recurso R3. Si se le concediese, llevaría al vector D a (1 0 0 0 ) que implica un interbloqueo. Por tanto, la petición de E debe retrasarse.

Al algoritmo del banquero se le ha achacado que teóricamente es correcto pero que en la práctica no es factible debido a:

1. Es muy costoso en tiempo de ejecución. Cada petición o liberación de recursos implica ejecutar el algoritmo.

2. En la práctica pocos procesos conocen sus necesidades máximas a priori.

### *3.4.3. Permitir que el sistema entre en un estado de interbloqueo y entonces intentar romperlo de alguna manera (detección y recuperación)*

Un sistema que no emplee ningún protocolo para evitar que se produzcan los interbloqueos, debe implementar un esquema de detección y recuperación de los mismos. Cuando se utiliza esta técnica, el sistema lo único que hace es monitorizar las peticiones y liberaciones de recursos. Cada vez que se concede o libera un recurso, se actualiza el grafo de recursos y se comprueba si existe interbloqueo o no. Si existe uno, se elimina uno de los procesos que intervenía en el interbloqueo. Si con esto no se ha solucionado el problema se elimina un proceso más. Esto continúa hasta que se logre eliminar el interbloqueo.

La estrategia de detección y recuperación se usa frecuentemente en grandes ordenadores, especialmente en sistemas batch en los que matar a un proceso y luego arrancarlo desde el principio es generalmente aceptable. Se de ben tomar precauciones para restaurar a su estado original cualquier fichero (o cualquier otro recurso) que haya sido modificado por el proceso recién muerto.

Los algoritmos de selección del proceso a eliminar (víctima) y la forma de volver a su estado original los recursos utilizados por la víctima caen fuera del alcance de este libro. Documentación sobre este tema se puede encontrar en la bibliografía [Pete85].

## 3.5. Interbloqueos con recursos temporales

Restringiremos la discusión con comunicación de procesos jerarquizados.

Consideraremos un sistema de procesos conectados únicamente (pg133) mediante buzones de mensajes. Ya

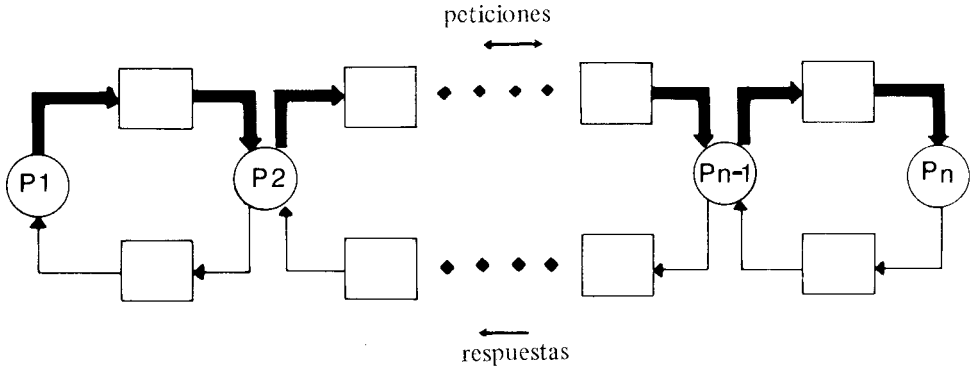



FIGURA 37

hemos visto que varios procesos productores y consumidores conectados a un único buzón, no pueden interbloquearse. Pero si conectamos circularmente varios procesos mediante buzones puede existir interbloqueo.

Para evitar esta circularidad recurriremos a jerarquizar los procesos en niveles. Cada proceso pertenece a un nivel Li. Los procesos de niveles inferiores se denominan «maestros» y pueden proporcionar mensajes («peticiones») a los procesos de niveles superiores, que llamaremos «seguidores». Estos últimos pueden proporcionar respuestas («réplicas») a sus maestros en contestación a sus peticiones.

Por tanto, las peticiones se envían en un solo sentido y las respuestas en sentido contrario (ver Fig. 37).

Aunque el sistema es jerárquico, todavía hay peligro de interbloqueos. Considere la situación mostrada en la Fig. 38.

P y Q pueden ser ambos incapaces de mandar mensajes y respuestas porque ambos buzones estén llenos o ser incapaces de recibir mensajes o respuestas porque los dos buzones estén vacíos: P y Q estarían interbloqueados.

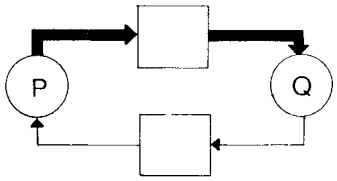



FIGURA 38

(pg134)

La regla de oro para prevenir interbloqueos es: «No intentar mandar un mensaje o una respuesta a menos que alguien lo vaya a recibir con el tiempo; y no intentar recibir un mensaje o una respuesta a menos que alguien vaya a mandarlo con el tiempo.»

Con esta regla podemos hacer la siguiente hipótesis: «cuando un maestro envía un mensaje a un seguidor, el último finalmente lo recibirá, y (si se lo pide) enviará a una respuesta a menos que sea retrasado indefinidamente por uno de sus propios seguidores».

Con esta suposición no pueden ocurrir interbloqueos con comunicación de procesos jerarquizados. La demostración se detalla a continuación:

1. La hipótesis anterior es cierta para cualquier proceso Pj del nivel más alto (j=max) porque no hay seguidores que puedan retrasarlo.

2. Consideramos ahora un proceso Pi-1 del nivel Li-1. Según la hipótesis, Pi-1 satisfará la hipótesis si no es retrasado indefinidamente por un seguidor Pj de un nivel mayor Lj donde $j \geq i$. Hay dos posibilidades para tales retrasos:

a) Pi-1 puede ser incapaz de enviar un mensaje a un seguidor Pj porque un buzón está lleno. Pero de acuerdo con la hipótesis, el seguidor Pj, de nivel Lj (donde $j \geq i$) finalmente recibirá un mensaje y permitirá que Pi-1 prosiga.

b) Pi-1 puede ser incapaz de recibir una respuesta de un seguidor Pj porque un buzón está vacío. De nuevo, de acuerdo con la hipótesis, el seguidor del nivel Lj (donde $j \geq i$) finalmente recibirá el correspondiente mensaje y producirá la respuesta que permita continuar a Pi-1.
Por tanto, los mensajes enviados a procesos de nivel Li-1 siempre serán recibidos, consumidos, y (si se pide) respondidos dentro de un intervalo finito. Por inducción se demuestra que esto se cumple a todos los niveles, desde L1 a Lmáx.

# *Capítulo 4*

# *Lenguajes de programación concurrente*

Como ya se dijo en ocasiones anteriores, las herramientas descritas en el capítulo 2 para manejar la concurrencia no están disponibles comercialmente en un único lenguaje de programación. Algunas como los buzones o los semáforos sí tienen cierto acomodo en algún lenguaje, pero con una sintaxis y una semántica diferentes a las aquí explicadas. Es necesario darse cuenta que las herramientas descritas por Hansen son de demasiado bajo nivel como para que su uso resulte atractivo en la programación concurrente. Constituyen una base conceptual para abordar los problemas inherentes a la concurrencia y también los elementos primarios para la construcción de herramientas más potentes.

En este capítulo se da una visión panorámica de los lenguajes de programación actuales que soportan la concurrencia de procesos. No se pretende dar una descripción del lenguaje, sino que nos limitaremos a una mera presentación informal de sólo aquellos aspectos relacionados con la programación concurrente.

Creemos que el conjunto elegido de lenguajes da una buena perspectiva de las herramientas actualmente disponibles en programación concurrente. Estos lenguajes son: Concurrent Pascal, Concurrent Euclid, CSP, Ada y Modula-2. (pg136)

## CONCURRENT PASCAL

El lenguaje Concurrent Pascal fue desarrollado por Per Brinch Hansen en el Institute of Thechnology de California durante el período 1972-1975. El primer compilador para este lenguaje fue escrito en Pascal secuencial por Al Hartmann y el primer intérprete (para el PDP-11/45) lo escribió Tom Zepko.

El Concurrent Pascal fue ideado para servir como herramienta de diseño e implementación de sistemas operativos. Está basado en el lenguaje Pascal al que añade estructuras para la programación concurrente (procesos y monitores), así como mecanismos de abstracción de datos (clases).

En Concurrent Pascal un programa consiste en un número fijo de procesos, monitores y clases. Estos componentes existen para siempre una vez inicializado el sistema. No existe creación/destrucción dinámica de procesos y monitores.

A continuación veremos con algo más de detalle cada uno de estos componentes.

### Procesos

Un tipo **process** define un programa secuencial. En su declaración pueden existir constantes, tipos, variables y procedimientos, bien entendido que estos objetos son privados al proceso que los declara. Por ejemplo:

type cardProcess = **process** (parámetros);
    < declaración de constantes >
    < declaración de tipos >
    < declaración de variables >
    < declaración de procedimientos >
    begin

```
    < sentencias >
end;
```

Inicialmente un programa en Concurrent Pascal se ejecuta como un único programa secuencial llamado el proceso inicial. Contiene la declaración de otros procesos y monitores, así como las sentencias para inicializarlos.

Un proceso se inicializa con la sentencia **init**, por ejemplo:

```
init reader (inBuffer);
```

(suponemos que, por ejemplo, existe en el proceso inicial la declaración:

```
var reader: cardProcess;
```

(pg137)

La sentencia init crea espacio para las variables privadas del proceso y arranca su ejecución. Los parámetros que lleva el proceso (inBuffer, en el ejemplo anterior) son los derechos de acceso que tiene ese proceso a otros componentes del sistema.

Un proceso se puede inicializar una sola vez y sus parámetros y variables privadas existen para siempre. De la misma manera, un proceso sólo puede acceder a sus variables locales y a sus parámetros. El compilador detecta errores como que un proceso intente acceder a variables de otros procesos (esto podría crear condiciones de carrera). Un proceso sólo puede comunicarse con otros procesos a través de monitores.

### Monitores

Un monitor permite comunicar datos entre procesos y también sincronizarlos.

Un tipo **monitor** define una estructura de datos compartida entre procesos y todas las operaciones que los procesos pueden ejecutar sobre dicha estructura. Estas operaciones se denominan **procedimientos monitor** o puntos de entrada al monitor. Estos procedimientos van acompañados de la palabra **entry** para diferenciarlos de aquellos otros que son locales al propio monitor. La estructura de un monitor es:

```
type nombreDeMonitor = monitor (parámetros);
    < declaración de constantes >
    < declaración de tipos >
    < declaración de variables >
    < declaración de procedimientos locales >
    < declaración de procedimientos entry >
    begin
        < sentencias de inicialización del monitor >
    end;
```

Un monitor se inicializa con la sentencia **init**. Esta sentencia asigna memoria a las variables compartidas y ejecuta las sentencias entre el begin y el end del monitor. Después de la inicialización, las variables compartidas del monitor existen para siempre (se denominan variables permanentes). Sin embargo, los parámetros y variables locales de los procedimientos monitor existen solamente mientras se ejecuta dicho procedimiento (se denominan variables temporales). (pg138)

Un procedimiento monitor sólo puede acceder a sus propias variables temporales y permanentes. Estas variables, por otro lado, sólo pueden ser accedidas por otros componentes del sistema invocando los procedimientos monitor, que se ejecutan como parte del proceso que los llamó.

Los monitores aseguran la exclusión mutua sobre las variables compartidas que protege. Si un proceso está ejecutando un procedimiento de un monitor M y otro proceso quiere ejecutar un procedimiento del mismo monitor, este último proceso será suspendido hasta que el primero abandone el monitor.

Además, los monitores proporcionan un mecanismo de sincronización mediante variables de tipo **queue**. Un procedimiento monitor puede retrasar a un proceso una cantidad arbitraria de tiempo ejecutando una operación **delay** sobre una variable de tipo queue. Solamente un proceso a la vez puede estar suspendido en una variable dada de tipo queue. Cuando un proceso se suspende por una operación delay abandona el monitor quedando, por tanto, éste libre. El proceso suspendido será reanudado cuando otro proceso ejecute una operación **continue** sobre la queue en la que estaba suspendido el primer proceso. En

Concurrent Pascal se obliga a que una operación continue sea la última sentencia de un procedimiento monitor. Su efecto es que el proceso llamante retorna del procedimiento monitor llamado, libera dicho monitor y reanuda (si existe) el proceso suspendido en la queue, proceso que entonces gana acceso al monitor.

Las variables queue deben ser declaradas como variables permanentes dentro del monitor.

### Clases

Un tipo **class** define una estructura de datos y un conjunto de operaciones asociadas a tal estructura. Es un tipo de componente de sistema que no puede ser llamado simultáneamente por varios componentes del sistema (a diferencia de los monitores). Al igual que los monitores, define un conjunto de puntos de entrada, tiene sentencias de inicialización (begin, end) y es necesario inicializarlas con la sentencia init.

## CONCURRENT EUCLID

Euclid fue diseñado en 1976 como lenguaje para el desarrollo de sistemas software verificables. La investigación y el uso de lenguaje llevó al diseño e implementación en 1981 de Concurrent Euclid. (pg139)

Concurrent Euclid (CE) fue diseñado para soportar la implementación de software altamente fiable y eficiente tal como sistemas operativos, compiladores y software empotrado para microprocesadores.

CE está basado en Pascal y añade (en lo referente a programación concurrente) monitores y procesos. También incorpora las sentencias signal y wait y proporciona mecanismos de bajo nivel para el acceso a todos los recursos hardware.

### Procesos

En CE un proceso se escribe con la palabra reservada **process** seguida del nombre del proceso. El resto es como un procedimiento. Por ejemplo:

```
process hola
    < declaraciones y sentencias >
end hola
```

En CE cualquier módulo puede contener procesos. Deben aparecer al final del módulo, siguiendo a la inicialización del mismo (si existe). Después de que el módulo se ha inicializado, todos los procesos comienzan a ejecutarse. Observe la similitud con el constructor COBEGIN-COEND.

### Monitores

En CE una variable **monitor** es declarada indicando que dicha variable es de tipo monitor. Por ejemplo:

```
VAR uno: monitor
            < importaciones >
            < exportaciones >
            < declaración de variables >
            < procedimientos monitor >
            < inicialización >
         end monitor
```

Es problema del compilador garantizar la exclusión mutua dentro del monitor. Un monitor en CE es similar a un módulo y puede tener una parte (pg140) de inicialización. Esta inicialización se ejecuta antes de que cualquier proceso entre al monitor. Un monitor no puede contener módulos, monitores o procesos. El acceso a las variables compartidas se realiza a través de los procedimientos que exporta el monitor. No se permite que un monitor exporte variables. De la misma forma, un punto de entrada al monitor no puede ser invocado desde dentro de éste. Por último, indicar que los procedimientos monitor son reentrantes.

**Sincronización**

Es frecuente que los procesos tengan que sincronizar sus actividades. Un ejemplo típico es el uso de un mismo recurso por parte de dos procesos. Si Pl quiere utilizar el recurso y éste lo tiene asignado P2, Pl debe bloquearse a la espera de que P2 libere el recurso. Cuando esto suceda, P2 debe indicar que el recurso está disponible y despertar a Pl que esperaba por él. Veamos el siguiente monitor escrito en CE (Holt, 1983):

```
var Resource:
    monitor
        exports (Acquire, Release)
        var inUse: Boolean = false
        var available: condition (signaled when not inUse)
        procedure Acquire =
            imports (var inUse, var available)
            begin
                if inUse then
                    wait (available)
                end if
                inUse:= true
            end Acquire
        procedure Release =
            imports (var inUse, var available)
            begin
                inUse:= false
                signal (available)
            end Release
    end monitor
```

(pg141)

Las sentencias signal y wait se aplican a conditions, que son colas de procesos. Un proceso que ejecuta wait se bloquea y abandona el monitor a la espera de que le despierten. Cuando un proceso ejecuta signal se comprueba la correspondiente cola. Si existen procesos en la cola se toma uno, que es reanudado inmediatamente. El proceso que ejecutó signal abandona el monitor y no se le permite continuar hasta que éste no quede libre. Si la condición señalada no contiene procesos, el efecto del signal es nulo. Sin embargo, antes de que el proceso señalante continúe, otros procesos pueden entrar y salir del monitor.

La razón por la que el proceso señalante cede control al proceso señalado es que de esta forma este último proceso está seguro de que la condición por la que fue despertado se sigue cumpliendo.

En CE existe una extensión a las conditions que se denomina **priority condition**. Se usan igual que las conditions ordinarias excepto que la sentencia wait debe especificar una prioridad. Por ejemplo:

```
var c: priority condition
.....
wait (c, 10)
…..
signal (c)
```

La sentencia signal despertará al proceso de la cola que se suspendió con el menor número de prioridad.

**CSP**

Communication Sequential Processes (CSP) es un lenguaje diseñado para la programación concurrente y que es idóneo para un entorno en red de microordenadores con memoria distribuida. Fue desarrollado por Hoare en 1978.

Los conceptos básicos del lenguaje son:

1) Un programa en CSP consiste de un número fijo de procesos secuenciales cuyos espacios de direccionamiento son mutuamente disjuntos.

2) La comunicación y sincronización se realiza a través de los constructores de entrada y salida.

3) Las estructuras secuenciales de control están basadas en los comandos **guardas** de Dijkstra. (pg142)

### Comunicación en CSP

La comunicación en CSP ocurre cuando un proceso nombra a otro proceso como destinatario de su salida y este último proceso nombre al primero como la fuente de su entrada (la operación de salida sería el equivalente a un send y la operación de entrada a un receive). El mensaje se copia del primer proceso al segundo proceso. Todos los mensajes tienen tipo y para que se establezca la comunicación, el tipo de mensaje de la salida debe ser igual al tipo de mensaje de la entrada. La transferencia de información ocurre solamente cuando los procesos fuente y destino han invocado sus operaciones de salida y entrada respectivamente. Por ello, o bien el proceso fuente o el proceso destino pueden quedar suspendidos a la espera de que el otro proceso invoque la operación correspondiente.

Este esquema se parece al rendez-vous de Ada (buzones de capacidad 0), pero aquí el mensaje se manda a un proceso en lugar de invocar un punto de entrada de ese proceso.

La forma en CSP de mandar un mensaje es:

nombreDelProcesoReceptor ! mensaje

De la misma forma, cuando un proceso quiere recibir un mensaje de otro proceso, ejecuta una sentencia de la forma:

nombreDelProcesoEmisor ? mensaje

### Comandos Guarda en CSP

En CSP el control secuencial se realiza a través de los comandos guardas de Dijkstra. Un comando guarda tiene la forma:

< guarda > = > < lista-comandos >

Un guarda consiste en una lista de declaraciones, expresiones booleanas y un comando de entrada (cada uno de estos componentes es opcional). Un guarda falla si todas sus expresiones tienen valor false, o si el proceso nombrado en su comando de entrada ha terminado. Si falla un guarda, el proceso en el que estaba definida esta sentencia se aborta. Si no falla, se ejecuta la lista de comandos.

Los comandos guarda pueden ser combinados en un comando de alternativas:

[G1 => C1 ▮ G2 => C2 ▮ … ▮ Gn => Cn] (pg143)

Un comando de alternativas especifica la ejecución de uno de sus comandos guarda constituyentes. Si fallan todos los guardas, el comando de alternativas falla y el proceso es abortado. Si se puede ejecutar más de un comando guarda en el mismo instante se selecciona aleatoriamente uno de ellos para ejecutarlo.

El comando de alternativas se puede ejecutar para siempre en un comando repetitivo que tiene la forma:

*[G1 => C1 ▮ G2 => C2 ▮ … ▮ Gn => Cn]

Este comando de alternativas se ejecuta para siempre mientras no falle. Cuando fallan todos sus guardas, el comando repetitivo termina y se cede control a la siguiente sentencia.

Veamos a continuación un ejemplo (Peterson): el problema del productor/consumidor con un buffer de 10 elementos. El buffer de 10 elementos se encapsula en un proceso CSP que llamaremos bounded-buffer y cuya definición es:

```
buffer: (0..9) item;
in, out: integer;
in:= 0;
out:= 0;
*[in < out+10; producer ? buffer (in mod 10)
    => in:= in + 1;
▮ out = in; consumer ? more()
    => consumer ! buffer (out mod 10);
        out:= out + 1
]
```

El proceso productor produce un elemento p y lo manda al proceso bounded-buffer ejecutando:

```
bounded-buffer ! p
```

El proceso consumidor recibe un elemento del proceso bounded-buffer ejecutando:

```
bounded-buffer ! more();
bounded-buffer ? q;
```

**ADA**

El Ministerio de Defensa de Estados Unidos consideró en 1974 desproporcionado (pg144) el gasto en software que estaba efectuando. Este era elevado debido a que las inversiones efectuadas en el desarrollo de sistemas empotrados se encarecían enormemente, por no contar con un lenguaje de desarrollo estándar que repercutiera positivamente en cada una de las etapas del ciclo de vida software de dichas aplicaciones.

A partir de sucesivos documentos (Strawman, Woodenman, Tinman, Steelman) y mediante un sistema de concurso con competencia entre los candidatos, así como contactos que generaban informes técnicos desde todo el mundo, se alcanzó, en 1980, la primera versión definitiva del lenguaje deseado: ADA.

Después de un proceso de estandarización se consiguió en 1983 el definitivo Manual de Referencia del Lenguaje (M.R.L.), el cual marca la guía a respetar por todos los fabricantes de compiladores del mencionado lenguaje.

Al igual que el resto de los lenguajes de programación que se están exponiendo en este capítulo, el lenguaje Ada también ofrece el procesamiento paralelo y de esta forma se puede decir que es un lenguaje completo, ya que, además de esto, tiene buenas características para la implementación de programas secuenciales. Existe una famosa frase de Ludwin Wittgenstein que dice: «Las limitaciones de mi lenguaje dan las limitaciones de mi mundo.»

El paralelismo en Ada se ofrece mediante sentencias que van implícitas en el lenguaje y, por tanto, se integran bastante bien con el resto de la sintaxis del lenguaje. De esta forma se consigue que la expresión formal de algo ritmos concurrentes sea fácil y su legibilidad sea alta. La razón de que no se haya elegido para Ada un conjunto de bibliotecas que contengan procedimientos para expresar el paralelismo son:

* el compilador no puede optimizar fácilmente el código.
* es necesario crear estas bibliotecas en ensamblador para obtener rapidez, por tanto se rompe el concepto de biblioteca.
* no es fácil expresar conceptos de concurrencia mediante conceptos que son para programación secuencial.

Antes de concluir esta breve introducción enumeramos algunas de las ventajas a destacar que ofrece este lenguaje:

* lleva consigo instrucciones de programación concurrente.
* permite la construcción de bibliotecas de tareas.
* contiene time-outs utilizados en el momento de interactuar dos tareas.
* interfaz con el hardware.
* excepciones.
* capacidad para abortar tareas y, de esta forma, poder tomar medidas de precaución, alarma, etc.

Estas razones, conjuntamente con las grandes posibilidades que ofrece a (pg145) nivel de programación secuencial, permite decir que el lenguaje de programación Ada es aceptable para la construcción de grandes Sistemas en Tiempo Real.

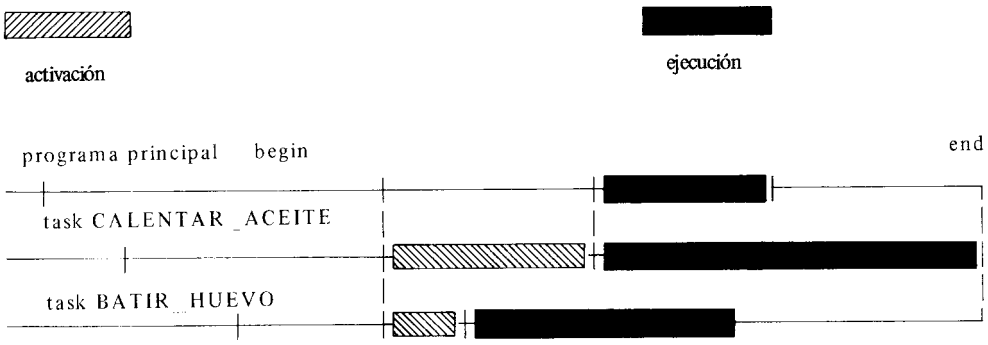
Los procesos se conocen como tareas, las cuales tendrán puntos de entrada mediante los cuales podremos establecer rendez-vous entre tareas. Rendez-vous es el mecanismo que ofrece Ada para la sincronización y comunicación entre tareas, como ya se vió en la sección 2.7. Una tarea está formada por una parte de especificación y un cuerpo, como ya se sabe, pero hay un dato que nos falta: ¿cuándo se activan las tareas?

### Activación de una tarea

La activación de las tareas es automática. A continuación se presenta un problema trivial, donde queda puesto de manifiesto la activación de las tareas.

```
procedure PREPARARNOS_PARA_FREIR_UN_HUEVO is
   task CALENTAR_ACEITE;
   task body CALENTAR_ACEITE is
   begin
      -- echar el aceite en la sarten
      -- encender el fuego
      -- poner la sartén sobre el fuego
   end CALENTAR_ACEITE;
   task BATIR_HUEVO;
   task body BATIR_HUEVO is
   begin
      -- coger un huevo y un plato
      -- echar el huevo en el plato
      -- batir el huevo que hay en el plato
   end BATIR_HUEVO;
begin -- activación de las tareas CALENTAR_ACEITE y BATIR_HUEVO
-- comienza la ejecución de las tareas declaradas
-- el padre comenzará a ejecutarse cuando haya finalizado la
-- activación de las tareas
   null;
end PREPARAR_PARA_FREIR_UN_HUEVO;
```

**Activar una tarea** significa elaborar las declaraciones que están incluidas en el cuerpo de dicha tarea. Por ello, el tiempo necesario para activar una tarea (pg146) no es el mismo para todas. La razón de que se activen cuando se llega al begin tiene por objeto poder «cazar» alguna posible excepción, que se pueda elevar en las partes declarativas de las tareas CALENTAR_ACEITE y BATIR_HUEVO. Gráficamente podemos verlo como se muestra a continuación.




El progenitor comenzará a ejecutarse cuando se hayan activado todas las tareas que dependan de él, en este caso CALENTAR_ACEITE y BATIR_HUEVO.

Antes de comenzar con el ejercicio siguiente vamos a introducir una nueva sentencia que tiene Ada: select. Esta sentencia la utilizaremos conjuntamente con la sentencia compuesta accept, de forma que podemos expresar muy claramente conceptos de concurrencia.

### Sentencia de espera selectiva

Permite seleccionar una tarea de entre varias citas posibles. La sentencia select termina de ejecutarse cuando se acabe de ejecutar una de sus ramas.

El formato de la sentencia select es: (pg147)

```
select
   accept A …;
   …                          alternativa 1
   end A;
or
   accept B …;
   …                          alternativa 2
   end B;
or
   accept C …;
   …                          alternativa 3
   end C;
end select;
```

Dentro de una sentencia select puede haber dos o más alternativas separadas por «or». Las reglas que se siguen para comprender el funcionamiento de esta sentencia son:

* si no hay llamadas a ninguna alternativa, la tarea que contiene la sentencia SELECT se suspende.

* si hay llamadas al punto de entrada A y no hay llamadas a ninguna otra alternativa se ejecuta la sentencia «accept A».

* si hay llamadas a varias alternativas, se selecciona la ejecución de una alternativa «arbitrariamente», aunque se tendrá en cuenta las prioridades de las citas. Un programa no puede depender del algoritmo de selección usado por el compilador, ya que sería erróneo.

Utilizando el ejercicio número 1 de la sección 2.7 vamos a ver cómo podemos insertar esta nueva sentencia en tal implementación del cliente y camarero.

Tal y como está en el ejercicio referenciado, el camarero obligatoriamente siempre tenía que devolver cambio a un cliente. También es verdad que las funciones de un camarero no sólo es la de devolver cambio, sino ade más poner comidas, por ejemplo. Por tanto, vamos a poner una nueva entrada al proceso camarero y utilizando la sentencia select veremos cómo podemos compaginar ambos puntos de entrada.

```
task CAMARERO is
entry DEVOLVER_CAMBIO(DINERO_CLIENTE: in out INTEGER);
entry ATENDER(MENSAJE: in COMIDAS;
                    HAY_EXISTENCIAS: out BOOLEAN):
end CAMARERO;                                                    (pg148)
task body CAMARERO is
   TODAS_LAS_CONSUMICIONES:INTEGER:=50; -- pesetas
begin
   PUT_LINE(" CAMARERO: Estoy atendiendo a clientes");
   select
      accept DEVOLVER_CAMBIO(DINERO_CLIENTE: in out INTEGER) do
            DINERO_CLIENTE:= DINERO_CLIENTE-TODAS_LAS_CONSUMICIONES;
      end DEVOLVER_CAMBIO;
      PUT_LINE(" CAMARERO: Un cliente me acaba de pagar y " & "me voy a hacer otras cosas");
   or
   accept ATENDER(MENSAJE: in COMIDAS;
                       HAY_EXISTENCIAS: out BOOLEAN) do
      if QUEDAN_EXISTENCIAS_DE_LA_COMIDA(MENSAJE) then
         HAY_EXISTENCIAS:=TRUE;
      else HAY_EXISTENCIAS:=FALSE;
      end ATENDER;
      -- ir a la cocina y preparar la comida MENSAJE
   end select;
end CAMARERO;
```

**Llamada a un punto de entrada temporarizada y condicional**

Se utiliza una sentencia select con dos únicas ramas, donde una llama a un punto de entrada y la otra contiene un conjunto de sentencias alternativas.

Son muy útiles cuando una tarea no quiere verse demorada porque una tarea servidora esté ocupada.

*Llamada temporarizada*

```
select
    llamada_a_un_punto_de_entrada;
or
    delay TIEMPO;
    otras cosas;
end select;
```
(pg149)

Si en un período de tiempo TIEMPO no se acepta la llamada a un punto de entrada, entonces se realizarán otras cosas.

*Llamada condicional*

```
select
    llamada_a_un_punto_de_entrada;
else
    delay TIEMPO;
    otras cosas;
end select;
```

Si no acepta la llamada al punto de entrada, entonces se espera el período de tiempo TIEMPO y realiza otras cosas.

Mediante estos nuevos conceptos que hemos introducido podemos hacer variantes al ejercicio anterior.

Podemos plantearnos que el cliente realiza una llamada temporarizada al camarero, es decir, si en el período de 2 minutos no le atiende, el cliente se irá sin pagar. Esto quedaría sencillamente implementado como:

```
task body CLIENTE is
    MONEDERO:INTEGER:=100; -- pesetas
begin
    ...
    select
        CAMARERO.DEVOLVER_CAMBIO(MONEDERO);
    or
        delay 2*MINUTOS;
        -- no me ha hecho caso el camarero y paso de pagarle
    end select;
    PUT("CLIENTE: Ahora me quedan ");
    PUT(MONEDERO);
    PUT LINE(" pesetas ");
    PUT_LINE("CLIENTE: Me voy del restaurante");
end CLIENTE;
```

Pero si el cliente es un poco más formal, en vez de irse sin pagar del restaurante lo que puede hacer es leer el periódico hasta que el camarero le atienda. (pg150)

```
task body CLIENTE is
    …
begin
    …
```

```
        loop
            select
                CAMARERO.DEVOLVER_CAMBIO(MONEDERO);
            else
                -- leer el periódico porque no me
                -- ha hecho caso el camarero
            end select;
        end loop;
        …
    end CLIENTE;
```

En este último ejemplo hay que señalar, que el cliente estará realizando polling continuamente hacia el camarero hasta que éste último le atienda.

**Sentencia select con guardas**

Es una sentencia select como la vista anteriormente, donde algunas de las alternativas pueden ir encabezadas por la palabra reservada « when» y a continuación de ésta irá una expresión booleana.

Las reglas que rigen el funcionamiento de la sentencia select con guardas son las que se exponen a continuación:

* se evalúan todas las condiciones de guarda.

* el comportamiento de la sentencia select con guardas es como el que se vió de la sentencia select, pero sólo intervienen aquellas ramas que tengan su condición a cierto y las que no tengan condición de guarda (las que no tengan «when»).

Las condiciones de guarda se reevalúan al principio de cada ejecución de la sentencia select. Si una guarda no existe se considerará como cierta.

Si la guarda contiene alguna variable global hay que tener en cuenta que otra tarea puede variar su valor y, por tanto, una condición que parezca cierta sea falsa; aquí aparecen las condiciones de carrera.

Una vez explicada la sentencia select con guardas podemos ver como sería el funcionamiento de un buzón cíclico donde sólo existirá un único escritor y un sólo lector.

La función del escritor será leer una cadena de caracteres desde el teclado (pg151) y enviarla al buzón. Una vez haya finalizado de mandar toda la cadena mandará un «\», indicando que no enviará más caracteres. Y la función del lector será la de leer del buzón los caracteres que éste contiene e ir escribiéndolos en la pantalla. En el momento que le llegue el carácter «\» no intentará leer más caracteres del buzón, ya que nadie va a escribir y, por tanto, se daría un interbloqueo.

```
with TEXT_IO;
use TEXT_IO;
-- Importamos de TEXT_IO: el GET_LINE, PUT, NEW_LINE
procedure MAIL is
        task BUFFER is
                entry METER(X:in CHARACTER);
                entry SACAR(X:out CHARACTER);
        end BUFFER;
        task PROCESO_1;
        task PROCESO_2;
        task body BUFFER is
                N:constant:=4;
                A:array(1..N) of CHARACTER;
                I,J:INTEGER range 1..N:=1;
                CUENTA:INTEGER range O..N:=O;
        begin
            loop
                select
```

```
                when CUENTA < N = > -- guarda
                accept METER(X:in CHARACTER) do
                        A(l):=X;
                end METER;
                I:= I mod N+1;
                CUENTA:=CUENTA+1;
            or
                when CUENTA > 0 = > - - guarda
                accept SACAR(X: out CHARACTER) do
                        X:=A(J);
                end SACAR;
                J:= J mod N + l;
                CUENTA:=CUENTA + 1;
            end select;
        end loop;
    end BUFFER;                                          (pgl52)
    task body PROCESO_1 is
        LINEA:STRING (1.30):=(others => ' ');
        ULTIMO:NATURAL;
    begin
        GET LINE(LINEA,ULTIMO);
        for I in L.ULTIMO loop
        PUT("PROCESO l:"&LINEA(I));NEW LINE;
        BUFFER.METER(LINEA(I));
    end loop;
    BUFFER. METER(LINEA(I));
  end PROCESO_1;
  task body PROCESO_2 is
    CAR:CHARACTER;
  begin
    loop
        BUFFER. SACAR(CAR);
        exit when CAR=' \ ';
        PUT("PROCESO 2:"&CAR);NEW LINE;
    end loop;
  end PROCESO_2;
begin
  null;
end MAIL;
```

Para finalizar con Ada, vamos a hablar sobre el tema del planificador de tareas. Si utilizamos Ada para programar problemas de concurrencia nos interesará saber cuál es la política de este planificador. Pero si además desea mos utilizar Ada dentro de los sistemas en tiempo real será necesario conocer las restricciones que tiene en el tiempo tal planificador.

El planificador Ada se puede realizar de forma implícita y explícita. El *planificador implícito* es el que se encuentra en el run-time de Ada y su política es no expulsora. Es totalmente transparente al programador, el cual identificará los procesos que tienen que intervenir en su programa, olvidándose por completo del planificador. Podremos asignar prioridades a cada una de las tareas, siendo la implementación la que define el rango válido de dichas prioridades. El *planificador explícito* puede ser implementado mediante una familia de puntos de entrada o utilizando un pragma que incluyen ciertos compiladores: TIME_SLICE. La familia de entradas es una declaración que podemos realizar en una tarea y nos permite implementar el planificador. La segunda opción, mucho más rápida de llevar a cabo, utilizando el (pg153) planificador implícito, nos permite indicar cuál va a ser la duración de las rodajas de tiempo para las tareas.

El Manual de Referencia al Lenguaje no comenta nada sobre detalles del planificador, dejándolo de la mano del implementador del compilador. Por tanto, cuando se realicen sistemas que requieren una gran precisión será necesario estudiarse el run-time del compilador que estemos utilizando.

**Planificador implícito**

Todo run-time de Ada mantendrá una lista de tareas seleccionables para su ejecución. «Si dos tareas con diferentes prioridades son ambas elegibles para su ejecución usando los mismos procesadores físicos y los mismos re cursos, entonces la tarea con prioridad más alta deberá ser ejecutada». En los casos donde las tareas tengan la misma prioridad, el orden de ejecución no está definido y, por tanto, depende del compilador.

Para tareas que no tengan explícitamente definida su prioridad, las reglas del planificador no están definidas, excepto en el caso de que alguna de estas tareas entre en rendez-vous con una tarea que tiene explícitamente una prio ridad. Desde el momento que varias tareas con diferentes prioridades pueden interactuar, Ada ofrece una serie de reglas para determinar la prioridad de la cita, la cual vendrá dada por:

* la más alta de las prioridades de dos tareas si sus prioridades se especifican.

* la prioridad de la tarea con la prioridad especificada si una tarea no tiene la prioridad especificada.

* indeterminada si ambas tareas no tienen prioridad especificada.

El momento de cambió de contexto entre tareas viene determinado por cuando se invoque al planificador.

El planificador se invocará únicamente en los siguientes puntos:

* inicialización de las tareas.

* terminación de las tareas.

* llamada a puntos de entrada.

* llegar a una sentencia accept, a donde no ha habido ninguna llamada.

* llegar a una sentencia select.

* finalización del rendez-vous.

* ejecución de una sentencia delay.

* terminación de la ejecución de una sentencia delay.

Para concluir esta exposición del lenguaje de programación Ada hay que resaltar la importancia de los paquetes genéricos con las tareas. (pg154)

Un paquete genérico puede ser considerado como una plantilla donde le indicamos, mediante sus parámetros formales, los tipos de datos con los que va a trabajar. Así, el intercambio del contenido de dos variables podría ser genérico ya que es independiente del tipo de datos.

Si este concepto, por ejemplo, lo aplicamos a un buffer cíclico como el visto en el ejercicio anterior, está claro que lo podíamos haber implementado como genérico, ya que nos da igual qué tipo de datos maneje, estableciéndose únicamente cómo los maneja.

A continuación, se describe un paquete genérico que soporta semáforos. Se ha hecho genérico porque la inicialización del semáforo se tiene que realizar en la parte secuencial del programa.

```
generic
    VALOR_INICIAL:NATURAL:=O; --parámetro formal del paq. genérico
package SEMAFORO is
    type SEMAPHORE is limited private;
    procedure WAIT(S:SEMAPHORE);
    procedure SIGNAL(S:SEMAPHORE);
private
    task type TASK_SEMAPHORE is
        entry WAIT;
        entry SIGNAL;
    end TASK_SEMAPHORE;
    type SEMAPHORE is record
            SEMAFORO:TASK_SEMAPHORE;
            end record;
end SEMAFORO;
```

```
package body SEMAFORO is
    procedure WAIT(S:SEMAPHORE) is
    begin
        S.SEMAFORO.WAIT;
    end WAIT;
procedure SIGNAL(S:SEMAPHORE) is
begin
    S.SEMAFORO.SIGNAL;
end SIGNAL;                                    (pg155)
task body TASK_SEMAPHORE is
    CONTADOR:NATURAL:=VALOR INICIAL;
            -- el contador se inicializará con el valor que le hayamos
            -- pasado mediante el parametro formal VALOR_INICIAL
begin
    loop
        select
            when CONTADOR > 0 = >
                accept WAIT;
                CONTADOR:=CONTADOR-1;
        or
            when (CONTADOR=0 and WAIT' COUNT=0) or (CONTADOR > 0) = >
                accept SIGNAL;
                CONTADOR:=CONTADOR+1;
        or
            when CONTADOR=O and WAIT'COUNT/=0 = >
                accept SIGNAL;
                accept WAIT;
        or
            terminate;
        end select;
    end loop;
end TASK_ SEMAPHORE;
end SEMAFORO;
```

A continuación, escribimos el programa principal que utilizará este paquete genérico de semáforos.

Pretendemos crear cuatro tareas, las cuales tendrán las siguientes restricciones para su ejecución.

* el proceso B se ejecutará siempre que se haya ejecutado A

* el proceso C se ejecutará siempre que se haya ejecutado A y B

* el proceso D se ejecutará siempre que se haya ejecutado B

```
with   TEXT_IO,IIO,SEMAFORO;
use    TEXT_IO,IIO;
procedure SEMAFO is
    package SEMAFORITO_LIB is new SEMAFORO(0);
            -- inicializamos el semáforo a cero
    use SEMAFORITO_LIB;                                    (pg156)
SB,                                                  -- asociado con el proceso B
SC,                                                  -- asociado con el proceso C
SD:SEMAFORITO_LIB.SEMAPHORE;                         -- asociado con el proceso D
task A;
task B;
task C;
task D;
task body A is
begin
    PUT_LINE ("*A* Hago cosas y permito que se ejecute B");
```

```
    SIGNAL(SB);
    PUT_LINE("*A* Hago cosas y permito que se ejecute C");
    SIGNAL(SC);
    PUT_LINE("Muere la tarea A");
end A;
task body B is
begin
    WAIT(SB);
    PUT_LINE("*B* Hago cosas y activo a C");
    SIGNAL(SC);
    PUT_LINE("*B* Hago cosas y activo a D");
    SIGNAL(SD);
    PUT_LINE("Muere la tarea B");
end B;
task body C is
begin
    WAIT(SC);
    WAIT(SC);
    PUT_LINE("*C* Me permiten ejecutarme A y B");
    PUT_LINE("Muere la tarea C");
end C;
task body D is
begin
    WAIT(SD);
    PUT_LINE("*D* me permite ejecutarme B");
    PUT_LINE("Muere la tarea D");
end D;                                                    (pg157)
begin
    null;
end SEMAFO;
```

La ejecución de este programa queda de la siguiente forma:

```
*A* Hago cosas y permito que se ejecute B
*B* Hago cosas y activo a C
*A* Hago cosas y permito que se ejecute C
*B* Hago cosas y activo a D
Muere la tarea A
*D* me permite ejecutarme B
Muere la tarea D
*C* Me permiten ejecutarme A y B
Muere la tarea C
Muere la tarea B
```

## MODULA-2

El lenguaje Modula-2 es un descendiente directo de Pascal y de Módula. Fue desarrollado por el prefesor Niklaus Wirth en el Institut für informatik de ETH Zürich. La definición formal del lenguaje se publicó en 1980 y el primer compilador disponible comercialmente apareció en 1981.

Modula-2 surgió de experimentos en multiprogramación con el objeto de poder disponer de un lenguaje que incluyera todos los aspectos de Pascal y añadiera el concepto de módulo y herramientas de multiprogramación.

En lo que sigue expondremos los aspectos del lenguaje Modula-2 relacionados con la programación concurrente.

### Corrutinas

Para implementar la concurrencia de procesos, Modula-2 se basa en las **corrutinas**.

Las corrutinas son procedimientos que se ejecutan independientemente (no concurrentemente). De hecho, son adecuadas para soportar la concurrencia simulada (un procesador multiplexado entre varios procesos).

Las corrutinas deben crearse antes de poder ser llamadas. Una corrutina se crea especificando el procedimiento que ejecutará y un área de memoria donde se ejecutará la corrutina. Una vez creada, la corrutina es ejecutable, pero no comienza a ejecutarse inmediatamente. Esto ocurrirá cuando otra corrutina le ceda control. (pg158)

En Modula-2, las corrutinas no se pueden acabar (aunque si pueden dejar de recibir control). Por ello siempre adquieren la forma de:

```
LOOP
   < sentencias >
END
```

Un proceso se crea mediante el procedimiento NEWPROCESS, cuya definición es:

PROCEDURE NEWPROCESS (P: PROC; a: ADDRESS; n: CARDINAL; VAR new: ADDRESS);

donde:

- P es el procedimiento que ejecutará el nuevo proceso. Debe ser un procedimiento sin parámetros declarado a nivel global.

- a y n especifican la dirección y el tamaño del área de datos en la que se ejecutará el proceso. Contendrá no sólo la pila de ejecución del proceso, sino también el estado de ejecución del mismo. Esta área normalmente se declara como un ARRAY OF WORD y su dirección y tamaño quedan establecidos mediante las funciones ADR y SIZE.

- new es el nombre del nuevo proceso.

Observe que hemos hablado de procesos y no de corrutinas que sería lo propio. Es más, en la segunda revisión de Modula-2, el procedimiento NEWPROCESS devolvía una variable de tipo PROCESS (antes hemos declarado new de tipo ADDRESS que es como está definido en la última revisión de Modula-2).

Se suele adoptar el término de proceso en lugar de corrutina, porque cuando se programan actividades concurrentes, es usual implementar también un planificador que se encargue de ceder control automáticamente a una u otra corrutina. Puesto que la transferencia de control ya no es explícita, podemos hablar de procesos. En cualquier caso, es conveniente no perder de vista que lo que Modula-2 ofrece como herramienta definida en el lenguaje es la corrutina y no el proceso.

La transferencia de control entre corrutinas se realiza mediante el procedimiento TRANSFER, cuya definición es:

PROCEDURE TRANSFER (VAR source, destination: ADDRESS); donde:

- source es la corrutina que cede control.

- destination, es la corrutina que gana control.

TRANSFER suspende al proceso actual, asigna su estado de ejecución a (pg159) la variable source y reanuda la ejecución del proceso identificado por destination.

**Procesos y señales**

Sin duda alguna, tanto NEWPROCESS como TRANSFER son mecanismos de demasiado bajo nivel como para que resulte atractiva su utilización en programación concurrente. Por ello, prácticamente todos los compilado res disponibles comercialmente incorporan un módulo con facilidades de alto nivel para trabajar con procesos. A continuación se detalla el módulo propuesto por Wirth:

```
DEFINITION MODULE Processes;
   TYPE SIGNAL;
   PROCEDURE StartProcess (P: PROC; n: CARDINAL);
      (* arranca un proceso concurrente con programa P y un área *)
```

```
        (* de trabajo de tamaño n. PROC es un tipo standard   *)
        (* definido como PROC=PROCEDURE  *)
    PROCEDURE SEND (VAR s: SIGNAL);
        (* se reanuda un proceso que estaba esperando por s *)
    PROCEDURE WAIT (VAR s: SIGNAL);
        (* espera a que algún proceso ejecute SEND sobre s *)
    PROCEDURE Awaited (s: SIGNAL): BOOLEAN;
        (* Awaited(s) = al menos un proceso está esperando por s *)
    PROCEDURE Init (VAR s: SIGNAL);
        (* inicialización obligatoria para las señales *)
END Processes.
```

No damos aquí la implementación de este módulo. El lector interesado puede recurrir a la bibliografía [Wirt83].

Si su compilador no tiene este módulo tendrá otro parecido. En cualquier caso, y por lo general, el programador suele implementar sus propios módulos de facilidades para la programación concurrente.

Observe en el módulo descrito anteriormente que mientras el equivalente de NEWPROCESS es StarProcess, no existe equivalencia con TRANSFER. Con este módulo, un proceso pierde únicamente control cuando ejecuta WAIT(s) o SEND(s). Estos dos nuevos procedimientos introducen facilidades para la comunicación entre procesos vía señales (la comunicación a través de variables compartidas la veremos más adelante). Mientras que los monitores son adecuados para el intercambio de información, las señales están diseñadas para la sincronización entre procesos. (pg160)

Es necesario darse cuenta que cada señal denota una cierta condición o estado entre las variables del programa, y mandar una señal significa que se ha dado esa condición. Un proceso esperando por una señal puede asumir que dicha condición se ha dado cuando recibió tal señal. Si hay varios procesos esperando por una señal S, tan sólo se reanudará a uno de ellos a la llegada de esta señal. Si no había nadie esperando, el efecto de SEND es nulo.

Para implementar el otro mecanismo de comunicación (variables compartidas), Modula-2 dispone de facilidades para soportar monitores. Como ya se dijo, un monitor proporciona acceso exclusivo sobre variables compar tidas. Para realizar monitores en Modula-2 basta con declarar un módulo con prioridad, definiendo en él las variables que van a ser compartidas y exportando el conjunto de procedimientos que consituyen los puntos de entrada la monitor. Esto podemos observarlo en el siguiente ejemplo:

```
MODULE miMonitor [2];
(* el [2] da prioridad al módulo y asegura la exclusión mutua *)
    EXPORT pl, p2; (* puntos de entrada *)
    VAR ..... (* variables compartidas y otras *)
    PROCEDURE pl (...);
        …..
    END p1;
    PROCEDURE p2 (...);
        …..
    END p2;
    (* otros procedimientos locales al monitor *)
    .....
    BEGIN
        (* sentencias de inicialización del módulo < monitor> *)
    END miMonitor.
```

# *Capítulo5*

# *Problemas de aplicación*

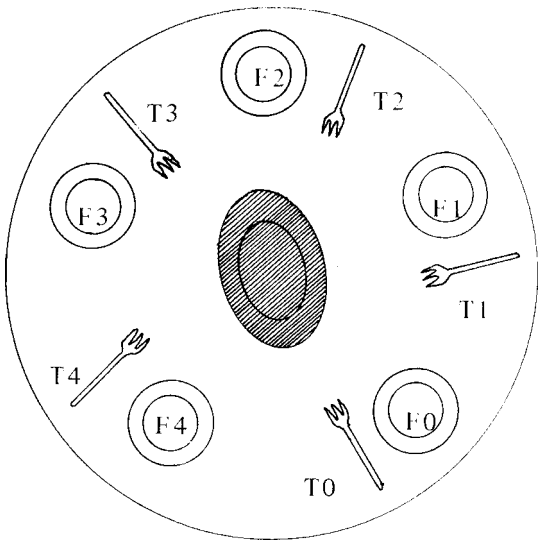
**Problema Ejemplo n.° 1**

El problema se plantea en un monasterio budista donde viven cinco monjes dedicados a «filosofar». Cada filósofo es feliz únicamente pensando y de cuando en cuando come. La vida de los filósofos es un ciclo del tipo:

```
REPEAT
    piensa
    come
FOREVER
```

El comedor común tiene una mesa como ilustra la Fig. 39:

FIGURA 39




(pg162)

En el centro de la mesa hay un plato de espagueti. Existen 5 platos y 5 tenedores. Cuando un filósofo desea comer, entra en el comedor, toma asiento, come y vuelve a su celda. Para comer espaguetis necesita 2 tenedores.

El problema consiste en inventar un protocolo que permita comer a los filósofos. Cada filósofo puede utilizar únicamente los dos tenedores adyacentes a su plato. El protocolo debe satisfacer las condiciones normales de:

1.° exclusión mutua (ningún filósofo intenta usar el mismo tenedor al mismo tiempo que otro).

2.° estar libre de deadlock y lockout.

(NOTA: si un filósofo está comiendo tiene dos tenedores).

**PROBLEMA Ejemplo n.º 1 (Solución)**

En un primer intento por resolver el problema vamos a utilizar semáforos. Asociaremos un semáforo a cada tenedor, de tal manera que cuando un filósofo quiera cogerlo deberá hacer una operación WAIT sobre dicho semá foro. Evidentemente, los semáforos los inicializaremos a 1 de forma que si el tenedor ya está cogido, una operación WAIT retrasará el proceso hasta que dicho tenedor quede libre. Cuando un filósofo suelte el tenedor, ejecutará un SIGNAL sobre el semáforo asociado a dicho tenedor. El programa resultante de esta solución es el algoritmo (1.1).

```
PROGRAM Filosofos; (* algoritmo (1.1): 1.° intento *)
   VAR tenedores: ARRAY [0..4] OF SEMAPHORE;
                  i: INTEGER;
   PROCEDURE Filosofo (i: INTEGER);
      BEGIN
         REPEAT
            Piensa;
            WAIT (tenedores[i]);
            WAIT (tenedores[(i+l) mod 5]);
            come;
            SIGNAL (tenedores[i]);
            SIGNAL (tenedores[(i+l) mod 5])
         FOREVER
      END;                                                    (pg163)
   BEGIN
      FOR i:=0 TO 4 DO INIT (tenedores[i], 1);
      COBEGIN
         Filosofo(0); Filosofo(1); Filosofo(2);
         Filosofo(3); Filosofo(4)
      COEND
   END.
```

La objeción que se puede hacer a esta solución es que pueden presentarse casos de interbloqueos, por lo cual la solución no es correcta. Considere lo que ocurre si los 5 filósofos se sientan a la mesa al mismo tiempo y ejecutan a la vez la operación WAIT (tenedores[i]). Después de esto todos los filósofos tienen el tenedor de su izquierda y quedan bloqueados a la espera que el filósofo de su derecha libere el tenedor que posee. Puesto que nadie es capaz de liberar su tenedor, todos se quedan en una espera indefinida.

Para evitar la situación de interbloqueo que hemos mencionado anteriormente seguiremos la siguiente política: en el comedor común podrá haber como máximo 4 filósofos. Esto asegura que al menos un filósofo estará comiendo, con lo que la anterior situación de interbIoqueo no se puede dar. (Se supone que los filósofos acaban de comer en un tiempo finito).

Para implementar esta solución declararemos aparte de un semáforo para cada tenedor, otro que actúe como contador y que controle el número de filósofos que están en el comedor. Cuando un filósofo quiera comer, deberá entrar al comedor, para lo cual ejecutará una operación WAIT sobre el semáforo 'puerta'. Cuando salga del comedor ejecutará una operación SIGNAL sobre dicho semáforo. Evidentemente, el semáforo 'puerta' se inicializará a 4. El programa que implementa esta solución es el algoritmo (1.2).

```
PROGRAM Filosofos; (* algoritmo (1.2): 2.° intento *)
   VAR  tenedores: ARRAY [0..4] OF SEMAPHORE;
            puerta: SEMAPHORE;
                 i: INTEGER;
   PROCEDURE Filosofo (i: INTEGER);
      BEGIN
         REPEAT
            Piensa;
            WAIT (puerta); (* entrada al comedor *)
            WAIT (tenedores[i]);
```

```
            WAIT (tenedores[(i+l) mod 5]);
            come;                                          (pg164)
            SIGNAL (tenedores[i]);
            SIGNAL (tenedores[(i+l) mod 5]);
            SIGNAL (puerta) (* salida del comedor *)
        FOREVER
    END;
BEGIN
    FOR i:=0 TO 4 DO INIT (tenedores[i], 1);
    INIT (puerta, 4);
    COBEGIN
        Filosofo(0); Filosofo(1); Filosofo(2);
        Filosofo(3); Filosofo(4)
    COEND
END.
```

Se puede demostrar de la siguiente manera que el algoritmo (1.2) está libre de interbloqueos [BenA82]:

**1.° Si Pi ejecuta WAIT (tenedores[¡]), finalmente completará el WAIT:**

Si Pi está detenido en el WAIT (tenedores[i]) significa que tenedores[i]= 0, lo que implica que Pi-1 está comiendo (está utilizando su tenedor derecho que siempre se coge después del izquierdo, lo que quiere decir que tiene dos tenedores y puede comer) y finalmente terminará con lo que Pi-1 ejecutará un SIGNAL (tenedores[i+1]) con lo que libera el tenedor izquierdo del proceso Pi y éste podrá cogerlo.

**2.° Si Pi espera indefinidamente en tenedores[i+1] significa que Pi+l espera indefinidamente en tenedores[i+2]:**

Solamente Pi y Pi+l compiten por el semáforo tenedores[i+1]. Si Pi+l está pensando entonces Pi no puede bloquearse por tenedores[i+1]. Similarmente, Pi y Pi+l no pueden simultáneamente bloquearse en el mismo se máforo tenedores[i+l] (o lo tiene uno o lo tiene otro). Así, si Pi está bloqueado en tenedores[i+1] y asumimos que nunca se hará un SIGNAL sobre tenedores[i+l], la única posibilidad es que Pi+l esté bloqueado indefinidamente en el semáforo tenedores[i+2].

**3.° Si Pi ejecuta WAIT (tenedores [i+1]), finalmente completará el WAIT y comerá:**

Por sucesivas aplicaciones del lema 2.°, si Pi espera indefinidamente en tenedores[i+l], entonces Pi+j esperan indefinidamente en tenedores[i+j+1] para j=1..4, pero esto contradice el invariante del semáforo 'puerta'.

Esta solución cumple con las dos condiciones formuladas en el enunciado del problema. Restringe, sin embargo, a que el número de filósofos en el comedor sea inferior a 5. (pg165)

La siguiente solución permite que entren los 5 filósofos a la habitación (aun sabiendo que sólo dos de ellos pueden estar comiendo al mismo tiempo). Para evitar la situación de interbloqueo que se producía en el algoritmo (1.1) vamos a utilizar asignación total. Es decir, un filósofo o coge los dos tenedores o no coge ninguno. Para implementar esta solución recurriremos a las RCC. En un array de 5 elementos (uno por cada filósofo) indicaremos cuantos tenedores tiene disponibles un filósofo. Inicialmente, puede coger los dos. Cuando alguno de sus filósofos adyacentes esté comiendo, el filósofo en cuestión tendrá disponibles 0 tenedores (si sus dos vecinos están comiendo) ó 1 tenedor (si sólo uno de sus vecinos está comiendo). El programa que implementa esta política se esboza en el agoritmo 1.3.

```
PROGRAM Filosofos; (* algoritmo (1.3): 3° intento *)
    VAR tenedores: SHARED ARRAY[0..4] OF 0..2;
                i: INTEGER;
    PROCEDURE Filosofo (i: INTEGER);
        BEGIN
            REPEAT
                Piensa;
                REGION tenedores DO BEGIN
```

```
                AWAIT tenedores[i] = 2;
                tenedores [(i+1) mod 5]:= tenedores [(i+1) mod 5] - 1;
                tenedores [(i+4) mod 5]:= tenedores [(i+4) mod 5] - 1
            END;
            come;
            REGION tenedores DO BEGIN
                tenedores [(i+1) mod 5]:= tenedores [(i+1) mod 5] + 1;
                tenedores [(i+4) mod 5]:= tenedores [(i+4) mod 5] + 1
            END;
        FOREVER
    END;
BEGIN
    FOR i:= 0 TO 4 DO tenedores[i]:= 2;
    COBEGIN
        Filosofo(0); Filosofo(1); Filosofo(2);
        Filosofo(3); Filosofo(4)
    COEND
END.
```

Otras posibilidades soluciones en esta línea son: (pg166)

```
PROGRAM Filosofos;
    VAR pensando: SHARED ARRAY [0..4] OF BOOLEAN;
        i: INTEGER;
    PROCEDURE Filosofo (i: INTEGER);
        BEGIN
            REPEAT
                Piensa;
                REGION pensando DO BEGIN
                    AWAIT (pensando[(i-1) mad 5]) & (pensando[(i+l) mod 5]);
                    pensando[i]:= FALSE
                END;
                come;
                REGION pensando DO pensando[i]:= TRUE
            FOREVER
        END;
    BEGIN
        FOR i:=0 TO 4 DO pensando[i]:= TRUE
        COBEGIN
            Filosofo(0); Filosofo(1); Filosofo(2);
            Filosofo(3); Filosofo(4)
        COEND
    END.
```

Y también es correcta la siguiente solución:

```
PROGRAM Filosofes;
    VAR tenedores: SHARED ARRAY [0..4] OF BOOLEAN;
        i: INTEGER;
    PROCEDURE Filosofo (i: INTEGER);
        BEGIN
            REPEAT
                Piensa;
                REGION tenedores DO BEGIN
                    AWAIT (tenedores[i]) & (tenedores[(i+l) mod 5]);
                    tenedores[i]:= FALSE
                    tenedores[(i+l) mod 5]:= FALSE
                END;                                          (pg167)
                come;
```

```
            REGION tenedores DO BEGIN
                tenedores[i]:= TRUE;
                tenedores[(i+l) mod 5]:= TRUE;
            END
        FOREVER
    END;
BEGIN
    FOR i:=0 TO 4 DO tenedores[i]:= TRUE;
    COBEGIN
        Filosofo(0); Filosofo(1); Filosofo(2);
        Filosofo(3); Filosofo(4)
    COEND
END.
```

La solución presentada en el algoritmo (1.3) puede originar lockout. Si dos filósofos se alían, pueden conseguir que el filósofo que esté entre los dos nunca coma. Considere qué ocurre si los filósofos 1 y 3 se alían para impedir comer al filósofo 2. En un instante puede estar comiendo el 1, por lo que el 2 no puede hacerlo. Antes de que el 1 termine, entra el 3 y come, con lo que cuando acabe el 1, el 2 sigue sin poder comer.

Para evitar el lockout diseñaremos una solución tal que el último filósofo que acabe de comer lo indique y si hay alguien que lo desea se le dé a este último prioridad. Para ello esbozamos la solución del algoritmo (1.4)

```
PROGRARM Filosofos; (* algoritmo (1.4): 4° intento *)
    VAR mesa: SHARED RECORD
                        tenedores: ARRAY [0..4] OF 0..2;
                        yoSoyElUltimo: event mesa
                    END;
        i:INTEGER
    PROCEDURE Filosofo (i: INTEGER);
        BEGIN
            REPEAT
                Piensa;
                REGION mesa DO BEGIN
                    AWAIT tenedores[i] = 2;
                    tenedores[(i+1) mod 5]:= tenedores [(i+1) mod 5] - 1;
                    tenedores[(i+4) mod 5]:= tenedores [(i+4) mod 5] - 1
                END;
                come;                                                    (pg168)
                REGION mesa DO BEGIN
                    tenedores[(i+1) mod 5]:= tenedores [(i+1) mod 5] + 1;
                    tenedores[(i+4) mod 5]:= tenedores [(i+4) mod 5] + 1
                    CAUSE (yoSoyElUltimo);
                    AWAIT (yoSoyElUltimo)
                END;
            FOREVER
        END;
    BEGIN
        FOR i:=0 TO 4 DO mesa. tenedores[i]:=2;
        COBEGIN
            Filosofo(0); Filosofo(1); Filosofo(2);
            Filosofo(3); Filosofo(4)
        COEND
    END.
```

En este algoritmo podemos apreciar que un filósofo no puede volver a comer hasta que otro lo haya hecho. Esto supone una restricción adicional al problema enunciado. Veamos ahora en el algoritmo (1.4) el escenario del al goritmo (1.3) que conducía al lockout. Si el filósofo 1 come primero, se dormirá en el

AWAIT (yoSoyElUltimo) hasta que otro filósofo haya comido. Supongamos que antes de que vaya a comer el filósofo 2 lo hace el filósofo 3. A continuación intenta comer el 2 y al no tener sus dos tenedores se duerme en el AWAIT tenedores[2]=2. En esta situación el filósofo 3 está comiendo; el 1 está esperando un evento y el dos esperando tener sus tenedores. Cuando acabe de comer el filósofo 3 liberará sus tenedores, despertará al 1 y se pondrá él mismo a dormir. En este momento, el filósofo 1 se irá a pensar y el 2 podría comer, puesto que ya tiene sus dos tenedores. No obstante, puede ocurrir que el filósofo 1 piense muy rápidamente y no le dé opción al 2 a coger sus tenedores (el proceso 1 se planifica antes que el 2 por tener mayor prioridad). En esta situación se puede producir el lockout.

Veamos, por tanto, que evitar el lockout no es trivial. A continuación se da una solución al problema que evita los interbloqueos y el lockout.

```
PROGRAM Filosofos;
   VAR v: SHARED RECORD
                        tenedor, esperando: ARRAY [0..4] OF BOOLEAN;
                        filo: ARRAY [0..4] OF EVENT v;
                    END;
       i: INTEGER;                                          (pg169)
   PROCEDURE filosofo (i: INTEGER);
      BEGIN
         REPEAT
            piensa;
            REGION v DO BEGIN
               WHILE (esperando[(i+l) mod 5]) OR (esperando[(i-1) mod 5]) DO AWAIT (filo[i]);
               WHILE NOT ( (tenedor[i]) & (tenedor[(i+l) mod 5])) DO BEGIN
                  esperando[i]:= TRUE;
                  AWAIT (filo[i])
               END;
               esperando[i]:= FALSE;
               tenedor[i]:= FALSE;
               tenedor[(i+l) mod 5]:= FALSE
            END; (* RC v *)
            come;
            REGION v DO BEGIN
               tenedor[i]:= TRUE;
               tenedor[(i+l) mod 5]:= TRUE;
               CAUSE (filo[(i+l) mod 5]);
               CAUSE (filo[(i-1) mod 5])
            END
         FOREVER
      END;
   BEGIN (* main *)
      FOR i:= 0 TO 4 DO BEGIN
         tenedor[i]:= TRUE;
         esperando[i]:= FALSE
      END;
      COBEGIN
         filosofo(0); filosofo(1); filosofo(2);
         filosofo(3); filosofo(4)
      COEND
   END.
```

**PROBLEMA Ejemplo n.° 2**

Sea una carretera por la que circulan coches en los dos sentidos. La carretera cruza un río donde sólo es posible la circulación de coches en un sentido (pg170)

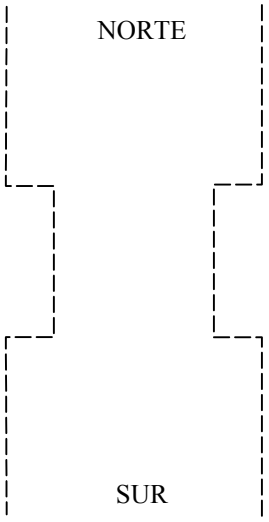



**FIGURA 40**

(ver Fig. 40). Sobre el puente pueden estar varios coches del mismo sentido.

Se pide que diseñe un protocolo que permita circular a los coches sobre el puente y que realice un programa siguiendo tal protocolo. En este programa cada coche estará representado por un proceso. Se debe conseguir que:

1) No haya interbloqueos. Dos coches de diferentes sentidos no pueden quedarse bloqueados en medio del puente.

2) No haya lockout. El protocolo que permite el paso de los coches sobre el puente debe ser justo. No se puede favorecer a los coches de algún sentido a costa de perjudicar a los otros.

**PROBLEMA Ejemplo n.° 2 (Solución)**

En una primera aproximación vamos a caracterizar el puente como una variable compartida que indicará cuantos coches del norte o del sur están en él. Así pues:

```
VAR puente: SHARED RECORD
                dentroN,                (* CN en el puente *)
                dentroS: CARDINAL.      (* CS en el puente *)
            END;
```

El problema se puede escribir como indica el algoritmo (2.1): (pg171)

```
PROGRAM Río;   (* algoritmo (2.1): 1.ª aproximación *)
   VAR puente: SHARED RECORD
                  dentroN, (* CN en el puente *)
                  dentroS: CARDINAL (* CS en el puente *)
               END;
   PROCEDURE CN (i: INTEGER);
      BEGIN
         REGION puente DO BEGIN
(1)         AWAIT dentroS= 0;
(2)         dentroN:= dentroN + 1
         END;
         (* cruzando el puente *)
         REGION puente DO
            dentroN:= dentroN - 1
      END;
   PROCEDURE CS (i: INTEGER);
      BEGIN
         REGION puente DO BEGIN
            AWAIT dentroN = 0;
```

```
            dentroS:= dentroS + 1
        END;
        (* cruzando el puente *)
        REGION puente DO
        dentroS:= dentroS - 1
    END;
BEGIN
    WITH puente DO BEGIN
        dentroS:= 0;
        dentroN:= 0
    END;
    COBEGIN
        CN(1); CN(2); CN(3); .....; CN(n);
        CS(I); CS(2); CS(3); .....; CS(n)
    COEND
END.
```

El programa anterior cumple con la primera condición del enunciado: no hay interbloqueos. Antes de que un coche se aventure a cruzar el puente se asegura que no hay otro coche del sentido opuesto cruzándolo. Sin embargo, no cumple con la segunda condición: puede existir lockout. Imagine que el primer coche que cruza el puente es uno del norte. Si continuamente (pg172) siguen llegando coches del norte, de tal manera que los del sur siempre encuentran a uno del norte dentro, los del sur jamás pasarán.

Independientemente de esto, es interesante observar qué ocurriría si en el algoritmo (2.1) invertimos el orden de las sentencias (1) y (2). Si esta inversión la realizamos tan sólo en uno de los procedimientos (por ejemplo, el que controla los coches del norte), podemos llegar al siguiente escenario: 'estando atravesando el puente coches del sur, el siguiente que llegue del norte pondrá dentroN a 1 y se esperará a que dentroS=0. Si llegan ahora más coches del sur comprobarán que dentroN es diferente de 0 y se esperarán. Esto no es correcto, ya que podrían haber cruzado. El error proviene de que consideramos que cuando un coche incrementa dentroN, a continuación entrará al puente, por lo que dentroN indica cuantos coches del norte hay en el puente. Al alterar el orden las sentencias (1) y (2) está variable, ya no significa cuantos coches del norte están dentro del puente.

Por otro lado, si cambiamos las sentencias (1) y (2) en los dos procedimientos, daremos lugar a interbloqueos. Si dentroS es mayor que 0 y llega uno del norte pone dentroN a 1 y espera. Salen del puente los del sur y rápidamente llegan más del sur haciendo dentroS diferente de 0. Ningún proceso puede avanzar. También es interesante observar que aunque las variables dentroN y dentroS sólo se modifican en sus respectivos procedimientos, sigue siendo obligatorio declararlas como compartidas, ya que existen n procesos de cada tipo.

Sigamos ahora con el problema. La solución dada sufre de lockout. Se trata de evitar que los coches de un sentido monopolicen el puente. Para ello, proponemos que se establezcan turnos para pasar. Es decir, después de que hayan pasado 10 coches de un sentido permitiremos pasar a 10 coches del otro y así sucesivamente. Para implementar esta solución deberemos llevar la cuenta del número de coches de un sentido que han pasado sobre el puente. Añadiremos dos variables más: pasadosN y pasadosS. El algoritmo (2.2) implementa esta nueva política:

```
PROGRAM Río; (* algoritmo (2.2): 2.ª aproximación *)
    VAR puente: SHARED RECORD
                    DentroN,                (* CN en el puente *)
                    dentroS: CARDINAL;      (* CS en el puente *)
                    pasadosN,               (* CN que han pasado el puente *)
                    pasadosS: CARDINAL      (* CS que han pasado el puente *)
                END;                                                    (pg173)
    PROCEDURE CN (i: INTEGER);
        BEGIN
            REGION puente DO BEGIN
                AWAIT (dentroS = 0) & (pasadosN < 10); dentroN:= dentroN + 1
            END;
            (* cruzando el puente *)
```

```
            REGION puente DO BEGIN
                dentroN:= dentroN - 1;
(1)             pasadosN:= pasadosN + 1;
(2)             IF pasadosN = 10
                THEN pasadosS:= 0
            END;
        END;
    PROCEDURE CS (i: INTEGER);
        BEGIN
            REGION puente DO BEGIN
                AWAIT (dentroN = 0) & (pasadosS < 10);
                dentroS:= dentroS + 1
            END;
            (* cruzando el puente *)
            REGION puente DO BEGIN
                dentroS:= dentroS – 1;
                pasadosS:= pasadosS + 1;
                IF pasadosS = 10
                THEN pasadosN:= 0
            END;
        END;
    BEGIN
        WITH puente DO BEGIN
            dentroS:= 0; dentroN:= 0;
            pasadosS:= 0; pasadosN:= 0;
        END;
        COBEGIN
            CN(1); CN(2); CN(3); .....; CN(n);
            CS(l); CS(2); CS(3); .....; CS(n)
        COEND
    END.
```

Este nuevo planteamiento es válido cuando por ambos sentidos llegan una gran cantidad de coches (después de pasar 10 coches en un sentido existen (pg1174) otros 10 coches del otro sentido esperando hacerlo). Si no ocurre así, puede que en un sentido estén esperando, mientras que en el otro no hay coches.

Analizemos el algoritmo (2.2). En estas condiciones no se cambia el sentido cada 10 coches. Si el puente es lo bastante largo como para albergar a más de 10 coches pueden entrar muchos, ya que pasadosN(S) se actualiza una vez cruzado el puente. Solución: poner las sentencias (1) y (2) en la RC que da el paso al puente (algoritmo (2.3)). Es decir, ahora en lugar de contabilizar al final del puente se indica cuantos coches han comenzado a pasarlo (en su inicio).

```
PROGRAM Río; (* algoritmo (2.3): 3.ª aproximación *)
    VAR puente: SHARED RECORD
                    dentroN, dentroS: CARDINAL;
                    pasadosN, pasadosS: CARDINAL
                END;
    PROCEDURE CN (i: INTEGER);
        BEGIN
            REGION puente DO BEGIN
                AWAIT (dentroS = 0) & (pasadosN < 10);
                dentroN:= dentroN + 1;
                pasadosN:= pasadosN + 1;
                IF pasadosN = 10
                THEN pasadosS:= 0
            END;
            (* cruzando el puente *)
            REGION puente DO
                dentroN:= dentroN - 1;
```

```
        END;
    PROCEDURE CS (i: INTEGER);
        BEGIN
            REGION puente DO BEGIN
                AWAIT (dentroN = 0) & (pasadosS < 10);
                dentroS:= dentroS + 1;
                pasadosS:= pasadosS + 1;
                IF pasadosS = 10
                THEN pasadosN:= 0
            END;                                                    (pg175)
             (* cruzando el puente *)
            REGION puente DO
                dentroS:= dentroS - 1
        END;
    BEGIN
        WITH puente DO BEGIN
            dentroS:= 0; dentroN:= 0;
            pasadosS:= 0; pasadosN:= 0;
        END;
        COBEGIN
            CN(1); CN(2); CN(3); .....; CN(n);
            CS(l); CS(2); CS(3); .....; CS(n)
        COEND
    END.
```

En esta solución pueden pasar intercaladamente coches de ambos sentidos. Inicialmente dentroN=0, dentroS=0, pasadosN=O y pasadosS=0. Si llega un CN pone dentroN y pasadosN a 1. Cuando salga del puente pone dentroN a 0. A continuación llega un CS y comprueba la condición, por lo que entra al puente. Esta situación no es la que se pretendía. Para conseguir el objetivo de que pasen en bloques de 10 coches es necesario alterar la condición de entrada al puente: la condición del Await. Debe comprobar que no está cruzando un coche del sentido opuesto y esperar a que el número de coches que han cruzado del sentido opuesto es igual a 10. Es decir, sustituir la condición del Await por:

```
"CN-i"
        .....
        AWAIT (dentroS=O) & (pasadosS=10)
        …..
                                                y
"CS-i"
        .....
        AWAIT (dentroN=O) & (pasadosN=10)

        .....
```

La inicialización la hacemos como: dentroN=O; dentroS=O; pasadosN=O; pasadosS=10. Con esta inicialización obligamos a que el primer coche que pase el puente sea uno del norte.

Para evitar dar prioridad al principio a uno de los dos sentidos, vamos a añadir dos variables que indiquen de quién es el turno (turnoN y turnoS) y que inicialmente pondremos a TRUE para dar la misma oportunidad a los (pg176) dos. El coche que consiga entrar primero negará el turno de los coches del sentido opuesto. Cuando detecte que ya han pasado 10 coches del mismo sentido, pondrá el turno de los otros a TRUE y el suyo a FALSE. Esta política se implementa en el algoritmo (2.4):

```
PROGRAM Río; (* algoritmo (2.4): 4.ª aproximación *)
    VAR puente: SHARED RECORD
                        dentroN,
                        dentroS: CARDINAL; pasadosN,
                        pasadosS: CARDINAL; turnoN,
                        turnos: BOOLEAN
                    END;
```

```
PROCEDURE CN (i: INTEGER);
    BEGIN
        REGION puente DO BEGIN
            AWAIT (dentroS = 0) & turnoN;
            turnoS:= FALSE;
            dentroN:= dentroN + 1;
            pasadosN:= (pasadosN + 1) mod 10;
            IF pasadosN = 0
            THEN BEGIN
                    turnoN:= FALSE;
                    turnoS:= TRUE
                END
        END;
        (* cruzando el puente *)
        REGION puente DO
            dentroN:= dentroN - 1;
    END;
PROCEDURE CS (i: INTEGER);
    BEGIN
        REGION puente DO BEGIN
            AWAIT (dentroN = 0) & turnoS;
            turnoN:= FALSE;
            dentroS:= dentroS + 1;
            pasadosS:= (pasadosS + 1) mod 10;
            IF pasadosS = 0
            THEN BEGIN                                    (pg177)
                    turnoS:= FALSE;
                    turnoN:= TRUE
                END
        END;
        (* cruzando el puente *)
        REGION puente DO
            dentroS:= dentroS - 1
    END;
BEGIN
    WITH puente DO BEGIN
        dentroS:= 0; dentroN:= 0;
        pasadosS:= 0; pasadosN:= 0;
        turnoN:= TRUE; turnoS:= TRUE
    END;
    COBEGIN
        CN(1); CN(2); CN(3);..... ; CN(n);
        CS(1); CS(2); CS(3);..... ; CS(n)
    COEND
END.
```

Esta solución es correcta, pero sigue presentando un inconveniente, como se aprecia en el siguiente escenario:

1) Han pasado 6 coches del norte y el puente está vacío.

2) En el sur hay 20 coches esperando a que pasen otros 4 más del norte.

3) Los 4 del norte que faltan por pasar para completar el bloque de 10, tardan muchísimo en llegar.

Lo que ha ocurrido es que hemos impuesto la justicia en el paso por el puente sin mirar si hay conflictividad o no. Para solucionarlo, tan sólo aplicaremos esta política de justicia (turnos) cuando en ambos sentidos haya coches esperando. La solución a implementar debe contemplar un nuevo componente: si en el sentido contrario hay o no hay coches esperando cruzar. Para ello declaramos dos nuevas variables,

esperandoN y esperandoS, que indican cuantos coches hay esperando en cada sentido. La variable compartida puente queda como:

```
VAR puente: SHARED RECORD
                dentroN, dentro S: CARDINAL;
                pasadosN, pasadosS: CARDINAL;
                turnoN, turnoS: BOOLEAN;
                esperandoN, esperandoS: CARDINAL
            END;
```
(pg178)

y el procedimiento que controla el paso de los coches del norte es el siguiente (para los coches del sur la solución es simétrica):

```
PROCEDURE CN (i: INTEGER);
   BEGIN
      REGION puente DO BEGIN
         esperandoN:= esperandoN + 1;
         AWAIT ((dentroS=O) & (turnoN OR esperandoS=O));
         esperandoN:= esperandoN - 1;
(1)      turnoS:= FALSE;
         dentroN:= dentroN + 1;
         pasadosN:= (pasadosN + 1) mod 10;
         IF pasadosN = 0
         THEN BEGIN
                 turnoN:= FALSE;
                 turnoS:= TRUE
              END
      END;
      (* cruzando el puente *)
      REGION puente DO
         dentroN:= dentroN - 1
   END;
```

Inicialmente, las variables se ponen a 0 y a TRUE.

Observe que un coche del norte se espera a que no haya ninguno del sur en el puente y a que o bien sea su turno, o bien que no siéndolo no haya ninguno del sur esperando.

Del estudio de esta solución obtenemos una posible situación de interbloqueo. Considere el siguiente escenario:

1) Han pasado 10 coches del norte: 9 han salido y el último está atravesando el puente. Luego:

dentroN=1; dentroS=0; esperandoN=0; esperandoS=0;

turnoN= FALSE; turnos=TRUE; pasadosN=0, pasadosS=0;

2) Sale el que estaba en el puente haciendo dentroN=0.

3) Llegan 3 coches del norte y a pesar de que no es su turno, como esperandoS=0, pasan y comienzan a atravesar el puente. Luego:

dentroN=3; dentroS=0, esperandoN=0; espeandoS=0;

turnoN=FALSE; turnoS=FALSE; pasadosN=3; pasadosS=0;

(* observe que los dos turnos valen FALSE *)

4) Viene 1 del sur poniendo esperandoS=1 y esperando en el Await. (pg179)

5) Salen del puente los 3 coches del norte poniendo dentroN=0

Ahora pueden entrar los del sur, pero antes de que lo hagan:

6) Llega uno del norte y pone esperandoN a 1 y se duerme en el Await.

Los del norte no pueden pasar porque aunque dentroS=0 no es su turno y hay esperando 1 en el sur. Los del sur no pueden pasar porque aunque dentroN=0 no es su turno y hay coches esperando en el norte.

El error proviene de la sentencia etiquetada como (1) que pone turnos a FALSE y que originó que los dos turnos fueran FALSE. Esta sentencia controlaba que entraran coches en bloques de 10. Al añadir una nueva condición en el Await, la sentencia ya no es válida.

Cuando el décimo coche del norte niega el paso a los de su mismo sentido (turnoN:= FALSE) y permite el paso a los del sur (turnos=TRUE), el undécimo del norte, que ahora puede pasar por la condición esperandoS=0, niega también el paso a los del sur

(1): turnos:= FALSE.

La solución es bien sencilla: quitar del programa la sentencia (1). Observe que ahora los coches de ambos sentidos pueden pasar intercaladamente y sólo se aplica la política de turnos cuando hay coches esperando en ambos sentidos. Así pues, el algoritmo que resuelve el problema enunciado es el (2.5):

```
PROGRAM Río; (* algoritmo (2.5): 5.ª aproximación *)
   VAR puente: SHARED RECORD
                     dentroN,                (* CN en el puente *)
                     dentroS: CARDINAL;      (* CS en el puente *)
                     pasadosN,
                     pasadosS: CARDINAL;
                     turnoN,
                     turnos: BOOLEAN;
                     esperandoN,
                     esperandoS: CARDINAL
                  END;
   PROCEDURE CN (i: INTEGER);
      BEGIN
         REGION puente DO BEGIN
            esperandoN:= esperandoN + 1;
            AWAIT ((dentroS = 0) & (turnoN OR esperandoS=O));
            esperandoN:= esperandoN -1;
            dentroN:= dentroN + 1;
            pasadosN:= (pasadosN + 1) mod 10;
            IF pasadosN = 0                                          (pg180)
            THEN BEGIN
                     turnoN:= FALSE;
                     turnoS:= TRUE
                 END
         END;
         (* cruzando el puente *)
         REGION puente DO
            dentroN:= dentroN - 1;
      END;
   PROCEDURE CS (i: INTEGER);
      BEGIN
         REGION puente DO BEGIN
            esperandoS:= esperandoS + 1;
            AWAIT ((dentroN = 0) & (turnoS OR esperandoN=O));
            esperandoS:= esperandoS - 1;
            dentroS:= dentroS + 1;
            pasadosS:= (pasadosS + 1) mod 10;
            IF pasadosS = 0
            THEN BEGIN
                     turnoS:= FALSE;
                     turnoN:= TRUE
                 END
         END;
```

```
        (* cruzando el puente *)
        REGION puente DO
            dentroS:= dentroS - 1
    END;
BEGIN
    WITH puente DO BEGIN
        dentroS:= 0; dentroN:= 0;
        pasadosS:= 0; pasadosN:= 0;
        turnoN:= TRUE; turnoS:= TRUE;
        esperandoN:= 0; esperandoS:= 0
    END;
    COBEGIN
        CN(1); CN(2); CN(3);..... ; CN(n);
        CS(1); CS(2); CS(3);..... ; CS(n)
    COEND
END.
```

Veamos a continuación cómo se pueden implementar las políticas reflejadas en los algoritmos (2.1) y (2.5) utilizando sucesos. (pg181)

En el algoritmo (2.1), los coches del norte esperaban a que dentro del puente no hubiese ninguno del sur. La variable dentroS se decrementa en la región de salida del procedimiento de los coches del sur. Por tanto, aquí se comprobará si es cero y en este caso se señalará un suceso a los coches del norte. La solución es simétrica para los coches del norte.

El algoritmo (2.6) refleja esta política:

```
PROGRAM Río; (* algoritmo (2.6) *)
    VAR puente: SHARED RECORD
                    dentroN, dentroS: CARDINAL;
                    permisoNorte, permisoSur: EVENT puente
                END;
    PROCEDURE CN (i: INTEGER);
        BEGIN
            REGION puente DO BEGIN
                IF dentroS > 0
                THEN AWAIT (permisoNorte);
                dentroN:= dentroN + 1
            END;
            (* cruzando el puente *)
            REGION puente DO BEGIN
                dentroN:= dentroN - 1;
                IF dentroN = 0
                THEN CAUSE (permisoSur)
            END
        END;
    PROCEDURE CS (i: INTEGER);
        BEGIN
            REGION puente DO BEGIN
                IF dentroN > 0
                THEN AWAIT (permisoSur);
                dentroS:= dentroS + 1
            END;
            (* cruzando el puente *)
            REGION puente DO BEGIN
                dentroS:= dentroS - 1;
                IF dentroS = 0
                THEN CAUSE (permisoNorte)
            END
```

```
        END;                                                    (pg182)
    BEGIN
        WITH puente DO BEGIN
            dentroS:= 0;
            dentroN:= 0
        END;
        COBEGIN
            CN(1); CN(2); CN(3);..... ; CN(n);
            CS(I); CS(2); CS(3);..... ; CS(n)
        COEND
    END.
```

Con la solución anterior asistimos a una falta de exclusión mutua: en el puente están a la vez coches del norte y del sur. Veamos el siguiente escenario:

1) En el puente hay 8 coches del sur. Luego: dentroN= 0; dentroS= 8

2) Cuando sale el octavo del sur ya había 3 coches del norte esperando.

3) Sale el último del sur haciendo dentroS=0 y ejecutando CAUSE(permisoNorte), lo que despierta a los del norte que estaban dormidos en el AWAIT.

4) Antes de que uno del norte pueda entrar, entra uno del sur (ya que dentroN=0) y pone dentroS=1 y empieza a atravesar el puente.

5) Los 3 del norte continúan haciendo dentroN=3 y atraviesan el puente.

Tenemos a 1 del sur y 3 del norte sobre el puente: falta de exclusión mutua. El error proviene que cuando se despertó a los del norte estos supusieron que dentroS=0 (correcto), pero entre que detectaron esta condición y entraron, la condición cambió (dentroS=1). Para evitar este problema, sustituimos los IF de entrada por WHILE, quedando el procedimiento CN como sigue (el CS es similar):

```
"CN-i"
        …..
        REGION puente DO BEGIN
            WHILE dentroS > 0 DO
                AWAIT (permisoNorte);
                dentroN:= dentroN + 1
        END;
        (* cruzando el puente *)
        REGION puente DO BEGIN
            dentroN:= dentroN - 1;
            IF dentroN = 0                                      (pg183)
            THEN CAUSE (permiso Sur)
        END;
        …..
```

Compruebe ahora que el escenario anterior que conducía a la falta de exclusión mutua no se puede producir.

La solución con sucesos al algoritmo (2.5) queda reflejada en el algoritmo (2.7)

```
PROGRAM Río; (* algoritmo (2.7) *)
    VAR puente: SHARED RECORD
                    dentroN, dentroS: CARDINAL;
                    esperandoN, esperandoS: CARDINAL;
                    turnoN, turnoS: BOOLEAN;
                    pasadosN, pasadosS: CARDINAL;
                    permisoNorte, permisoSur: EVENT puente
                END;
    PROCEDURE CN (i: INTEGER);
```

```
        BEGIN
          REGION puente DO BEGIN
            esperandoN:= esperandoN + 1;
            WHILE ((dentroS > 0) OR (turnoS & esperandoS > 0)) DO AWAIT(permisoNorte);
            esperandoN:= esperandoN - 1;
            dentroN:= dentroN + 1;
            pasadosN:= (pasadosN + 1) mod 10;
            IF pasadosN = 0
            THEN BEGIN
                    turnoN:= FALSE;
                    turnoS:= TRUE
                 END
          END;
          (* cruzando el puente *)
          REGION puente DO BEGIN
            dentroN:= dentroN - 1;
            IF ((dentroN=O) & (turnoS OR esperandoN=O))
            THEN CAUSE (permisoSur)
          END;
        END;
      PROCEDURE CS (i: INTEGER);
        BEGIN                                                    (pg184)
          REGION puente DO BEGIN
            esperandoS:= esperandoS + 1;
            WHILE ((dentroN > 0) OR (turnoN & esperandoN > 0)) DO AWAIT (permisoSur);
            esperandoS:= esperandoS - 1;
            dentroS:= dentroS + 1;
            pasadosS:= (pasadosS + 1) mod 10;
            IF pasadosS = 0
            THEN BEGIN
                    turnoS:= FALSE;
                    turnoN:= TRUE
                 END
          END;
          (* cruzando el puente *)
          REGION puente DO BEGIN
            dentroS:= dentroS - 1;
            IF ((dentroS=O) & (turnoN OR esperandoS=O))
            THEN CAUSE (permisoNorte)
          END;
        END;
      BEGIN (* Rio *)
        WITH puente DO BEGIN
          dentroN:= 0; dentroS:= 0;
          esperandoN:= 0; esperandoS:= 0;
          turnoN:= TRUE; turnoS:= TRUE;
          pasadosN:= 0; pasadosS:= 0
        END;
        COBEGIN
          CN(1); CN(2); CN(3);..... ; CN(n);
          CS(1); CS(2); CS(3);..... ; CS(n)
        COEND
      END.
```

Este último algoritmo parece correcto, pero no lo es. Si hay un coche del norte cruzando el puente y tres coches del sur esperando, el siguiente (y siguientes) coche(s) del norte esperan (lo cual ya es erróneo). Es más, cuando salga del puente el coche del norte, los coches del sur siguen sin poder entrar, porque turnoN

es TRUE y esperandoN es mayor que 0; por su lado otros coches del norte tampoco podrán entrar porque turnos es TRUE y esperandoS es mayor que 0. Luego estamos en una situación de interbloqueo.

El problema está en que hemos traducido, de la solución anterior, la expresión: (pg185)

AWAIT (dentroS=O) & (turnoN OR esperandoS=O)

por:

WHILE (dentroS > 0) OR (turnoS & esperandoS > 0) DO..

mientras que la traducción correcta es:

WHILE (dentroS > 0) OR (NOT turnoN & esperandoS > 0) DO...

Es obvio que NOT turnoN no es equivalente a turnoS, puesto que inicialmente los dos tienen el valor TRUE.

**PROBLEMA Ejemplo n.° 3**

Tenemos un sistema con un conjunto de N tareas (procesos): tarea1, tarea2, ...., tareaN. Cada tarea debe ejecutarse periódicamente cada cierto intervalo de tiempo. Por ejemplo: la tarea1 debe ejecutarse cada segundo, la tarea2 cada 10 segundos, etc. Los intervalos de tiempo para cada tarea están predefinidos y se almacenan como datos del programa (por ejemplo en un array). Durante estos intervalos las tareas están dormidas.

Se PIDE: programar un SCHEDULER (planificador) que arranque las tareas cuando les corresponda.

(NOTA: las tareas ejecutarán algún proceso trivial).

(PISTA: la temporalización se puede resolver con un reloj software -un contador que se va decrementando-).

**PROBLEMA Ejemplo n.° 3 (Solución)**

Del enunciado se desprende que vamos a tener los siguientes procesos:

a) las N tareas que estarán dormidas hasta que reciban una señal del planificador momento en el cual se pondrán a ejecutar su trabajo. Al terminarlo quedan de nuevo dormidas a espera de otra señal.

b) El planificador que estará dormido hasta que reciba una señal del reloj diciendo que ya ha pasado un segundo. En ese momento decrementa en uno el intervalo de tiempo durante el cual cada tarea debe estar dormida. Si alguno de esos intervalos se hace 0 manda una señal a la tarea correspondiente para despertarla.

c) El reloj software que contará ticks (1 segundo = 60 ticks). Al llegar a 60 manda una señal al planificador. (pg186)

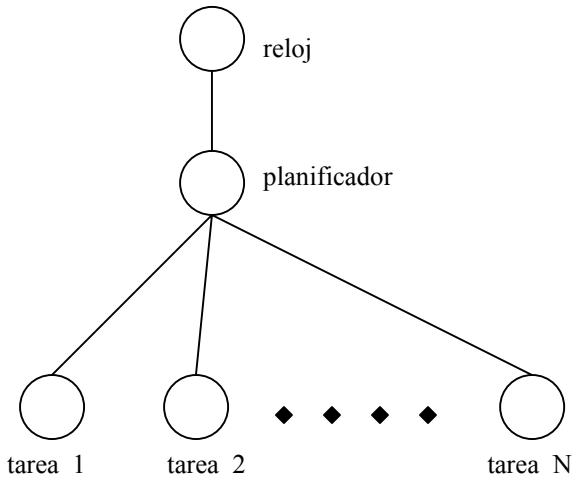



**FIGURA 41**

Estos procesos y sus relaciones se muestran en la Fig. 41.

Se supone que si una tarea hay que despertarla cada 10 segundos, el tiempo durante el cual esta tarea está ejecutándose es menor a 10 segundos. Es decir, cuando vayamos a despertarla estamos seguros de que está dormida.

Por otro lado, el intervalo de tiempo se mide de manera absoluta y no desde que la tarea ha acabado de realizar su trabajo. Es decir, supongamos que a la tarea1 hay que despertarla cada 5 segundos. Pasados éstos se la despierta y comienza a ejecutarse. Al mismo tiempo que se ejecuta se sigue contabilizando si han pasado 5 segundos desde la última vez que se la despertó.

Puesto que el planificador debe esperar a que pase un segundo, vamos a utilizar un semáforo (timer) donde esperará. Cuando haya pasado 1 segundo el reloj mandará una señal a este semáforo. Por otro lado, las tareas esperarán en un semáforo diferente cada una a que haya pasado el tiempo para su activación. Definiremos un array de semáforos (comienzo) para este objetivo. La información sobre el intervalo de activación para cada tarea lo mantendremos en un array llamado tiempos. Para cada tarea figurará su tiempo de espera y una variable auxiliar que el planificador decrementará cada segundo.

El programa pedido se muestra a continuación: (pg187)

```
PROGRAM Planifica;
   CONST n = 30; (* numero de tareas *)
   TYPE tareas = [l..n];
        unaTarea = RECORD
                        (* segundos que faltan para activar tarea *)
                        faltan,
                        (* intervalo de tiempo entre activaciones *)
                        intervalo: INTEGER
                   END;
   VAR  tiempos: ARRAY tareas OF unaTarea;
        comienzo: ARRAY tareas OF SEMAPHORE;
        timer: SEMAPHORE;
        i: INTEGER;
   PROCEDURE reloj;
      CONST unSegundo = 60; (* ticks *)
      VAR i:INTEGER;
      BEGIN
         REPEAT
            i:= unSegundo;
            WHILE i > 0 DO i:= i - 1;
            SIGNAL (timer); (* indica que ha pasado 1 segundo *)
         FOREVER
      END;
   PROCEDURE planificador;
      VAR i: INTEGER;
      BEGIN
         REPEAT
            WAIT (timer); (* espera que pase 1 segundo *)
            FOR i:= 1 TO n DO
               WITH tiempos [i] DO BEGIN
                  faltan:= faltan - 1;
                  IF faltan = 0 (* hay que activar la tarea i *)
                  THEN BEGIN
                           faltan:= intervalo; (* restaurar faltan *)
                           SIGNAL (comienzo[¡])
                       END
               END
         FOREVER
      END;
   PROCEDURE tarea (i: INTEGER);                                    (pg188)
      BEGIN
         REPEAT
            WAIT (comienzo[i]);
            ejecuta el proceso-i
         FOREVER
```

```
    END;
BEGIN (* planifica *)
    INIT (timer, 0);
    «rellena adecuadamente la variable tiempos con los valores para cada tarea»
    FOR i:= 1 TO n DO INIT (comienzo[i], 0);
    COBEGIN
        reloj;
        planificador;
        tarea(1); tarea(2); ...; tarea(n)
    COEND
END.
```

**PROBLEMA Ejemplo n.° 4**

En un tablero de ajedrez queremos colocar 8 damas sin que se amenacen mutuamente.

Se pide programar una solución recursiva y concurrente al problema anterior tal que obtenga todas las combinaciones posibles que cumplan el enunciado.

**PROBLEMA Ejemplo n.° 4 (Solución)**

La solución recursiva nos es a todos conocida. La idea para añadir una búsqueda concurrente es la siguiente: se crea un proceso para cada posible lugar que pueda ocupar una dama en el tablero. Obviamente muchos de estos procesos morirán al comprobar que la situación de las damas es incorrecta.

Inicialmente el tablero está vacío. Creamos 8 procesos cada uno de los cuales tiene un tablero (para buscar una solución) con una dama colocada en la fila primera, columna i-esima. Así pues, el proceso 1 tiene una dama en la fila 1, columna 1; el proceso 2 tiene una dama en la fila 1, columna 2; el proceso 8 tiene una dama en la fila 1, columna 8. Ahora cada uno de esos ocho procesos crea a otros 8 tal que estos últimos asumen la posición de las damas en el tablero de su padre y añaden una segunda dama en la fila 2 columna i. Los procesos hijos comprueban si las damas se amenazan y si es así mueren (se abandona la búsqueda puesto que la situación actual es incorrecta). En (pg189)

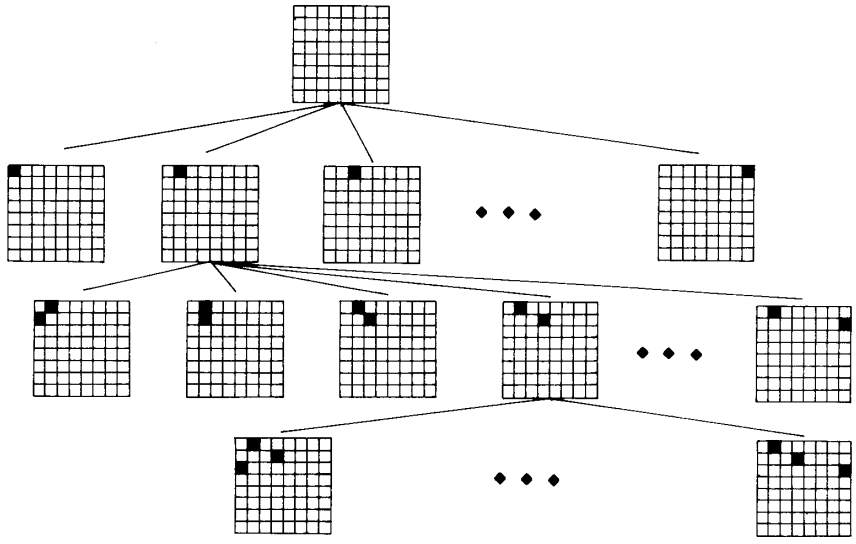


FIGURA 42

caso de que las damas no se amenacen se sigue el proceso hasta que estén puestas las 8, momento en el cual se imprime la solución obtenida (ver Fig.42).

El programa que sigue este protocolo se esboza a continuación:

```
PROGRAM OchoReinas;
    TYPE nmrColumna = 0..8;
         tab = RECORD
```

```
                nmrDama: 0..8;
                fila: ARRAY [1..8] OF nmrColumna
            END;
    VAR i: INTEGER;
        tabInit: tab;
    PROCEDURE ponOchoDamas (tablero: tab);
        VAR t: ARRAY [1..8] OF tab;
            i: INTEGER;
        BEGIN
            IF seComen (tablero)
            THEN «muérete»                                    (pg190)
            ELSE IF tablero.nmrDama = 8
                    THEN escribeSolución (tablero)
                    ELSE BEGIN
                            tablero.nmrDama:= tablero.nmrDama + 1;
                            FOR i:=1 TO 8 DO BEGIN
                                t[i]:= tablero;
                                t[i].fila[tablero.nmrDama]:= i
                            END;
                            COBEGIN
                                ponOchoDamas (t[1]);
                                ponOchoDamas (t[2]);
                                ponOchoDamas (t[3]);
                                ponOchoDamas (t[4]);
                                ponOchoDamas (t[5]);
                                ponOchoDamas (t[6]);
                                ponOchoDamas (t[7]);
                                ponOchoDamas (t[8]);
                            COEND
                            END
        END; (* ponOchoDamas *)
    BEGIN (* main *)
        WITH tabInit DO BEGIN
            nmrDama:= 0;
            FOR i:=1 TO 8 DO fila[i]:= 0
        END;
        ponOchoDamas (tabInit)
    END.
```

Observe que en este algoritmo un tablero define también cuantas damas hay sobre él. La parte central del algoritmo consiste en la copia de 8 tableros y la colocación de la siguiente dama. A continuación se lanzan en concurrencia a ocho procesos, cada uno con un tablero diferente.

**PROBLEMA Ejemplo n.º 5**

Un recurso único está compartido por dos tipos de procesos llamados lectores y escritores. Los lectores pueden utilizar el recurso simultáneamente, pero cada escritor deber tener acceso exclusivo a el. Los dos tipos de procesos se excluyen mutuamente en el acceso al recurso y en caso de conflicto los escritores tienen prioridad sobre los lectores. (pg191)

**PROBLEMA Ejemplo n.º 5 (Solución)**

Vamos a considerar que el ciclo de vida de ambos tipos de procesos es el siguiente:

pide recurso

usa recurso

libera recurso

Desde que un proceso pide el recurso hasta que lo libera diremos que está activo. Cuando está usando el recurso además diremos que está trabajando. En adelante seguiremos la siguiente notación:

la: lectores activos lt: lectores trabajando

ea: escritores activos et: escritores trabajando

En una primera aproximación no tendremos en cuenta que los escritores se excluyan mutuamente cuando acceden al recurso.

Los criterios de corrección que debe cumplir la solución al problema son:

1) puesta en cola de los procesos a la hora de utilizar el recurso; es decir: ( $0 \leq lt \leq la$ ) & ( $0 \leq et \leq ea$ )

2) los lectores y escritores se excluyen mutuamente en el acceso al recurso; es decir:NOT ($lt > 0$ & $et > 0$)

3) no interbloqueo de procesos activos; es decir: si ($et=0$ & $lt=0$) & ($la > 0$ OR $ea > 0$) entonces, en un tiempo finito ($lt > 0$ OR $et > 0$)

4) los escritores deben tener prioridad sobre los lectores; es decir:

conceder el recurso a un lector activo si: $ea = 0 \rightarrow (et = 0)$

conceder el recurso a un escritor activo si: $lt=0 \rightarrow (la \geq 0)$

En una primera aproximación vamos a utilizar semáforos para sincronizar el acceso al recurso. El algoritmo que implementa la solución es el (5.1).

```
PROGRAM lectoresYEscritores; (* algoritmo (5.1): 1.ª aprox. *)
   VAR estado: SHARED RECORD
                    la, lt
                    ea, et: INTEGER
               END;
       leyendo, escribiendo: SEMAPHORE;
   PROCEDURE permisoLectura;
      BEGIN
         IF ea = 0                                          (pg192)
         THEN WHILE lt < la DO BEGIN
                    lt:= lt + 1;
                    SIGNAL (leyendo)
               END
      END;
   PROCEDURE permisoEscritura;
      BEGIN
         IF lt =0
         THEN WHILE et < ea DO BEGIN
                    et:= et + 1;
                    SIGNAL (escribiendo)
               END
      END;
   PROCEDURE lector (i: INTEGER);
      BEGIN
         REGION estado DO BEGIN
            la:= la + 1;
            permisoLectura
         END;
         WAIT (leyendo);
         (*** accede al recurso y lee ***)
         REGION estado DO BEGIN
            la:= la - 1;
            lt:= lt - 1;
            permisoEscritura
         END
```

```
    END;
PROCEDURE escritor (i: INTEGER);
    BEGIN
        REGION estado DO BEGIN
            ea:= ea + 1;
            permisoEscritura
        END;
        WAIT (escribiendo);
        (*** accede al recurso y escribe ***)
        REGION estado DO BEGIN
            ea:= ea - l;
            et:= et - 1;
            permisoLectura
        END
    END;                                                    (pg193)
BEGIN (* lectoresYEscritores *)
    INIT (leyendo, 0); INIT (escribiendo, 0);
    WITH estado DO BEGIN
        la:= 0; lt:= 0;
        ea:= 0; et:= 0
    END;
    COBEGIN
        lector (1); lector (2); ...; lector (n);
        escritor (1); escritor (2); ...; escritor (n)
    COEND
END.
```

A continuación vamos a verificar que el algoritmo anterior satisface los criterios de corrección enunciados. Lo haremos utilizando **asertos**.

En el algoritmo (5.1) vemos que «la» y «ea» se incrementan en 1 por cada petición del recurso y se decrementan en 1 por cada liberación del mismo. Por tanto indican los lectores y escritores que están activos según definimos.

Por otro lado «lt» es el número de operaciones signal sobre el semáforo leyendo menos el número de liberaciones del recurso. Por tanto refleja el número de procesos que han pasado (o pueden pasar) el wait (leyendo) menos el número de liberaciones del recurso. Puesto que el invariante de los semáforos nos asegura que el número de lectores que han pasado o pueden pasar el wait (leyendo) es igual al número de signal (leyendo), deducimos que es el número de lectores trabajando. Lo mismo se aplican para «et».

Probaremos a continuación que se cumplen los criterios (1) y (2).

(1): $(0 \leq lt \leq la)$ & $(0 \leq et \leq ea)$

(2): NOT $(lt > 0$ & $et > 0)$

Inicialmente es cierto (1) y (2) ya que la=0; ea=O; lt=0; et=0. Después de que un lector haga una petición = > la:= la + 1 y sigue cumpliéndose que $0 \leq lt < la$. Por tanto:

```
«lector»
        REGION estado DO BEGIN
            (* (1) & (2) *)
            la:= la + 1
            (* (1) & (2) *)
            permisoLectura
            (* ? *)
        END;                                                (pg194)
```

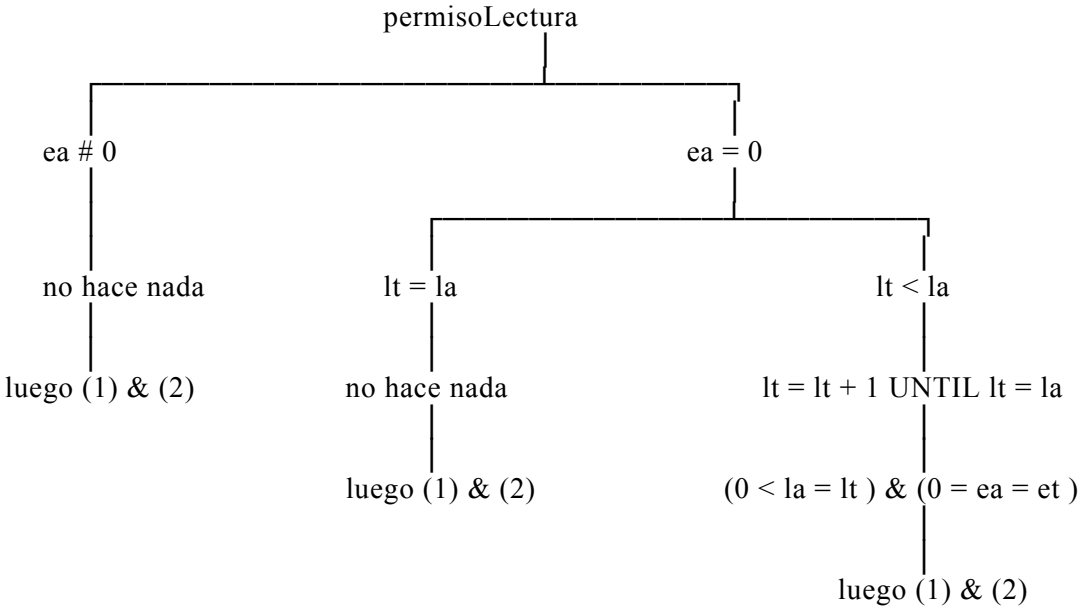



FIGURA 43

Cuando se intente conceder la lectura tenemos la situación mostrada en la Fig. 43.

Luego «?» implica que se cumplen (1) y (2). Cuando el lector libere el recurso tenemos que:

```
«lector»
      REGION estado DO BEGIN
         (* (1) & (2) & lt > 0 *)
         la:= la - 1;
         lt:= lt - 1;
         (* ?? *)
         permisoEscritura;
         (* ??? *)
      END                                                    (pg195)
```

(1) & (2) & lt > 0 => (0 < lt ≤ la) & (0 = et ≤ ea)

Luego «??» implica que se cumplen (1) y (2).

Cuando se intenta conceder la escritura tenemos la situación mostrada en la Fig. 44.

La corrección del criterio (3) se esboza a continuación

(3) (lt = 0 & et = 0) & (la > 0 OR ea > 0) => en un tiempo finito => (lt > 0 OR et > 0)

Es fácil demostrar que:

D ≡ (lt = 0 => et = ea) & (ea = 0 => lt= la)

Si el recurso está desocupado se cumple que:

I ≡ (lt = 0 & et = 0)

De ello:

I & D => ea = et = 0

I & D & (ea = 0) => la = lt = 0

En una situación en la que nadie utiliza el recurso (lt = 0 y et = 0) es porque nadie lo quiere utilizar (la = 0 y ea = 0).

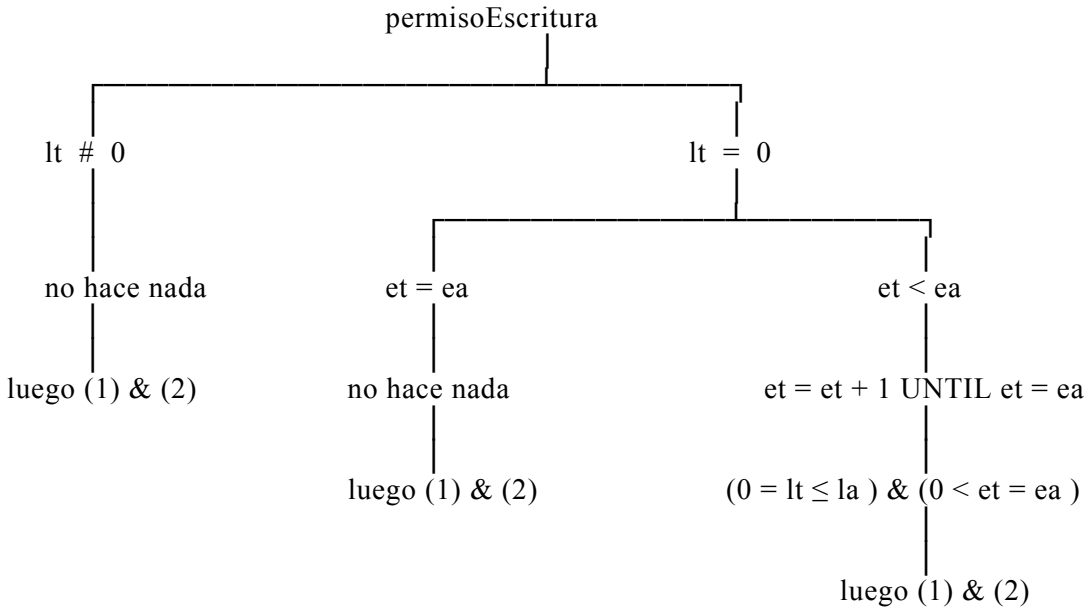



FIGURA 44 (pg196)

En cuanto al criterio (4) observe que sólo se concede el recurso a un lector activo si ea=0. Pero si ea ≥ 0 y la ≥ 0 el recurso se le concede al escritor, ya que lo único que lo restringe es que lt > 0, mientras que a los lectores se les niega el recurso con sólo que ea > 0.

A continuación vamos a intentar resolver el problema utilizando RCC. Los lectores se esperarán a que ea=0 y los escritores a que lt=0. El programa que implementa esta solución es el algoritmo (5.2).

```
PROGRAM lectoresYEscritores; (* algoritmo (5.2) *)
   VAR estado: SHARED RECORD
                     lt, ea: INTEGER
               END;
   PROCEDURE lector (i: INTEGER); .
      BEGIN
         REGION estado DO BEGIN
            AWAIT ea = 0;
            lt:= lt + 1
         END;
         (*** usa el recurso ***)
         REGION estado DO
            lt := lt - 1
      END;
   PROCEDURE escritor (i: INTEGER);
      BEGIN
         REGION estado DO BEGIN
            ea:= ea + 1;
            AWAIT lt = 0
         END;
         (*** usa recurso ***)
         REGION estado DO
            ea:= ea - 1
      END;
   BEGIN   (* lectoresYEscritores *)
      WITH estado DO BEGIN
         lt:= 0; ea:= 0
      END;
```

```
    COBEGIN
        lector (1); lector (2); ...; lector (n);
        escritor (1); escritor (2); ...; escritor (n)
    COEND
END.
```

(pg197)

Lo único que nos falta es conseguir que los escritores se excluyan mutuamente en el acceso al recurso. Para ello basta declarar una variable compartida (escritura) y poner una RC cuando un escritor use el recurso:

REGION escritura DO (*** usa el recurso ***)

Como puede comprobar la exclusión entre escritores no afecta para nada a los lectores.

**PROBLEMA Ejemplo n.° 6**

Tenemos tres máquinas: un ordenador personal y dos placas controladoras con sus respectivos «timer». El ordenador personal está conectado con cada placa controladora mediante una línea de transmisión bidireccional.

Las líneas de transmisión se asimilan a dos buzones (uno para cada dirección), y cada «timer» se asimila a un proceso con un ciclo y un semáforo sobre el que hace «signal». (Suponemos que de esta asimilación se encarga el compilador y un código ya escrito.)

Se pide escribir un programa (en pascal con primitivas de concurrencia: cobegin..coend, semáforos, buzones, región crítica, ...) que se comporte de acuerdo con las especificaciones que vienen a continuación.

**a)** Cada placa controladora tiene una salida:

calefacción: (encendida, apagada)

y dos entradas:

temperatura habitación: (-20 .. 60)

ventana: (abierta, cerrada).

Inicialmente cada placa tiene una «temperatura objetivo». Cada minuto, la placa debe leer la temperatura de la habitación. Si la temperatura de la habitación es menor en dos o más grados que la «temperatura objetivo» y la ventana está cerrada pondrá la calefacción a encendido. Si la temperatura de la habitación es mayor o igual que la temperatura objetivo pondrá la calefacción a apagado.

**b)** (Además ..)

Cada hora, cada placa debe enviar al ordenador personal las temperaturas mínima y máxima de esa hora. El ordenador personal debe presentar (cada hora) en pantalla las temperaturas mínima y máxima leídas en cada placa en las dos últimas horas.

**c)** (Además ..)

El ordenador personal leerá eventualmente un comando del teclado para modificar la temperatura objetivo de una placa. Debe transmitir la nueva temperatura objetivo a la placa correspondiente, que la tomará como nueva referencia.

(pg198)

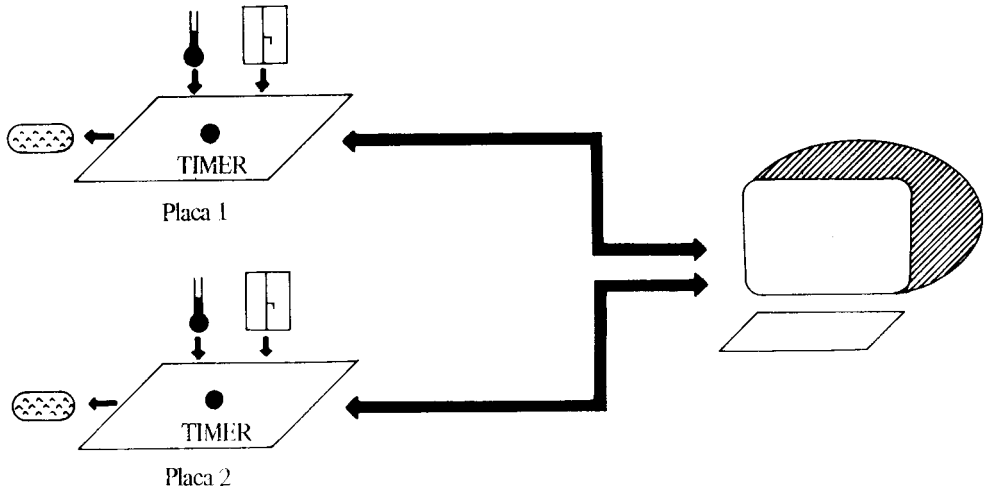



FIGURA 45

**PROBLEMA Ejemplo n.° 6 (Solución)**

El problema lo podemos representar como se muestra en la Fig. 45.

**a)** Supondremos que disponemos de los adecuados convertidores analógico/digitales para interactuar con el sistema calefactor y para tomar medida de la temperatura y del estado de la ventana. La comunicación de una placa con estos dispositivos se realizará a través de ciertos puertos de E/S de los que dispone la placa. Por ello, bastarán operaciones de lectura y escritura sobre determinadas direcciones de memoria para controlar los citados dispositivos.

Con este razonamiento NO necesitamos crear un proceso que controle cada dispositivo (temperatura, ventana, calefacción). Bastará, por tanto, un proceso principal en cada placa que actuará sincronizado con el «timer».

Así pues, los procesos necesarios para implementar esta primera parte y sus relaciones se muestran gráficamente en la Fig. 46:

El diagrama mostrado en la Fig. 46 se refiere a una sola placa. Idéntico esquema existirá en la otra placa.

El proceso «timer» se limitará a indicar el paso del tiempo (un minuto en este caso). Su implementación es la siguiente:

```
PROCEDURE timer;
   CONST unSegundo = 60;  (* ticks *)
   VAR i: INTEGER;
   BEGIN
      REPEAT                    (pg199)
         i:= unSegundo;
         WHILE i > 0 DO i:= i - 1;
         SIGNAL haPasadoUnSegundo)
      FOREVER
   END;
```

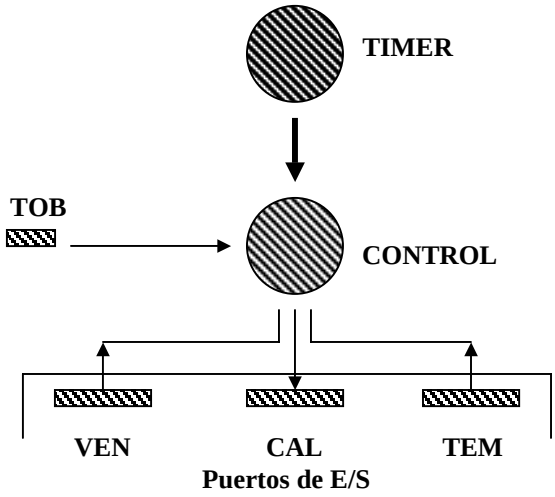



**FIGURA 46**

El proceso «control» se ejecutará cada segundo y comprobará la temperatura de la habitación y el estado de la ventana, actuando en consecuencia sobre el sistema calefactor. Para interactuar con estos dispositivos suponemos definidos los siguientes procedimientos:

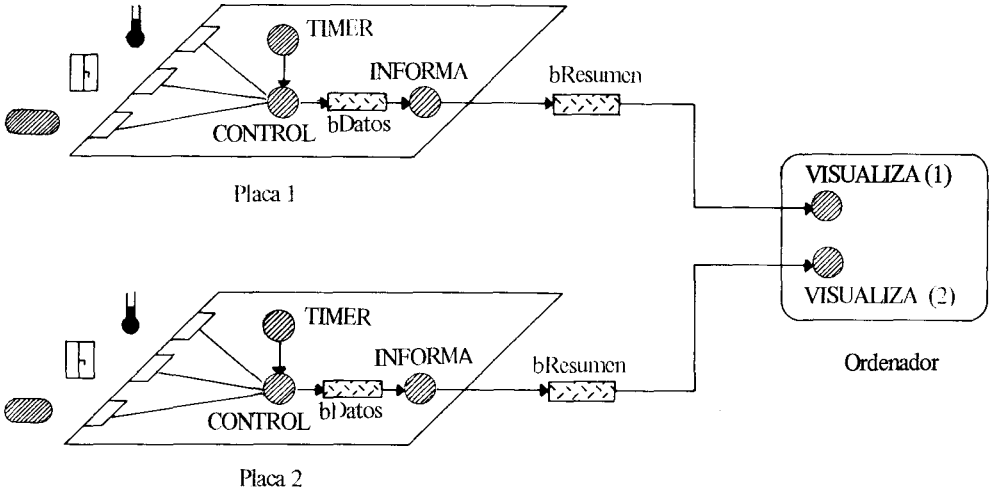
PROCEDURE leeTemperatura (VAR temperaturaActual: unaTemperatura);
PROCEDURE estadoDeLaVentana (VAR estado: unEstadoDeVentana);
PROCEDURE calefacción (acción: unaAcción);

siendo:

TYPE unaTemperatura = -20..60;
unEstadoDeVentana = (abierta, cerrada);
unaAcción = (encender, apagar);

Con esto, el programa que implementa estas primeras especificaciones es el siguiente:

```
PROGRAM controlTemperatura;
  TYPE unaTemperatura = -20..60;
       unaPlaca = 1..2;                                    (pg200)
  PROCEDURE placa (i: unaPlaca);
    TYPE unEstadoDeVentana = (abierta, cerrada);
         unaAcción = (encender, apagar);
    VAR temperaturaObjetivo: unaTemperatura;
        haPasadoUnSegundo: SEMAPHORE;
    PROCEDURE timer;
      CONST unSegundo = 60;  (* ticks *)
      VAR i: INTEGER;
      BEGIN
        REPEAT
          i:= unSegundo;
          WHILE i > 0 DO i:= i - 1;
          SIGNAL (haPasadoUnSegundo)
        FOREVER
      END; (* timer *)
    PROCEDURE control;
      VAR temperaturaHabitacion: unaTemperatura;
          ventana: unEstadoDeVentana;
      BEGIN
        REPEAT
          WAIT (haPasadoUnSegundo);
          LeeTemperatura (temperaturaHabitacion);
          estadoDeLaVentana (ventana);
          IF (temperaturaObjetivo - temperaturaHabitacion ≥ 2) & (ventana = cerrada)
          THEN calefaccion (encender)
          ELSE IF temperaturaHabitacion ≥ temperaturaObjetivo
                 THEN calefaccion (apagar)
        FOREVER
      END; (* control *)
    BEGIN (* placa *)
      temperaturaObjetivo:= 25; (* por ejemplo *)
      INIT (haPasadoUnSegundo, 0);
      COBEGIN
        timer;
        control;
      COEND
    END;(*placa*)                                          (pg201)
```




```
BEGIN (* controlTemperatura *)
   COBEGIN
      placa (1);
      placa (2)
   COEND
END. (* controlTemperatura *)
```

**b)** Estas nuevas especificaciones implican la creación de nuevos procesos:

- En cada placa hay que crear un proceso «informa» cuya función será contrastar las 60 mediciones de temperatura realizadas por el proceso «control» y mandar al ordenador personal las temperaturas máxima y mínima de esa hora.

- En el ordenador personal hay que crear dos procesos (idénticos) que reciban la información de los procesos «informa» y cada hora visualicen en pantalla las temperaturas máxima y mínima de cada habitación en las dos últimas horas.

La comunicación entre procesos la implementamos con buzones. El nuevo diagrama de conexión entre procesos es el mostrado en la Fig. 47.

Así pues, definimos para cada placa el buzón «bDatos» como:

```
VAR bDatos: BUFFER 1 OF unaTemperatura;
```

El proceso «control» mandará a este buzón la temperatura leída cada minuto. La decisión de dotar a este buzón con una capacidad de 1 mensaje se debe a que el intervalo de tiempo entre dos SEND consecutivos (1 minuto) (pg202) es lo bastante amplio como para que el proceso «informa» tenga tiempo de recoger el mensaje. De hecho, el buzón «bDatos» casi siempre estará vacío. En este mismo sentido cabe comentar que en el intervalo de 1 minuto, el proceso «control» gastará la mayor parte de ese tiempo bloqueado en el semáforo «haPasadoUnSegundo», de la misma forma que el proceso «informa» estará casi siempre bloqueado esperando un mensaje del proceso «control». Con esto queremos decir que el tiempo empleado por los procesos «control» e «informa» en ejecutar su código es muy pequeño en comparación con 1 minuto. Si existiese la posibilidad de que el proceso «informa» no pudiese recoger regularmente (cada minuto) el mensaje enviado por el proceso «control», el buzón «bDatos» debería tener una capacidad mayor que 1. Ello sería necesario para evitar que el proceso «control» se quedase bloqueado al hacer un SEND y no pudiera recoger la medida de temperatura en el siguiente minuto. Para determinar qué tamaño debería tener el buzón «bDatos» deberíamos hacer estimaciones de tiempos de ejecución sobre el proceso «informa» y el subsistema del que forma parte. Entramos con esto en el área del Tiempo Real.

En el problema que nos ocupa, el código que debe ejecutar el proceso «informa» se ejecutará lo bastante rápido como para hacer válida la elección de dar capacidad 1 al buzón «bDatos». Sólo si «informa» se quedase bloqueado al hacer SEND en el buzón «bResumen» podríamos tener problemas. Veremos a continuación que esto no es posible.

Al buzón «bResumen» se mandará, cada hora, un mensaje con las temperaturas máxima y mínima producidas durante ese intervalo. Por otra parte, el proceso «visualiza» (en el ordenador) estará siempre esperando este mensaje, salvo cuando haya pasado 1 hora, en cuyo caso tiene que escribir unas cuantas

líneas en pantalla. El tiempo para escribir estas líneas es mucho menor que 1 hora, razón por la cual el proceso «informa» nunca encontrará lleno el buzón «bResumen» y por ello no podría bloquearse debido a esta circunstancia.

El anterior razonamiento también justifica dar capacidad de 1 mensaje al buzón «bResumen», cuya declaración es:

```
TYPE unResumen = RECORD
                    temperaturaMaxima, temperaturaMinima: unaTemperatura
                 END;
VAR bResumen: BUFFER 1 OF unResumen;
```

Así pues, el programa que resuelve este apartado es el siguiente: (pg203)

```
PROGRAM controlTemperatura;
  TYPE una Temperatura = -20..60;
       unaPlaca = 1..2;
       unResumen = RECORD
                      temperaturaMaxima, temperaturaMinima: unaTemperatura
                   END;
       bResumen = BUFFER 1 OF unResumen;
  VAR resumenHora: ARRAY unaPlaca OF bResumen;
  PROCEDURE placa (i: unaPlaca);
    TYPE unEstadoDeVentana = (abierta, cerrada);
         unaAccion = (encender, apagar);
    VAR temperaturaObjetivo: unaTemperatura;
        haPasadoUnSegundo: SEMAPHORE;
        bDatos: BUFFER 1 OF unaTemperatura;
    PROCEDURE timer;
      CONST unSegundo = 60;  (* ticks *)
      VAR i: INTEGER;
      BEGIN
        REPEAT
          i:= unSegundo;
          WHILE i > 0 DO i:= i - l;
          SIGNAL (haPasadoUnSegundo)
        FOREVER
      END; (* timer *)
    PROCEDURE control;
      VAR temperaturaHabitacion: unaTemperatura;
          ventana: unEstadoDeVentana;
      BEGIN
        REPEAT
          WAIT (haPasadoUnSegundo);
          leeTemperatura (temperaturaHabitacion);
          estadoDeLaVentana (ventana);
          IF (temperaturaObjetivo-temperaturaHabitacion ≥ 2) & (ventana = cerrada)
          THEN calefaccion (encender)
          ELSE IF temperaturaHabitacion > temperaturaObjetivo
                  THEN calefaccion (apagar);
          SEND (temperaturaHabitacion, bDatos)
        FOREVER
      END; (* control *)                                    (pg204)
    PROCEDURE informa;
      VAR t: unaTemperatura;
          registro: unResumen;
          i: INTEGER;
      BEGIN
        REPEAT
          RECEIVE (t, bDatos);
```

```
                WITH registro DO BEGIN
                   temperaturaMaxima:= t;
                   temperaturaMinima:= t;
                   FOR i:= 1 TO 59 DO BEGIN
                      RECEIVE (t, bDatos);
                      IF t > temperaturaMaxima
                      THEN temperaturaMaxima:= t
                      ELSE IF t < temperaturaMinima
                            THEN temperaturaMinima:= t
                   END (* FOR *)
                END; (* WITH *)
                SEND (registro, resumenHora [i])
             FOREVER
          END (* informa *)
       BEGIN (* placa *)
          temperaturaObjetivo:= 25; (* por ejemplo *)
          INIT (haPasadoUnSegundo, 0);
          COBEGIN
             timer;
             control;
             informa
          COEND
       END; (* placa *)
    PROCEDURE ordenador;
       VAR pantalla: SHARED BOOLEAN;
       PROCEDURE visualiza (i: unaPlaca);
          VAR tUltimas, tPenultimas: unResumen;
          BEGIN
             RECEIVE (tUltimas, resumenHora [i]);
             REPEAT
                tPenultimas:= tUltimas;
                RECEIVE (tUltimas, resumenHora [i]);
                aConsola (i, tPenultimas, tUltimas)
             FOREVER
          END; (* visualiza *)                                    (pg205)
       PROCEDURE aConsola (i: unaPlaca; penul, ultim: unResumen);
          BEGIN
             REGION pantalla DO BEGIN
                Writeln;
                Writeln ("De placa ", i);
                Write (" Penultima Hora: ");
                Write ("maxima: ", penul.temperaturaMaxima);
                Writeln (" minima: ", penul.temperaturaMinima);
                Write (" Ultima Hora: ");
                Write ("maxima: ", ultim.temperaturaMaxima);
                Writeln (" minima: ", ultim.temperaturaMinima);
                Writeln
             END
          END; (* aConsola *)
       BEGIN (* ordenador *)
          COBEGIN
             visualiza (1);
             visualiza (2);
          COEND
       END; (* ordenador *)
    BEGIN (* controlTemperatura *)
       COBEGIN
          placa (1);
```

```
        placa (2);
        ordenador
    COEND
END. (* controlTemperatura *)
```

**c)** Para satisfacer los nuevos requerimientos debemos añadir a cada placa un proceso que reciba la nueva temperatura objetivo enviada por el ordenador. Por su parte, en el ordenador hay que añadir un proceso que se en cargue de recoger del teclado la orden de cambio de temperatura. La comunicación entre este último proceso y los dos procesos receptores de las placas lo implementaremos con dos buzones: «bRecepciónOrden» cuya definición es:

```
VAR bRecepcionOrden: ARRAY unaPlaca OF bOrden;
```

siendo:

```
TYPE bOrden = BUFFER 10 OF unaTemperatura;
```

(pg206)

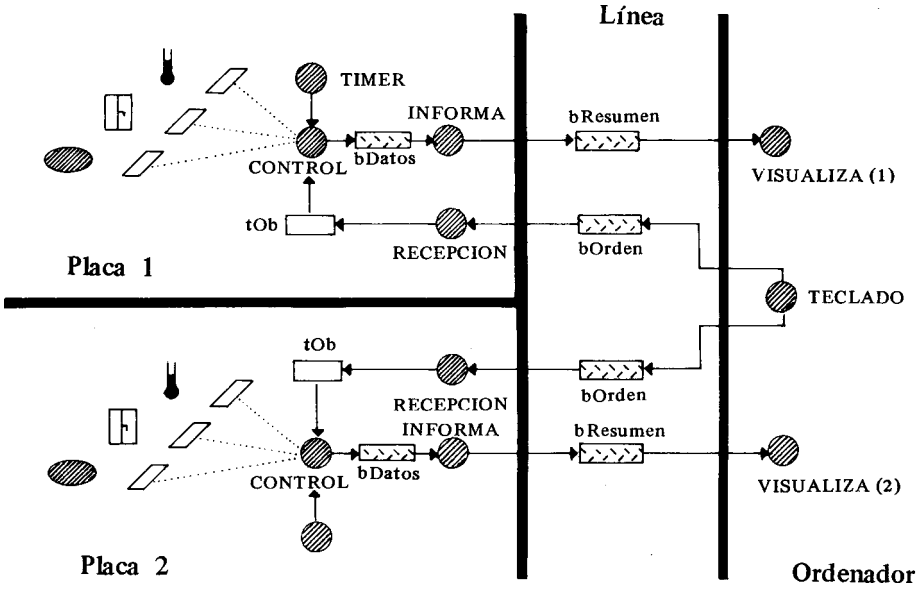



FIGURA 48

Con esto, el nuevo diagrama de conexión entre procesos queda tal y como se expresa en la Fig. 48.

Como puede observarse en esta figura, la variable «temperaturaObjetivo» ahora es accedida también por el proceso «recepción», luego debemos declararla como compartida:

```
VAR temperaturaObjetivo: SHARED unaTemperatura;
```

Por otro lado, la capacidad de «borden» se ha puesto de 10 mensajes porque no sabemos a qué velocidad pueden generarse órdenes de cambio de la temperatura objetivo desde el teclado. Si bien es cierto que el proceso «recepción» se limitará a recoger el mensaje del buzón y cambiar la variable «temperaturaObjetivo», también es cierto que para realizar esta última operación debe entrar a una región crítica y ésta puede estar ocupada. Para evitar bloquear (en la medida de lo posible) al proceso «teclado» es necesario disponer de un buzón de tamaño mayor que 1.

Así pues, el programa que resuelve este apartado es el siguiente:

```
PROGRAM controlTemperatura;
    TYPE unaTemperatura = -20..60;
         unaPlaca = 1..2;
         unResumen = RECORD
                        temperaturaMaxima, temperaturaMinima: unaTemperatura
                     END;
         bResumen = BUFFER 1 OF unResumen;
         bOrden = BUFFER 10 OF unaTemperatura;
    VAR resumenHora: ARRAY unaPlaca OF bResumen;
         bRecepcionOrden: ARRAY unaPlaca OF bOrden;
```

(pg207)

```
PROCEDURE placa (i: unaPlaca);
  TYPE unEstadoDeVentana = (abierta, cerrada);
       unaAccion = (encender, apagar);
  VAR temperaturaObjetivo = SHARED unaTemperatura;
       haPasadoUnSegundo: SEMAPHORE;
       bDatos: BUFFER 1 OF unaTemperatura;
  PROCEDURE timer;
    CONST unSegundo = 60;    (* ticks *)
    VAR i: INTEGER;
    BEGIN
      REPEAT
        i:= unSegundo;
        WHILE i > 0 DO i:= i - 1;
        SIGNAL (haPasadoUnSegundo)
      FOREVER
    END; (* timer *)
  PROCEDURE control;
    VAR temperaturaHabitacion, tOb: unaTemperatura;
        ventana: unEstadoDeVentana;
    BEGIN
      REPEAT
        WAIT (haPasadoUnSegundo);
        leeTemperatura (temperaturaHabitacion);
        estadoDeLaVentana (ventana);
        REGION temperaturaObjetivo DO tOb:= temperaturaObjetivo;
        IF (tOb - temperaturaHabitacion ≥ 2) & (ventana = cerrada)
        THEN calefaccion (encender)
        ELSE IF temperaturaHabitacion ≥ tOb
             THEN calefaccion (apagar);
        SEND (temperaturaHabitacion, bDatos)
      FOREVER
    END; (* control *)
  PROCEDURE informa;
    VAR t: unaTemperatura;
        registro: unResumen;
        i: INTEGER;                                    (pg208)
    BEGIN
      REPEAT
        RECEIVE (t, bDatos);
        WITH registro DO BEGIN
          temperaturaMaxima:= t;
          temperaturaMinima:= t;
          FOR i:= I TO 59 DO BEGIN
            RECEIVE (t, bDatos);
            IF t > temperaturaMaxima
            THEN temperaturaMaxima:= t
            ELSE IF t < temperaturaMinima
                 THEN temperaturaMinima:= t
          END (* FOR *)
        END; (* WITH *)
        SEND (registro, resumenHora [i])
      FOREVER
    END (* informa *)
  PROCEDURE recepcion;
    VAR t: unaTemperatura;
    BEGIN
      REPEAT
        RECEIVE (t, bRecepcionOrden [i]);
```

```
                REGION temperaturaObjetivo DO temperaturaObjetivo:= t
            FOREVER
        END; (* recepcion *)
    BEGIN (* placa *)
        temperaturaObjetivo:= 25; (* por ejemplo *)
        INIT (haPasadoUnSegundo, 0);
        COBEGIN
            timer;
            control;
            informa;
            recepcion
        COEND
    END; (* placa *)
PROCEDURE ordenador;
    VAR pantalla: SHARED BOOLEAN;
    PROCEDURE visualiza (i: unaPlaca);
        VAR tUltimas, tPenultimas: unResumen;
        BEGIN
            RECEIVE (tUltimas, resumenHora [i]);
                REPEAT                                          (pg209)
                    tPenultimas:= tUltimas;
                    RECEIVE (tUltimas, resumenHora [i]);
                    aConsola (i, tPenultimas, tUltimas)
                FOREVER
        END; (* visualiza *)
    PROCEDURE aConsola (i: unaPlaca; penul, ultim: unResumen);
        BEGIN
            REGION pantalla DO BEGIN
                Writeln;
                Writeln ("De placa ", i);
                Write ("Penultima Hora: ");
                Write ("maxima: ", penul.temperaturaMaxima);
                Writeln (" minima: ", penul.temperaturaMinima);
                Write (" Ultima Hora: ");
                Write ("maxima: ", ultim.temperaturaMaxima);
                Writeln (" minima: ", ultim.temperaturaMinima);
                Writeln
            END
        END; (* aConsola *)
    PROCEDURE teclado;
        VAR quePlaca: unaPlaca;
                t: unaTemperatura;
        BEGIN
            REPEAT
                Read (quePlaca, t);
                SEND (t, bRecepcionOrden [quePlaca])
            FOREVER
        END; (* teclado *)
    BEGIN (* ordenador *)
        COBEGIN
            visualiza (1);
            visualiza (2);
            teclado
        COEND
    END; (* ordenador *)
BEGIN (* controlTemperatura *)
    COBEGIN
        placa (1);
```

```
            placa (2);
            ordenador
        COEND
    END. (* controlTemperatura *)
```

# *Capítulo 6*

# *Enunciados de problemas*

**PROBLEMA N.º 1 [1]**

Realizar la siguiente operación:

a = fl(f2(f3(x,y),f4(z,w)),f3(x,y),f5(f3(x,y),f6(r,t)))

calculando en paralelo las funciones según se pueda y convenga (se supone que el cálculo de cada una de las funciones consume mucho tiempo).

**PROBLEMA N.º 2 [4]**

Un ascensor en el que caben cuatro personas atiende las llamadas que se le hacen desde varios pisos. En estos pisos llegan personas que quieren subir o bajar a otros pisos. Programar la situación anterior.

(PISTA: habrá un proceso ascensor y procesos persona).

**PROBLEMA N.º 3 [1]**

Tenemos un programa que se va a ejecutar en un sistema de multiprogramación. El ciclo de vida del programa junto con los recursos que necesita se indican en la Fig. 49. (pg212)

"Programa"
fase A;
fase B;
fase C;
fase D;
fase E;
fase F;

| | Ploter | Perf. Papel | Disco | Cinta |
|---|---|---|---|---|
| a | -- | 1 | -- | 1 |
| b | -- | -- | 3 | 2 |
| c | 1 | -- | -- | 2 |
| d | 1 | -- | 4 | -- |
| e | 1 | 1 | 4 | -- |
| f | -- | 1 | 4 | -- |

FIGURA 49

Queremos evitar interbloqueos, para lo cual utilizamos asignación jerárquica de recursos. Por otra parte liberaremos un recurso cuando sea posible.

Asimilando recursos a variables compartidas, escribir el código correspondiente a la asignación y liberación de recursos para el programa anterior.

**PROBLEMA N.º 4 [2]**

Cuatro procesos quieren usar de vez en cuando cinco recursos C,P,B,A,R según la tabla de la Fig. 50.

| | C | P | B | A | R |
|---|---|---|---|---|---|
| P1 | SI | SI | | | |
| P2 | | SI | SI | SI | |
| P3 | | | SI | SI | SI |
| P4 | SI | | SI | SI | |

FIGURA 50 (pg213)

Cada proceso necesita tener todos los recursos indicados para poder trabajar. Su ciclo de vida es:

```
repeat
    usar varios recursos;
    otras actividades
forever
```

Se pide:
a) Poner algún escenario de interbloqueo utilizando los grafos de asignación de recursos.
b) Resolver el problema utilizando asignación jerárquica de recursos.
c) Idem, pero utilizando asignación total.

**PROBLEMA N.° 5 [3]**

Considere tres procesos: C, R y F (Comerciante, Representante y Fabricante). El comerciante realiza pedidos al representante en forma de mensajes. El representante tramita pedidos al fabricante (mediante mensajes) y el fabricante envía los productos al comerciante (el justificante va como mensaje).

Establecer unas normas para la comunicación entre los procesos y escribir un programa siguiendo estas normas.

**PROBLEMA N.°6 [1]**

Considerar el programa siguiente:

```
VAR v: SHARED RECORD
            …..
            llegada: EVENT v
        END;
  COBEGIN
     «Pl» REPEAT
            S1;
            REGION v DO BEGIN
                S2;
                CAUSE (llegada);
                AWAIT (llegada);
                S3
            END                                    (pg214)
        FOREVER
     «P2» REPEAT
            S4;
```

```
        REGION v DO BEGIN
            S5;
            CAUSE (llegada);
            AWAIT (llegada);
            S6
        END
        FOREVER
COEND
```

Estudiarlo y comentar cómo se comportan los procesos P1 y P2.

**PROBLEMA N.º 7 [2]**

Un usuario de un centro de cálculo usa disco y pantalla (1'), pantalla (10') y disco y pantalla (1') sucesivamente. Hay 2 unidades de disco y 8 de pantalla y más de 8 usuarios potenciales. Se pide:

a) Plantear alguna situación de interbloqueo y describirla mediante un grafo de asignación de recursos.
b) ¿Conviene utilizar asignación total de recursos?
c) ¿Qué solución tenemos_ para evitar los interbloqueos? Escribirla.

**PROBLEMA N.º 8 [2]**

En un cierto lenguaje tenemos:

- el tipo semáforo

- los procedimientos: wait(s), signal(s) y comienza(unaTarea).

Si la tarea T1 es:

```
        begin
            s11;
            s12
        end
```

el efecto de comienza(T1) en:

```
        S1;
        comienza(T1);
        S2;
        S3;
        …..
```

es el mostrado en la Fig. 51. (pg215)

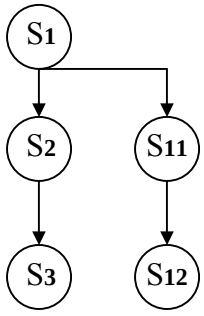



**FIGURA 51**

SE PIDE: traducir a este lenguaje el siguiente programa:

```
VAR v: SHARED Tr;
BEGIN
    S0;
    COBEGIN
        BEGIN
            S1;
```

```
            REGION v DO S2;
            S3
        END;
        BEGIN
            S4;
            REGION v DO S5;
            S6;
        END;
        S7
    COEND;
    S8
END.
```

**PROBLEMA N.º 9 [1]**

Tenemos el problema de los lectores/escritores. (Los lectores son incompatibles con los escritores. Los escritores son incompatibles entre sí. Los escritores tienen prioridad sobre los lectores.)

1) Nos han dado como solución el programa siguiente:

```
«lector-i»                  «escritor-i»
REPEAT                      REPEAT
    WAIT(s);                    WAIT(s); WAIT(s); WAIT(s); WAIT(s);
    lee;                        escribe;                                (pg216)
    SIGNAL(s);                  SIGNAL(s); SIGNAL(s); SIGNAL(s); SIGNAL(s);
    otras cosas                 otras cosas
FOREVER                     FOREVER
```

siendo:

```
        VAR s: SEMAPHORE
```

y teniendo en el programa principal:

```
    BEGIN
        INIT (s, 4);
        COBEGIN
            lector1;lector2;lector3;lector4;
            escritor1; escritor2
        COEND
    END.
```

**Se pide**: Estudie la corrección de esta solución.

2) Estudiar si la solución sería correcta con un solo escritor.

**PROBLEMA N.º 10 [2]**

Tenemos N procesos de la forma: «P_i»

```
REPEAT                          COBEGIN
    otras cosas                     P1; P2; P3; ...; PN
    usa recursoUnico                COEND
FOREVER
```

Queremos que, para usar el recurso, Pi tenga más prioridad que Pj si i > j.

Programe un algoritmo para conseguir gestionar explícitamente esta prioridad.

**PROBLEMA N.º 11 [1]**

Dados dos procesos P1 y P2 cuya programación tiene la siguiente forma:

```
COBEGIN          «P1»          «P2»
   P1;             BEGIN         BEGIN
   P2                 a;            c;
COEND                 (1)           (2)
                      b             d
                   END           END
```
(pg217)

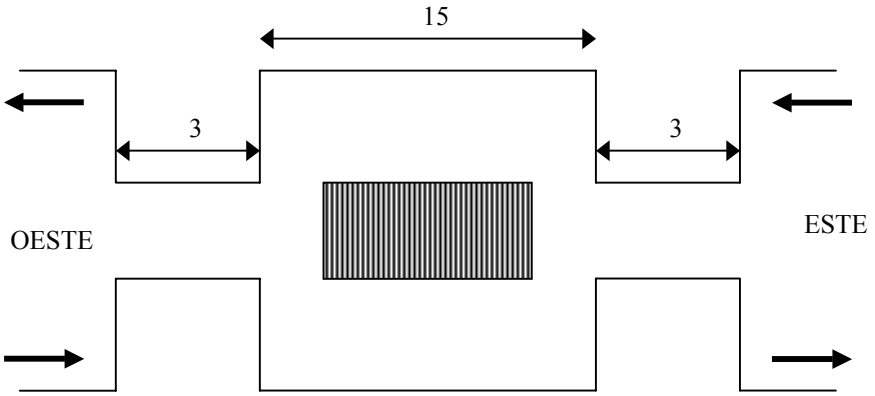



**FIGURA 52**

Completar los programas para que se cumplan las siguientes condiciones de sincronización:

- si P1 llega a (1) antes que P2 a (2), P1 esperará a que P2 llegue a (2).
- si P2 llega a (2) antes que P1 a (1), P2 esperará a que P1 llegue a (1).

**PROBLEMA N.º 12 [3]**

Una carretera cruza dos puentes de una sola vía como se indica en la Fig. 52.

Se pide: programe el comportamiento de los coches del este y del oeste tal que la solución no tenga interbloqueos.

**PROBLEMA N.º 13 [3]**

Tenemos tres tipos de recursos: R1, R2 y R3 y dos unidades de cada tipo. Queremos programar doce procesos concurrentes:

```
p11   p12   p13   p14   (* p1i *)
p21   p22   p23   p24   (* p2i *)
p31   p32   p33   p34   (* p3i *)
```

Los procesos están caracterizados como:

```
«P1i»
   REPEAT
      usa un recurso R1;
      no usa ningún Ri;
      usa un recurso R1 y un recurso R2;
      no usa ningún Ri
   FOREVER
```
(pg218)

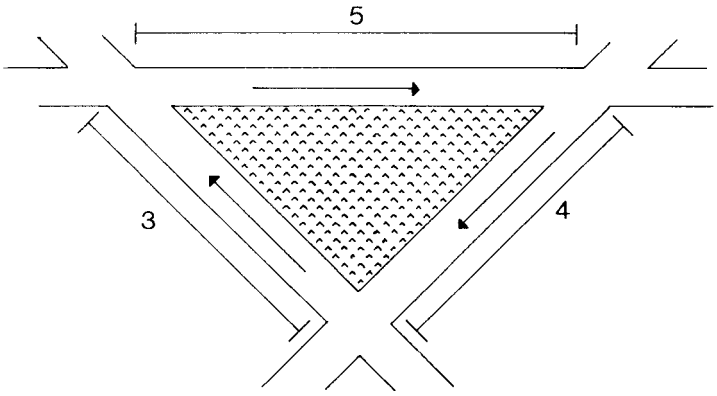



**FIGURA 53**

```
«P2i»
   REPEAT
      usa un recurso R2;
      no usa ningún Ri;
      usa un recurso R2 y un recurso R3;
      no usa ningún Ri
   FOREVER
«P3i»
   REPEAT
      usa un recurso R3;
      no usa ningún Ri;
      usa un recurso R3 y un recurso RI;
      no usa ningún Ri
   FOREVER
```

SE PIDE:
a) Describir una situación de interbloqueo utilizando grafos de asignación de recursos.
b) Escribir un programa correspondiente a la situación anterior.
c) Dar unas normas para evitar los interbloqueos en este problema.
d) Escribir los procedimientos correspondientes a estas normas utilizando regiones críticas condicionales.
e) Idem que d) pero utilizando sucesos.

## PROBLEMA N.º 14 [3]

Tenemos 3 carreteras de una sola vía que se cruzan según indica la Fig. 53. (pg219)

Se pide:
a) Describir alguna situación de interbloqueo y programarla.
b) Proponer alguna regla para evitar los interbloqueos.
c) Programar el comportamiento de los vehículos de acuerdo con la regla anterior.

## PROBLEMA N.º 15 [1]

Dado el programa:

```
VAR v: SHARED RECORD
            sc: event v;
            .....
            .....
       END;
.....
.....
COBEGIN
   pl;
```

```
    p2;
    p3
COEND;
```

siendo:

```
"Pl, P2"                          "P3"
   REPEAT                            REPEAT
      S0;                               S4;
      REGION v DO BEGIN                 REGION v DO BEGIN
         S1;                               S5;
         AWAIT (sc);                       CAUSE (sc);
         S2                                S6
      END;                              END;
      S3                                S7
   FOREVER                           FOREVER
```

SE PIDE: escribir un programa equivalente sin utilizar sucesos.

**PROBLEMA N.º 16 [2]**

Considere un sistema compuesto por tres procesos fumadores y un proceso agente. Cada fumador continuamente hace un cigarro y se lo fuma. Pero para poder fumarse un cigarro se necesitan tres ingredientes: papel, tabaco y cerillas. Uno de los fumadores tiene papel, el otro tabaco y el otro cerillas. El agente tiene una cantidad infinita de los tres ingredientes. El agente pone dos de los ingredientes en la mesa. El fumador que tiene el ingrediente (pg220) que falta puede hacer un cigarro y fumárselo, indicando al agente cuando termine que se lo ha fumado. El agente entonces pone otros dos ingredientes y el ciclo se repite.

SE PIDE: escriba un programa para sincronizar al agente y a los fumadores.

(NOTA: cuando algún fumador no puede fumar se dedica a construir un 'rompecabezas'; es decir, no se duerme).

**PROBLEMA N.º 17 [3]**

Tres agentes y un contable trabajan sin parar en una sociedad que cambia (compra y vende) dólares por otras monedas. En sus operaciones consideran el «cambio» (del dólar) que consiste en un precio de compra y un precio de venta.

El contable se informa de las operaciones realizadas, echa sus cuentas y si considera necesario modificar el «cambio» escribe su nuevo valor en una pizarra para información de los agentes.

El contable utiliza un procedimiento para calcular el nuevo «cambio»:

```
PROCEDURE ajustaCambio (operacion: una0peracion;
                        VAR cambioNuevo: unCambio);
```

Es posible que N llamadas sucesivas a ajustaCambio devuelvan el mismo valor para cambioNuevo, mientras que a la llamada N+1 suba o baje el precio de venta y/o el de compra.

Cada uno de los agentes recibe peticiones de cambio, mira la cotización en la pizarra, realiza la operación y comunica el contable la operación realizada.

SE PIDE: escribir un programa que se comporte según lo hacen los agentes y el contable.

(NOTA1: se considerarán los procesos:

```
COBEGIN
   agente(1); agente(2); agente(3);
   contable
COEND
```

donde contable tiene la forma:

```
REPEAT
   se informa de una operación;
```

```
            ajustaCambio (operación ... );
            .....
        FOREVER
).                                                    (pg221)
```

(NOTA 2: las peticiones de cambio al agente (i) las programaremos como:

```
        RECEIVE (operación, ventanilla[i])
```

y se ignoran los procesos clientes que hacen el SEND correspondiente).

**PROBLEMA N.º 18 [3]**

Cuatro estudiantes están resolviendo problemas. Dos encargados manejan sendas fotocopiadoras.

Cada estudiante resuelve cada problema sin distraerse. Una vez acabado cada problema le interesa fotocopiarlo, pero si hay cola para fotocopiar empieza otro problema, excepto si tiene cinco problemas sin fotocopiar.

Los encargados de las fotocopiadoras cuando no trabajan pasan a espera no activa (duermen). Cuando llega un cliente y ambos están durmiendo, uno de ellos realizará las fotocopias. Cuando hay dos clientes que quieren fotocopiar deben estar despiertos los dos encargados.

SE PIDE: escribir un programa concurrente que se comporte como estos personajes.

```
COBEGIN
    est(1); est(2); est(3); est(4);
    encar(1); encar(2)
COEND;
```

**PROBLEMA N.º 19 [2]**

Tenemos un ordenador llamado TRON que transmite caracteres por una línea serie (bit a bit). Nos han regalado una impresora que recibe los caracteres a imprimir en paralelo (caracter a caracter). Queremos conectar el TRON con la impresora y para ello intercalamos entre los dos un AIM-65, que recibirá bit a bit del TRON y cuando tenga un caracter completo (8 bits) lo transmitirá a la impresora. Como restricciones tenemos:

1) hasta que el AIM-65 no haya tomado el bit que le transmitió el TRON, éste no puede seguir enviado (contención del TRON).

2) hasta que la impresora no haya recibido el caracter que le mandó el AIM65, éste no puede enviarlo otro caracter (contención del AIM-65).

El diagrama de conexión es el mostrado en la Fig. 54.

SE PIDE: programar el comportamiento del sistema descrito.

(NOTA: se ignoran los caracteres de control para efectuar el protocolo de transmisión, así como la velocidad de la misma.) (pg222)

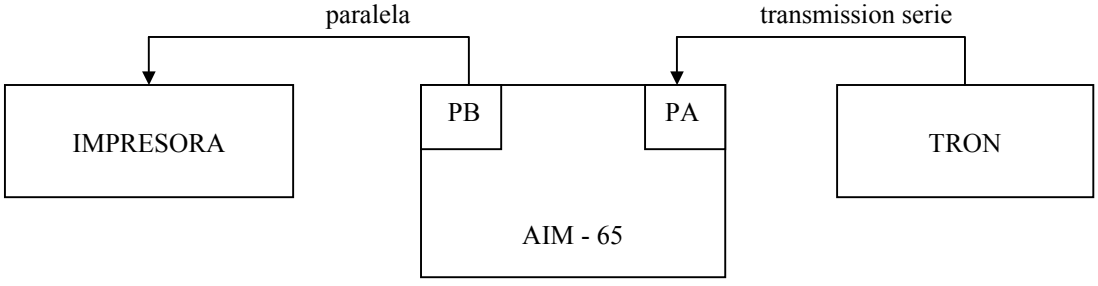



**FIGURA 54**

**PROBLEMA N.º 20 [3]**

Para gestionar la cola de listados en un ordenador con dos impresoras tenemos el siguiente módulo:

```
TYPE nombreDeFichero = ARRAY [1..14] OF CHAR;
VAR bufferListar: BUFFER 10 OF nombreDeFichero;
      colaListar: SHARED RECORD
                        …..
                        …..
                        hayAlgo: EVENT colaListar
                  END;
PROCEDURE aCola;
   VAR nombreFichero: nombreDeFichero;
       numeroLineas: 0...maxInt;
   BEGIN
      REPEAT
         RECEIVE (nombreFichero, bufferListar);
         numeroLineas:= nL (nombreFichero);
         REGION colaListar DO BEGIN
            meteEnColaListar (nombreFichero, numeroLineas);
            CAUSE (hayAlgo)
         END
      FOREVER
   END;
PROCEDURE impresora (i: INTEGER);
   .....
   .....
   BEGIN
      REPEAT
         .....
         .....
         imprimeFichero ( (* cuyo nombre es *) nombreFichero);        (pg223)
         .....
         .....
      FOREVER
   END;
```

Al comienzo se crean entre otros procesos los siguientes:

```
COBEGIN
   ... aCola; impresora (1); impresora (2)
COEND
```

Los usuarios del sistema, cuando lo necesiten, harán:

```
SEND (nombreFichero, bufferListar);
```

SE PIDE:

1) Escribir el procedimiento impresora (i).

2) Completar y modificar el programa para que visualice en pantalla el nombre de los ficheros que se están imprimiendo y los que están en cola esperando a ser imprimidos.

(NOTA: El procedimiento meteEnColaListar mete el nombre del fichero especificado, que tiene númeroDeLíneas, en 'ficheros', poniéndolo en orden por número de líneas.)

**PROBLEMA N.º 21 [4]**

La empresa GUASA se compone de cuatro departamentos: ventas, fabricación, transporte y almacén. Los departamentos se intercambian información; para ello disponen cada uno de un cartero que tiene como función llevar/recoger mensajes a/de otros departamentos. Por atención a los carteros se ha instalado un cajetín de correos con cuatro casilleros (uno para cada departamento). Cuando al cartero del departamento X se le ordena llevar un mensaje al departamento Y, va al cajetín de correos y deja dicho mensaje en el casillero del Y. Asimismo, cuando se le ordena que recoja los posibles mensajes de otros departamentos para el suyo (el X), va al cajetín de correos y recoge lo que haya en el casillero X.

La empresa decide automatizar el envío de correo. Para ello, en cada departamento se instala un terminal conectado a un ordenador. Para mandar un mensaje, se teclea el comando MANDAR seguido del nombre del departamento destinatario y del mensaje. Por ejemplo:

MANDAR fabricación "necesito 100 bombas"

Para recibir todo el correo que haya se teclea el comando RECIBIR. Si hay mensajes para ese departamento se visualizarán por pantalla acompañados (pg224) del nombre del departamento que originó el mensaje. Por ejemplo, si el departamento de ventas tiene correo del de fabricación y del de almacén, cuando en ventas se teclee RECIBIR, aparecerá en su pantalla:

fabricación "hay huelga de embotelladores"
almacén "no me quedan escobas"

Si no hay mensajes se imprimirá:

no tienes correo.

La compañía Todo-Lo-Hago-Bien ha propuesto el programa que se da a continuación para resolver la automatización. Mirando este programa, conteste a las siguientes preguntas:

1) ¿La solución dada cumple con los requerimientos del problema? Razone su respuesta.

2) En caso de que haya respondido negativamente al apartado anterior, realice las modificaciones que crea necesarias para que el programa satisfaga las condiciones del problema propuesto.

3) Para evitar que los procesos "cartero" pierdan tiempo mientras se imprimen mensajes por consola, se propone que la impresión la realice el proceso Imprimir, del cual hay una copia por cada consola que existe y cuya declaración es:

PROCEDURE Imprimir (Dep: TDepartamento);

Programe este proceso y modifique el programa de forma conveniente para incorporar este nuevo proceso.

```
PROGRAM CorreoElectronico; (* programa propuesto *)
  TYPE TDepartamento =(ventas, fabricacion, almacen, transporte);
       TComando = (MANDAR, RECIBIR,....); (* y otros comandos *)
       TOrdenConsola = RECORD
                          Comando: TComando;
                          Argumentos: ARRAY [1..80] OF CHAR
                       END;
       TMensaje = ARRAY [0..40] OF CHAR;
       TOrdenCartero = RECORD
                          Comando: TComando;
                          Destinatario: TDepartamento;
                          Mensaje: TMensaje
                       END;
       TBuzonCartero = BUFFER 10 OF TOrdenCartero;
       TCarta = RECORD
                   Remitente: TDepartamento;
                   Mensaje: TMensaje
                END;                                          (pg225)
```

```
        TBuzon = BUFFER 20 OF TCarta;
    VAR CajetinDeCorreos: ARRAY TDepartamento OF TBuzon;
        BuzonesCarteros: ARRAY TDepartamento OF TBuzonCartero;
    PROCEDURE InterpretaComandosDeConsola (Consola: TDepartamento);
      VAR Orden: TOrdenConsola;
          OrdenCartero: TOrdenCartero;
      BEGIN
        REPEAT
          recogeOrdenDelTeclado (Orden); (* y lo deja en Orden *)
          IF Orden.Comando = MANDAR
          THEN BEGIN
                 OrdenCartero.Comando:= MANDAR;
                 (* extrae de Argumentos: destinatario y mensaje *)
                 OrdenCartero.Destinatario:= DameDestinatario (Orden.Argumentos);
                 OrdenCartero.Mensaje:= DameMensaje (Orden.Argumentos);
                 SEND (OrdenCartero, BuzonesCarteros[Consola])
               END
          ELSE IF Orden.Comando = RECIBIR
               THEN BEGIN
                      OrdenCartero.Comando:= RECIBIR;
                      SEND (OrdenCartero, BuzonesCarteros[Consola])
                    END
               ELSE (* interpreta el comando que sea *)
        FOREVER
      END;
    PROCEDURE Cartero (Departamento: TDepartamento);
      VAR OrdenCartero: TOrdenCartero;
          Carta: TCarta;
      BEGIN
        REPEAT
          RECEIVE (OrdenCartero, BuzonesCarteros[Departamento]);
          IF OrdenCartero.Comando = MANDAR
          THEN BEGIN
                 Carta. Remitente:= Departamento;
                 Carta.Mensaje:= OrdenCartero.Mensaje;
                 SEND (Carta, CajetinDeCorreos[OrdenCartero.Destinatario])
               END                                                    (pg226)
          ELSE BEGIN (* OrdenCartero.Comando = RECIBIR *)
                 RECEIVE (Carta, CajetinDeCorreos[Departamento]);
                 ImprimeEnConsolaDel (Departamento, Carta)
               END
        FOREVER
      END;
    BEGIN (* main *)
      COBEGIN
        InterpretaComandosDeConsola (ventas);
        InterpretaComandosDeConsola (fabricacion);
        InterpretaComandosDeConsola (almacen);
        InterpretaComandosDeConsola (transporte);
        Cartero (ventas);
        Cartero (fabricacion);
        Cartero (almacen);
        Cartero (transporte)
      COEND
    END.
```

**PROBLEMA N.º 22 [2]**

En una aplicación hay tres procesos productores (p l, p2 y p3) y dos procesos consumidores (cl y c2). Los procesos se lanzan como:

```
COBEGIN
   pl; p2; p3;
   cl; c2;
   (* otros procesos *)
COEND;
```

Los procesos productores tienen la forma:

```
"P1"                    "P2"                    "P3"
VAR n: INTEGER;         VAR n: INTEGER;         VAR n: INTEGER
REPEAT                  REPEAT                  REPEAT
   n:= calculos (..);      n:= calculos(..);       n:= calculos(.);
   SEND (n, bl)            SEND (n, b2)            SEND (n, b3)
FOREVER                 FOREVER                 FOREVER
```

y la forma de los procesos consumidores es:

```
"C1"                    "C2"
VAR n: INTEGER;         VAR n: INTEGER;
REPEAT                  REPEAT
   RECEIVE (n, b11);       RECEIVE (n, b12);                    (pg227)
   utiliza (n)             utiliza (n)
FOREVER                 FOREVER
```

siendo b1, b2, b3, b11, b12: buffer 10 OF INTEGER;

Queremos que los mensajes de b1 vayan a b11, los de b2 a b12, los de b3 cuyo contenido sea impar a b11 y los mensajes de b3 cuyo contenido sea par a b12.

SE PIDE:

a) Programe uno o varios (especificar cuántos) procesos distribuidores para cumplir con la regla del párrafo anterior.

b) En cierto ordenador disponemos de un compilador con una primitiva de sincronización "buzón2", tal que sus variables se declaran, se recibe y se envía igual que con buzón, y que además permite preguntar si el buzón está lleno o vacío.

FUNCTION vacio (b: buzon2): BOOLEAN;

FUNCTION lleno (b: buzon2): BOOLEAN;

Se pide programar uno o varios (especificar cuántos) procesos distribuidores aprovechando las características de esta nueva primitiva.

c) Comente brevemente las ventajas e inconvenientes de la primitiva buzón2.

**PROBLEMA N.º 23 [1]**

En cierto ordenador se están ejecutando N procesos, cuyo ciclo de vida

es:

```
"Pi"
   REPEAT
      pideMemoria (cantidadl);
      realiza operaciones;
      pideMemoria (cantidad2);
      realiza operaciones;
      liberaMemoria (cantidad1 + cantidad2);
      otras cosas
   FOREVER
```

La memoria del ordenador se declara como:

```
VAR memoria: SHARED RECORD
                memoriaLibre: INTEGER;
                (* otros campos *)
             END;
```

e inicialmente, toda la memoria del ordenador (1M palabras) está libre. (pg228)

SE PIDE:
a) Programe los procedimientos pideMemoria y liberaMemoria.
b) ¿En qué situación se puede producir un interbloqueo?
c) ¿Bajo qué condiciones podríamos evitar el interbloqueo?

**PROBLEMA N.º 24 [2]**

Realizar uno o dos seguimientos de los programas siguientes, justificando la salida que producen.

programa 1)

```
PROGRAM bla (output);
   VAR impresora : SHARED BOOLEAN;
   PROCEDURE di (n: INTEGER);
      BEGIN
         IF di = 0
         THEN REGION impresora DO WriteLn('bla')
         ELSE COBEGIN
                 di (n-1);
                 di (n-1)
              COEND
      END; (* di *)
   BEGIN
      di (3)
   END.
```

programa 2)

```
PROGRAM fib (output);
   VAR impresora: SHARED BOOLEAN;
   PROCEDURE ram (g, p: INTEGER);
      BEGIN
         IF g < 1
         THEN REGION impresora DO WriteLn('fin',g)
         ELSE BEGIN
                 REGION impresora DO WriteLn(g);
                 COBEGIN
                    ram (p, g-p);
                    ram (g+p, 2*p+g)
                 COEND
              END
      END; (* ram *)
   BEGIN
      ram (5, 3)
   END.
```

(pg229)

(PISTA: pintando un árbol es más fácil).

**PROBLEMA N.º 25 [3]**

Un brazo articulado tiene un conjunto de «N» motores que le permiten efectuar los movimientos para los que fue diseñado. Para efectuar un movimiento, el brazo programa los motores, teniendo en cuenta que no se puede programar un nuevo movimiento sin haber terminado el anterior.

No se exigirá al brazo que un determinado motor intervenga más de una vez en un movimiento.

El brazo obtiene la descripción de los movimientos de un buzón alimentado exteriormente por un hipotético usuario (que no es necesario implementar).

Si en un movimiento es necesario que intervengan «n» motores, se deben recibir «n» mensajes del usuario que programa el brazo.

Programar el procedimiento brazo y el procedimiento motor, teniendo en cuenta que si se programa el movimiento de un motor es necesario indicarle la amplitud del giro y el sentido del mismo (basta con un número entero para ello).

Los motores que intervienen en un movimiento deben comenzar a funcionar simultáneamente.

NOTA: Una posible declaración de tipos sería la siguiente:

```
CONST
   TotalMotores = ... ;
TYPE
   NumMotores = 1..TotalMotores;
   UnMotor = RECORD
                Programado : BOOLEAN;
                GiroYSentido : INTEGER;
             END;
   Mensaje = RECORD
                NumMotor, GiroYSentido : INTEGER;
                UltimoMotorAProgramar : BOOLEAN
             END;
```

(* UltimoMotorProgramar se utilizará para determinar en qué momento hemos recibido la información necesaria para comenzar un movimiento *) (pg230)

**PROBLEMA N.º 26 [3]**

Diseñar y realizar una implementación de la región crítica condicional utilizando como únicas primitivas de sincronización las definidas para los semáforos. Para ello, se debe indicar el código y las estructuras de datos que generaría en los puntos p0, pl, p2 y pa, un compilador que implementase el siguiente esquema de región crítica condicional, utilizando la solución propuesta:

```
VAR v: SHARED RECORD          REGION v DO BEGIN (* p1, entrada *)
          .....                  SI;
          .....                  AWAIT condición; (* p2, espera *)
          (* p0, variables       Sn;
             auxiliares*)      END (* p3, salida)
```

Una vez realizada la implementación, contestar de forma concisa y concreta a las siguientes preguntas:

a) ¿Cuántos semáforos se han utilizado y para qué se ha empleado cada uno de ellos?
b) ¿Cuántas variables auxiliares (no semáforos) se han utilizado y para qué se ha empleado cada una de ellas?
c) En la solución propuesta, ¿cómo se han implementado las colas de procesos existentes en la estructura original de la región crítica condicional?
d) Si se piensa que la implementación propuesta no trabaja adecuadamente en determinadas circunstancias, o que tiene alguna restricción adicional, comentarlo brevemente.

SUGERENCIA: Para simplificar se puede empezar suponiendo que en cualquier momento tan sólo habrá como máximo un proceso esperando por la "condición" de un await. Una vez resuelto este caso, podemos analizar qué problemas se presentan cuando hay varios procesos esperando, e intentar solucionarlos.

**PROBLEMA N.º 27 [2]**

Escriba un algoritmo para resolver el problema de los filósofos utilizando para ello monitores.

**PROBLEMA N.º 28 [3]**

Queremos escribir una herramienta que le permita a un proceso esperar N posibles avisos de N procesos diferentes, la comunicación sería la mostrada en la Fig. 55. (pg231)

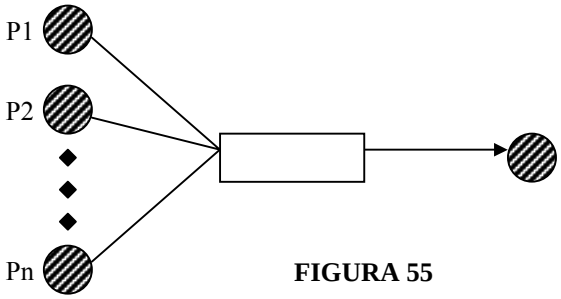



**FIGURA 55**

Queremos que un proceso que avisa se retrase si el que recibe no está preparado, y que el que recibe se bloquee si no hay nadie que avisa (comunicación con rendezvous). Para ello se escribe el siguiente trozo de programa:

```
TYPE NumAvisos = l..N;
     Avisos = SHARED RECORD
                 EsperandoRecibir : BOOLEAN;
                 EsperandoEnviar : ARRAY [NumAvisos] OF BOOLEAN
              END;
VAR Aviso : Avisos;
PROCEDURE Avisa (i : NumAvisos);
   BEGIN
      REGION Aviso DO
      IF EsperandoRecibir
      THEN EsperandoRecibir:=FALSE
      ELSE BEGIN
              EsperandoEnviar[i]:=TRUE;
              AWAIT NOT EsperandoEviar[i]
           END
   END;
FUNCTION Recibe : NumAvisos;
   (* Devuelve el número de aviso recibido *)
   VAR i : NumAvisos;
   BEGIN
      REGION Aviso DO BEGIN
         i:=1;
         (* Miramos si hay alguien retrasado *)
         WHILE NOT EsperandoEnviar[i] AND (i < N) DO i:=i+l;
         IF (i = N) AND NOT EsperandoEnviar[i]
         THEN BEGIN (* No hay nadie *)                                  (pg232)
              EsperandoRecibir:=TRUE;
(*1*)         AWAIT NOT EsperandoRecibir;
           END
      ELSE BEGIN (* Habia alguien *)
              EsperandoEnviar[i]:=FALSE;
              Recibe:=i
           END
   END
END;
```

Suponiendo que:
- La variable Aviso tiene todos sus campos a FALSE.
- Un único proceso llama a Recibe (y no llama a Avisa)
- No hay dos procesos que llamen a Avisa con el mismo parámetro.

Se piede:

1. Si el proceso que llama a Recibe queda bloqueado en (*l*) dejará el valor devuelto por Recibe indeterminado al terminar la función. Modificar el trozo de programa para que esto no suceda.

2. ¿A qué procesos Pi se les da mayor prioridad en esta implementación?

**PROBLEMA N.º 29 [1]**

Dado el siguiente programa:

```
PROGRAM p (input, output);
   VAR impresora,contador : SHARED INTEGER;
       a : INTEGER;
       s : SEMAFORO;
   PROCEDURE Escribe;
      BEGIN
         REGION impresora DO
            REGION contador DO
               WriteLn(contador)
      END;
   PROCEDURE p (a: INTEGER);
      BEGIN
         IF a=1
         THEN BEGIN
                 Wait(s);
                 REGION impresora DO BEGIN                    (pg233)
                    WriteLn('valor de a',a);
                    Escribe
                 END
              END
         ELSE IF a = 2
              THEN BEGIN
                      REGION contador DO
                         contador:=contador+1;
                         Escribe;
                         Signal(s)
                      END
              ELSE BEGIN
                      Wait(s);
                      REGION impresora DO WriteLn('valor de a',a)
                   END
      END;
   BEGIN
      Init(s,0);
      contador:=1;
      Read(a);
      COBEGIN
         p(a);
         p(a MOD 3 + 1)
      COEND
   END.
```

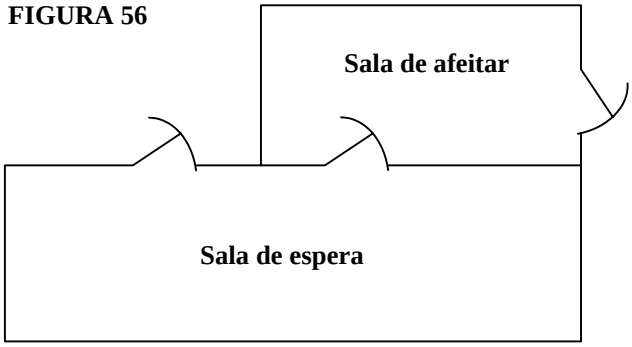
Determinar su salida para las siguientes entradas:

a. 1

b. 2

c. 3

**FIGURA 56**




**PROBLEMA N.º 30 [3]**

Tenemos una peluquería con dos salas como indica la Fig. 56. En esta peluquería (pg234) hay un barbero que afeita a los clientes que llegan, según las siguientes normas:

BARBERO: cuanto termina con un cliente le muestra la salida. Luego comprueba la habitación de espera. Si hay clientes, lleva uno a la sala de afeitar; si no, se duerme en la sala de afeitar.

CLIENTE: entra en la sala de espera. Si hay otros clientes espera con ellos; si no, abre la puerta de la sala de afeitar para ver si el barbero está ocupado. Si el barbero está ocupado, cierra la puerta y espera en la sala de espera; si no lo está, entra y despierta al barbero.

SE PIDE: programar un proceso barbero y un proceso cliente que se comporten según las normas descritas.

**PROBLEMA N.º 31 [1]**

Resolver el problema de los filósofos utilizando asignación jerárquica de recursos.

**PROBLEMA N.º 32 [4]**

En el patio de una casa se encuentran cuatro niños que continuamente efectúan las siguientes tareas:

Inicialmente se encuentran jugando. Cuando un niño tiene hambre recurre a un plato y toma una porción de queso del mismo. Acto seguido duerme durante 30 unidades de tiempo y a continuación vuelve a sus juegos. Este ciclo se repite indefinidamente.

Si cuando un niño intenta comer, encuentra el plato vacío, avisa a la madre de este suceso, esperando entonces a que la madre ponga comida en el plato.

La madre, entre otras cosas, se encarga de realizar las siguientes dos tareas:

- despertar a los niños que, después de comer, han dormido 30 unidades de tiempo (los niños son muy dormilones y no se despertarían de otra forma).

- Reponer el plato, cuando esté vacío, con cuatro porciones de queso, si es que algún niño solicita comida.

SE PIDE: implementar un programa en el que existan cuatro procesos «niño» y un proceso «madre», los cuales actúen siguiendo el comportamiento anteriormente descrito. La solución propuesta debe estar libre de interbloqueos e inanición. (pg235)

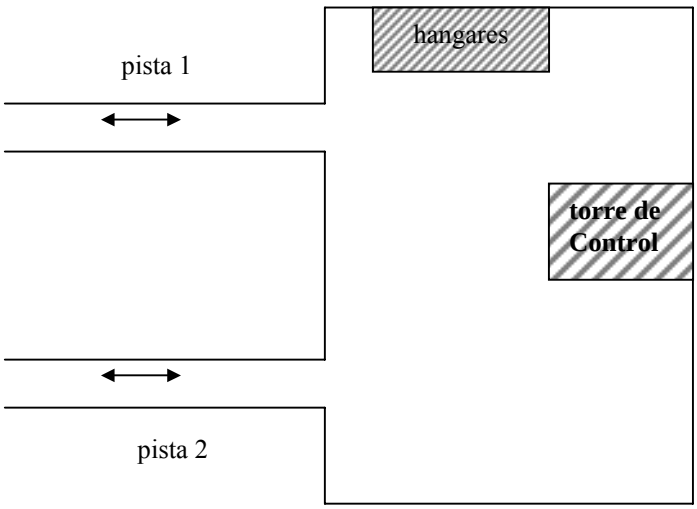



**FIGURA 57**

**PROBLEMA N.º 33 [3]**

En un aeropuerto existen dos pistas de aterrizaje/despegue y una torre de control (donde hay un solo controlador), como se indica en la Fig. 57.

Hay 20 aviones que realizan vuelos de reconocimiento y que tienen este aeropuerto como base de operaciones. El ciclo de vida de un avión es:

```
REPEAT
    indica a la torre que quieres despegar;
    espera permiso de la torre, que además dice qué pista utilizar;
    rueda por la pista indicada y despega;
    indica a la torre que ya despegastes;
    haz vuelo de reconocimiento durante 2 horas;
    indica a la torre que quieres aterrizar;
    espera permiso de la torre, que además dice qué pista utilizar;
    aterriza en la pista indicada y para;
    indica a la torre que ya estás en el hangar;
    otras cosas (* mantenimiento *)
UNTIL pasen 20 años;
```

Los procesos que existen son:

```
COBEGIN
    avion(1); avion(2); avion(3); ...; avion(20);
    torre
COEND;
```

(pg236)

SE PIDE: Programe el proceso avión y el proceso torre de tal manera que se eviten las colisiones en las pistas (una misma pista no puede ser utilizada simultáneamente por más de un avión).

**PROBLEMA N.º 34 [3]**

Blancanieves y los 7 enanitos viven en una casa donde sólo existen 4 sillas, que los enanitos utilizan para comer.

Cuando un enanito vuelve de trabajar en la mina comprueba si hay una silla libre para sentarse. Si existe una silla libre, entonces indica a Blancanieves que ya está sentado, y espera pacientemente SU TURNO a que le sirva. Cuando le ha servido, Blancanieves le indica que puede empezar a comer. El enanito come y cuando acaba, deja la silla libre y vuelve a la mina. Por su parte, Blancanieves cuando no tiene a ningún enanito pendiente de servirle, se va a dar una vuelta (no duerme).

SE PIDE: programar los procesos enanitos y Blancanieves para que se comporten según lo anteriormente descrito, utilizando como guía el pseudocódigo siguiente:

```
"Enanito (i)"
        REPEAT
            trabajar en la mina;
            esperar a coger silla libre;
            indicar a Blancanieves que está sentado;
            esperar SU TURNO a que le sirva;
            comer;
            abandonar la silla
        FOREVER
"Blancanieves"
        REPEAT
            IF hay alguien esperando
            THEN BEGIN
                        recoger, por orden, la siguiente petición;
                        servirle la comida;
                        indicarle que puede comer
                    END
            ELSE me doy una vuelta
        FOREVER                                                    (pg237)
```

**PROBLEMA N.º 35 [2]**

Escriba el código de un monitor (*), que permita suspender a un proceso un intervalo de tiempo dado. El monitor debe tener dos «entry points»:

```
1) PROCEDURE entry retrasame (quienSoy: unProceso; cuantoTiempo: CARDINAL);
        (* suspende al proceso identificado por "quienSoy" *)
        (* durante "cuantoTiempo" segundos     *)
2) PROCEDURE entry haPasadoUnSegundo;
        (* es invocado por un proceso reloj, cuando ha pasado *)
        (* 1 segundo. Debe reanudar a los procesos cuyo *)
        (* intervalo de suspensión haya expirado     *)
```

siendo:

```
CONST nmrProcesos = 20;

TYPE unProceso = [1..nmrProcesos];
```

(*) El monitor es del tipo definido por Hoare:

- se puede ejecutar más de una operación «continue» en un procedimiento monitor.

- en una estructura «queue» pueden estar encolados más de un proceso.

**PROBLEMA N.º 36 [3]**

Tenemos un sistema con n procesos (n >> 20) que quieren usar un recurso. Interesa que puedan utilizar el recurso simultáneamente el máximo número de procesos.

```
COBEGIN
    p1; p2; ...; pn
COEND
```

```
"Pi"
    REPEAT
        sus asuntos;
        usa el recurso
    FOREVER
```

Por otra parte, nos piden que evitemos la eventualidad (y si no es posible, que la minimicemos) de que 13 procesos se encuentren usando el recurso.

SE PIDE:

a) Escribir un programa que cumpla en lo posible los requisitos anteriores.

b) ¿Es posible evitar que 13 procesos se encuentren utilizando el recurso? Razone su respuesta.

**PROBLEMA N.º 37 [4]**

El metropolitano de Madrid ha construido una línea de metro «CIRCULAR». (pg238) Dicha línea sólo consta de una vía y, por tanto, los trenes sólo pueden ir en un solo sentido. Se supone que existen «t» trenes y «e» estaciones.

Cada tren está compuesto únicamente por un vagón, y cada vagón sólo tiene una entrada por donde sólo cabe un viajero a la vez. Los viajeros que llegan al andén de una estación cualquiera esperan a la llegada de un tren para tomarlo, indicando la estación de destino a la que se dirigen. Cuando un tren llega a una estación, los viajeros que van en el tren y tienen como estación destino la actual, abandonan el tren.

Por otra parte, para que no existan colisiones entre trenes hay que instalar un sistema de sincronización entre los trenes:

- Un tren no abandona la estación en que está hasta que se le asegure que en la siguiente estación no hay otro tren.

- Los trenes paran dos minutos en cada estación para dar tiempo a que los viajeros suban y bajen del tren, a no ser que tengan que esperar más tiempo por cuestiones de sincronización con otros trenes. Se supone que los trenes una vez que parten de la estación inicial recorren la línea indefinidamente.

SE PIDE: programar el comportamiento de los trenes y de los viajeros, según el protocolo anteriormente esbozado.

**PROBLEMA N.º 38 [1]**

Estudie el siguiente programa (Hyman, 1966):

```
PROGRAM Exclusi6nMutua;
   VAR flag: ARRAY [0...1] OF BOOLEAN;
         turno: 0..1;
   PROCEDURE P (i: INTEGER);
      BEGIN
         REPEAT
            flag[i]:= TRUE;
            WHILE turno < > i DO BEGIN
               WHILE flag[1-i] DO;
                  turno:= i
            END;
            usa recurso;
            flag[i]:= FALSE;
            otras cosas
         FOREVER
      END;                                                    (pg239)
   BEGIN (* main *)
      flag[0]:= FALSE; flag[1]:= FALSE;
      turno:= 0;
      COBEGIN
         P(0); P(1)
      COEND
   END. (* main *)
```

¿Hay exclusión mutua entre P(0) y P(I) en el uso del recurso? Justifique su respuesta.

**PROBLEMA N.º 39 [1]**

Deseamos implementar buzones con comportamiento "no bloqueante" con respecto a las operaciones send y receive.

Alguien propone añadir dos nuevas operaciones a realizar sobre variables de tipo buzón:

```
1) PROCEDURE vacío (b:buzon): BOOLEAN;
       (* devuelve TRUE si el buzón está vacío    *)
       (* devuelve FALSE si el buzón no está vacío *)
2) PROCEDURE lleno (b:buzon): BOOLEAN;
       (* devuelve TRUE si el buzón está lleno    *)
       (* devuelve FALSE si el buzón no está lleno *)
```

Con estas nuevas operaciones se implementan las primitivas envía y recibe (sobre una variable de tipo buzón) de la siguiente manera:

```
PROCEDURE envía (VAR b:buzon; m: mensaje; VAR enviado: BOOLEAN);
   BEGIN
      IF NOT lleno(b)
      THEN send (m, b);
             enviado:= TRUE
      ELSE enviado:= FALSE
      END
   END envia;
PROCEDURE recibe (VAR b: buzon; VAR m: mensaje; VAR recibido:BOOLEAN);   (pg240)
   BEGIN
      IF NOT vacio(b)
      THEN receive (m, b)
             recibido:= TRUE
      ELSE recibido:= FALSE
      END
   END recibe;
```

¿Cree que es correcta la implementación de "envía" y "recibe"? Justifique su respuesta.

# *Bibliografía*


[Barn87] J. G. P. Barnes, Programación en Ada. Ed. Diaz de Santos, 1987.

[BenA82] M. Ben-Ari, Principles of Concurrent Programming. Prentice-Hall,

[BriH73] Brinch Hansen, P. Operating System Principles. Prentice-Hall, 1973.

[BriH77] Brinch Hansen, P. The Architecture of Concurrent Programs. Prentice-Hall, 1977.

[DOD83] Reference Manual for the Ada Programming Language. ANSI/MIL-STD 1815A, D.O.D., 1983.

[Glea84] Richard Gleaves, Modula-2 for Pascal Programmers. Springer-Verlag, 1984.

[Holt78] R. C. Holt, G. S. Graham, E. D. Lazowska, M. A. Scott, Structured Concurrent Programming with Operating Systems Applications. Addison-Wesley, 1978.

[Holt83] R. C. Holt, Concurrent Euclid. The Unix System and Tunis, Addison-Wesley, 1983.

[Math84] Mathai Joseph, V. R. Prasad and N. Natarajan, A Multiprocessor Operating System. Prentice-Hall, 1984.

[Nara84] Narain Gehani, Ada Concurrent Programming. Prentice-Hall, 1984.

[Pete85] Peterson J. L. & Silverschatz A. Operating System Concepts, 2.° ed., AddisonWestley, 1985.

[Tane84] Andrew S. Tanenbau, Structured Computer Organization, 2.° ed. Prentice-Hall, 1984.

[Tane87] Andrew S. Tanenbaum, Operating Systems, design and implementation. PrenticeHall, 1987.

[Wirt85] Niklaus Wirth, Programming in Modula-2, Springer-Verlag. 3rd corrected edition, 1985.